



Dissertation on
Question Tagging and Vulnerability Detection

*Submitted in partial fulfilment of the
requirements for the award of degree of*

**Bachelor of Technology
in
Computer Science and Engineering**

UE19CS390A – Capstone Project Phase - 2

Submitted by:

Anurag Khanra	PES1UG19CS072
Arvind Krishna	PES1UG19CS090
Jeevan Samrudh LH	PES1UG19CS196
Rahul D Makhija	PES1UG19CS368

Under the guidance of

V.R. Badri Prasad
Associate Professor

August - December 2022

**DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING
FACULTY OF ENGINEERING
PES UNIVERSITY**

(Established under Karnataka Act No. 16 of 2013)
100 Feet Ring Road, Bengaluru – 560 085, Karnataka, India



PES
UNIVERSITY

PES UNIVERSITY

(Established under Karnataka Act No. 16 of 2013)
100ft Ring Road, Bengaluru – 560 085, Karnataka, India

FACULTY OF ENGINEERING

CERTIFICATE

This is to certify that the dissertation entitled

Question Tagging and Vulnerability Detection

is a bona fide work carried out by:

Anurag Khanra	PES1UG19CS072
Arvind Krishna	PES1UG19CS090
Jeevan Samrudh LH	PES1UG19CS196
Rahul D Makhija	PES1UG19CS368

in partial fulfilment for the completion of seventh semester Capstone Project Phase - 2
(UE19CS390A) in the Program of Study - **Bachelor of Technology in Computer Science and Engineering** under rules and regulations of **PES University, Bengaluru** during the period August.
2022 – December 2022. It is certified that all corrections / suggestions indicated for internal
assessment have been incorporated in the report. The dissertation has been approved as it satisfies
the 7th semester academic requirements in respect of project work.

Signature
V.R. Badri Prasad
Associate Professor

Signature
Dr. Shylaja S S
Chairperson

Signature
Dr. B K Keshavan
Dean of Faculty

External Viva

Name of the Examiners

Signature with Date

1.

2.

DECLARATION

We hereby declare that the Capstone Project Phase - 2 entitled “**Question Tagging and Vulnerability Detection**” has been carried out by us under the guidance of Prof. V.R. Badri Prasad, Associate Professor and submitted in partial fulfilment of the course requirements for the award of degree of **Bachelor of Technology in Computer Science and Engineering** of **PES University, Bengaluru** during the academic semester August - December 2022. The matter embodied in this report has not been submitted to any other university or institution for the award of any degree.

Anurag Khanra **PES1UG19CS072**

Arvind Krishna **PES1UG19CS090**

Jeevan Samrudh LH **PES1UG19CS196**

Rahul D Makhija **PES1UG19CS368**

ACKNOWLEDGEMENT

We would like to express our gratitude to Prof. V. R. Badri Prasad, Department of Computer Science and Engineering, PES University, for her/his continuous guidance, assistance, and encouragement throughout the development of this UE19CS390A - Capstone Project Phase – 2.

We are grateful to the project coordinator, Prof. Mahesh H.B., for organizing, managing, and helping with the entire process.

We take this opportunity to thank Dr. Shylaja S S, Chairperson, Department of Computer Science and Engineering, PES University, for all the knowledge and support we have received from the department. We would like to thank Dr. B.K. Keshavan, Dean of Faculty, PES University for his help.

We are deeply grateful to Dr. M. R. Doreswamy, Chancellor, PES University, Prof. Jawahar Doreswamy, Pro Chancellor – PES University, Dr. Suryaprasad J, Vice-Chancellor, PES University for providing to us various opportunities and enlightenment every step of the way. Finally, this project could not have been completed without the continual support and encouragement we have received from our family, friends and the technical staff of our department.

ABSTRACT

With the increase in the usage of the internet for gaining and providing information and knowledge, questionnaire forums are becoming popular means of the same for both teenagers and adults. Users post questions and mark them with the topic that they are related to (known as tags). Post this, users always expect quick answers/solutions from known and reliable sources. However, a large number of posted questions remain unanswered due to erroneous and huge number of tags.

In our system, we propose an automated method to generate these tags using machine learning and deep learning techniques. This mainly helps in standardizing the tag content for similar questions and the total number of tags that the system needs to deal with. These tags can be generated from the content of the question provided by the user.

