

# Towards a Block-Level ML-Based Python Vulnerability Detection Tool\*

Amirreza Bagheri<sup>ab</sup> and Péter Hegedűs<sup>ac</sup>

## Abstract

Computer software is driving our everyday life, therefore their security is pivotal. Unfortunately, security flaws are common in software systems, which can result in a variety of serious repercussions, including data loss, secret information disclosure, manipulation, or system failure. Although techniques for detecting vulnerable code exist, the improvement of their accuracy and effectiveness to a practically applicable level remains a challenge. Many existing methods require a substantial amount of human expert labor to develop attributes that indicate vulnerabilities. In previous work, we have shown that machine learning is suitable for solving the issue automatically by learning features from a vast collection of real-world code and predicting vulnerable code locations. Applying a BERT-based code embedding, LSTM models with the best hyperparameters were able to identify seven different security flaws in Python source code with high precision (average of 91%) and recall (average of 83%). Upon the encouraging first empirical results, we go beyond this paper and discuss the challenges of applying these models in practice and outlining a method that solves these issues. Our goal is to develop a hands-on tool for developers that they can use to pinpoint potentially vulnerable spots in their code.

**Keywords:** deep learning, vulnerability detection, source code embedding, data mining

## 1 Introduction

In today's applications, security bugs (i.e., vulnerabilities) in software are becoming increasingly difficult to detect, allowing hackers and attackers to profit from

---

\*The research was supported by the Ministry of Innovation and Technology NRD Office within the framework of the Artificial Intelligence National Laboratory Program (RRF-2.3.1-21-2022-00004) and by project TKP2021-NVA-09, implemented with the support provided by the Ministry of Innovation and Technology of Hungary from the National Research, Development and Innovation Fund, financed under the TKP2021-NVA funding scheme.

<sup>a</sup>Institute of Informatics, University of Szeged, Hungary

<sup>b</sup>E-mail: [bagheri@inf.u-szeged.hu](mailto:bagheri@inf.u-szeged.hu), ORCID: 0000-0001-9691-7937

<sup>c</sup>E-mail: [hpeter@inf.u-szeged.hu](mailto:hpeter@inf.u-szeged.hu), ORCID: 0000-0003-4592-6504

their exploits. Tens of thousands of such flaws are discovered and fixed each year. Manually auditing source code and discovering vulnerabilities is time-consuming at best, if not impossible.

In our previous work [2], we have shown that machine learning is suitable for solving the issue automatically by learning features from a vast collection of real-world code and predicting vulnerable code locations. The dataset was gathered from GitHub and contains Python code with a variety of vulnerabilities that is embedded into a vector space using one of three embedding models (word2vec, fasttext, BERT) [5, 16, 8]. Individual code tokens and their context are extracted from the source code of the vulnerable files to provide data samples for fine-grained analysis. Then, we trained various machine learning (ML) models to see how effective they were at identifying vulnerable code parts.

The entire training process can be divided into two parts: first, an embedding model is trained using its parameters, such as min-count (how frequently a token must appear in the training corpus to be assigned a vector representation), and second, the system is trained using its parameters, such as min-count or iterations. After that, the code blocks can only be encoded in their vector representations. We found that LSTM models were the most suitable, thus we used them and trained them with different hyperparameters, such as the number of neurons or dropout, in the second stage. Applying a BERT-based code embedding, LSTM models performed the best, they were able to identify seven different security flaws in Python source code with high precision (average of 91%) and recall (average of 83%).

Following a successful empirical evaluation, the results must be implemented in practice. However, there are several difficulties in putting the above-described method into practice and making it available as a developer tool. The training data samples are code snippets (extracted from vulnerability fixing commits), but when we use vulnerability identification in practice, we use the entire program as input. To use code embedding and model prediction, we need a method for efficiently locating code blocks within the program. Furthermore, because these code blocks may overlap, we require a method for aggregating block-level predictions.

In this paper, we focus on overcoming these challenges and outline a potential developer tool that developers can use. We apply a small focus area and a sliding context window to divide the code into blocks. The focus area moves through the code, and with each step, the model gathers surrounding information, generates a prediction based on that context as input, and uses that prediction to determine the vulnerability rating of the focus area. In a developer tool, the different classification confidence levels may be represented by different colors. To summarize our contributions, we provide a block-level vulnerability prediction method that is practically applicable to Python code, in contrast to the majority of other research initiatives, which are primarily focused on Java, C, C++, or PHP and do not provide guidance on practical application. Furthermore, existing vulnerability prediction approaches provide predictions at higher abstraction levels, such as methods, classes, or files, whereas we aim for finer-grained, smaller block-level predictions.

## 2 Related Work

This section describes previous works in finding vulnerabilities and also attempts a classification, although there are many different criteria under which approaches can be compared. The advantages and disadvantages of the previous approaches are described.

### 2.1 Vulnerability Prediction Based on Software Metrics

What characteristics should be considered while determining whether or not code is vulnerable? For a long time, the most popular features were software and developer metrics. Code churn, developer activity, coupling, amount of dependencies, and legacy metrics are examples [22]. Such metrics are widely employed as features in fault prediction models [13], and they are extremely important in the field of software quality and reliability assurance. Nagappan et al. [23], for example, use organizational measures to predict software failures.

Although it appears that those data may be employed in vulnerability prediction, there are significant issues with this. For starters, two pieces of code could have the same characteristics but completely different behaviors, resulting in a distinct risk of being exposed. They also tend not to transfer well from one software project to the next. The most serious complaint is that such measurements fail to capture the semantics of the code [30], and that this method ignores the source code, program behavior, and data flow. The method effectively applies a presumption that particular meta characteristics will be linked to security issues, which is not always accurate [15].

Many vulnerabilities can, for example, be found in extremely simple programs. The simplest or most direct solution to an algorithmic problem frequently lacks the safeguards and measures necessary to prevent attacks, which is precisely why software engineers working under time constraints or with little familiarity with security issues have difficulty. Code complexity isn't a perfect indicator of security problems, and there are analogous arguments and counterexamples for the other measures as well. However, it must be accepted that software metrics can provide at least some insight. This is demonstrated in the following studies, which employ machine learning techniques and code metrics to anticipate the occurrence of software security problems. Shin et al. [30] estimate vulnerabilities in JavaScript projects using nine complexity measures, with a low false positive rate but a large false negative rate. Using linear discriminant analysis and Bayesian networks, the authors used code complexity, code churn, and developer metrics to identify vulnerabilities in a later paper [29], attaining 80 percent recall and 25 percent false positives. Chowdhury et al. [4] use complexity, coupling, and cohesion metrics to try to anticipate software vulnerabilities using methodologies that have previously been used for fault detection. They conduct a study on Mozilla Firefox releases and anticipate vulnerabilities using decision trees, random forest, logistic regression, and naive Bayes models, with precision and recall of roughly 70% and 70%, respectively. Zimmerman et al. contributed to the list by looking into code churn, code

complexity, code coverage, organizational metrics, and actual dependencies [40].

They discovered a weak but statistically significant link between the indicators studied and utilized logistic regression to identify vulnerabilities based on them, with an emphasis on Windows Vista's proprietary code. The measures were able to anticipate vulnerabilities with a median precision of 60%, but a recall of 40%, which was disappointing. When using support vector machines to anticipate vulnerabilities using import statements, Neuhaus et al. [24] found an average precision of 70% and recall of 40% when using import statements in the Mozilla project. Yu et al. [38] include a variety of factors, including software metrics like the number of subclasses or methods in a file, as well as crash features and code tokens with associated tf-idf scores. As a result, their strategy incorporates a variety of perspectives. They can forecast vulnerabilities at the file level and get very good results in reducing the amount of code that needs to be reviewed by humans to detect a vulnerability.

Other researchers have made predictions based just on commit messages. Zhou et al. [39] use a K-fold stacking technique to examine commit messages to forecast whether or not a commit contains vulnerabilities. In contrast, Russel et al. [26] discovered that humans and Machine Learning systems both struggled to identify build failures or defects based just on commit messages. Our approach, the suggested method, does not use external code measurements and instead learns characteristics directly from the source code.

## 2.2 Anomaly Detection Approaches for Finding Vulnerabilities

The task of characterizing normal and anticipated behavior and finding deviations from it is known as anomaly detection. The assumption is that code that does not follow the indicated criteria is frequently the source of a bug. To evaluate source code and uncover normal coding patterns, data mining techniques were applied. For instance, Li et al. [19] developed PR-Miner, a tool that can discover code patterns in any programming language and has shown to be highly beneficial. Their method, which is based on associating programming patterns that are used in tandem, is independent of the language used, and the violations detected by their tool have been confirmed as flaws in Linux, PostgreSQL, and Apache. However, a basic issue is that faults that are themselves normal patterns are routinely neglected, resulting in common vulnerabilities going undetected [36].

Rare programming patterns or API usages, on the other hand, may be labeled as false positives simply because they are uncommon. Several anomaly detection methods have a high risk of false positives [10]. Anomaly detection in code is not a simple way for detecting security vulnerabilities, because it is difficult to tell when a violation of typical code patterns has a security implication and when it does not. The method utilized in this study varies from traditional anomaly detection in that explicit labels are used to train a model on both vulnerable and secure code, avoiding the dubious assumption that "normal" = "right." It falls under the heading of susceptible code pattern analysis.

## 2.3 Vulnerable Code Pattern Analysis and Similarity Analysis

In comparison to learning about abstract metrics or the notion of proper code, it seems almost natural to just try to answer the question: What does vulnerable code often look like? Vulnerable code pattern analysis and similarity analysis are two slightly different approaches to answering that question. The name suggests that similarity analysis achieves exactly that. The purpose is to locate the most comparable code fragments to a susceptible code snippet, presuming that they are at risk of sharing the vulnerability. This method works well for identical or almost identical code clones in which the compared code fragments' inherent structure is quite close [18], a circumstance that occurs frequently, especially in the open-source community.

Susceptible code pattern analysis examines vulnerable code segments using data mining and machine-learning techniques to discover common characteristics. These characteristics are patterns that can be used to detect vulnerabilities in new code portions. As detailed by Ghaffarian et al. [10], most of the work in this area gathers a huge dataset, analyzes it to extract feature vectors, and then applies machine-learning techniques to it. Both methodologies are often used to analyze source code without running it, which is known as a static analysis, while some academics also use a dynamic analysis. The bottom line is that, unlike 'conventional' static analysis, the characteristics are generated automatically or semi-automatically, removing the need for subjective human specialists. An unbiased model can be developed by learning directly from a dataset of code what susceptible code includes.

In many cases, those approaches use a coarse granularity, classifying entire programs [11], files [29], components [24], or functions [37], making pinpointing the specific position of a vulnerability hard. Some researchers, such as Li et al. [18] and Russell et al. [26], employ a finer representation of the code. Furthermore, the approaches differ in several ways, including the language used, the data source, the dataset size, the labeling process, the granularity level of the analysis (whole files down to code tokens), the machine learning model used, the types of vulnerabilities examined, and whether the model can be used in cross-project predictions or only on the project it was created for. To begin, some fundamental approaches utilizing various machine learning techniques will be discussed.

Following that, deep learning-based techniques are investigated in further depth. Morrison et al. [22] look at security flaws in Windows 7 and Windows 8.8 using a variety of machine learning approaches such as logistic regression, naive Bayes, and others. With very unsatisfactory results, support vector machines, and random forest classifiers. As a result, the precision and recall levels were quite poor. Pang et al. [25] extract labels from an internet database in a fairly basic manner. To classify the entire Java code, utilize a combination of feature selection and n-gram analysis. susceptible or not vulnerable classes. They use a simple n-gram model in combination with feature selection methods to integrate related features and limit the number of irrelevant features taken into account on a relatively small dataset of four Java android applications. After that, they use support vector machines

as their learning algorithm, with 92 percent accuracy, 96 percent precision, and 87 percent recall inside the same project, and 65 percent in cross-project prediction (training on one project and trying to classify vulnerable files in another one).

Shar et al. [28] use machine learning to detect XSS and SQLI vulnerabilities in PHP code and reduce false positives. They manually select specific code attributes before training a multi-layer perceptron to augment static analysis tools. They discovered fewer vulnerabilities than a static analysis tool, but with reduced false positive rates, resulting in a satisfactory outcome. They adopt a hybrid technique with dynamic analysis in their later work [28], which significantly improves their earlier results, as tested on six large PHP projects. They also try unsupervised predictors, which are less accurate but still a fascinating study topic.

Raw source code is analyzed as text by Hovsepian et al. [15]. They used an Android email client built in Java as an example, focusing on evaluating the source code as if it were a natural language and processing files as a whole. They convert files into feature vectors made up of Java tokens with their respective counts in the file after filtering out comments. These feature vectors are classed as susceptible or clean in a binary approach. Finally, a support vector machine classifier is trained to determine whether a file is vulnerable. This classifier has an accuracy of 87 percent, with 85 percent precision and 88 percent recall. Their success demonstrates that evaluating source code as natural text and gaining insight without sophisticated models of code representation is possible. Unfortunately, the application on a single software repository limits their work. For a comparable job, they later utilized decision trees, k-nearest-neighbor, naive Bayes, random forest, and support vector machines [27].

### 2.3.1 Deep Learning for Vulnerability Prediction

Several papers have successfully used deep learning models to automatically learn features for fault prediction [35]. The following works show how this approach can be applied to vulnerability detection. Russell et al. [26] employ recurrent and convolutional neural networks to scrape a large codebase of C projects from GitHub, the Debian Linux distribution, and synthetic examples from the SATE IV Juliet test suite, resulting in a database of over 12 million functions. They produce the binary labels 'vulnerable' and 'not vulnerable' for the routines using three separate static tools, as well as a randomly initialized one-hot embedding for lexing. Convolutional and recurrent neural networks are used for feature extraction at the core of their research, followed by a random forest classifier. The best results came from convolutional neural networks, which allowed for fine-tuning of precision and recall against each other.

Russel et al. are not only among the first researchers to use deep representation learning directly on source code from a large codebase, but they are also able to use a convolutional feature activation map to highlight suspicious parts of the code, rather than simply classifying a whole function as vulnerable, with their work. The work of Liu et al. [20] is based on the notion that violations that are consistently remedied are genuine positives, whereas violations that are disregarded are likely

to be either not important or false positives. They look into changes in 730 Java projects, use the static bug detection tool Findbugs to find changes that are fixing a violation reported by that tool, then follow the violations across versions to see if they are addressed or ignored. They can use this information to determine which tool-reported infractions are consistently disregarded over several revisions and which are addressed almost immediately. They collect the code patterns that correlate to infractions using an abstract syntax tree as a representation.

Liu et al employ an unsupervised learning approach to extract features of code, focusing primarily on patches to learn fix patterns, rather than building a binary classifier on 'vulnerable' or 'not vulnerable.' As a result, their method may be characterized as a type of similarity analysis. The discovered coding patterns are encoded into a vector space using an embedding layer, the discriminating features are learned using a convolutional neural network, and violations with learned features are clustered using an X-means clustering technique. They discovered that, while security-related breaches are uncommon, they are common in 30 percent of the projects. Furthermore, the research shows that only a small percentage of breaches are corrected. Liu et al. discovered that for 90% of fixed breaches, a chunk of merely 10 lines of code or fewer is adequate to capture the relevant context. The CNN produces patterns that are nearly identical to the tool's violation description and are used to build fixed patterns. One of the top five suggested fix patterns can fix roughly one-third of a test set of violations. Liu et al. also chose 10 open-source Java projects to offer proposals to based on the modifications proposed by their program, with 67 of the 116 suggestions being accepted right away. Of course, their technology can only suggest patches that match previously discovered fix patterns.

