Source Code Vulnerability Analysis Using Automation

Khushi Garg

October 2022

1 Introduction

A deep learning-based vulnerability detection system called VUDENC automatically picks up features from a sizable body of real-world code. Its foundation is a word2vec model, and a sizable Python corpus was used to train it specifically for this task. Individual code tokens and their context are taken from the susceptible files' source code to construct the data samples, which enables a fine-grained examination. To identify the characteristics of susceptible code, a Long-Short-Term-Memory network is trained on samples for each type of vulnerability. It is then utilised to find vulnerabilities in source code. With the use of this method, VUDENC is able to identify seven different security flaws with high precision and recall (on average 91 percent) and to indicate the exact sections of the code that are thought to be vulnerable.

This work begins by outlining the vulnerability challenge, goes on to explain the technical context in Section 2, and then presents related work in Section 3. The task is finished with a model evaluation in Section 6, a discussion of the findings in Section 7, and then a conclusion. It is looked into whether a dataset of appropriate Python source code can be mined from Github for typical vulnerability types and what kinds of vulnerabilities may be looked into.

2 Related Work

In this paper, Russell et al.[18] showed how to use machine learning to quickly discover source code vulnerabilities. To do this, they created a rich C/C ++ source code dataset from Debian and GitHub repositories, tagged with curated vulnerability discoveries from various static analysis tools, and SATE IV data. Combined with the set. They have created a special C/C ++ lexer to generate an easy-to-understand and comprehensive representation of the function source code suitable for ML training. They modified various machine learning techniques inspired by the problem of natural language classification and found that features trained by convolutional neural networks classified using the ensemble tree algorithm yielded the best results overall. In this paper, Guo et al.[7] chose

C/ C as the target languages for vulnerability discovery. Since numerous beginning software and systems are written in C/ C, once these vulnerabilities are exploited, they can have severe consequences. Their vulnerability discovery model is erected on code criteria and deep literacy. The model excerpts the target function's code standard, excerpts the code metric's features by a convolutional neural network, and also learns the point vector by LSTM. The features of the source code are presented in the form of code criteria, and different code criteria have a different granularity of abstraction conception of the source code.

In this paper, Bilgin et al.[2] developed a source code representation system that allowed us to perform intelligent analysis on source code in the Abstract Syntax Tree (AST) form, as well as investigate whether ML can distinguish vulnerable and nonvulnerable code fractions. To perform a comprehensive performance evaluation, they used a public dataset that contains a large number of function-position real source code corridors booby-trapped from open-source systems and precisely labeled according to the type of vulnerability if any. They compared the efficacy of their proposed system for vulnerability vaticination from source code to state-of-the-art styles by conducting total and realistic trials under various administrations. In this paper, Wang et al.[22] created ContractorWard a model for effectively and efficiently detecting six types of vulnerabilities of smart contracts based on extracted static characteristics. The experimental results demonstrate the effectiveness and effectiveness of ContractWard. Its discovery speed is about 4 seconds on a smart contract on average, much more snappily than Oyente and Security. ContractWard can be applied to descry vulnerabilities in all high-position languages such as reliability, Serpent, and LLL because high- position languages can all be converted into opcodes. Eventually, we elect the model that takes XGBoost as the multi-label classifier and takes SMOTETomke as the slice system in our ContractWard.

In this paper, Wang et al.[21] Introduced funded, a new literacy frame for erecting vulnerability discovery models. Funded leverages advances in graphical neural networks to develop new graph-grounded literacy styles for capturing and inferring program control, data, and call dependences. Unlike former work that treats a program as a successional sequence or untyped graph, funded learns and manipulates the graph representation of the program's source law. In this graph representation, individual statements are connected to other statements by the relational edge. It combined probabilistic literacy with statistical evaluation to automatically collect high-quality training exemplifications from open source systems. It provided numerous real-world vulnerability law training exemplifications to condense the limited vulnerability law exemplifications available in the standard vulnerability database. In this paper, Li et al.[12] proposed an automated and intelligent system for source code vulnerability discovery grounded on minimum intermediate representation literacy. First, he converted the source code format illustration to a minimum intermediate representation to count inapplicable rudiments and reduce the length of the reliance. Pre-training in the extended corpus also transforms the intermediate representation into a real-valued vector, conserving structural and semantic information. The vector was also transferred to three concatenated convolutional neural networks to get

a high position of vulnerability characteristics. Eventually, the classifier was trained to use the learned features. trials were conducted to corroborate this vulnerability discovery system. Experience had shown that our system outperforms fine-granulated methods compared to traditional and ultramodern smart methods.

In this paper, Saccente et al.[19] introduced Project Achilles, a Java source code security vulnerability detection tool built upon LSTM RNN models. It can be trained using vulnerability source code datasets, analyze Java programs and predict security vulnerabilities at the method level. Their preliminary experimental evaluation indicated that it performs effectively and accurately. LSTMs prove to be a promising way of performing static analysis over source code. However, due to the shallow understanding that LSTMs derive from the actual semantics of the source code, perhaps static analysis is not an effective method for identifying software defects. In this paper, Akram et al.[1] proposed a vulnerability detection technique to detect vulnerabilities in software, as well as participated libraries at the source code position. They crawl the vulnerable source code by tracing and locating the patch lines from different web sources according to their CVE- figures and erected a point indicator of 2931 vulnerable lines. also, they developed a vulnerability discovery approach grounded on code clone discovery fashion and descry hundreds of vulnerabilities in thousands of GitHub open source systems, which aren't been noticed before as vulnerable. We detected vulnerabilities in some veritably notorious lately available software, including rearmost interpretation of Linux, HTC- kernel, FindX-8.1-kernel, and in 7- TB of C/ C source code(,823 open source systems).