In addition to this, for questionnaire forums related to programming questions like StackOverflow and StackExchange, our system aims to use the code snippets provided by the user to detect some common software vulnerabilities using deep learning techniques. Software Vulnerabilities are flaws in a computer system that weakens the overall security of the device/system. Detection of these vulnerabilities beforehand will benefit the user and prevent any sort of security threats in their system.

TABLE OF CONTENTS

1. INTRODUCTION	8
2. PROBLEM STATEMENT.....	11
3. LITERATURE SURVEY.....	13
4. DATA	19
4.1 Question Tagging Dataset	19
4.2 Software Vulnerability Detection Dataset	19
5. METHODOLOGY	21
5.1 Question Tagging –	21
5.1.1 Data Processing and Preparation:	21
5.1.2 Model Building.....	26
5.2 Question Forwarding	28
5.3 Vulnerability Detection	29
5.3.1 Pre-processing and tokenizing of software vulnerability dataset	29
5.3.2 Training model to detect software vulnerability	30
6. RESULTS AND DISCUSSIONS	37
6.1 Question Tagging	37
6.2 Vulnerability Detection	40
7. PLAN OF WORK FOR CAPSTONE PROJECT PHASE-3	42
7.1 Deliverable	42
7.2 Expected Result	42
REFERENCES/BIBLIOGRAPHY	43
APPENDIX: DEFINITIONS, ACRONYMS, AND ABBREVIATIONS	44

LIST OF FIGURES

Figure 1 - Multilabel classification of different Software Vulnerabilities.....	20
Figure 2 - Dataset of questions and their tags	21
Figure 3 - Popularity of different tags	22
Figure 4 - Number of answers for different tags	23
Figure 5 - Co-occurrence of tags	24
Figure 6 - Question tagging dataset after initial pre-processing.....	24
Figure 7 - Final pre-processed dataset for Question tagging	25
Figure 8 - Questions with their binary representation of op 50 tags.....	25
Figure 9 - Question Tagging ANN	26
Figure 10 - Question Tagging LSTM	26
Figure 11 - Question Tagging CNN	27
Figure 12 - Static analysis with a correlation matrix	27
Figure 13 - Function to retrieve domain experts for a tag	28
Figure 14 - unimportant tokens present in code	29
Figure 15 - Software Vulnerability dataset.....	29
Figure 16 - Relevant tokens remaining in the data	30
Figure 17 - Software Vulnerability pre-processed data.....	30
Figure 18 - SW Vuln. ANN model.....	31
Figure 19 - SW Vuln. LSTM model.....	32
Figure 20 - Summary of the LSTM model	32
Figure 21 - Highly optimized CNN model for detecting SW vulnerabilities	34
Figure 22 - Results of the CNN model for SW vulnerability	34
Figure 23 - API Schema	35
Figure 24 - Initialization of SVD model.....	35
Figure 25 - Function to detect vulnerability	35
Figure 26 - Results of different models trained to detect question tags	37
Figure 27 - Comparison of our best model with the state-of-the-art results.....	38
Figure 28 - Evaluation metrics for models	38
Figure 29 - Precision and Recall of each tag	39
Figure 30 - Example of question tagging model	40
Figure 31 - Results for SVD models	40
Figure 32 - comparison of State-of-the-art models with our best SVD model.....	41
Figure 33 - Evaluation metrics for SVD model.....	41
Figure 34 - Derived metrics of test dataset.....	41

1. INTRODUCTION

StackOverflow is a questionnaire forum where users can ask questions regarding programming domain which will be forwarded to the relevant people to be answered. They do not check the validity and security of the same. Most of the questions that were answered in the forum will be on point to the question asked. Although this is usually a good thing, this might backfire if the question asked by the user has hidden issues within itself. Although there is a good probability that the question will be solved, and the user will get what they desire, there will be a chance that the user will use the incomplete solution which might lead to vulnerabilities in the same, which if unpatched might be devastating to the user or the application. Although code with vulnerability could be technically right and actually get the job done, in almost all cases, it doesn't change the fact that the way of usage is not suggested. In some of the most common vulnerabilities in C, the first stack overflow solution that will be suggested by google is sometimes incomplete in the same sense, which might be devastating to the usage of a new developer trusting StackOverflow to be complete in all senses.