### 2.3.2 Long-short Term Memory Networks

Although Gupta et al. [12] and Dam et al. [6] have demonstrated that extended short-term memory networks are well suited to modeling source code and correcting faults in C code, the latter was likely the first to do so. to learn features automatically for anticipating security vulnerabilities [7]. They Extract the code from a publicly available dataset including 18 Java applications. utilizing Java Abstract Syntax Tree to replace all methods in the source file Some tokens are available in generic versions. They then employ LSTM to train syntactic and semantic skills.

A random forest classifier and semantic characteristics. They got over 91 percent precision for within-project vulnerability prediction, and after training a model on one project, it got over 80 percent precision and recall in at least four of the other 17 projects. VulDeePecker [18] is a deep learning-based vulnerability detection method. The authors propose the first dataset of vulnerabilities targeted for deep learning algorithms, which is derived from the National Vulnerability Database and the Software Assurance Reference Dataset maintained by the NIST and come from popular C and C++ open-source products.

Li et al. want to design a tool that doesn't rely on humans to determine features but still has a low rate of false positives and false negatives. They divided files into code gadgets, which are semantically related lines of code that are grouped,

focusing on critical areas of library and function API calls in a very sophisticated manner. Only two sorts of vulnerabilities are evaluated: buffer errors and resource management problems. On different subsets of their data, Li et colleagues chose bidirectional long short-term memory networks, attaining a precision of roughly 87 percent, with better results if the network is trained on manually selected function calls. Harer et al. [14] used LSTM networks to detect and resolve C vulnerabilities in the synthetic SATE IV code base. They were able to use a sequence-to-sequence strategy to develop solutions for discovered vulnerabilities, albeit measuring and comparing their success is difficult. Similarly, Gupta et al. [12] employ RNNs in a sequence-to-sequence configuration to remedy flawed C code, while not focusing on security vulnerabilities, fixing 27 percent of their applications completely and 19 percent partially.

### 3 Approach

Our approach to vulnerability detection is to analyze code tokens and their surrounding tokens to determine the context in which they exist. Using embedding layer models, the code is embedded into semantically meaningful numerical vectors. After that, an LSTM network is used to recognize vulnerable code features and categorize code as vulnerable or not vulnerable. The overview of the approach is shown in Figure 1.

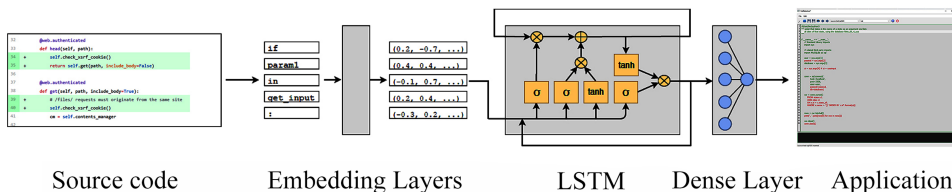


Figure 1: Overview of the approach

#### 3.1 Data Source

In prior research, the researchers found that applying their model to code from the same project that it was trained on yielded better results in detecting vulnerabilities [25]. Cross-project prediction resulted in a significant reduction in precision and recall. In the works of Russel et al. [26] and Li et al. [18], the best results were obtained when dealing with a synthetic data set rather than code from real applications. Nonetheless, because such a vulnerability detection tool appears to be the most desirable and final result, our strategy attempts to leverage a huge dataset of real-life source code to train a model that can be applied to any code, not just one project.

For numerous reasons, the entire dataset was compiled from publicly available GitHub projects: First, because GitHub is the world’s largest repository of source



code, the amount of meaningful data accessible is unlikely to be insufficient for this application. Second, unlike synthetic code bases, nearly all GitHub projects contain 'natural' source code in the sense that they are real-world projects. Third, the data is open, making it easier to re-examine and reproduce the work, which is difficult in studies that focus on proprietary code, for example. Because GitHub is primarily a version control system, it is centered on commits, and as Zhou et al. [39] explains, it is possible to detect vulnerabilities by looking at commits. Patches are commits that remedy a defect or vulnerability and consist of two versions, one buggy and one updated and correct. Vulnerable code patterns can be discovered by evaluating the differences between the old and new versions.

## 3.2 The Data Collection Process

Commits that fixed numerous vulnerabilities were gathered. Each vulnerability needed its dataset after the data had been collected and filtered. Table 1 summarizes the fundamental data of the dataset, such as the number of repositories and commits that comprise it, the number of modified files that contain known vulnerabilities, the number of lines of code, the number of distinct functions they contain, and the total number of characters. The following sections will demonstrate the suitability of this dataset by using it to train the model.

Table 1: Vulnerability Dataset

Vulnerability	Repo.	Commits	Files	Functions	LOC	Chars
SQL Injection	457	582	721	7452	102558	5960074
XSS	52	89	102	983	18916	1236587
Command injection	125	225	354	3561	48031	2740339
XSRF	112	189	384	6418	76198	3368206
Remote code execution	71	88	186	4198	40591	1955087
Path disclosure	175	204	332	4596	62303	2814413

### 3.2.1 Python Vulnerabilities

**Injection attacks** - An injection attack is based on user input that causes unexpected or harmful behavior when processed or executed. A user can sometimes access or change data without permission by exploiting an injection vulnerability, which usually allows the user to have the interpreter (such as the server or the operating system) execute arbitrary commands. Injection attacks can be avoided by vetting all user input and employing so-called "sanitization" techniques that convert harmful to harmless inputs, such as filtering out special characters.

**SQL injection** - The OWASP foundation lists SQL injections as one of the top security flaws, ranking them among the most prevalent and dangerous flaws affecting web applications. The Common Weakness Enumeration defines SQL injection

as "when a SQL command is provided to a downstream component, the software generates all or a portion of it using externally influenced input from an upstream component, but fails to neutralize or does so in a way that may cause it to change the intended SQL command." When user-controllable input contains SQL syntax that has not been removed, it can be misinterpreted and executed as a SQL statement. This can be used to alter searches, such as accessing files that should not be accessible or adding new statements that can alter or destroy databases. If its sanitization is not thorough, any form of a database-driven website could end up being the subject of such an exploit.

**Command injection** - According to the Common Weakness Enumeration, the software constructs all or a portion of a command using externally affected input from an upstream component, but it fails to neutralize or does so incorrectly specific aspects that could change the intended command when it is sent to a downstream component. This is another instance of untrusted data being executed, but instead of being directed at a SQL database, it is directed at a command run by the system being attacked, such as the server shell. An attacker could then read, modify, or delete files that they shouldn't have access to.

**Remote code execution** - The primary distinction between command injection and remote code execution is that command injection executes an OS system command, whereas remote code execution executes actual programming code on the target machine. It is also sometimes used to define a hacking goal rather than a vulnerability, in the sense that an attacker can execute arbitrary commands on a system by exploiting a vulnerability.

**Various types of session hijacking** - The main goal of session hijacking is to allow an attacker to enter a client's connection with a server, typically by obtaining or guessing a valid session token and then posing as a trusted client. To connect to a client using a maliciously set session ID by a third party, a user must be tricked into clicking a link that contains the session ID as a parameter. Because the malicious third party now has access to the session token, the active session can be accessed. An attacker could even gain access to a logged-in account. By using cross-site scripting to obtain a session token, the attacker can hijack the shared session between the client and server.

Man-in-the-middle attacks are also included in session hijacking. An attacker pretends to be the connection partner on both sides of a conversation between two systems, possibly a client and a server. Because they are effectively acting as a proxy, the attacker can view and occasionally change the content of the communication. By utilizing appropriate encryption and certifications, man-in-the-middle attacks are avoided. The term "replay attack" refers to an attack in which the attacker, posing as the original originator of the transmission, records a legitimate portion of communication between two parties (such as a client and a server) and sends it again later. The attacker can access features and data that were only intended to be accessible to the original sender if suitable protective measures are not in place (primarily secret one-time session IDs).

**Cross-site scripting** - Cross-site scripting, also known as XSS, is one of the most serious flaws in web applications. It frequently appears on OWASP's top ten

list of vulnerabilities. Unsanitized data is also central to the cross-site scripting problem. This time, a user adds custom code to a website or URL before passing it on to other users, who will see the code as part of the page and run it in their browser. The CWE defines cross-site scripting as follows: The program either does not neutralize user-controllable input at all or neutralizes it incorrectly before including it in output that is used to create a web page that is served to other users.

A guest book that accepts arbitrary input is a simple example of stored cross-site scripting. A visitor may post plain text or Javascript, which is saved on the website permanently and distributed to other users, who will receive and run the Javascript code. Of course, changes can be made, such as using different input methods and producing executable code in languages such as Flash. Another example is an email that contains a link to another website, but the URL contains malicious Javascript that, when clicked, executes the malicious code. To prevent XSS attacks, user-generated content should be sanitized with tools such as HTML escape and others.

**Cross-Site request forgery** - The CWE defines cross-site request forgery as follows: The web application does not, or is unable to, thoroughly verify whether the request was submitted by a well-formed, legitimate, consistent user. This is further explained below: If a web server is designed to accept requests from clients without any means of verifying that they were made voluntarily, an attacker may be able to trick a client into sending an unintended request to the website that would be treated as a legitimate request. This can expose data or result in accidental code execution and can be accomplished via a URL, image load, XMLHttpRequest, or other means.

**Directory traversal/path disclosure** -When a user changes the input in such a way that paths of a file system that were not intended to be accessed are exposed, this is known as a path traversal or directory traversal vulnerability. According to the CWE, the software does not properly neutralize special elements in the pathname that could cause it to resolve to a location outside of the restricted directory. The software generates a pathname from external input to identify a file or directory that is located beneath a restricted parent directory. A common example of this vulnerability is a website that displays a file whose path is specified in a URL parameter. The attacker can explore the file system and possibly show files that weren't intended to be accessible by altering this parameter to contain some `"../.."`.

### 3.2.2 Scraping GitHub

The first step is to build a dataset, or more precisely, to find a large number of commits that address a security issue. Because the goal is to cover a wide range of vulnerabilities, each vulnerability type requires multiple examples. Commits are the main topic of interest in our work because the process of patching a flaw indicates the presence of the flaw in the first place and provides the basis for labeling the data afterward. The GitHub search API can only handle certain types of requests, and the number of results for each request is limited to 1000. Filters cannot be

implemented in the search API, unlike the regular search available to users, so filtering for only the programming language is not possible. As a result, after obtaining the results and selecting the few relevant and useful ones from among them, this filtering must be done manually. As a result, the approach taken here is to write a script that searches the Github API for contributions containing various security-related search phrases, then filters out everything that isn't relevant, such as code written in a different programming language or configuration files. The script is included in the repository and authenticates with an API token.

Initially, a lengthy list of security-related terms was used. These terms are based on prior research citezhou2017automated, the CVE database, and the OWASP foundation's list of security risks. To collect the data, a script was written that connected to the GitHub API via the requests library. The keyword list must be supplied at the beginning of the script. This first set of keywords will be combined with a second set of keywords related to improvements, repairs, or modifications in order to consider every possible combination of the first and second set elements. Because the second set of terms denotes a problem or a solution, the combinations should be useful (but not sufficient) in distinguishing genuine security improvements from numerous other mentions of vulnerabilities, such as examples in showcase projects for educational purposes.

However, it quickly became clear that only a few of those keyword combinations were truly relevant to the task at hand. Some, like 'vuln', 'XXE', 'malicious', or 'CVE', were overly broad and yielded a wide range of results; others, like 'dos' (as an abbreviation for denial of service), yielded completely unrelated results due to overlap of meanings (in this case, 'dos' referring to an old Windows operating system, and, even more frequently, the very common Portuguese *As*). As a result, the available options were significantly reduced. After combining every keyword from the revised first set with every keyword from the second set, a search request is sent to Github for each of the combinations. It should be noted that this only means that the names (and thus URLs) of commits and repositories are gathered; no actual source code or even a diff file is downloaded at this time. After combining every keyword from the revised first set with every keyword from the second set, a search request for each combination is submitted to Github. It should be noted that this only collects the names (and thus URLs) of commits and repositories: no actual source code or even a diff file has been retrieved.

### **3.2.3 Filtering the Results**

The second priority was to find projects that display security flaws, exhibit exploits, or serve as tools for attacking or preventing exploits. While those works frequently include useful examples of vulnerabilities, they rarely include commits that repair them, but rather commits that introduce them into the codebase on purpose. Furthermore, they run counter to the work's methodological assumptions, as the goal is to learn about vulnerable code as it appears in real-world projects where developers make legitimate mistakes. As a result, an attempt is made to screen such projects out.

With the `-b` parameter, the script could include a list of keywords to indicate projects that should be avoided. The repository names were searched for the following keywords: "offensive", "pentest", "vulnerab", "security", "hack", "exploit", "ctf", "capture the flag", "attack". The README files for the remaining projects are then downloaded from Github. The next step is to obtain the diff files. In the GNU diff model and similar representations used by GitHub, a diff is a text file that represents the changes made in a commit. It contains some metadata (such as the filename and change line number), the modified lines, and three lines of code before and after. A '+' at the beginning of a line denotes a new and fixed line, whereas a '-' denotes a line that was eliminated in favor of the repair [20]. A commit on Github might include modifications to multiple files at once.

A single HTTP request can be used to download the diff for a commit URL. This is a far easier way than cloning the entire repository and selecting individual files from a certain point in the project's history, which appears to be impossible at this time due to the size of the dataset and computational and temporal constraints. The preceding phase produces a large number of code diffs that can be used to recreate important lines of code in the state before and after the modification. The diff from GitHub includes the modified lines as well as three lines before the first change and three lines after the last change for each changed file, so there isn't much context for the change. However, the vast number of changes that can be mined with this method may more than compensate for the comparatively limited context offered for each change. The time it took to run all of those queries was over 80 hours.

To build the dataset, we obtained only the diff files and recreated the 'before version' and 'after version' of the required code snippet, each with the modified lines and three lines above and below them. The goal was to classify the first version as vulnerable and the second version as 'not vulnerable', which yielded some pleasing results. The classifier that had learned from the training set was able to accurately classify the validation set samples and determine whether they belonged in the 'previous/vulnerable' or 'after/fixed' categories. When the model was applied to a new file containing source code, it went through several parts of it and tried to identify them, and the problem became evident.