In this paper, Tang et al.[20] introduced ELM, which trained the discovery model with a noniterative training medium. To ameliorate perfection performance, they also introduced the kernel system. They also proposed a multilevel symbolization system for emblematic representation and introduced doc2vec for vector representation. In detail, they first gained emblematic representations of the source canons related to vulnerabilities through three symbolizations. Using three situations of symbolization can significantly reduce the noise introduced by inapplicable information of vulnerable canons. also, they used doc2vec to automatically transfigure emblematic representation of source canons to corresponding vector representation. In this paper, Kong et al.[11] described a source code static analysis technology based on data emulsion for vulnerability discovery. This technology allows different results to validate each other by parsing and creating data emulsion on the outgrowth of different stationary analysis styles, which greatly reduces false cons and false negatives. This system is supported by detailed explanations. A scalable source code analysis system prototype is designed and implemented, which can also automatically search for stylish results based on feedback from the stoner commerce. The entire system is scalable and platform agnostic. Trial and error have shown that this system outperforms a single system with lower false positives and false negatives and advanced effectiveness.

3 Methodology

3.1 Background

3.1.1 What Are Source Code Analysis Tools And how does they work?

Tools for analysing source code vulnerabilities offer an automated way to assess application code. They can be applied to find weaknesses or spot chances for development to create a higher-quality finished product. Instead of manually analysing each line of code, it uses automated static analysis to evaluate the source code as a whole.

The writing of the code comes first. Then, an automated code analyzer examines your code and compares it to pre-established coding standards. The source of this might be coding standards. Or it may be a team-developed internal coding standard. Code that deviates from accepted coding practises is discovered by a code analyzer. The outcome may then be examined. There can be erroneous warnings that need to be dismissed. And certain problems require more attention than others. Some tools, such as Helix QAC and Klocwork, prioritise violations for you. then take care of the problems. Start with the most crucial corrections.

These options offer a time- and money-saving way to pinpoint software development flaws without the need for test cases or a sizable investment of effort. These technologies offer a solid, repeatable option for benchmarking if subpar quality, missed faults, or compliance problems are becoming a significant problem for your business.

3.1.2 Types of source code analysis

Either static or dynamic examination of source code is possible. Without actually running the program, static analysis analyses and debugs code. As a consequence, problems may be found early on in the creation of a program, frequently eliminating the need for several future patches. We use dynamic analysis to find more subtle bugs and vulnerabilities after static analysis. Real-time program testing is a component of dynamic analysis.

This method's key benefit is that it spares the developer from having to rely on informed guesses in error-prone scenarios. Other advantages include removing pointless software components and guaranteeing that the application being tested is compatible with other programs that will probably be running concurrently.

3.1.3 Workflow diagram

3.1.4 Software or tools available to implement source code vulnerability analysis

- Fortify Static Code Analyzer

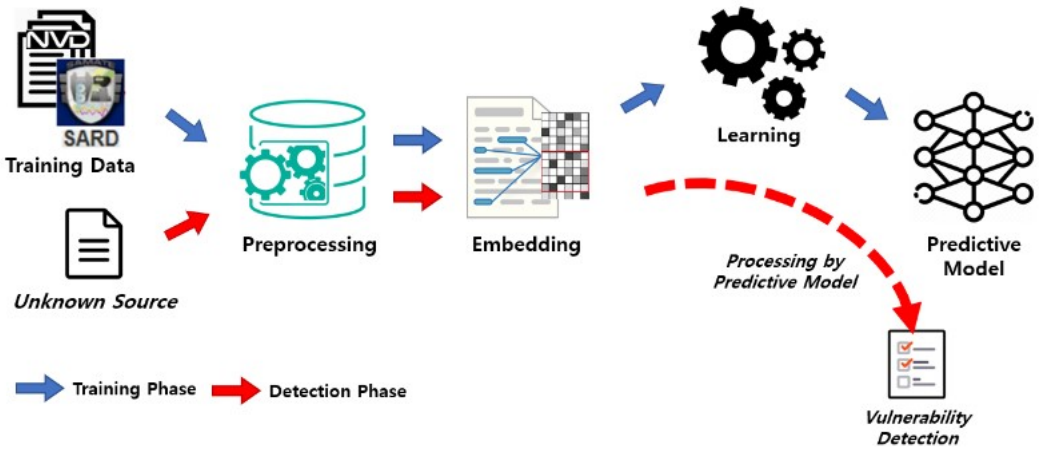

Figure 1: Workflow of the model

- Checkmarx CxSAST
- Coverity Scan
- Veracode
- AppScan
- Reshift Security
- Brakeman

3.1.5 Use of automatic source code vulnerability detector in cyber security

The process of locating, analyzing, managing with, and reporting security flaws in software and devices is understood as vulnerability management. By reducing the regions of their operations where they're vulnerable, organisations may better protect themselves from possible dangers. The method of finding and repairing vulnerabilities in systems on a network may be automated with the use of vulnerability management software. System's vulnerabilities are compiled and located using vulnerability scanners and sometimes endpoint agents. When vulnerabilities are identified, it's important to consider how they could affect various scenarios before deciding how to effectively resolve them. Verifying vulnerabilities, as an example , vulnerability validation can be used in contextualizing true severity of vulnerabilities.