Example:

`strcpy` is used to copy an array of bytes that are terminated by a NULL byte (a C string) from one address (the second parameter) to another address (the first parameter). Usually, it is used to copy a string from one memory location to a totally different memory location:

```
char string[] = "stackflow";
char copied_string[10]; // length of "stackflow" + NULL byte

strcpy(copied_string, string);
puts(copied_string);
```

In the given example, the user asked the way to copy strings in c, the following is an accepted solution. Although the solution is absolutely correct in the given context, it is incomplete since it doesn't not consider the length of the string as a parameter. So, if the string to be copied is longer than the size of "copied string" the null character that is supposed to terminate the string at the end might be overwritten. This will cause undefined behaviour in the same way leading to page faults or improper memory access which can be used by the attacker to gain unauthorized entry to user's memory space.

Our solution to the same is to use Deep learning to detect possible vulnerabilities that could arise from the code, and tagging them along with the default tags that could be generated by using the question

itself entirely. This will not only warn the user about possible mistakes and issues that could arise. We are planning to train a deep learning model which can detect various common vulnerabilities in C/C++ (since most of the high-level languages don't have the same issues that can be seen in C/C++), and we will pass the code provided by the user in StackOverflow to the model to see the presence of any of the same.

The most common C/C++ vulnerabilities are:

- Improper usage of string copy function leading to buffer overflow - <https://cwe.mitre.org/data/definitions/676.html>
- Dangling and expired pointer access: <https://cwe.mitre.org/data/definitions/825.html>
- Improper usage of inherently dangerous buffer input function: <https://cwe.mitre.org/data/definitions/242.html>
- Null pointer dereferences: <https://cwe.mitre.org/data/definitions/476.html>
- Out of bound Read/Write in arrays: <https://cwe.mitre.org/data/definitions/125.html> and <https://cwe.mitre.org/data/definitions/787.html> respectively

We are using a dataset with most of these vulnerable codes and names as values, which will be used to train the model (explained in detail in the DATA section) the same will be tested on stack overflow code, which will give the list of possible vulnerabilities in the same.

Adding to the same, Users post questions and mark them with the topic that they are related to (known as tags). Post this, users always expect quick answers/solutions from known and reliable sources. However, a large number of posted questions remain unanswered due to erroneous and huge number of tags.

For example:

object	60824
programming-challenge	3181
beginner	7207

These are the most commonly used tags in the StackExchange website

The tags like “object” “programming-challenge” or “beginner” provide no insight into the question or the problem that is being faced by the user.

One more issue could be duplicate tags, for example

The tags “python” and “python3” or “python 3.x” are used almost always in pairs. this is not only a waste of time while providing no insights, this also takes up one full valuable slot in the five possible tags that could be used added to the question

The possibility of wrong tags is also considerable, since the user might not entirely know what the issue with the code exactly is, which will lead to personal bias while adding the question.

The solution to both of the above problems is to generate tags automatically using machine learning techniques and remove user from changing the legitimacy and relevancy of the tags that will be linked to the given question

2. PROBLEM STATEMENT

Automated vulnerability detection and tagging of StackOverflow questions using Deep Learning techniques

Humans are erroneous, we have our own personal bias over everything we do. Although sometimes this could be a good thing, this might backfire when we are trying to express a common ground to communicate. The personal bias of the user might hinder them from expressing exactly what they want and lead to miscommunication. This could be either because of incomplete knowledge over the subject matter or due to misbelief over a certain subject leading to false beliefs and assumptions. The problem we are trying to tackle here is the personal bias of the user in using a programming related questionnaire forum like stackoverflow.com or any of the various StackExchange forums. These are the few problems that the users bias might lead to in the same:

- Incomplete questioning (assuming the problem is somewhere else than the actual issue)
- Wrong tagging or selecting the domain of the problem
- Incomplete problem (leading to incomplete solution which could be assumed to be absolute)

We are trying to solve point 2 and 3 partially by completely removing the user from the tagging process, and also checking for various vulnerabilities and issues that could arise from the code (which might get the job done, but could be wrong as well).

- **Wrong tagging or selecting the domain of the problem:**

The users might assume wrong tags for the given problem in their hands. We have seen instance of users putting up redundant tags (overlapping tags like Pytorch and Python, where Pytorch obviously implies the issue is in python) or wrong tags entirely, since the issue might not be clear in the error or problem they are facing (low level language where there is no clear exception handling). We are trying to solve both of these by using a trained model to generate the tags automatically without taking input from the user.