That endeavor resulted in an astonishing number of false positives. The reason behind this is that the dataset had the same number of (actual) positives and negatives, whereas in reality, Figure 2: In between numerous lines of 'clean' code, retrieving the snippet in the state before and after the commit from a git diff, the old vulnerable version in red, the new vulnerable version in green The dataset does not accurately reflect the class unbalanced nature of the data to which the classifier should be applied. Of course, this was clear from the start, but owing to the aforementioned time and processing resource constraints, it appeared that collecting the diffs was simply the best technique that could be done at all. This was not accurate, and a better solution may be found.

```

@@ -368,12 +368,11 @@ def _make_flat_wins_csv(self, **kwargs):
368 368     Note that this view removes win, notification and customer response entries
369 369     that might have been made inactive in duecourse
370 370     """
371     - sql_str = "SELECT id FROM wins_completed_wins_fy"
372     - if self.end_date:
373     -     sql_str = f"{sql_str} where created <= '{self.end_date.strftime('%m-%d-%Y')}'"
374     -
375 371     with connection.cursor() as cursor:
376     - cursor.execute(sql_str)
372 + if self.end_date:
373 +     crusor.execute("SELECT id FROM wins_completed_wins_fy where created <+ %s", (self.end_date,))
374 + else:
375 +     crusor.execute("SELECT id FROM wins_completed_wins_fy)
377 376     ids = cursor.fetchall()
378 377
379 378     wins = Win.objects.filter(id__in=[id[0] for id in ids]).values()
    
```

Vulnerable  
(Before)

```

- sql_str = "SELECT id FROM wins_completed_wins_fy"
- if self.end_date:
-     sql_str = f"{sql_str} where created <= '{self.end_date.strftime('%m-%d-%Y')}'"
-     cursor.execute(sql_str)
    
```

Not Vulnerable  
(After)

```

+ if self.end_date:
+     crusor.execute("SELECT id FROM wins_completed_wins_fy where created <+ %s", (self.end_date,))
+ else:
+     crusor.execute("SELECT id FROM wins_completed_wins_fy)
    
```

Figure 2: Vulnerable and Not Vulnerable parts selection

### 3.2.4 Downloading the Dataset

We noticed that downloading the source code in a reasonable amount of time was possible if all of the filterings were done beforehand in a clever way to keep the number of downloaded repositories to a bare minimum. First, the commit is examined to see if it contains keywords related to the vulnerability. The diff file is then examined to see if any files with the code language of choice are affected. If this is not the case, the commit can be ignored because only commits that change specific language source code files are taken into account. The commit is then compared to the previously downloaded commits. By definition, many open-source repositories are forks or clones of one another, or they contain the commit history of other projects. Duplicates are excluded. The distinction is then thoroughly examined. Each change in the commit has an effect on a specific file. The filename is reviewed for each modification to see if it contains terms that indicate it is a showcase project - a file called "SQL exploit" is more likely to be part of a project exhibiting vulnerabilities than a patch that fixes an inadvertent vulnerability.

The body of the diff file is then processed. If HTML tags or the keywords 'sage' are used, the diff is no longer considered. Although HTML code is sometimes embedded in some files, the vulnerabilities in those files are almost never in the same code. Sage is an open-source mathematics system, and some commits include

parameters and variables that are useful to it but not relevant to this project. Finally, the change is checked to see if any lines of code have been removed or replaced. If there are only minor changes, the algorithm will struggle to determine which lines are vulnerable. Finally, after much deliberation, it is determined which commits are truly worth downloading.

Pydriller [32], a tool for downloading repositories containing intriguing commits and traversing their commitments to identify all the matches with the commits that remain in the collection of interesting ones, is only now being used. Some checks are performed anew for each commit. The commit is skipped if the prior file is empty. The commit is also avoided if the previous file is longer than 30,000 characters. Similar to the file name, the commit message is reviewed for suspicious terms. Finally, the source code for the dataset is downloaded and saved.

### 3.2.5 Flaws in the Data

When we dug deeper into the data, we discovered that the process of collecting vulnerability samples based on commit messages is far from perfect. There were still some (albeit minor) commits that contained exploit implementations rather than fixes, such as setups for capturing the flag, attack demonstrations, or cyber security tools like Burp Suite. Some commit messages, for example, read 'fix remote code execution,' and this vulnerability is repaired somewhere, but the same commit also contains, for example, eight other files with minor and significant changes that may or may not be related to the issue indicated in the commit message. It's difficult to tell whether modifications are related to the commit message's stated goal without human supervision or predetermined knowledge.

The answers for several keywords were just unspecific. There were many results for the phrase brute force in which a brute force strategy was utilized to solve a problem rather than a defense against a brute force attack. As a result, the findings were not particularly useful. A similar issue arose with the phrase tampering, which was used seldom and for a variety of reasons (including DNS tampering, but also game data manipulation for cheating purposes). The term "keyword hijacking" was frequently used in a figurative sense, for example, to describe a person or application that inserted undesirable but authorized material, or to describe data fields or entries that were used by the developers for other purposes as intended. Many fixes and changes relating to developers traversing their file structures, not an attacker attempting to do so, were found using the phrase directory traversal.

Changes were occasionally overly convoluted and spanned numerous files, including those that were not written in Python. The more complex the modifications and the more lines changed, the more difficult it is to model and learn from the sample. Another issue is that many vulnerabilities are defined by the lack of specific defense mechanisms, such as XSRF tokens or nonces/counters that prevent replay attacks. Fixing those vulnerabilities sometimes does not alter or remove a susceptible section of code, resulting in insights into what vulnerable code looks like, but instead adds a few extra lines. In other circumstances, those lines can be positioned in a variety of ways, with a variety of ways to provide the needed

functionality. Learning to notice the lack of something vague that is required is far more difficult than learning to recognize a very explicit erroneous piece of code that is there.

Commits using replay attacks typically had both of the aforementioned issues: They're dispersed throughout a lot of files, and they usually add new lines rather than alter an existing, broken code segment. As a result, this kind of vulnerability had to be ruled out. There were just a few results for man-in-the-middle attacks that were trying to harden an application against them rather than performing them. And the defense systems were so specialized that they yielded little usable information. The majority of the unauthorized commits were likewise not related to fixing susceptible code segments, but instead invoked methods or handled errors that were not particularly tied to a vulnerability.

There were simply too many applications outside the realm of security and vulnerabilities that were only concerned with pretty formatting of outputs rather than preventing vulnerabilities exploiting format strings, and there were simply too many applications outside the realm of security and vulnerabilities that were only concerned with pretty formatting of outputs rather than preventing vulnerabilities exploiting format strings. Other types of vulnerabilities, such as cross-site scripting, command injection, cross-site request forgery, path disclosure, remote code execution, open redirect vulnerabilities, SQL injection, and so on, did provide excellent learning opportunities.

### 3.2.6 Filtering the Data

Individual samples were subjected to specific constraints to improve the dataset's quality. Only files with a length of fewer than 10,000 characters were considered. This offers some advantages: Long portions of comments, docstrings, and manually specified variables are common features of very long files. Furthermore, they act as a form of 'long tail' in terms of computing costs, requiring a significant amount of time to analyze for very small advantages. Finally, certain manual examinations revealed that they do not appear to contain the best quality code. Commits that removed or changed a file in more than 10 different locations were removed from the sample to improve the dataset's quality even more. Such bulk modifications are likely to affect several different concerns at once, rather than just one. Of course, such steps lowered the number of samples. In the case of SQL injections, for example, the dataset was reduced from 842 repositories and 903 commits affecting 2354 files totaling 212913 lines of code to 457 repositories, 582 commits, 721 files, and 102558 lines of code. The quality of the data did not suffer as a result of the reduction, as a test of the final model with the non-trimmed dataset yielded no better results.

A severe flaw in the code was introduced at this time, which was only discovered and repaired late in the process. After identifying which lines of code in the diff file were susceptible, they were removed from the source code and labeled as such. The rest of the file was then divided into even blocks of the same length as the vulnerable code snippets on average, and tagged as 'not vulnerable'. Notice



how the splitting occurs initially, followed by the separation of vulnerable and non-vulnerable individuals. The issue with this method was that it regarded susceptible code areas differently than non-vulnerable code areas. The procedure of constructing a code block differed: vulnerable blocks were extracted directly from the source code, whereas clean blocks were constructed using the block-splitting algorithm. As a result, the vulnerable blocks developed some specific characteristics that the trained classifier could easily recognize.

Some of the susceptible areas were probably very long (with entire functions removed, for example) Since the majority were relatively brief (one or two lines modified), resulting in an average of medium length, thus the clean code was divided into medium-length blocks, which doubled as a proxy for their vulnerability status. When the classifier was applied to a new source code file cut into even blocks and should determine which were vulnerable, the outcome was excessively high precision and recall numbers, as well as poor performance.

### 3.3 Labeling

The data is tagged using information from the commit context, similar to Li et al. [18]. The bits of code that were altered or deleted in such a commit can be labeled as vulnerable, and the version after the fix, as well as all the data around the affected component, can be labeled as not susceptible. Of course, there are times when a repair fails to cure an issue, when many vulnerabilities exist at the same time, or when a new vulnerability is introduced. This strategy ignores all of it because the key goal is simple automation without the requirement for human expert oversight. Furthermore, everything marked as 'not-vulnerable' should be regarded as 'at least not demonstrated to be vulnerable'.

### 3.4 Representation of the Source Code

Simple techniques like a bag of words representations have previously shown unsatisfactory results and are unable to capture the semantic context of code by design. They may be promptly rejected. Others, such as Russell et al. [26] and Hovsepian et al. [15], claim that an AST representation is required to mine patterns from code, while others, such as Liu et al. [20], argue that this is not the case. Furthermore, Dam et al. [7] claim that, in addition to human-engineered features and software metrics, ASTs may be unable to capture the semantics buried deep within source code. Code is sequential data akin to natural text, and long short-term memory networks are created specifically for modeling such data, with excellent results.

Given all of this, our technique is built to work directly on source code as text. Because code snippets are used as samples, the method could be compared to an n-gram technique, however, the snippets are far longer than those used in n-grams. To account for the code's locality aspect [34], the context surrounding each code token will be emphasized for learning features.

### 3.4.1 Choosing Granularity

As Morrison et al. [22] explain, binary-level predictions and analysis on the level of whole files provide little insight because developers often already know which files are vulnerable to security issues, and developers prefer, if possible, a much finer approach at the level of lines or instructions. Dam et al. [7] start their paper with some persuasive examples, suggesting that there are files with comparable metrics, structure, and even almost identical tokens, one of which may be clean and the other vulnerable, despite the same metrics. A technique that 'zooms in' to study small bits of code individually may be more promising than a top-down approach that looks at entire files. Our method employs a fine-grained approach, examining each character in the code as well as its context. Only in this way can the specific position of the vulnerability be pinpointed.

### 3.4.2 Preprocessing the Source Code

Tokens at the source-code level in languages like Python include identifiers, keywords, separators, operators, literals, and comments. While some researchers [25] omit separators and operators, others [37] remove a large number of tokens and keep only API nodes or function calls. Comments are removed from this work because they do not affect the program's behavior. Even if they have some predictive value for vulnerability status, this is not the type of data that should be learned by the model, which is designed to discover vulnerable code. Otherwise, the source code remains unchanged. Hovsepian et al [15] take a similar strategy. Generic names are not used to substitute variables or literals; everything is taken exactly as it is represented. Because neural networks work with numerical vectors of uniform size, it's vital to represent code tokens as vectors that keep the semantic and syntactic information from the code. Furthermore, the vector's variables must be chosen in such a way that the vectors are manageable in size.

Li et al. [18] apply carefully constructed code gadgets, Hovsepian et al. [15] use a simple bag-of-words strategy, Russell et al. [26] train a randomly initialized one-hot-embedding, and Liu et al. [20] use word2vec. A naive one-hot encoding is one possibility, but it is utterly oblivious to the semantic meaning of tokens. An embedding layer, on the other hand, uses vectors with high cosine similarity to represent semantically comparable code elements. A code snippet is turned into a list of representations of its tokens in our method. Language keywords, identifiers such as function names and variables, integers, operators, and even whitespaces, brackets, and indentations are examples of these. Every one of the tokens must be embedded, or represented by a numeric vector.

As a result, a complete portion of the code is converted into a vector of vectors of numbers. All the embedding layers have previously been used successfully for similar projects [20]. Aside from the conceptual advantages over a one-hot encoding, it also requires significantly smaller vector sizes, making it computationally less expensive. It was picked as the best embedding method for our strategy. Because no pre-trained language model for Python code is currently available, embedding

layers must be trained first. A corpus of high-quality Python code is obtained for this purpose, once again from GitHub. The embedding layer model is trained on this corpus to prepare it for the task of encoding Python code tokens as vectors.

The vulnerable and non-vulnerable components had to be treated the same the entire way up to the labeling stage to properly analyze the data. The data was divided into equal chunks, which were then tagged as vulnerable if they overlapped with one of the vulnerable code segments, otherwise as clean. The technique of breaking down source code into blocks has been given in a simplified manner thus far. Initially, the comments are filtered out of the code, similar to the work of Hovsepyan et al. [15] and many others, because they are unlikely to alter the vulnerability of a file. A small focus window iterates over the entire source code in  $n$ -steps. To avoid tokens being split in half, the focus window always starts and stops at a character that represents the end of a token in Python, such as a colon, bracket, or whitespace. The surrounding context of roughly length  $m$ , starting and stopping at the border of code tokens, is determined for this focus window, with  $m > n$ . The context will largely lie behind the focus window if it is at the beginning of the file, and if it is in the middle, the surrounding context will span a snippet that spans equally before and after the focus window.

As a result, there are a lot of overlapping blocks. It is labeled as vulnerable if the entire block contains partially vulnerable code, otherwise, it is labeled as clean. This ensures that code snippets that contain a vulnerability are identified. The parameters  $n$  and  $m$  will be optimized, and their ideal values will be found through experimentation. According to Liu et al. [20], a portion of merely 10 lines of code is usually enough to capture the important context for a vulnerability. The next step is to convert those code blocks, which are simply lists of Python characters, into numerical vector lists.

### 3.5 Embedding Layers

A suitable embedding layer model trained on Python source code is required to encapsulate the code tokens in a numerical vector. A substantial training base of code, ideally made up of clean, working Python code, is necessary to train this model. This research follows the heuristic that popular code projects are of high quality, similar to Bhoopchand et al. [3] and Allamanis et al. [1]. It is worth noting that those repositories are likely to include minimal security flaws and defects in general. We propose our recommended embedding layers in our previous paper and test them with different hyperparameters to see the effectiveness of each of them [2].

### 3.6 Selecting the Machine Learning Model

Many machine learning models and methodologies have been applied to vulnerability detection, with inconsistent results, including SVMs, decision trees, random forest, and naive Bayes models. However, not all of those models are equipped with the needed features. Our technique aims to construct a model that can learn

vulnerability aspects from code token sequences. Source code is sequential data by definition, as the effect of each line is highly dependent on the effects of the instructions around it. This will result in many false positives when trying to detect a vulnerability. Rather, the idea is to discover that a token is 'bad' when used in a specific way, with tokens that have come before it.

Deep learning-based models, particularly RNNs, are particularly well-suited to representing code locality while also being able to capture far more context than ngrams [6]. Deep neural networks, particularly recurrent neural networks and long short-term memory networks, have numerous advantages. To recap, such networks may describe sequential data by using an internal state as a "memory" to keep track of prior inputs and contextualize data. RNNs, on the other hand, suffer from vanishing or exploding gradients, making it difficult to train them on longer sequences since the distance between the occurrence of a piece of information and the point at which it becomes relevant exceeds the RNN's capabilities. LSTM, on the other hand, were created to cope with problems like this since they can learn how long information should be preserved. They have been effectively employed in modeling code and are designed for the type of task required in Our method. As a result, an LSTM is used as the model in this study. We decided to utilize Bi-LSTM for the final version of the tool, but because we started with LSTM, we will explain it first.

### 3.7 Preparing the Data for Classification

The information gathered is still in the form of code snippets that are vulnerable and not vulnerable. The snippets are translated into a list of tokens, and each token is replaced with its vector representation based on the chosen embedding layer model. Each vector has a binary label, with '0' indicating vulnerable and '1' indicating not vulnerable or unknown status. The data is divided into three sets: training, validation, and testing. 70% of the data is chosen at random as a training set, 15% is chosen as a test set for validation, and 15% is kept aside for a final evaluation after the experiments. Dam et al. [7] utilized the same ratios, Russell et al. [26] split their dataset into 80 percent training, 10% validation, and 10% final test set, and Li et al. [18] used an 80-20 split between train and test set.