3.2 Problem Statement

Tools for automated source code analysis offer a repeatable, objective method of application bench marking assessments. It is simple to overlook serious soft-

ware bugs due to reusable functionality, test shortcuts, and short deadlines. The benchmark metrics for software quality, developer productivity, application complexity, technical debt, and software risk are provided by the Source Code Vulnerability Analysis Tool. The number of vulnerabilities recorded in a single year has never been larger than in 2020, according to figures from the Common Vulnerabilities and Exposures (CVE) group and the National Vulnerability Database (NVD) Report [17](18,103). It is clear that over the past three years, the number of vulnerabilities has peaked.

Semantic feature learning, high false negatives and false positives, data set, cross-project vulnerability detection, and code metric are some limitations of deep learning used in source code security research. Fine-grained program feature representation, DL paired with static and dynamic program analysis technology, creating an open source unified data-set standard library, transfer learning, and new code attribute are some of the potentials there. Source code-based code similarity-based vulnerability detection may identify numerous clone kinds; binary code-based detection is more accurate but has a greater risk of false positives (source) and analytical complexity (binary). And for code pattern-based vulnerability detection, the static technique gets higher code; the coverage dynamic method detects more quickly, but it also has poor code coverage and lacks run-time information (dynamic).

3.3 Proposed Solution

Finding vulnerabilities should ideally be automated in a method that is both highly accurate and much quicker than manual code examination. Instead of requiring arbitrary characteristics that must be manually defined, it would automatically learn features from existing code. Such a tool would assist human specialists by decreasing or obviating the requirement for the most time-consuming and error-prone procedures in vulnerability detection. If such a tool could identify potential vulnerabilities at their positions in the source code, it would be very helpful to developers because it has a low probability of false positives and false negatives.Reusable functionality, test shortcuts, and tight deadlines make it easy to overlook critical software bugs.

Comparing machine learning techniques to static analysis and other conventional methods, it is true that they are not yet commonly employed. Machine learning for vulnerability feature identification has been studied, although many of the research work with synthetic code samples, have extremely tiny data sets, or are only relevant to a small number of projects. Many popular programming languages haven't gotten any attention thus far. Additionally, a lot of suggested methods just classify whole files, which is simpler to accomplish but less helpful for developers. In order to describe code and build models, a variety of strategies have been used up to this point, but some of them have failed to capture the sequential nature and semantic structure of source code.

In light of this, there is still much work to be done. By suggesting VUDENC, a program that uses a deep neural network to learn vulnerability traits from a sizable code base mined from GitHub, this study seeks to make a contribution

to the field. In order to truly be useful to real-life applications, VUDENC concentrates on Python code and only learns from natural code. Using word2vec to convert the source code into a numerical vector representation, it operates directly on the source code itself. VUDENC employs a fine-grained method that analyses individual code tokens in order to categorise certain sections of the code. A lengthy short term memory network that has the capacity to employ an internal "memory" to capture the semantic context of code tokens is used to simulate vulnerabilities. A proof-of-concept for a deep learning-based vulnerability detection tool, VUDENC demonstrates a viable methodology. The application to code is illustrated by a number of instances.