- **Incomplete problem:**

The user might not know hidden issues in the code which they are using (either from question or the answer section of the forum). This might be devastating if the code used by the user although works properly in the given scenario might lead to unexpected or plain wrong behaviour in a different situation (these are called vulnerabilities, which might get abused by an attacker). The user couldn't be blamed for the same since they are at the benefit of doubt of innocence. The solution to this is to train a model to detect the most common vulnerabilities that could arise automatically from

the code snippet if uploaded. This could be used not only to warn the user about the possible underlying issues but also might help the person answering the question.

3. LITERATURE SURVEY

R. Russell et al. [1] aims to detect software vulnerabilities in code snippets in languages like C and C++. Machine Learning and Deep Learning techniques are used to detect software vulnerabilities.

Alternate mechanisms to detect vulnerabilities are as follows:

- Static Analysers - They can detect vulnerabilities without having to execute the code
- Dynamic Analysers - They can detect vulnerabilities by executing the code and monitoring its execution against a fixed number and type of test cases.

As software vulnerability detection is a complex task, data has to be compiled and curated from a large number of sources. This must also include the different varieties of data structures, algorithms and coding patterns. A million functions of C and C++ were compiled from SATE IV Juliet Test Suite, GitHub and Debian based Linux distributions. A total of 12 million function samples were collected from the above-mentioned sources.

For initial pre-processing, source lexing was done as the entire code snippet is huge and contains redundant and unimportant information and hence cannot be used as an input to a machine learning model. The authors built a custom lexer to remove redundant features from code snippets and extract only important features. Standard lexers like that of C and C++ captured too much information which led to overfitting of the model. The total vocabulary size of the custom lexer was 156 tokens. Secondly, as part of data curation, the data was compiled from a variety of different sources and the major challenge faced by the authors was the removal of duplicate functions/inputs. This was handled as part of the data preparation process. The custom lexer built was applied on all of the input functions and duplicates/near duplicates were detected using their lexed representation.

Lastly, for labelling of data, the functions collected had to be labelled into the vulnerabilities they contain. This would compose the training set to be used by the machine learning model. The authors used static analysers, dynamic analysers and commit message tagging to perform this task.

The first method used was Neural network classification and representation learning. In this method, the authors used a word to vec based embedding with $k=13$, and then used CNN and RNN for feature extraction. A fully connected classifier was used after feature extraction. Two hidden layers in the neural network were used of 64 and 16 and then the softmax layer for the output. Both the networks were trained with a batch size of 128, Adam optimization and cross entropy loss. Dataset split - 80:10:10. The second method was ensemble learning on neural representations

In this method, the output of the CNN and RNN layers were passed to ensemble classifiers like Random Forest.

The results showed that CNN models performed better than RNN with a higher recall and Ensemble classifiers performed better than Neural Networks. However, there was scope for improvement to

include features like Improved Labels, Detection of style violations, Commit categorization and Algorithm/Task Classification.

J. R. Cedeño González, J. J. Flores Romero, M. G. Guerrero and F. Calderón [2] aimed to generate tags for questions using a multi class classification technique. The authors mentioned two major methods already implemented in the same field, the use of TF-IDF on a bag of words and then calculation of similarity between questions and tags to evaluate the performance of their model and the use of discriminant classifiers where a classifier was generated for each of the 500 most common question tags and predictions were generated using a selection scheme.

The authors used a dataset generated using the StackOverflow website.

Traditional NLP pre-processing techniques were used. Some of them are lower casing, punctuation removal, stopword removal, regular expression matching and stemming. In addition to the standard techniques, a Computer Science related lexicon was used to preserve certain terms. A bag of words vectorizer was used to convert the remaining terms into vectors. Linear Support Vector Classifier (LSVC) feature selection method was used with the L1 norm as a penalty function.

As part of data preparation, each tag in the training set was assigned a position. As the system was made to generate the top 5 tags, the authors used a separate classifier for each tag/each of the 5 positions. Two classification models were used, - SVM and Naive Bayes. The model was trained with the 100, 200 and 500 most frequent tags. Dataset split - 90:10

In general, it was noticed that the SVC classifier performed better than the NB classifier.

An interesting observation that was noted was that the feature selection method did not impact the classifier performance to a large extent.

L. Ruggahakotuwa et al. [3] found that attacks due to lack of cyber security are a common occurrence. Unverified or improper configurations in the source code causes flaws in security, which is a target for attackers to exploit the vulnerable code and makes the system breached. This paper is aimed at automating the detection and rectification of vulnerabilities in the source code. The model is used in a plugin that can detect and solve the vulnerable source code using the technologies specified in the paper below.