It is worth noting that the validation set is not used to learn parameters; instead, it is used to assess the model's performance after it has learned its parameters on the training set. This evaluation is taken into account while adjusting the model's hyperparameters, and all findings are finally presented using the final test set, which the model has never seen before. To obtain an equal length of vectors for each sample, the lists of vectors are shortened and padded.

## 3.8 Training the LSTM

### 3.8.1 Architecture of the Model

A sequential model is built using the Keras package. The most crucial aspect comes first: the LSTM layer. The goal of this layer is to discover features that are linked to a code snippet's vulnerability state. There is no need for a separate dropout layer because the LSTM layer is susceptible to numerous hyperparameters, including dropout and recurrent dropout. The classes should be weighed appropriately because the data is inherently class imbalanced (there are far more clean code blocks in the training data than vulnerable ones). This ensures that, even though there are many more instances of clean code, the examples of vulnerable code are handled appropriately in training. The class weights are determined automatically using the scikit-learn library's class weight function. The activation layer, a dense output layer with a single neuron, follows the LSTM layer. Because the purpose is to produce a forecast between 0 and 1 for the two classes of non-vulnerable and vulnerable code, the activation function utilized here is a sigmoid activation function.

### 3.8.2 Selecting Hyperparameters for the LSTM

Many choices must be taken when it comes to the LSTM hyperparameters. The hyperparameters are adjusted and tested empirically to identify the ideal configurations after calculating some plausible beginning values based on other research and common sense. Technically, the metric and loss functions are hyperparameters Figure 3. Because our approach prefers a fair balance of false positives and false negatives and the classes are already weighed, the F1 metric appears to be particularly well suited to evaluate overall performance. As a result, the F1 score is chosen as the LSTM model's optimization criterion. In the scripts, the F1 metric and its accompanying loss function are custom defined. The number of neurons is, of course, a key hyperparameter that defines the model. It has an impact on learning ability. More neurons let the model learn a more complex structure, but training the model takes longer. The dimensionality of the output space is likewise determined by the number of neurons.

The batch size specifies how many samples are displayed to the network before the weights are changed again. As a result, when making a prediction later, the model should not be trained with a batch size smaller than the number of samples used at the time. To compare the outcomes, a range of various batch sizes are used. A batch size of one single sample or a batch size of the entire training set is the most extreme value. Batch sizes of 32, 64, and 128 samples are commonly employed in the middle of the two. LSTM, like many other models, can be overfitted with training data, lowering their predictive effectiveness. Dropout is a regularization strategy in which input and recurrent connections to LSTM units are occasionally randomly omitted from the next step of the training, preventing the network from updating its weights. This decreases the risk of the network overfitting by depending too heavily on a few inputs.

There are two types of dropout in LSTM: the standard dropout describes the proportion of units that are dropped from the inputs; the cheval dropout describes the fraction of units that are dropped from the inputs. The recurrent dropout is the percentage of units that leave the recurrent condition. A normal dropout rate is between 10% and 50%. Experimentation will establish the ideal dropout. Finally, the number of epochs, or the number of times the learning algorithm will run through the entire training data set, must be changed. In the literature, epochs are commonly referred to as 10, 100, 500, or even 1000. Several optimizers from the adam family will be tried to discover which produces the best results. A model with optimal configurations can be calculated when all of those hyperparameters have been modified.

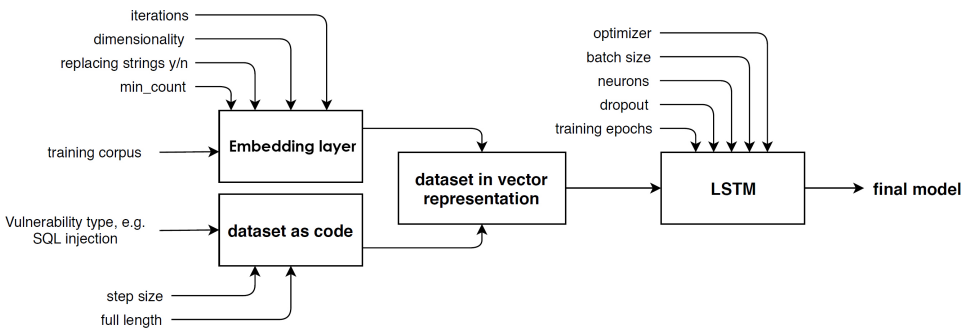


Figure 3: Creating models with different hyperparameters

### 3.8.3 Selecting the Optimizer

The objective for the F1 score was chosen as a criterion for the model's performance because our strategy is to attain high precision and recall at the same time. To determine how 'wrong' the forecasts are at a given point, a loss function based on the F1 score will be employed. The optimizer must update the model parameters until the global minimum is found to minimize the loss function. Simply remove the gradient of the loss concerning the weights multiplied by a modest amount called the 'learning rate' from the weights to be improved. With each iteration of the optimization, the gradient is calculated for a distinct sub-sample of the data and is thus subject to statistical fluctuation, which is why this approach is called Gradient Descent" (SGD). However, if the loss function is not convex or there are ill-conditioned regions, SGD can become stuck in a local minimum. This can be changed by lowering the learning rate. A slow learning rate, on the other hand, suggests that the network will not learn rapidly enough. What factors should be considered when determining the learning rate?

Fortunately, the learning rate does not have to be set in stone and may be dynamically adjusted. The adam optimizer dynamically selects a learning rate. It was

first published in 2014 [17], and it is built primarily for deep neural networks, where it produces excellent results quickly and is frequently used as a go-to optimization approach for a variety of issues. It considers prior updates as well as the first and second moments of the gradient, which are defined as the expected value of that variable to the power of one or two, respectively - the mean and centered variance of the gradient. It combines the advantages of the Adaptive Gradient Algorithm (adagrad) [9] and Root Mean Square Propagation (RMSprop) [33], according to the authors. Adagrad adjusts the learning rate for different characteristics and performs exceptionally well on sparse datasets with a large number of missing samples. Its disadvantage is that it has a very slow learning rate. RMSprop, a variant of adagrad, adjusts learning rates based on recent gradient magnitudes for the weight and works well on both online and non-stationary issues.

When calculating momentum, it only considers gradients in a fixed window. Other optimizers, such as adadelta, nadam, adamax, NAG, and others, will not be discussed in length here. Since the chosen loss function, the F1 score is not convex, SGD will probably not converge towards the optimal solution. According to IBM, the adam family of optimizers (which includes RMSprop, adagrad, and others) should converge under certain conditions. Li et al. [18], Russell et al. [26], and Dam et al. [7] employ adamax, Russell et al. [26] use the conventional adam optimizer, and Dam et al. [7] utilize RMSprop, however, the applicability depends heavily on the dataset's peculiarities. The Adam optimizer is utilized as a starting point for our technique, and it may be empirically compared to other optimizers to see which provides the best results in practice.

### 3.9 Evaluation

True positives, true negatives, false positives, and false negatives are frequently the basis for evaluation when it comes to prediction and categorization. They have been referenced before, but they will be adequately clarified here. Positive and negative refer to the prediction, so a prediction of 'vulnerable' would be positive, and a prediction of 'not vulnerable' would be negative in this work. True and false refer to whether the forecast matches the actual value or an external evaluation. As a result, a false positive is a piece of clean code that the classifier incorrectly labels as vulnerable, a true positive is a vulnerability that was correctly identified, a false negative is an actual vulnerability that was not classified as such, and a true negative is a piece of code that was classified as 'not vulnerable' and is free from vulnerabilities. Precision and recall are two metrics that are directly derived from those four numbers.

The rate of genuine positives within all positives is the precision. It assesses how accurate the model is in terms of how many of the predicted positives are true positives, or, to put it another way, how much trust can be placed in the positive categorization and how many false alarms are generated. The recall, also known as sensitivity, is a metric that compares the percentage of correctly detected positives to the total number of positives. It could be interpreted as a measure of how diligently the classifier looks for all positives - or how much is missed.

When the data set is class imbalanced, meaning there are many more positives than negatives or vice versa, accuracy does not provide much insight. When it comes to vulnerability detection, the majority of code fragments will be clean, and vulnerabilities will be uncommon. For example, Morrison et al. [22] discovered that only 0.003 percent of their Windows code was vulnerable, while Shin et al. [31] found that 3% of their Firefox files were vulnerable. When genuine positives are few and true negatives are common, a classifier can attain high accuracy ratings even though it misses the majority of the positives because the many true negatives make the total result appear to be extremely accurate. As a result, the accuracy alone is insufficient for this application. The F1 score is a balanced score that considers precision and memory. The F1 score is better suitable for class-imbalanced data sets since it is less easily influenced by a large number of true negatives.

In an ideal, perfect world, the model would have a near-zero percent rate of false positives and false negatives, implying that precision and recall, as well as accuracy and F1 score, are all close to one. The accuracy, precision, recall, and F1 score will be used to evaluate the model in this study, although many previous studies on similar themes only use the first three of those four variables. Precision and recall values of 70% are feasible for prediction models, according to some studies [31], [22], however, current techniques have shown some more astonishing outcomes. Precision and recall of more than 65% seem like a good target for this project.

## 4 Study Results

A significant amount of contributions that addressed vulnerabilities were gathered. Each vulnerability needed its dataset after the data had been collected and filtered. The table below provides a summary of their basic information, including the number of repositories and commits that make up the dataset, the number of modified files that contain security holes, the number of lines of code, the number of distinct functions they contain, and the total number of characters. By using it to train the model, the next parts will show that this dataset is appropriate. Since some configurations must be used as a starting point, even though their hyperparameters are not optimum, they can be used to show how alternative hyperparameters lead to better or worse results. The ideal combination of all parameters can be found after going through each hyperparameter and describing how it impacts performance.

The baseline model analyzes the dataset for SQL injections using a focus region step size  $n$  of 5 and a context length  $m$  of 200. It has 30 neurons and is trained using the Adam optimizer for 10 epochs with a dropout and recurrent dropout of 20% and a batch size of 200. Even though training a model for more epochs would almost surely produce superior results, this was not possible due to the need to test numerous combinations, which would have taken more than an hour. As a result, only the resultant "best" model is trained for more epochs. The classification performance of the resulting LSTM model's F1 score, which offers a balanced score that considers precision and recall, is used to compare results. It should be noted that the same model can be trained on the same data two times, one right after



the other, and the resulting scores for precision, accuracy, recall, etc. can deviate by roughly 1-3% due to the nondeterministic character of the entire process. As a result, all of the results in the following tables are only estimates and could vary somewhat.

## 4.1 Hyperparameters of the Embedding Layers

Are our models useful as an embedding, and how do their hyperparameters affect the overall outcomes? The tests that follow look into this. The training corpus has 69,517,343 (almost 70 million) unique tokens that were extracted from multiple Python projects. In various configurations, the hyperparameters vector length, min count, and training iterations are tested. The results of retaining strings as-is versus replacing them with generic string tokens are also compared. Since the baseline model is employed, all hyperparameters are, unless otherwise provided, selected using this default setup. In general, the method entails training an embedding layer model, using it to embed the data, and then training an LSTM model on it. Since the embedding layer itself cannot be evaluated by any kind of number, the quality of the embedding is assessed using the performance of the LSTM model. Its ability to be applied in the situation for which it was designed determines how effective it is. A poor embedding will produce a poor LSTM model that is unable to interpret the data that is given to it. However, a functional LSTM model demonstrates that the embedding layers were appropriate.

### 4.1.1 Vector Dimensionality

The code tokens are transformed into numerical vectors of a specific length or dimensionality when utilizing embedding layers. The more distinct "axes" there are for relating words to one another, the longer those vectors are, and the better the models can capture more intricate connections. It is doubtful that a vector with a size of under 100 can represent Python's semantics well understood, judging by comparisons to jobs involving natural language, where 200-point vector sizes are usual. The minimal count of a token to occur in the vector is used to compare different vector lengths. The models' training iterations are set at 100 and their vocabulary to 1000.

### 4.1.2 String Replacement

As some other researchers have done, strings found in the Python training file can either be replaced with a generic "string" token or left alone. It is difficult to predict which option will perform better in advance. Replacing them could lessen the level of detail in the model while maintaining them could focus too much on the particular content of string tokens. The embedding vectors are set to have a length of 200 to compare the two methods. The comparison is made between a min count of 10, 100, and 5000 with training iterations between 1 and 300. The Average F1 score for the embedding layers encoding that retains strings is indicated in Table 2

by the value before the slash (/), while the score for the model that substitutes a generic string token for strings is indicated by the value after the slash (/). These outcomes demonstrate that the variant without string substitution consistently produces better outcomes.

Table 2: F1 score for different min-count and iteration number w and w/o string

Minimum count	1 Iter.	10 Iter.	100 Iter.	300 Iter.
10	64% / 58%	78% / 72%	82% / 72%	84% / 74%
100	58% / 49%	75% / 66%	73% / 69%	82% / 74%
5000	50% / 49%	67% / 64%	75% / 73%	76% / 73%

### 4.1.3 Minimum Count

The minimum count specifies the minimum number of times a token must appear in the training corpus before a vector representation is given to it. Less frequently occurring tokens will not be encoded and will instead be skipped over later when entire lists of tokens are transformed into lists of vectors. This largely serves to ignore illegitimate identifiers such as uncommon variable names, strings, and other identifiers. To train the embedding layer model, strings are left unchanged for 100 iterations with a 200-vector training set. It could have appeared logical to believe that disregarding unusual tokens would enhance performance, but this was not the case. When tokens are seldom disregarded, the model performs better, Figure 4.

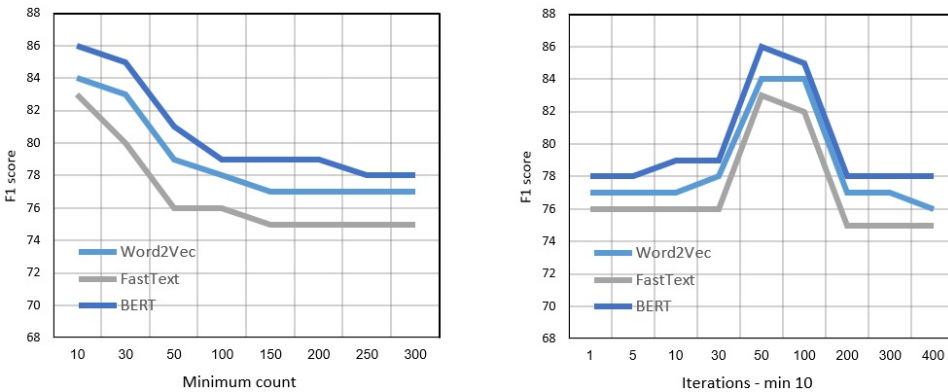


Figure 4: Iterations and minimum count in the different models

#### 4.1.4 Iterations

The quantity of repetitions in training is determined by the number of iterations. It is reasonable to anticipate that there will be no additional advantage from additional training after a certain number of iterations. Using the same parameters as before—a corpus of original strings, a dimensionality of 200, and a min count of 10—the model is trained. Up until 50 or 100 iterations, it seems that more iterations improve the model’s performance. There is no need to increase the iterations to 300 because doing so decreases rather than improves model performance and necessitates a significant increase in training time. It should be noted that the overall trend for greater performance is a smaller min count.