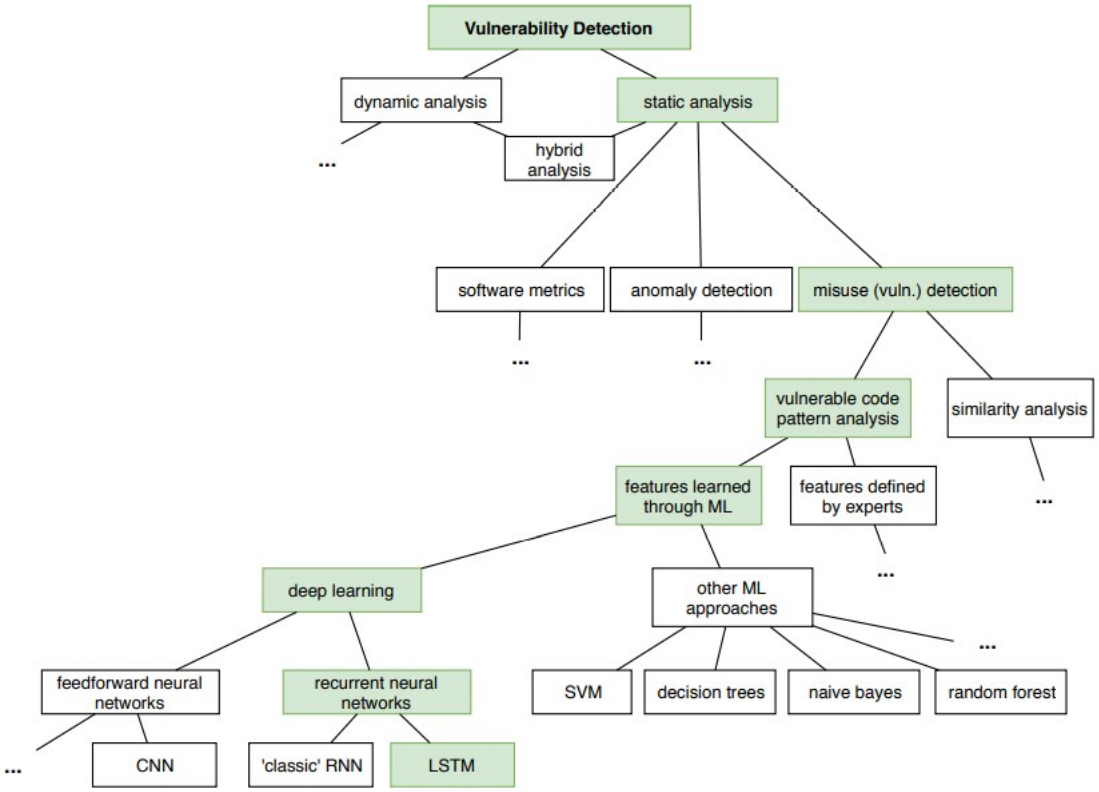

Figure 2: One way to structure different approaches for vulnerability detection

3.3.1 Architecture

To provide a succinct summary: To comprehend the context in which a token occurs, the VUDENC technique uses code tokens and the tokens around it. Using a word2vec paradigm, the code is encoded in numerical vectors. Following that, an LSTM network is used to identify susceptible code characteristics and categorise code as vulnerable or not vulnerable (using a final dense layer with a single output neuron).

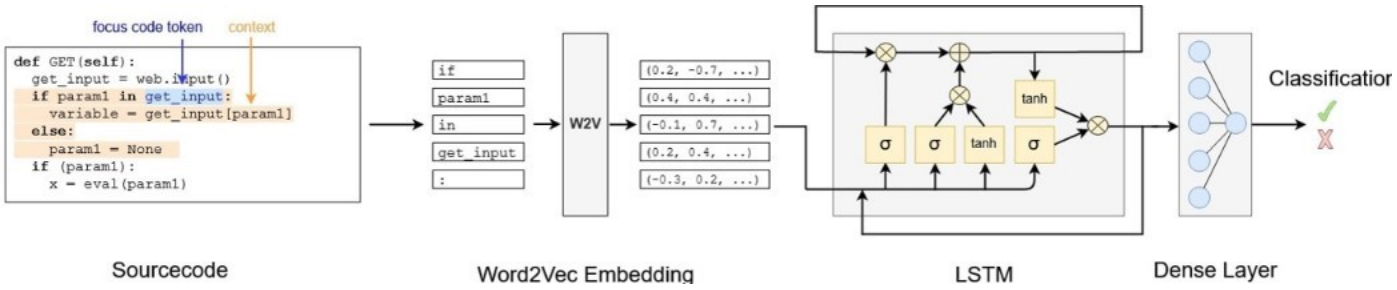

Figure 3: Architecture of the model

- Source Code : For a number of reasons, the whole dataset is compiled from projects that are publicly accessible on GitHub. First off, Github is the largest source code repository in the world, thus it is doubtful that there won't be enough relevant data for this application. Second, almost all projects on GitHub contain "natural" source code in the sense that they are genuine projects that are utilized in practise, as opposed to synthetic code bases. Third, because the data is open to the public, it is simpler to review and repeat the work than it is for studies that, for example, rely on proprietary code.

- word2vec Embedding : In 2013 [14], word2vec embedding was created. Each unique token is also given a unique vector representation, although semantics are taken into consideration by providing comparable tokens with similar representations. Since Word2vec is a two-layer network, it must be trained independently by being given a corpus of training data from which it may learn semantics. Word2vec assigns vectors with high cosine similarity to tokens that share semantic properties based on how words combine and relate to one another. Complete similarity or equivalent is stated as a 0-degree-angle, whereas total similarity or equivalence is expressed as a 90-degree-angle if two terms have nothing in common and have no link to one another at all.

- LSTM : In 1997 [14], Sepp Hochreiter and Jürgen Schmidthuber proposed the concept of LSTMs. Across the use of a gradient-based method that guarantees consistent error flow through the internal states, prohibiting

both bursting and vanishing, they provide a solution to the vanishing gradient problem. Other LSTM variations include so-called "peephole connections," which enable even the gate layers to account for the present state of the cell memory [9], and gated recurrent units [5], which combine certain gates and alter and simplify the LSTM's architecture. With astonishing dexterity, the Open project AI also taught an LSTM how to instruct a robot hand to manipulate items.

- Dense Layer : Neuron density might differ between layers. The last layer may just contain one neuron if the output is a single value, as is the case in several classification tasks. A layer is said to be "dense" or completely connected when every neuron in the layer above is linked to every neuron in the layer in question. Layer 4 in the figure is a dense layer; all other layers are sparse, or less dense. Only the activation layer, a dense output layer with a single neuron, follows the LSTM layer. Because the objective is to anticipate a value between 0 and 1 for the two groups of vulnerable or not vulnerable, a sigmoid activation function is utilised here.

- Classification : Therefore, it will be simpler to do classification and prediction on artificial data sets, or data sets that have been carefully chosen and vetted, than it will be to train a model on potentially messy real-world code from sources like Github. Due to its clear and consistent style and structure, Russel et al. were able to reach 84 percent on the SATE test suite with roughly the same methodology while only achieving 56 percent on natural code from Github. Considering that the VUDENC technique only uses natural, real-life code, it may be argued that it performs rather well.

3.3.2 Implementation

Algorithm for the model : Static code analysis uses generalization and abstract principles to examine a program's form, structure, content, or documentation[13] without running it. As a result, static techniques can identify the specific reason why a vulnerability was found [8]. They can identify dangers resulting from problems with information flow, access control, and (in)proper usage of APIs (such as cryptographic libraries) using a variety of different techniques[16]. The fundamental concept is to extract patterns from massive volumes of data, use a machine learning algorithm to find vulnerability traits, and then categorize code as necessary. This means that vulnerability detection may be automated and potentially used very early in the software's lifetime, which can drastically lower the costs of locating and repairing mistakes. [3] [8] A appropriate representation must be established and the data must be formatted in a specified way before being provided to a machine learning algorithm. Depending on how the model is set up, any bit of data that is stored in this representation may turn into a feature.[6] Machine learning has the advantage that it can produce features without relying on human judgment or expertise. Such techniques can evaluate big repositories without generating the source code, much like a static

analysis. However, they do not suffer from the false positive problem as much because of their ability to precisely adjust accuracy and recall, giving them an edge over both static and dynamic techniques that is at least promising. [18].

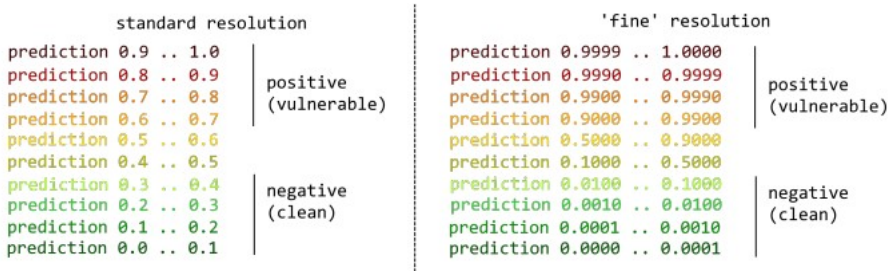

Figure 4: colors and confidence levels

```
def run_batch_mode(tweaks, args):
    for t in tweaks:
        if os_supported(t['os_v_min'], t['os_v_max']) \
                and is_executable(t['group'], args.groups, is_admin()) \
                and t['group'] != 'test':
            run_command(t['set'])


def run_command(cmd):
    try:
        subprocess.run(cmd, shell=True, timeout=60, check=True)
        dglogger.log_info(str(cmd))
    except subprocess.CalledProcessError as e:
#         dglogger.log_error(e, file=sys.stderr)
        dglogger.log_error(str(e)) # figure out deal w/file=sys.stderr!
    except subprocess.TimeoutExpired as e:
        dglogger.log_error(e, file=sys.stderr)
    except OSError as e:
        dglogger.log_error(e, file=sys.stderr)
    except KeyError as e:
        dglogger.log_error(e, file=sys.stderr)
```

Figure 5: Code with a command injection vulnerability with colored classifications (fine resolution)

The trained classifier's usefulness is next evaluated using the source code. The code is divided into chunks just as it was in the past (using a small focus area and a sliding context window as described above). The focus area moves through the code, and with each successive step, the surrounding context is taken into account. The model then uses that context as input to produce a forecast, and that prediction serves as the focus area's vulnerability categorization. The various categorization confidence levels are highlighted using different colors in the figures.

standard resolution		'fine' resolution	
Label 0 (not vulnerable)		Label 0 (not vulnerable)	
prediction 0.9 .. 1.0	false positive	prediction 0.9999 .. 1.0000	false positive
prediction 0.8 .. 0.9		prediction 0.9990 .. 0.9999	
prediction 0.7 .. 0.8		prediction 0.9900 .. 0.9990	
prediction 0.6 .. 0.7		prediction 0.9000 .. 0.9900	
prediction 0.5 .. 0.6		prediction 0.5000 .. 0.9000	
prediction 0.4 .. 0.5		prediction 0.1000 .. 0.5000	
prediction 0.3 .. 0.4	true negative	prediction 0.0100 .. 0.1000	true negative
prediction 0.2 .. 0.3		prediction 0.0010 .. 0.0100	
prediction 0.1 .. 0.2		prediction 0.0001 .. 0.0010	
prediction 0.0 .. 0.1		prediction 0.0000 .. 0.0001	
Label 1 (vulnerable)		Label 1 (vulnerable)	
prediction 0.5 .. 1.0	true positive	prediction 0.5 .. 1.0	true positive
prediction 0.0 .. 0.5	false negative	prediction 0.0 .. 0.5	false negative

Figure 6: Colors and confidence levels taking labels into account

```
    original_names = {}
    for index in source.indexes:
        # make sure the name is < 63 chars with the suffix
        name = hashlib.sha256(temp + index.name.encode('utf-8') + suffix).hexdigest()[0:60]
        original_names[name] = index.name
        columns = []
        for column in index.columns:
            columns.append(next(x for x in destination.columns if x.name == column.name))
        new = sqlalchemy.Index(name, *columns)
        new.unique = index.unique
        new.table = destination
        new.create(bind=self._connection)
    self.insert(destination.name, dataframe, batch_size=batch_size)
    self.execute("BEGIN; SET LOCAL statement_timeout = '1min'; ANALYZE %s; COMMIT;" % table)

    with self as transaction:
        backup = sqlalchemy.Table(table + '_b', self.metadata)
        backup.drop(bind=self._connection, checkfirst=True)
        source.rename(name=source.name + '_b', connection=self._connection)
        destination.rename(name=table, connection=self._connection)
        for index in source.indexes:
            index.rename(index.name[0:-2] + '_b', connection=self._connection)
        for index in destination.indexes:
            index.rename(original_names[index.name], connection=self._connection)

for func in _after_replace_callbacks:
    func(destination, source)
```