The LSTM model, which uses different embedding layers, performs noticeably differently depending on the hyperparameters, as shown by the tables above, with a difference between the best and worst parameters in the LSTM’s F1 score of almost 25 percentage points. Therefore, careful evaluation of the hyperparameter values was not a waste of effort, as the final model’s ability to learn features is influenced by the quality of the embedding. The final model will require a min count of merely 10 for tokens to be included, encode code tokens in 200-dimensional numerical vectors, not alter any strings, and be trained for 100 iterations, Figure 4.

## 4.2 Parameters in Creating the Dataset

The collection is made up of samples, each of which is a brief section of code built around a single token. Different step sizes  $n$  can be selected while shifting the focus point through the source code. Higher total samples and more sample overlap result from a smaller step size. The second argument, the complete length of a code sample  $m$ , determines the size of the context window surrounding the token in focus. Characters are used to measure both. The default settings are applied to all hyperparameters of the LSTM model, and the previously established ideal model is employed. Consistently lower outcomes follow bigger  $n$ . This is most likely because there will not be much overlap between the focus points’ surrounding context, and the moving window that contains the code snippets if the gaps between them are wide.

A single token will appear multiple times if the emphasis shifts in very short steps because the code snippets have a lot of overlap. For example, a token can appear at the end of one snippet, in the middle of the next, and at the beginning of the one after that. This implies that there are samples that demonstrate the pertinent code with more information before and after it for each vulnerability. Making it somewhat simpler for the model to figure out which component is the real source of the vulnerability. With a longer whole length  $m$  of the code snippet constituting one sample, the model performs better. Again, a bigger  $m$  results in more overlap. The drawback of this is that the prediction may become less accurate, as a significant portion of text around a token may be identified as vulnerable because it is located within a snippet of length  $m$ . However, a bigger  $m$  also has the benefit that more token context may be taken into account, which is precisely why

the LSTM was initially chosen. The samples, which were already rather numerous and relatively huge in size, continued to grow for a whole length of more than 200, outpacing the machines' computing power. Moving forward, the settings for building the training set were fixed to a step length of  $n=5$  and a full context window length of  $m=200$ .

### 4.3 Hyperparameters and Performance of the LSTM model

It is necessary to choose appropriate hyperparameters for the LSTM model to respond to the study question, "How effective is our technique in finding vulnerabilities as measured with accuracy, precision, and recall?" All additional LSTM hyperparameters are evaluated using the baseline model with the following values:  $n=5$ ,  $m=200$ , 30 neurons, 10 epochs, dropout 20%, and Adam optimizer. The code examples are embedded using the embedding layer models with the optimal configuration predetermined.

#### 4.3.1 Number of Neurons

The model can represent more complicated structures with a larger number of neurons, but training takes longer. In general, a model performs better with more neurons, with diminishing results beyond 50 to 70 neurons. When all other factors are held constant, the training time nearly doubles from 1 neuron to 100 neurons, then again from 100 neurons to 250 neurons. The machines the models are trained on reached their limits after more epochs and bigger datasets, sometimes terminating the operation. The optimal arrangement is therefore determined to be 100 neurons, Figure 5.

#### 4.3.2 Batch Size

The following outcomes (Figure 5) were achieved using the baseline model with standard batch sizes (32, 64, and 128) as well as some very small and very large batch sizes: The size of the batch does not appear to have a significant impact on the model's overall performance. Only very large batch sizes of above 1000 result in performance degradation. On the other hand, the batch size had a big impact on how long it took to train the model. While training with a batch size of 5000 took 45 seconds each epoch, a batch size of 200 took 130 seconds, a batch size of 64 required 270 seconds, and the smallest practicable batch size required roughly 370 seconds. The model had to be trained for more than twenty minutes with a batch size of 10, hence the training was stopped. Conclusion: It can be said that for batch sizes less than 64, no improvement in accuracy and recall would warrant spending the additional time required for training with such little chunks of samples. From now on, a batch size of 128 will be regarded as ideal.

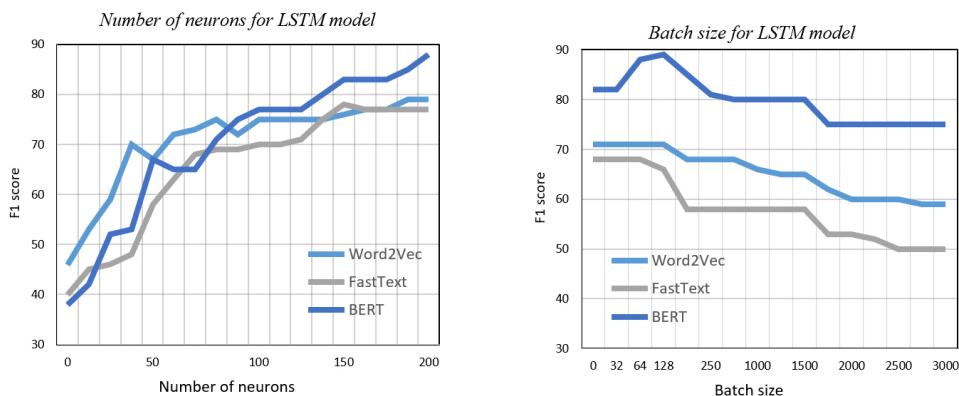


Figure 5: Hyperparameters for the LSTM model

### 4.3.3 Optimizer

In addition to the common Adam optimizer, the Keras model also provides similar optimizers like RMSprop and Adagrad, as well as NAdam and Adamax. To assess each person's performance, they are all put to the test. Due to their suitability for online issues, Adam, NAdam, and RMSprop appear to perform slightly better than Adagrad and Adamax. The SGD's performance is significantly worse. It takes around three hours to train each of the top three optimizers, therefore they are compared once more with 50 epochs. Adam has been selected as the preferred standard optimizer since it was a very close call. All things being equal, this optimizer is more likely to be employed in other studies, making comparisons easier.

### 4.3.4 Dropout

The terms dropout and repeat dropout are combined. The baseline model is trained once more but for 30 epochs this time. A fluctuation of about 2 percent points can still be accounted for by a few remaining variances in the outcome. The model functions well up to a 25% dropout. Performance gradually declines as there is a greater random loss of neurons. Therefore, setting the default dropout at 20% seems like a sensible decision, minimizing overfitting while yet allowing for adequate model performance, Figure 6.

### 4.3.5 Number of Training Epochs

Up to a certain point, training the model for additional epochs improves performance (Figure 6). 100 neurons were used in the model's training. Keep in mind that the performance on the validation set is used to calculate the accuracy, precision, recall, and F1 score. Additionally, the model has a 20% dropout, which should help avoid overfitting. Naturally, using more epochs lengthens the time required to train the entire model. Lengthier training sessions result in noticeable benefits.

However, beyond 100 epochs, there is not much to be gained, hence 100 epochs are selected for the model.

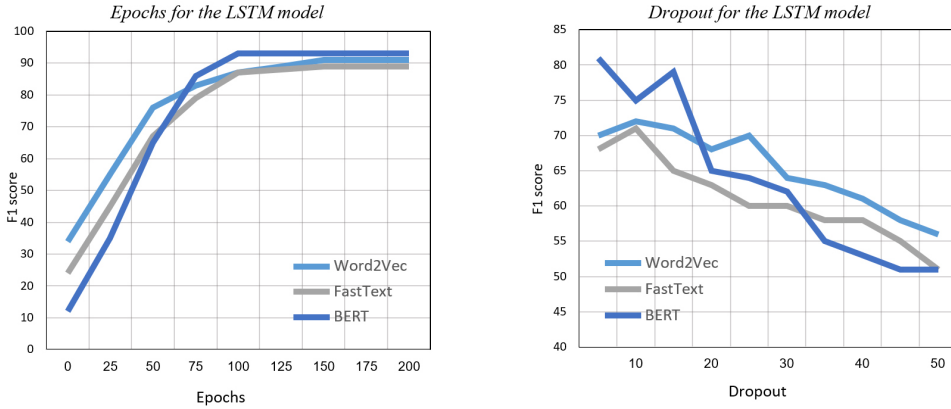


Figure 6: Hyperparameters for the LSTM model

#### 4.3.6 Optimal Configuration

Given the dataset, limitations on processing capacity, and storage space, the recommended hyperparameter settings are:

- 100 neurons
- Training for 100 epochs
- About 20% of dropouts and repeat dropouts
- Batch size 128
- Utilizing the Adam Optimizer

These hyperparameters enable the model to be trained on all vulnerabilities for the best outcomes.

### 4.4 Performance for Subsets of Vulnerabilities

To respond to our study question, what categories of vulnerabilities are detectable, we looked at each vulnerability group separately. Several of the initial considerations for vulnerabilities have to be eliminated. There were relatively few results for the keywords cross-origin, buffer overflow, function injection, clickjack, eval injection, cache overflow, smurf, and denial of service, and no dataset of any size could be produced. Numerous commits that were unrelated to security vulnerabilities were produced by the keywords brute force, tampering, directory traversal, hijacking, replay attack, man-in-the-middle, format string, unauthorized, and sanitize.

A manual review of a few randomly chosen samples revealed that the majority of those commits dealt with other problems unrelated to thwarting an exploit.

Table 3: LSTM+word2vec results for each vulnerability categories

Vulnerability	Accuracy	Precision	Recall	F1
SQL Injection	92.5%	86.2%	86.0%	86.1%
XSS	91.2%	87.9%	80.8%	84.2%
Command injection	90.3%	88.0%	82.3%	84.0%
XSRF	90.1%	87.6%	84.4%	85.9%
Remote code execution	90.0%	86.0%	85.1%	85.8%
Path disclosure	89.3%	89.0%	86.4%	86.1%
Average	91.0%	88.2%	86.1%	85.6%

Therefore, it was unable to produce a high-quality dataset for those vulnerabilities. Seven vulnerabilities are left for which a dataset might be produced. The LSTM model is trained on the training sets using the determined ideal hyperparameters, with the optimizers set to minimize the F1 scores. Finally, the performance of the model is assessed, this time using the final test dataset that the models have never "seen". The findings are shown in Tables 3, 4 and 5. It seems that while the optimizer is attempting to reduce the F1 score, it is more straightforward to do so by increasing precision while the recall is a little lower. Figure 7 displays the exact meanings of the colors. In the sections that follow, one example for each vulnerability is also provided.

Table 4: LSTM+fastText results for each vulnerability categories

Vulnerability	Accuracy	Precision	Recall	F1
SQL Injection	91.2%	82.2%	88.0%	85.1%
XSS	92.8%	83.8%	80.8%	82.2%
Command injection	91.2%	89.0%	87.3%	88.1%
XSRF	92.3%	82.7%	81.3%	81.9%
Remote code execution	90.2%	86.0%	82.8%	83.7%
Path disclosure	89.8%	82.0%	81.1%	81.5%
Average	91.8%	86.4%	85.1%	84.0%

#### 4.4.1 SQL Injection

With 96041 samples for training and 20581 samples for testing, the data for the SQL injection vulnerability was divided into a training set and a test set. 10.9% or so of those code fragments have some susceptible code in them.

Table 5: LSTM+BERT results for each vulnerability categories

Vulnerability	Accuracy	Precision	Recall	F1
SQL Injection	92.5%	82.2%	78.0%	80.1%
XSS	93.8%	91.9%	80.8%	86.0%
Command injection	95.8%	94.0%	87.2%	90.5%
XSRF	92.2%	92.9%	85.4%	89.0%
Remote code execution	91.1%	96.0%	82.6%	88.8%
Path disclosure	91.3%	92.0%	84.4%	88.1%
Average	93.8%	91.4%	83.2%	87.1%

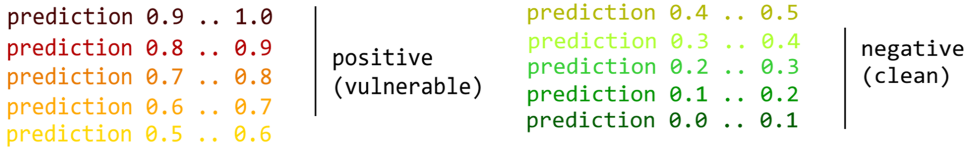


Figure 7: Color codes and confidence levels

With the aforementioned hyperparameters, the LSTM model trained for 100 iterations on the training set, yielding accuracy, precision, recall, and F1 scores of 92.5%, 82.2%, and 78.0%, 83.5% respectively within the test set. Figure 8 shows a tiny example of a SQL injection repair on GitHub. The SQL query stored in the variable `SQL str`, which is formed by directly concatenating other variables into a string, is executed by the instruction `cursor.execute` in the exposed code snippet. Figure 9 shows the detection of this vulnerability with help of our model.

#### 4.4.2 Cross-site Scripting

A rate of 8.9% vulnerable samples was obtained after splitting and processing the data for cross-site scripting, producing 17010 training samples and 3645 test samples. Following training on the training set, the model performed on the test set with accuracy, precision, recall, and F1 score of 97.7%, 91.9%, 80.8%, and 86.0%. For an illustration of how the model finds an XSS vulnerability, see Figure 10. The variable `self.content` is used to create dynamically generated HTML code for a comment area. This code needs to be escaped to prevent script injection. Figure 11 shows the detection on the source code.



```

        that might have been made inactive in duecourse
        """
-       sql_str = "SELECT id FROM wins_completed_wins_fy"
-       if self.end_date:
-           sql_str = f"{sql_str} where created <= '{self.end_date.strftime('%m-%d-%Y')}'"
-
        with connection.cursor() as cursor:
-       cursor.execute(sql_str)

        ids = cursor.fetchall()

        wins = Win.objects.filter(id__in=[id[0] for id in ids]).values()
    that might have been made inactive in duecourse
    """

    with connection.cursor() as cursor:
+       if self.end_date:
+           cursor.execute("SELECT id FROM wins_completed_wins_fy where created <= %s", (self.end_date,))
+       else:
+           cursor.execute("SELECT id FROM wins_completed_wins_fy")
+       ids = cursor.fetchall()

    wins = Win.objects.filter(id__in=[id[0] for id in ids]).values()

```

Figure 8: Commit for vulnerability (SQL injection)

```

def _make_flat_wins_csv(self, **kwargs):
    """
    Make CSV of all completed Wins till now for this financial year, with non-local data flattened
    remove all rows where:
    1. total expected export value = 0 and total non export value = 0 and total odi value = 0
    2. date created = today (not necessary if this task runs before end of the day for next day download)
    3. customer email sent is False / No
    4. Customer response received is not from this financial year
    Note that this view removes win, notification and customer response entries
    that might have been made inactive in duecourse
    """
    sql_str = "SELECT id FROM wins_completed_wins_fy"
    if self.end_date:
        sql_str = f"{sql_str} where created <= '{self.end_date.strftime('%m-%d-%Y')}'"

    with connection.cursor() as cursor:
        cursor.execute(sql_str)
        ids = cursor.fetchall()

    wins = Win.objects.filter(id__in=[id[0] for id in ids]).values()

    for win in wins:
        yield self._get_win_data(win)

def get(self, request, format=None):
    end_str = request.GET.get("end", None)
    if end_str:
        try:
            self.end_date = models.DateField().to_python(end_str)
        except ValidationError:
            self.end_date = None