Figure 7: Code with SQL vulnerability with colored classifications using labels

True positives, false positives, true negatives, and false negatives may all be visually identified if the labels are accessible (perhaps because the code derives from the labeled data set). In this instance, the colors for the code that was designated as being secure remain the same. This translates to real positives being in hues of green while orange and red indicate false positives. Blue indicates vulnerable code that has been identified as such (true positive), whereas brilliant purple indicates undetected vulnerability (false negative). Another code snippet containing a vulnerability is highlighted using the labels in the illustration. The vulnerabilities were identified as evidenced by the blue hue, while the majority of other areas are dark green since they are not susceptible and are not seen as such. There is some difference just beyond the susceptible area's bounds (magenta and red). The fact that the vulnerable code fragment's core was appropriately identified as such, however, is what matters. The GitHub repository has a tonne of examples that are posted online along with instructions on how to get the same outcomes. [23]

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 test set. 10.9percent or so of the code fragments have some susceptible code in them. With the above-mentioned hyper parameters, the LSTM model was trained on the training set for 100 iterations, yielding accuracy, precision, recall, and F1 score improvements of 92.5percent, 82.2percent, 78.0percent, and 80.1percent, respectively. Figures show a tiny example of a SQL injection patch on Github and the model's identification of the susceptible code section.

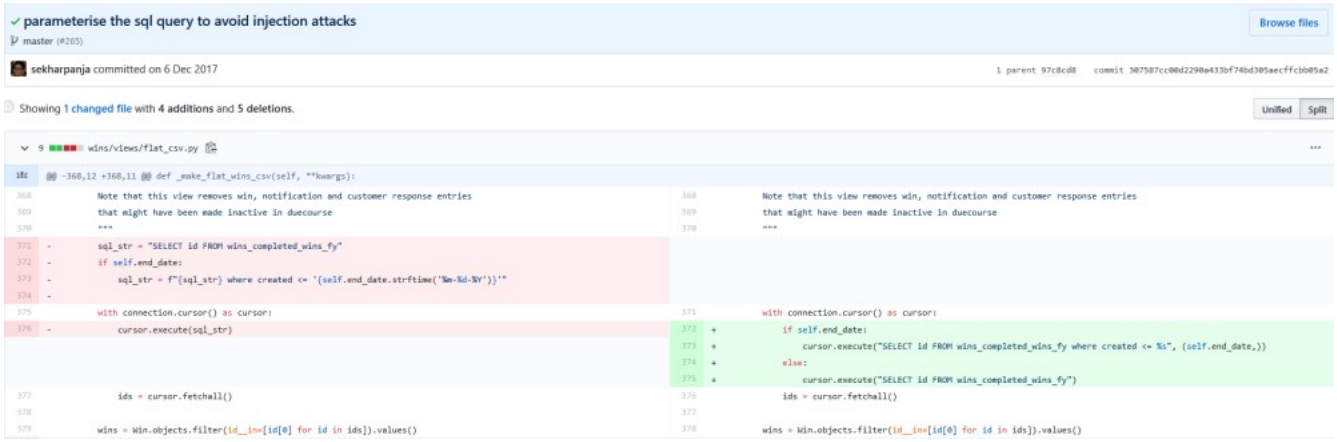

Figure 8: Commit for vulnerability (SQL injection)

```
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)

Cross-site scripting : A rate of 8.9percent susceptible 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 86.0percent, 97.7percent, and 91.9percent. For an illustration of how the model finds an XSS vulnerability, see Figures 25 and 26. The variable self. content is used to create dynamically generated HTML code for a comment area. This code has to be escaped to prevent script injection.

Figure 10: Commit for vulnerability (XSS)

```
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 (XSS)

Hardware and Software Requirements : Machine Learning Approach (CNN, RNN, LSTM) and Real software packages including Github and the Debian Linux distribution and benchmark datasets are used to find vulnerabilities. Processor: 10th Gen Intel Core i5-1035G1 processor, 1.0 Ghz base speed, 3.6 Ghz max speed, 4 Cores, 6Mb Smart Cache Operating System: Pre-loaded Windows 10 with lifetime validity, Memory and Storage: 8 GB RAM Storage 512 GB SSD.

4 Evaluation

The models are trained on the training data-sets, and the validation data-set is used to assess the models' performance with various hyper-parameter values. The final test data set is utilised for the results.

4.1 The Data

Commits that fixed several vulnerabilities were compiled. Each vulnerability needed its own data-set 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 data-set, 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 data-set is appropriate.