```

Figure 9: Detection of vulnerability (SQL injection)

```

15     import json
16
17 - from django.utils import safestring
18     from django.utils.translation import ugettext_lazy as _
19     from django.utils.translation import ungettext_lazy
20
21 @@ -75,7 +74,7 @@ def get_rules_as_json(mapping):
22
23     rules = getattr(mapping, 'rules', None)
24     if rules:
25         rules = json.dumps(rules, indent=4)
26 - return safestring.mark_safe(rules)
27
28
29
30
31 class MappingsTable(tables.DataTable):
32
33     import json
34
35
36
37 from django.utils.translation import ugettext_lazy as _
38 from django.utils.translation import ungettext_lazy
39
40
41
42
43
44
45
46
47
48
49
50
51
52
53
54
55
56
57
58
59
60
61
62
63
64
65
66
67
68
69
70
71
72
73
74     rules = getattr(mapping, 'rules', None)
75     if rules:
76         rules = json.dumps(rules, indent=4)
77 + return rules
78
79
80
81
82
83
84
85
86
87
88
89
90
91
92
93
94
95
96
97
98
99
100
101
102
103
104
105
106
107
108
109
110
111
112
113
114
115
116
117
118
119
120
121
122
123
124
125
126
127
128
129
130
131
132
133
134
135
136
137
138
139
140
141
142
143
144
145
146
147
148
149
150
151
152
153
154
155
156
157
158
159
160
161
162
163
164
165
166
167
168
169
170
171
172
173
174
175
176
177
178
179
180
181
182
183
184
185
186
187
188
189
190
191
192
193
194
195
196
197
198
199
200
201
202
203
204
205
206
207
208
209
210
211
212
213
214
215
216
217
218
219
220
221
222
223
224
225
226
227
228
229
230
231
232
233
234
235
236
237
238
239
240
241
242
243
244
245
246
247
248
249
250
251
252
253
254
255
256
257
258
259
260
261
262
263
264
265
266
267
268
269
270
271
272
273
274
275
276
277
278
279
280
281
282
283
284
285
286
287
288
289
290
291
292
293
294
295
296
297
298
299
300
301
302
303
304
305
306
307
308
309
310
311
312
313
314
315
316
317
318
319
320
321
322
323
324
325
326
327
328
329
330
331
332
333
334
335
336
337
338
339
340
341
342
343
344
345
346
347
348
349
350
351
352
353
354
355
356
357
358
359
360
361
362
363
364
365
366
367
368
369
370
371
372
373
374
375
376
377
378
379
380
381
382
383
384
385
386
387
388
389
390
391
392
393
394
395
396
397
398
399
400
401
402
403
404
405
406
407
408
409
410
411
412
413
414
415
416
417
418
419
420
421
422
423
424
425
426
427
428
429
430
431
432
433
434
435
436
437
438
439
440
441
442
443
444
445
446
447
448
449
450
451
452
453
454
455
456
457
458
459
460
461
462
463
464
465
466
467
468
469
470
471
472
473
474
475
476
477
478
479
480
481
482
483
484
485
486
487
488
489
490
491
492
493
494
495
496
497
498
499
500
501
502
503
504
505
506
507
508
509
510
511
512
513
514
515
516
517
518
519
520
521
522
523
524
525
526
527
528
529
530
531
532
533
534
535
536
537
538
539
540
541
542
543
544
545
546
547
548
549
550
551
552
553
554
555
556
557
558
559
560
561
562
563
564
565
566
567
568
569
570
571
572
573
574
575
576
577
578
579
580
581
582
583
584
585
586
587
588
589
590
591
592
593
594
595
596
597
598
599
600
601
602
603
604
605
606
607
608
609
610
611
612
613
614
615
616
617
618
619
620
621
622
623
624
625
626
627
628
629
630
631
632
633
634
635
636
637
638
639
640
641
642
643
644
645
646
647
648
649
650
651
652
653
654
655
656
657
658
659
660
661
662
663
664
665
666
667
668
669
670
671
672
673
674
675
676
677
678
679
680
681
682
683
684
685
686
687
688
689
690
691
692
693
694
695
696
697
698
699
700
701
702
703
704
705
706
707
708
709
710
711
712
713
714
715
716
717
718
719
720
721
722
723
724
725
726
727
728
729
730
731
732
733
734
735
736
737
738
739
740
741
742
743
744
745
746
747
748
749
750
751
752
753
754
755
756
757
758
759
760
761
762
763
764
765
766
767
768
769
770
771
772
773
774
775
776
777
778
779
780
781
782
783
784
785
786
787
788
789
790
791
792
793
794
795
796
797
798
799
800
801
802
803
804
805
806
807
808
809
810
811
812
813
814
815
816
817
818
819
820
821
822
823
824
825
826
827
828
829
830
831
832
833
834
835
836
837
838
839
840
841
842
843
844
845
846
847
848
849
850
851
852
853
854
855
856
857
858
859
860
861
862
863
864
865
866
867
868
869
870
871
872
873
874
875
876
877
878
879
880
881
882
883
884
885
886
887
888
889
890
891
892
893
894
895
896
897
898
899
900
901
902
903
904
905
906
907
908
909
910
911
912
913
914
915
916
917
918
919
920
921
922
923
924
925
926
927
928
929
930
931
932
933
934
935
936
937
938
939
940
941
942
943
944
945
946
947
948
949
950
951
952
953
954
955
956
957
958
959
960
961
962
963
964
965
966
967
968
969
970
971
972
973
974
975
976
977
978
979
980
981
982
983
984
985
986
987
988
989
990
991
992
993
994
995
996
997
998
999
1000

```

Figure 10: Commit for vulnerability (Cross-site scripting)

```

class MappingFilterAction(tables.FilterAction):
    def filter(self, table, mappings, filter_string):
        """Naive case-insensitive search."""
        q = filter_string.lower()
        return [mapping for mapping in mappings
                if q in mapping.ud.lower()]

def get_rules_as_json(mapping):
    rules = getattr(mapping, 'rules', None)
    if rules:
        rules = json.dumps(rules, indent=4)
    return safestring.mark_safe(rules)

class MappingsTable(tables.DataTable):
    id = tables.Column('id', verbose_name=_('Mapping ID'))
    description = tables.Column(get_rules_as_json,
                               verbose_name=_('Rules'))

class Meta(object):
    name = "idp_mappings"
    verbose_name = _("Attribute Mappings")
    row_actions = (EditMappingLink, DeleteMappingsAction)
    table_actions = (MappingFilterAction, CreateMappingLink,
                    DeleteMappingsAction)

```

Figure 11: Detection of vulnerability (Cross-site scripting)

### 4.4.3 Command Injection

The accuracy, precision, recall, and F1 score of the command injection model's performance on the test set were 97.8%, 94.0%, 87.2%, and 90.5%, respectively. With a rate of 4.6% samples containing a vulnerability, 51763 training samples, and 11073 test samples were generated from the dataset. One illustration can be found in Figure 12. Here is an example of some code that uses `subprocess.call` to run the Java compiler when given a command with the detection part in Figure 13. Extra items can be handled as additional arguments to the shell because the command is passed to it as a string and with the option "shell=True," which enables the injection of other commands.

```

89     print("[*] Compiling modified backdoor...")
90 -   if subprocess.call("javac -cp tmp/ tmp/%s" % backdoor, shell=True) != 0:

91     print("[!] Error compiling %s" % backdoor)
92     print("[*] Compiled modified backdoor")
93
89     print("[*] Compiling modified backdoor...")
90 +   #if subprocess.call("javac -cp tmp/ tmp/%s" % backdoor, shell=True) != 0:
91 +   if subprocess.call(['javac', '-cp', 'tmp/', 'tmp/%s'%backdoor], shell=False) != 0:
92     print("[!] Error compiling %s" % backdoor)
93     print("[*] Compiled modified backdoor")

```

Figure 12: Commit for vulnerability (Command injection)

### 4.4.4 Cross-site Request Forgery

68434 training samples and 14665 test samples were used to process the data, and 5.9% of the samples contained susceptible code. The model performed quite well on the test data set for XSRF, achieving an accuracy of 97.2%, a precision of 92.9%, a recall of 85.4%, and an F1 score of 89.0%. Figure 14 shows an example of an XSRF vulnerability and Figure 15 shows the detection done by our approach. In this instance, an XSRF attack prevention check for proper XSRF cookies was merely absent.

### 4.4.5 Remote Code Execution

There were 9797 test samples and 45723 training samples in the data for remote code. 5.3% or so of the samples were vulnerable, the remainder were uncontaminated. The model was performed on the final test set with an accuracy of 98.1%, a precision of 96.0%, a recall of 82.6%, and an F1 score of 88.8% after being trained on the training set. Similar to the previous vulnerabilities, a specific illustration is provided here. Figure 16 illustrates a situation in which a command created by concatenating strings is executed using a call to `os.system`.

It is preferable to give the command as a sequence rather since just the first element of the sequence will be considered as a program to run. Figure 17 shows

```

print("[*] Modifying provided backdoor...")
inmain=False
level=0
bd=open(backdoor, "r").read()
with open("tmp/%s" % backdoor, 'w') as f:
    for l in bd.split("\n"):
        if "main(" in l:
            inmain=True
            f.write(l)
        elif "}" in l and level<2 and inmain:
            f.write("%s.main(args);}" % oldmain)
            inmain=False
        elif "}" in l and level>1 and inmain:
            level-=1
            f.write(l)
        elif "{" in l and inmain:
            level+=1
            f.write(l)
        else:
            f.write(l)
print("[*] Provided backdoor successfully modified")

print("[*] Compiling modified backdoor...")
if subprocess.call("javac -cp tmp/ tmp/%s" % backdoor, shell=True) != 0:
    print("[!] Error compiling %s" % backdoor)
print("[*] Compiled modified backdoor")

if(len(oldmain)<1):
    print("[!] Main-Class manifest attribute not found")
else:
    print("[*] Repackaging target jar file...")
    createZip("tmp", outfile)
    print("[*] Target jar successfully repackaged")
    shutil.rmtree('tmp/')

if __name__ == "__main__":
    main(sys.argv[1:])

```

Figure 13: Detection of vulnerability (Command injection)

the detection of this vulnerability with help of our model.

#### 4.4.6 Path Disclosure

With 11802 test samples and 55072 training samples, this vulnerability had a rate of 7.13% vulnerable samples. The model's performance on the test set was 97.3% accurate, 92.0% precise, 84.4% recall, and 88.0% overall F1 score. An example is shown in Figure 18 and the detection example in Figure 19. Using the `commonprefix` function to determine whether the requested path is located inside the web root directory prevented a path disclosure in the example.

#### 4.4.7 Open Redirect

There were 38189 training samples and 8184 test samples after the data had been processed. 6.4% of the samples have a vulnerability in them. An accuracy of 96.8%, a precision of 91.0%, a recall of 83.9%, and an F1 score of 87.3% were attained for this last vulnerability.

Figure 20 shows a common and simple case in which the session's next URL is requested without being sanitized, allowing untrusted URL strings to contain redi-

```

33     def head(self, path):
34 -     self.get(path, include_body=False)

35
36     @web.authenticated
37     def get(self, path, include_body=True):

38         cm = self.contents_manager
39
33     def head(self, path):
34 +     self.check_xsrft_cookie()
35 +     return self.get(path, include_body=False)
36
37     @web.authenticated
38     def get(self, path, include_body=True):
39 +     # /files/ requests must originate from the same site
40 +     self.check_xsrft_cookie()
41     cm = self.contents_manager

```

Figure 14: Commit for vulnerability (Cross-site request forgery)

```

class FilesHandler(IPythonHandler):
    """serve files via ContentsManager

    Normally used when ContentsManager is not a FileContentsManager.

    FileContentsManager subclasses use AuthenticatedFilesHandler by default,
    a subclass of StaticFileHandler.
    """

    @property
    def content_security_policy(self):
        # In case we're serving HTML/SVG, confine any Javascript to a unique
        # origin so it can't interact with the notebook server.
        return super(FilesHandler, self).content_security_policy + \
            "; sandbox allow-scripts"

    @web.authenticated
    def head(self, path):
        self.get(path, include_body=False)

    @web.authenticated
    def get(self, path, include_body=True):
        cm = self.contents_manager

        if cm.is_hidden(path) and not cm.allow_hidden:
            self.log.info("Refusing to serve hidden file, via 404 Error")
            raise web.HTTPError(404)

        path = path.strip('/')
        if '/' in path:
            _, name = path.rsplit('/', 1)
        else:

```

Figure 15: Detection of vulnerability (Cross-site request forgery)

rect parameters that route users to pages other than the ones they were supposed to see and detection sample in Figure 21.

```

15 - import common, sqlite3, subprocess, NetworkManager, os, crypt, pwd, getpass, spwd
16
17 # fetch network AP details
18 nm = NetworkManager.NetworkManager
19
20 @@ -61,7 +61,8 @@ def get_allAPs():
21
22
23
24
25
26
27
28
29
30
31
32
33
34
35
36
37
38
39
40
41
42
43
44
45
46
47
48
49
50
51
52
53
54
55
56
57
58
59
60
61
62 def add_user(username, password):
63     encPass = crypt.crypt(password,"22")
64 - os.system("useradd -G docker,wheel -p "+encPass+" "+username)
65
66
67
68
69
70
71
72
73
74
75
76
77
78
79
80
81
82
83
84
85
86
87
88
89
90
91
92
93
94
95
96
97
98
99
100
101
102
103
104
105
106
107
108
109
110
111
112
113
114
115
116
117
118
119
120
121
122
123
124
125
126
127
128
129
130
131
132
133
134
135
136
137
138
139
140
141
142
143
144
145
146
147
148
149
150
151
152
153
154
155
156
157
158
159
160
161
162
163
164
165
166
167
168
169
170
171
172
173
174
175
176
177
178
179
180
181
182
183
184
185
186
187
188
189
190
191
192
193
194
195
196
197
198
199
200
201
202
203
204
205
206
207
208
209
210
211
212
213
214
215
216
217
218
219
220
221
222
223
224
225
226
227
228
229
230
231
232
233
234
235
236
237
238
239
240
241
242
243
244
245
246
247
248
249
250
251
252
253
254
255
256
257
258
259
260
261
262
263
264
265
266
267
268
269
270
271
272
273
274
275
276
277
278
279
280
281
282
283
284
285
286
287
288
289
290
291
292
293
294
295
296
297
298
299
300
301
302
303
304
305
306
307
308
309
310
311
312
313
314
315
316
317
318
319
320
321
322
323
324
325
326
327
328
329
330
331
332
333
334
335
336
337
338
339
340
341
342
343
344
345
346
347
348
349
350
351
352
353
354
355
356
357
358
359
360
361
362
363
364
365
366
367
368
369
370
371
372
373
374
375
376
377
378
379
380
381
382
383
384
385
386
387
388
389
390
391
392
393
394
395
396
397
398
399
400
401
402
403
404
405
406
407
408
409
410
411
412
413
414
415
416
417
418
419
420
421
422
423
424
425
426
427
428
429
430
431
432
433
434
435
436
437
438
439
440
441
442
443
444
445
446
447
448
449
450
451
452
453
454
455
456
457
458
459
460
461
462
463
464
465
466
467
468
469
470
471
472
473
474
475
476
477
478
479
480
481
482
483
484
485
486
487
488
489
490
491
492
493
494
495
496
497
498
499
500
501
502
503
504
505
506
507
508
509
510
511
512
513
514
515
516
517
518
519
520
521
522
523
524
525
526
527
528
529
530
531
532
533
534
535
536
537
538
539
540
541
542
543
544
545
546
547
548
549
550
551
552
553
554
555
556
557
558
559
560
561
562
563
564
565
566
567
568
569
570
571
572
573
574
575
576
577
578
579
580
581
582
583
584
585
586
587
588
589
590
591
592
593
594
595
596
597
598
599
600
601
602
603
604
605
606
607
608
609
610
611
612
613
614
615
616
617
618
619
620
621
622
623
624
625
626
627
628
629
630
631
632
633
634
635
636
637
638
639
640
641
642
643
644
645
646
647
648
649
650
651
652
653
654
655
656
657
658
659
660
661
662
663
664
665
666
667
668
669
670
671
672
673
674
675
676
677
678
679
680
681
682
683
684
685
686
687
688
689
690
691
692
693
694
695
696
697
698
699
700
701
702
703
704
705
706
707
708
709
710
711
712
713
714
715
716
717
718
719
720
721
722
723
724
725
726
727
728
729
730
731
732
733
734
735
736
737
738
739
740
741
742
743
744
745
746
747
748
749
750
751
752
753
754
755
756
757
758
759
760
761
762
763
764
765
766
767
768
769
770
771
772
773
774
775
776
777
778
779
780
781
782
783
784
785
786
787
788
789
790
791
792
793
794
795
796
797
798
799
800
801
802
803
804
805
806
807
808
809
810
811
812
813
814
815
816
817
818
819
820
821
822
823
824
825
826
827
828
829
830
831
832
833
834
835
836
837
838
839
840
841
842
843
844
845
846
847
848
849
850
851
852
853
854
855
856
857
858
859
860
861
862
863
864
865
866
867
868
869
870
871
872
873
874
875
876
877
878
879
880
881
882
883
884
885
886
887
888
889
890
891
892
893
894
895
896
897
898
899
900
901
902
903
904
905
906
907
908
909
910
911
912
913
914
915
916
917
918
919
920
921
922
923
924
925
926
927
928
929
930
931
932
933
934
935
936
937
938
939
940
941
942
943
944
945
946
947
948
949
950
951
952
953
954
955
956
957
958
959
960
961
962
963
964
965
966
967
968
969
970
971
972
973
974
975
976
977
978
979
980
981
982
983
984
985
986
987
988
989
990
991
992
993
994
995
996
997
998
999
1000

```

Figure 16: Commit for vulnerability (Remote code execution)

```

def get_allAPs():
    """
    nmcli con | grep 802-11-wireless
    """
    ps = subprocess.Popen('nmcli -t -f SSID,BARS device wifi list', shell=True,stdout=subprocess.PIPE).comr
    wififrows = ps.split('\n')
    wifi = []
    for row in wififrows:
        entry = row.split(':')
        print(entry)
        wifi.append(entry)
    return wifi
    # wifi_aps = []
    # for dev in wians:
    #     for ap in dev.AccessPoints:
    #         wifi_aps.append(ap.Ssid)
    # return wifi_aps

def add_user(username, password):
    encPass = crypt.crypt(password,"22")
    os.system("useradd -G docker,wheel -p "+encPass+" "+username)

def add_newWifiConn(wifiname, wifipass):
    print(wians)
    wlan0 = wians[0]
    print(wlan0)
    print(wifiname)
    # get selected ap as currentwifi
    for dev in wians:
        for ap in dev.AccessPoints:
            if ap.Ssid == wifiname:
                currentwifi = ap
    print(currentwifi)
    # params to set password
    params = {
        "802-11-wireless": {
            "security": "802-11-wireless-security",

```

Figure 17: Detection of vulnerability (Remote code execution)

```

39
40     rel_dst = dst
41     if os.path.isabs(dst):
42 -         _root = os.path.commonprefix([www_root_abs, dst])
43
44     if _root is not www_root_abs:
45         msg = "Destination path is absolute and is not a subdirectory of web root. {}".format([www_root, dst])
46         logger.critical(msg)
47         raise ContentInstallerException(msg)
47 -         rel_dst = os.path.relpath(www_root_abs, dst)
48
49     else:
50         _dst = os.path.join(www_root_abs, dst)
51         _dst = os.path.realpath(_dst)
52 @@ -53,7 +55,8 @@ def _sanity_check_path(self, src, dst, www_root):
53         msg = "Destination is a relative path that resolves outside of web root. {}".format([www_root_abs, dst])
54         logger.critical(msg)
55         raise ContentInstallerException(msg)
56 -         rel_dst = os.path.relpath(www_root_abs, _dst)
57
58     rel_dst = dst
59     if os.path.isabs(dst):
60 +         _dst = os.path.realpath(dst)
61 +         _root = os.path.commonprefix([www_root_abs, _dst])
62
63     if _root is not www_root_abs:
64         msg = "Destination path is absolute and is not a subdirectory of web root. {}".format([www_root, dst])
65         logger.critical(msg)
66         raise ContentInstallerException(msg)
67 +         rel_dst = os.path.relpath(_dst, www_root_abs)
68
69     else:
70         _dst = os.path.join(www_root_abs, dst)
71         _dst = os.path.realpath(_dst)
72
73         msg = "Destination is a relative path that resolves outside of web root. {}".format([www_root_abs, dst])
74         logger.critical(msg)
75         raise ContentInstallerException(msg)
76 +

```

Figure 18: Commit for vulnerability (Path disclosure)

## 4.5 Application on Source Code

Our approach expands the work in the area of vulnerable code pattern analysis. A large dataset of source code written in Python is collected from Github, filtered, preprocessed, and labeled based on the information from commits. Several different types of vulnerabilities are taken into consideration, and source code from many different projects is collected. The resulting dataset of natural code containing vulnerabilities is made available for further research. Samples are generated by dividing the code into overlapping snippets that capture the immediate context of some tokens. The samples are embedded in numerical vectors using different embedding layers.

A long short-term memory network is trained to extract features and then applied to classify code that was not used in training, highlighting the exact locations within the code that are potentially vulnerable. We combine all of the

```

if not os.path.isdir(www_root):
    msg = "Web root % s does not exist or is not a directory." % src
    logger.critical(msg)
    raise ContentInstallerException(msg)

www_root_abs = os.path.abspath(www_root)

rel_dst = dst
if os.path.isabs(dst):
    rroot = os.path.commonprefix([www_root_abs, dst])
    if rroot is not www_root_abs:
        msg = "Destination path is absolute and is not a subdirectory of web root. {}".format([www_root])
        logger.critical(msg)
        raise ContentInstallerException(msg)
    rel_dst = os.path.relpath(www_root_abs, dst)
else:
    _dst = os.path.join(www_root_abs, dst)
    _dst = os.path.realpath(_dst)
    _root = os.path.commonprefix([www_root_abs, _dst])
    if _root is not www_root_abs:
        msg = "Destination is a relative path that resolve outside of web root. {}".format([www_root_abs])
        logger.critical(msg)
        raise ContentInstallerException(msg)
    rel_dst = os.path.relpath(www_root_abs, _dst)

abs_dst = os.path.join(www_root_abs, rel_dst)
if os.path.exists(abs_dst):
    msg = "Destination directory already exists : {}".format(abs_dst)
    logger.critical(msg)
    raise ContentInstallerException(msg)

return (src,rel_dst, www_root_abhs)

```

Figure 19: Detection of vulnerability (Path disclosure)

```

110         request.session['oidc_nonce'] = nonce
111
112         request.session['oidc_state'] = state
113 -         request.session['oidc_login_next'] = request.GET.get(redirect_field_name)
114
115         query = urlencode(params)
116         redirect_url = '{url}?{query}'.format(url=self.OIDC_OP_AUTH_ENDPOINT, query=query)
141         request.session['oidc_nonce'] = nonce
142
143         request.session['oidc_state'] = state
144 +         request.session['oidc_login_next'] = get_next_url(request, redirect_field_name)
145
146         query = urlencode(params)
147         redirect_url = '{url}?{query}'.format(url=self.OIDC_OP_AUTH_ENDPOINT, query=query)

```

Figure 20: Commit for vulnerability (Open redirect)

trained models into a single, straightforward text editor, called VulDetective, and test them using a variety of features, including which embedding layer and which vulnerabilities they are vulnerable to. Additionally, the tool displays the content color coded, Figure 22, including gray for comments, green for not vulnerable, and red for vulnerable. We aim to keep it as straightforward as we can because the tool’s goal is better detection; as a result, we spend a lot of time training various models and experimenting with various embedding layers and hyperparameters.



```

params = {
    'response_type': 'code',
    'scope': 'openid',
    'client_id': self.OIDC_RP_CLIENT_ID,
    'redirect_uri': absolutify(
        request,
        reverse('oidc_authentication_callback')
    ),
    'state': state,
}

if import_from_settings('OIDC_USE_NONCE', True):
    nonce = get_random_string(import_from_settings('OIDC_NONCE_SIZE', 32))
    params.update({
        'nonce': nonce
    })
    request.session['oidc_nonce'] = nonce

request.session['oidc_state'] = state
request.session['oidc_login_next'] = request.GET.get(redirect_field_name)

query = urlencode(params)
redirect_url = '{uri}?{query}'.format(uri=self.OIDC_OP_AUTH_ENDPOINT, query=query)
return HttpResponseRedirect(redirect_url)

class OIDCLogoutView(View):
    """Logout helper view"""

    http_method_names = ['get', 'post']

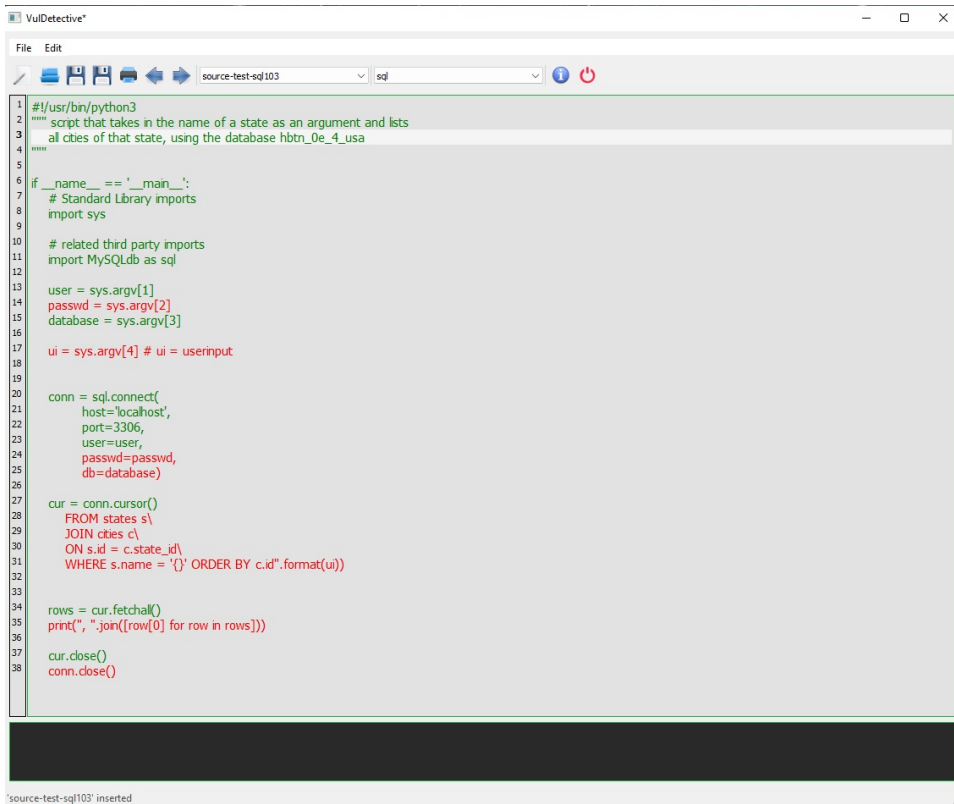
    @property
    def redirect_url(self):
        """Return the logout url defined in settings."""
        return import_from_settings('LOGOUT_REDIRECT_URL', '/')

```

Figure 21: Detection of vulnerability (Open redirect)

The application we have created differs in some respects from all other previous work. Unlike the approaches of Li et al. [18], Pang et al. [25], Hovsepyan et al. [15], and Dam et al. [7], it uses a broad code base rather than a select number of projects. The predictions are not only applicable within the same file or project, but can be generalized to any other source code. In contrast to these four works, a fine granularity is also chosen. The aforementioned works all classify entire files or, as in the case of Li et al. [18], consider only API and function calls. Our approach is more in line with the work of Russell et al. [26] and Ma et al. [21] in that vulnerabilities are detected at specific locations in the code rather than just at the file level, which is likely to be more useful to developers; different tokens can even be color-coded depending on the confidence level of the classification. Similar to the research of Hovsepyan et al. [15] and in contrast to the work of Ma et al. [21], Yamaguchi et al. [37], and Liu et al. [20], this work does not convert the source code into a structure such as an abstract syntax tree but assumes that it is plain text. It follows the natural hypothesis and aims to use as few assumptions as possible, leaving the extraction of features from the source code entirely to the trained model.

The labels for the dataset are not generated using a static analysis tool, as is the case in the work of Russel et al. [26], Dam et al. [7], and Hovsepyan et al. [15]. The basic idea of our approach is independence from manually designed features,



```

1 #!/usr/bin/python3
2 """
3 script that takes in the name of a state as an argument and lists
4 all cities of that state, using the database hbtn_0e_4_usa
5 """
6 if __name__ == '__main__':
7     # Standard Library imports
8     import sys
9
10    # related third party imports
11    import MySQLdb as sql
12
13    user = sys.argv[1]
14    passwd = sys.argv[2]
15    database = sys.argv[3]
16
17    ui = sys.argv[4] # ui = userinput
18
19
20    conn = sql.connect(
21        host='localhost',
22        port=3306,
23        user=user,
24        passwd=passwd,
25        db=database)
26
27    cur = conn.cursor()
28    FROM states s\
29    JOIN cities c\
30    ON s.id = c.state_id\
31    WHERE s.name = '{0}' ORDER BY c.id".format(ui)
32
33
34    rows = cur.fetchall()
35    print(", ".join([row[0] for row in rows]))
36
37    cur.close()
38    conn.close()

```

Figure 22: Overview of the VulDetective application

which is the major limitation of previous static analysis tools. The goal is not to model an existing static tool, but to learn features without initial assumptions. Therefore, it is based on a similar assumption as Liu et al. [20], namely that code that has been patched or patched was most likely vulnerable before the fix. The flag is based solely on the Github commits, which (at least in theory) allows the discovery of vulnerability patterns that have not yet been manually included in static analysis tools. The dataset used as a basis for training consists of natural code from real software projects, rather than synthetic databases designed to provide clear examples of vulnerabilities.

This makes the whole task more difficult, as real code is much messier and less clean than synthetic code. In this respect, our method differs from the approaches of Russell et al. [26] and Li et al. [18]. However, this also makes our approach independent of specific projects with their characteristics and therefore robust to some degree to the threats to validity that would arise from a narrower approach. The machine learning model used is an LSTM and Bi-LSTM, as also used by

Li et al. [18] and Dam et al. [7]. Compared to the latter, the architecture and preprocessing of the data in our approach are much simpler. Many other approaches use either different deep learning models (CNNs and RNNs in the case of the work by Russell et al. [26]) or completely different machine learning approaches (support vector machines in the case of the work by Pang et al. [25]).

To conclude the list of contributions: The focus is on code written in Python, unlike most other research projects that are primarily concerned with Java, C, C++, or PHP. No other approach has been found that uses even remotely similar techniques and works with Python. Of course, the proposed approach could be applied to other languages as well. The various embedding layer models that have been trained for Python are another contribution to this work.