Vulnerability	**rep.**	**commits**	**files**	**functions**	**LOC**	**chars**
sql injection	336	406	657	5388	83558	3960074
xss	39	69	81	783	14916	736567
command injection	85	106	197	2161	36031	1740339
xsrf	88	141	296	4418	56198	2682206
remote code execution	50	54	131	2592	30591	1455087
path disclosure	133	140	232	2968	42303	2014413
open redirect	81	93	182	1762	26521	1295748

4.2 The baseline model

A baseline model was developed to demonstrate the consequences of various hyper-parameter adjustments. Since some configurations must be used as a starting point, even when their hyper parameters are not optimum, they may be used to show how alternative hyper parameters lead to better or worse outcomes. The ideal combination of all parameters may be found after going through each hyper parameter and describing how it impacts the performance. The baseline model analyses the data-set for SQL injections using a focus region step size n of 5 and a context length m of 200. It contains 30 neurons and is trained using the Adam optimizer for 10 epochs with a dropout and recurrent dropout of 20percent and a batch size of 200. Results are contrasted using the classification performance of the resulting LSTM model's F1 score, which provides a balanced value that takes accuracy and recalls into account. It should be stressed that the nondeterministic nature of the entire process can cause scores for precision, accuracy, recall, etc. to vary by around 1percent to 3percent when the same model is trained on the same data twice, one right after the other.

4.3 Performance for subsets of vulnerabilities

Several of the initial considerations for vulnerabilities have to be eliminated. There were relatively few hits 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 using the keywords brute force, tampering, directory traversal, hijacking, replay attack, man-in-the-middle, formatstring, unauthorised, and sanitise. A thorough review of a few randomly chosen samples revealed that the majority of those commits dealt with other problems unrelated to thwarting an attack. Therefore, it was unable to produce a high-quality dataset for such 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 the table below.

Vulnerability	**Accuracy**	**Precision**	**Recall**	**F1**
SQL injection	92.5%	82.2%	78.0%	80.1%
XSS	97.8%	91.9%	80.8%	86.0%
Command injection	97.8%	94.0%	87.2%	90.5%
XSRF	97.2%	92.9%	85.4%	89.0%
Remote code execution.	98.1%	96.0%	82.6%	88.8%
Path disclosure	97.3%	92.0%	84.4%	88.1%
Open redirect	96.8%	91.0%	83.9%	87.3%
Average	*96.8%*	*91.4%*	*83.2%*	*87.1%*

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.

In the GitHub repository, there are three examples provided for each vulnerability that may be used to check out how the model predicts vulnerabilities in source code, in order to show what the model actually does when doing predictions on code. There are programs that apply the models to the source code, producing colored highlights in the code that indicate the locations where the model thinks a vulnerability may exist. The figure illustrates the exact meanings of the colors.

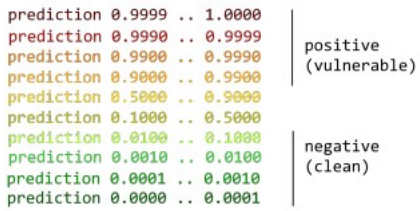

Figure 12: color key for the predictions

4.4 Comparison with other works

To give a framework for the assessment of this study, the next part includes comparisons with related research in the area. Each technique has inherent variances, hence it is impossible to directly compare them. The following comparison criteria are used for approaches:

- Language: what language is subject of the classification efforts
- Data basis: does the data stem from real-life projects or from synthetic databases (such as benchmark data sets).
- 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)
- Machine Learning Approach: what class of neural network or machine learning
- Vulnerability types: which kinds of vulnerabilities are detected
- Size of data set: how many functions, projects, classes etc. make up the dataset
- Scope and applicability: Has the model been trained on a single project and can it only classify files within that application, or is it generally applicable to any code from a large variety of sources

characteristics of approach							resulting metrics			
Name	Language	Data	Labels	Scope	Granularity	Method	Acc.	Pre.	Rec.	F1
Russel et al.	C/C++	real & synth.	static tool	general	token level (fine)	CNN, RNN				57%
Pang et al.	Java	real	pre-existing	4 apps	whole classes	SVM	63%	67%	63%	65%
VuRLE	Java	real	manually identified	general	edits (fine)	10-fold CV		65%	66%	65%
VulDeePecker	C/C++	real & synth.	patches & manual	general	API/function calls	BLSTM				85%-95%
Dam et al.	Java	real	static tool	18 apps	whole file	LSTM	4 / 17 (see above)			
Hovsepyan et al.	Java	real	static tool	1 project	whole file	grid search	87%	85%	88%	85%
VUDENC	Python	real	patches	general	token level	LSTM	97%	91%	83%	87%

It follows that focusing on predictions inside a single project, as done by Hovsepyan et al.[10] and Dam et al.[4] when they train a classifier to anticipate vulnerabilities within the same application, would make it much easier to improve metrics for accuracy, recall, and F1. A classifier that can be used to identify vulnerabilities, in general, is far more challenging to train, but the payoff is also much greater. It should be emphasized that neither of these two approaches nor the one employed by Pang et al.[15], can pinpoint the exact location of the vulnerability. Instead, they all aim to predict whether or not an entire file is susceptible. VUDENC has a considerably more challenging mission to complete since it seeks to provide a generic vulnerability detector that can be used at the fine granularity of code tokens.

Additionally, the calibre of the underlying data has a significant impact on a model's quality. It will thus be simpler to classify and forecast using artificial data sets, or data sets that have been carefully chosen and vetted than it will be to train a model using potentially messy real-world code from sources like GitHub. Due to its clear and consistent style and structure, Russel et al. were able to reach 84percent on the SATE test suite with roughly the same methodology while only achieving 56percent on natural code from Github. Given that the VUDENC technique only applies to realistic, real-life code, it may perform fairly well.