## 4.6 Result Comparison with Other Works

To give a framework for the assessment of this study, Table 6 and 7 includes comparisons with related research in the field. Each approach has inherent variances, hence it is difficult to directly compare them. Approaches are compared under the following aspects:

- Language: what language is subject of the classification efforts
- Data: does the data stem from real-life projects or synthetic databases
- Labels: how are the labels for the training data originally generated
- Granularity: is the code evaluated on a rough granularity (whole classes or files) or a fine granularity (lines or tokens)
- Method: what class of neural network or machine learning approach is used (CNN, RNN, LSTM)

## 5 Conclusion

This paper presents a vulnerability detection method based on deep learning on source code. Its purpose is to relieve human vulnerability detection experts of the time-consuming and subjective effort of manually defining vulnerability detection criteria. Via LSTM models, this research demonstrates the feasibility of learning vulnerability attributes straight from source code using machine learning. It can detect seven different types of errors in Python source code. We were able to identify specific sections of code that are likely to be vulnerable, as well as provide confidence levels for our predictions. We get an accuracy of 93.8%, a recall of 83.2%, a precision of 91.4%, and an F1 score of 87.1% on average. We also demonstrate how the trained model can be applied in practice, therefore opening up the possibility of building a hands-on developer tool for detecting vulnerable code blocks in arbitrary Python programs. Moreover, the presented method is language agnostic, it can be adapted to other languages as well. Higher measurements in precision, recall, and

Table 6: Comparisons with related researches

Name	Lang.	Data	Labels	Scope	Gran.	Method
Russel et al. [26]	C/C++	real and synth.	static tool	general	token level	CNN RNN
Pang et al. [25]	Java	real	pre existing	4 apps	whole classes	SVM
VuRLE [21]	Java	real	manually	general	edits (fine)	10-fold CV
VulDeePecker [18]	C/C++	real and synth.	patches and manual	general	API function calls	BLSTM
Dam et al. [7]	Java	real	static tool	18 apps	whole file	LSTM
Hovsepyan et al. [15]	Java	real	static tool	1 project	whole file	grid search
Bagheri et al.	Python	real	patches	general	token level	LSTM

Table 7: Comparisons with related researches

Name	Accuracy	Precision	Recall	F1
Russel et al.	-	-	-	57%
Pang et al.	63%	67%	63%	65%
VuRLE	-	65%	66%	65%
VulDeePecker	-	-	-	85%-95%
Dam et al.	81%	82%	76%	80%
Hovsepyan et al.	87%	85%	88%	85%
Bagheri et al.	93%	91%	83%	87%

F1 are a lot simpler to accomplish if the methodology centers around forecasts inside a solitary task, as Hovsepyan et al. [15] and Dam et al. [7] do when they train a classifier to predict vulnerabilities inside the same application. Preparing a classifier that is relevant for general recognition of vulnerabilities is a lot harder - yet additionally prompts a substantially more valuable final product. Note that similar two methodologies, as well as the one taken by Pang et al. [25], are likewise simply attempting to predict regardless of whether an entire record is defenseless without having the option to bring up the specific area of the vulnerability. Since it expects to foster an overall vulnerability identifier that can be utilized at the fine granularity of code tokens, it has a significantly more confounded undertaking to satisfy. With basically a similar methodology, Russel et al. [26] accomplished 56% on regular code from Github, yet 84% on the Satisfy test suite because of its spotless and predictable structure and design. our methodology seemingly performs all around given that it works absolutely on normal real-life source code.

## References

- [1] Allamanis, M. and Sutton, C. Mining source code repositories at massive scale using language modeling. In *2013 10th Working Conference on Mining Software Repositories (MSR)*, pages 207–216. IEEE, 2013. DOI: [10.1109/MSR.2013.6624029](https://doi.org/10.1109/MSR.2013.6624029).
- [2] Amirreza, B. and Hegedűs, P. A Comparison of Different Source Code Representation Methods for Vulnerability Prediction in Python. In *Proceedings of the 14th International Conference on the Quality of Information and Communications Technology (QUATIC 2021)*, 2021. DOI: [10.48550/arXiv.2108.02044](https://doi.org/10.48550/arXiv.2108.02044).
- [3] Bhoopchand, A., Rocktäschel, T., Barr, E., and Riedel, S. Learning Python code suggestion with a sparse pointer network. *arXiv preprint arXiv:1611.08307*, 2016. DOI: [10.48550/arXiv.1611.08307](https://doi.org/10.48550/arXiv.1611.08307).
- [4] Chowdhury, I. and Zulkernine, M. Using complexity, coupling, and cohesion metrics as early indicators of vulnerabilities. *Journal of Systems Architecture*, 57(3):294–313, 2011. DOI: [10.1016/j.sysarc.2010.06.003](https://doi.org/10.1016/j.sysarc.2010.06.003).
- [5] Church, K. W. Word2vec. *Natural Language Engineering*, 23(1):155–162, 2017. DOI: [10.1017/S1351324916000334](https://doi.org/10.1017/S1351324916000334).
- [6] Dam, H. K., Tran, T., Grundy, J., and Ghose, A. Deepsoft: A vision for a deep model of software. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 944–947, 2016. DOI: [10.1145/2950290.2983985](https://doi.org/10.1145/2950290.2983985).
- [7] Dam, H. K., Tran, T., Pham, T., Ng, S. W., Grundy, J., and Ghose, A. Automatic feature learning for vulnerability prediction. *arXiv preprint arXiv:1708.02368*, 2017. DOI: [10.48550/arXiv.1708.02368](https://doi.org/10.48550/arXiv.1708.02368).

- [8] Devlin, J., Chang, M.-W., Lee, K., and Toutanova, K. Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805*, 2018. DOI: [10.48550/arXiv.1810.04805](https://doi.org/10.48550/arXiv.1810.04805).
- [9] Duchi, J., Hazan, E., and Singer, Y. Adaptive subgradient methods for online learning and stochastic optimization. *Journal of Machine Learning Research*, 12(7), 2011. DOI: [10.1109/TSMC.2021.3097714](https://doi.org/10.1109/TSMC.2021.3097714).
- [10] Ghaffarian, S. M. and Shahriari, H. R. Software vulnerability analysis and discovery using machine-learning and data-mining techniques: A survey. *ACM Computing Surveys (CSUR)*, 50(4):1–36, 2017. DOI: [10.1145/3092566](https://doi.org/10.1145/3092566).
- [11] Grieco, G., Grinblat, G. L., Uzal, L., Rawat, S., Feist, J., and Mounier, L. Toward large-scale vulnerability discovery using machine learning. In *Proceedings of the Sixth ACM Conference on Data and Application Security and Privacy*, pages 85–96, 2016. DOI: [10.1145/2857705.2857720](https://doi.org/10.1145/2857705.2857720).
- [12] Gupta, R., Pal, S., Kanade, A., and Shevade, S. Deepfix: Fixing common C language errors by deep learning. In *Thirty-First AAAI Conference on Artificial Intelligence*, 2017. DOI: [10.1609/aaai.v31i1.10742](https://doi.org/10.1609/aaai.v31i1.10742).
- [13] Hall, T., Beecham, S., Bowes, D., Gray, D., and Counsell, S. A systematic literature review on fault prediction performance in software engineering. *IEEE Transactions on Software Engineering*, 38(6):1276–1304, 2011. DOI: [10.1109/TSE.2011.103](https://doi.org/10.1109/TSE.2011.103).
- [14] Harer, J., Ozdemir, O., Lazovich, T., Reale, C., Russell, R., Kim, L., et al. Learning to repair software vulnerabilities with generative adversarial networks. In Bengio, S., Wallach, H., Larochelle, H., Grauman, K., Cesa-Bianchi, N., and Garnett, R., editors, *Advances in Neural Information Processing Systems*, Volume 31, 2018. URL: [https://proceedings.neurips.cc/paper\\_files/paper/2018/file/68abef8ee1ac9b664a90b0bbaff4f770-Paper.pdf](https://proceedings.neurips.cc/paper_files/paper/2018/file/68abef8ee1ac9b664a90b0bbaff4f770-Paper.pdf).
- [15] Hovsepian, A., Scandariato, R., Joosen, W., and Walden, J. Software vulnerability prediction using text analysis techniques. In *Proceedings of the 4th International Workshop on Security Measurements and Metrics*, pages 7–10, 2012. DOI: [10.1145/2372225.2372230](https://doi.org/10.1145/2372225.2372230).
- [16] Joulin, A., Grave, E., Bojanowski, P., Douze, M., Jégou, H., and Mikolov, T. Fasttext.zip: Compressing text classification models. *arXiv preprint arXiv:1612.03651*, 2016. DOI: [10.48550/arXiv.1612.03651](https://doi.org/10.48550/arXiv.1612.03651).
- [17] Kingma, D. P. and Ba, J. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014. DOI: [10.48550/arXiv.1412.6980](https://doi.org/10.48550/arXiv.1412.6980).
- [18] Li, Z., Zou, D., Xu, S., Ou, X., Jin, H., Wang, S., Deng, Z., and Zhong, Y. Vuldepecker: A deep learning-based system for vulnerability detection. *arXiv preprint arXiv:1801.01681*, 2018. DOI: [10.48550/arXiv.1801.01681](https://doi.org/10.48550/arXiv.1801.01681).

- [19] Li, Z. and Zhou, Y. PR-Miner: Automatically extracting implicit programming rules and detecting violations in large software code. *ACM SIGSOFT Software Engineering Notes*, 30(5):306–315, 2005. DOI: [10.1145/1095430.1081755](https://doi.org/10.1145/1095430.1081755).
- [20] Liu, K., Kim, D., Bissyandé, T. F., Yoo, S., and Le Traon, Y. Mining fix patterns for findbugs violations. *IEEE Transactions on Software Engineering*, 47(1):165–188, 2018. DOI: [10.1109/TSE.2018.2884955](https://doi.org/10.1109/TSE.2018.2884955).
- [21] Ma, S., Thung, F., Lo, D., Sun, C., and Deng, R. H. Vurle: Automatic vulnerability detection and repair by learning from examples. In *European Symposium on Research in Computer Security*, pages 229–246. Springer, 2017. DOI: [10.1007/978-3-319-66399-9\\_13](https://doi.org/10.1007/978-3-319-66399-9_13).
- [22] Morrison, P., Herzig, K., Murphy, B., and Williams, L. Challenges with applying vulnerability prediction models. In *Proceedings of the 2015 Symposium and Bootcamp on the Science of Security*, pages 1–9, 2015. DOI: [10.1145/2746194.2746198](https://doi.org/10.1145/2746194.2746198).
- [23] Nagappan, N., Murphy, B., and Basili, V. The influence of organizational structure on software quality. In *2008 ACM/IEEE 30th International Conference on Software Engineering*, pages 521–530. IEEE, 2008. DOI: [10.1145/1368088.1368160](https://doi.org/10.1145/1368088.1368160).
- [24] Neuhaus, S., Zimmermann, T., Holler, C., and Zeller, A. Predicting vulnerable software components. In *Proceedings of the 14th ACM Conference on Computer and Communications Security*, pages 529–540, 2007. DOI: [10.1145/1315245.1315311](https://doi.org/10.1145/1315245.1315311).
- [25] Pang, Y., Xue, X., and Namin, A. S. Predicting vulnerable software components through n-gram analysis and statistical feature selection. In *2015 IEEE 14th International Conference on Machine Learning and Applications (ICMLA)*, pages 543–548. IEEE, 2015. DOI: [10.1109/ICMLA.2015.99](https://doi.org/10.1109/ICMLA.2015.99).
- [26] Russell, R., Kim, L., Hamilton, L., Lazovich, T., Harer, J., Ozdemir, O., Ellingwood, P., and McConley, M. Automated vulnerability detection in source code using deep representation learning. In *2018 17th IEEE International Conference on Machine Learning and Applications (ICMLA)*, pages 757–762. IEEE, 2018. DOI: [10.1109/ICMLA.2018.00120](https://doi.org/10.1109/ICMLA.2018.00120).
- [27] Scandariato, R., Walden, J., Hovsepyan, A., and Joosen, W. Predicting vulnerable software components via text mining. *IEEE Transactions on Software Engineering*, 40(10):993–1006, 2014. DOI: [10.1109/TSE.2014.2340398](https://doi.org/10.1109/TSE.2014.2340398).
- [28] Shar, L. K. and Tan, H. B. K. Predicting SQL injection and cross site scripting vulnerabilities through mining input sanitization patterns. *Information and Software Technology*, 55(10):1767–1780, 2013. DOI: [10.1016/j.infsof.2013.04.002](https://doi.org/10.1016/j.infsof.2013.04.002).

- [29] Shin, Y., Meneely, A., Williams, L., and Osborne, J. A. Evaluating complexity, code churn, and developer activity metrics as indicators of software vulnerabilities. *IEEE Transactions on Software Engineering*, 37(6):772–787, 2010. DOI: [10.1109/TSE.2010.81](https://doi.org/10.1109/TSE.2010.81).
- [30] Shin, Y. and Williams, L. An empirical model to predict security vulnerabilities using code complexity metrics. In *Proceedings of the Second ACM-IEEE International Symposium on Empirical Software Engineering and Measurement*, pages 315–317, 2008. DOI: [10.1145/1414004.1414065](https://doi.org/10.1145/1414004.1414065).
- [31] Shin, Y. and Williams, L. Can traditional fault prediction models be used for vulnerability prediction? *Empirical Software Engineering*, 18(1):25–59, 2013. DOI: [10.1007/s10664-011-9190-8](https://doi.org/10.1007/s10664-011-9190-8).
- [32] Spadini, D., Aniche, M., and Bacchelli, A. Pydriller: Python framework for mining software repositories. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 908–911, 2018. DOI: [10.5281/zenodo.1327411](https://doi.org/10.5281/zenodo.1327411).
- [33] Tieleman, T. and Hinton, G. Lecture 6.5-rmsprop, coursera: Neural networks for machine learning. Technical report, University of Toronto, 2012.
- [34] Tu, Z., Su, Z., and Devanbu, P. On the localness of software. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 269–280, 2014. DOI: [10.1145/2635868.2635875](https://doi.org/10.1145/2635868.2635875).
- [35] Wang, S., Liu, T., and Tan, L. Automatically learning semantic features for defect prediction. In *2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE)*, pages 297–308. IEEE, 2016. DOI: [10.1145/2884781.2884804](https://doi.org/10.1145/2884781.2884804).
- [36] Yamaguchi, F., Lottmann, M., and Rieck, K. Generalized vulnerability extrapolation using abstract syntax trees. In *Proceedings of the 28th Annual Computer Security Applications Conference*, pages 359–368, 2012. DOI: [10.1145/2420950.2421003](https://doi.org/10.1145/2420950.2421003).
- [37] Yamaguchi, F., Rieck, K., et al. Vulnerability extrapolation: Assisted discovery of vulnerabilities using machine learning. In *5th USENIX Workshop on Offensive Technologies (WOOT 11)*, 2011. DOI: [10.5555/2028052.2028065](https://doi.org/10.5555/2028052.2028065).
- [38] Yu, Z., Theisen, C., Williams, L., and Menzies, T. Improving vulnerability inspection efficiency using active learning. *IEEE Transactions on Software Engineering*, 47(11):2401–2420, 2019. DOI: [10.1109/TSE.2019.2949275](https://doi.org/10.1109/TSE.2019.2949275).
- [39] Zhou, Y. and Sharma, A. Automated identification of security issues from commit messages and bug reports. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, pages 914–919, 2017. DOI: [10.1145/3106237.3117771](https://doi.org/10.1145/3106237.3117771).



- [40] Zimmermann, T., Nagappan, N., and Williams, L. Searching for a needle in a haystack: Predicting security vulnerabilities for Windows Vista. In *2010 Third International Conference on Software Testing, Verification and Validation*, pages 421–428. IEEE, 2010. DOI: [10.1109/ICST.2010.32](https://doi.org/10.1109/ICST.2010.32).