5 Conclusion

VUDENC is a deep learning-based vulnerability detection solution that learns from a natural code base. It can forecast 7 different kinds of vulnerabilities and runs on Python source code. Github commits were mined for a sizable dataset, and the commit context was used to label the commits. The dataset is freely accessible and can be used to repeat this study. A word2vec model has been trained on a sizable corpus of Python code to be able to carry out semantically preserved code token embeddings.

Systematic tests reveal that VUDENC achieves 96.8 percent accuracy, 83 percent recall, 91 percent precision, and 87 percent F1 score. Future research should concentrate on enhancing the method for labelling the data and gathering

the dataset.

References

[1] J. Akram and P. Luo. Sqvdt: A scalable quantitative vulnerability detection technique for source code security assessment. *Software: Practice and Experience*, 51(2):294–318, 2021.

[2] Z. Bilgin, M. A. Ersoy, E. U. Soykan, E. Tomur, P. Çomak, and L. Karaçay. Vulnerability prediction from source code using machine learning. *IEEE Access*, 8:150672–150684, 2020.

[3] H. K. Dam, T. Tran, T. Pham, S. W. Ng, J. Grundy, and A. Ghose. Automatic feature learning for vulnerability prediction. *arXiv preprint arXiv:1708.02368*, 2017.

[4] H. K. Dam, T. Tran, T. Pham, S. W. Ng, J. Grundy, and A. Ghose. Automatic feature learning for vulnerability prediction. *arXiv preprint arXiv:1708.02368*, 2017.

[5] F. A. Gers and J. Schmidhuber. Recurrent nets that time and count. In *Proceedings of the IEEE-INNS-ENNS International Joint Conference on Neural Networks. IJCNN 2000. Neural Computing: New Challenges and Perspectives for the New Millennium*, volume 3, pages 189–194. IEEE, 2000.

[6] I. Goodfellow, Y. Bengio, and A. Courville. *Deep learning*. MIT press, 2016.

[7] J. Guo, Z. Wang, H. Li, and Y. Xue. Detecting vulnerability in source code using cnn and lstm network. *Soft Computing*, pages 1–11, 2021.

[8] M. K. Gupta, M. Govil, and G. Singh. Static analysis approaches to detect sql injection and cross site scripting vulnerabilities in web applications: A survey. In *International Conference on Recent Advances and Innovations in Engineering (ICRAIE-2014)*, pages 1–5. IEEE, 2014.

[9] S. Hochreiter and J. Schmidhuber. Long short-term memory. *Neural computation*, 9(8):1735–1780, 1997.

[10] A. Hovsepyan, R. Scandariato, W. Joosen, and J. Walden. Software vulnerability prediction using text analysis techniques. In *Proceedings of the 4th international workshop on Security measurements and metrics*, pages 7–10, 2012.

[11] D. Kong, Q. Zheng, C. Chen, J. Shuai, and M. Zhu. Isa: a source code static vulnerability detection system based on data fusion. In *InfoScale*, page 55, 2007.

[12] X. Li, L. Wang, Y. Xin, Y. Yang, and Y. Chen. Automated vulnerability detection in source code using minimum intermediate representation learning. *Applied Sciences*, 10(5):1692, 2020.

[13] B. Liu, L. Shi, Z. Cai, and M. Li. Software vulnerability discovery techniques: A survey. In *2012 fourth international conference on multimedia information networking and security*, pages 152–156. IEEE, 2012.

[14] T. Mikolov, I. Sutskever, K. Chen, G. S. Corrado, and J. Dean. Distributed representations of words and phrases and their compositionality. *Advances in neural information processing systems*, 26, 2013.

[15] Y. Pang, X. Xue, and A. S. Namin. 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, 2015.

[16] M. Pistoia, S. Chandra, S. J. Fink, and E. Yahav. A survey of static analysis methods for identifying security vulnerabilities in software systems. *IBM systems journal*, 46(2):265–288, 2007.

[17] REDSCAN. Nist security vulnerability trends in 2020: an analysis. 2020.

[18] R. Russell, L. Kim, L. Hamilton, T. Lazovich, J. Harer, O. Ozdemir, P. Ellingwood, and M. McConley. 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.

[19] N. Saccente, J. Dehlinger, L. Deng, S. Chakraborty, and Y. Xiong. Project achilles: A prototype tool for static method-level vulnerability detection of java source code using a recurrent neural network. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering Workshop (ASEW)*, pages 114–121. IEEE, 2019.

[20] G. Tang, L. Yang, S. Ren, L. Meng, F. Yang, and H. Wang. An automatic source code vulnerability detection approach based on kelm. *Security and Communication Networks*, 2021, 2021.

[21] H. Wang, G. Ye, Z. Tang, S. H. Tan, S. Huang, D. Fang, Y. Feng, L. Bian, and Z. Wang. Combining graph-based learning with automated data collection for code vulnerability detection. *IEEE Transactions on Information Forensics and Security*, 16:1943–1958, 2020.

[22] W. Wang, J. Song, G. Xu, Y. Li, H. Wang, and C. Su. Contractward: Automated vulnerability detection models for ethereum smart contracts. *IEEE Transactions on Network Science and Engineering*, 8(2):1133–1144, 2020.

[23] L. Wartschinski. Vudenc - python corpus for word2vec, 2. 12. 2019.