A minor flaw, such as misplacing a script tag, might result in significant data loss. Or a single quote. Veracode, OWASP, Source Clear, and other industry solutions have emerged to address this issue.

Static - Static code analysers detect flaws in source code without running the program.

Peng et al. [1] examined the use of deep learning in programme analysis. Investigations using natural language granularity, which might produce insufficient information, and instead encode abstract syntax tree nodes into vector representations that categorise an AST node as a single neural layer.

These are used by CNN for classification. The qualitative evaluation of deep learning-based classification yields a slightly better result than logistic regression and support vector machines.

Kremenk et al. can automatically summarise the program features using code analysis and graph probability models.

Yamaguchi et al. investigated identifying anomalous or missing criteria linked to security-basic articles in source code via statically tainted source code.

Dynamic - Over and again execute the projects with numerous test inputs and identify the weakness. Dynamic analysis produces very accurate results in vulnerability detection, it is very hard to find the relevant path to activate the vulnerability.

VDiscover is a machine learning-based programme that extracts dynamic information from a binary to forecast whether a vulnerability test case contains software vulnerabilities.

According to Tarmas et al. the random forest machine learning classification method outperforms logistic regression, multi-layer perceptron, and other classification algorithms.

Bug fixing and Patching - Vulture is a tool that automatically learns from the areas of past vulnerabilities to foresee the future vulnerabilities of new segments before they are completely implemented.

Prophet is an automated patch generating framework that acquires patches from open-sourced software, repositories and creates the correct code. Detection of vulnerabilities can be done using static analysis tools based on machine learning, code matrix based on support vector machines for android devices, static and dynamic code aspects for web applications.

Kazuki et al. [4] have built up a framework for source code appreciation strategies and metrics to foresee the understanding exertion in programming upkeep and advancement of the tasks.

Qosai et al. [5] have done an examination on the effect of improving the source code on programming measurements. They have assessed the capability to break down the procedures and tools of programming source code, identifying the potential imperfections of the product items. And furthermore, they have assessed the effect of applying alterations suggested by programming code analysing tools (SCA) on programming measurements.

The objective is to build an open-source plugin that will detect the top ten OWASP vulnerabilities. With a warning indication, it can precisely and effectively detect the susceptible source code parts. The source code that was found to be susceptible is replaced with the right source code. This paper fails to recognise the accuracy of the model built but is a compilation of major software vulnerability detection models.”

J. Dietrich et al. [6] showed in their paper that tags may or may not be connected with posts, and tags may or may not relate to programming language names. When posting about a cross-language integration problem or a comparison inquiry, numerous tags may be used. Users may ask a question

in a certain language, but the code is inaccurate (in the sense that it cannot be processed), therefore the tag is technically invalid because it describes the user's intent rather than the fragment. Cross-reference programming language tags with tags supplied by linguists on GitHub.

They found code snippets using the SOTorrent data set which showed that there were 48, 807, 762 code snippets with at least one tag associated. The top 20 tags were found using TIOBE index. Then the tags were standardized for example python-3 was made into python. The top tags generated were #java, #c, #python, #c++, #javascript, #c#, #php, #sql, #objective-c, #matlab, #r, #perl, #assembly, #swift, #go, #ruby and #plsql. which resulted in 29, 920, 851 snippets with at least one of these tags. #c may be linked to #assembly by code semantics, but #c may be linked to #java by code syntax.

They implemented the project by loading SOTorrent data into MySQL dataset and creating two tables' Snippets with tags from SOTorrent and Tags generated using Linguist. Used Java and R scripts to process the intermediate data.

The limitation was that tag normalization isn't completely accurate (removal of suffixes) and all languages are not represented. Also, leveraging the co-occurrence of SOTorrent and Linguist tagging to compare both techniques If we simply look at the top linguist match, we see that it only matches any of the user tags in about half of the cases (45%). The curve flattens out rapidly as planned, albeit not as quickly as anticipated. This suggests that the two categorization schemes are significantly incompatible. This implies that depending on one strategy, and especially a mix of systems, requires extreme caution.

People label pairs of languages that are used in the similar situation. The most common pair is #php and #javascript, followed by #php and #sql and #c / #c++. They tried to see if programming practices are transitive along the path of how languages changed and if this reflection is exact for all types of languages.

The findings achieved using the two methodologies, however, are frequently inconsistent. This suggests that both should be used with caution, and that a hybrid method combining the strengths of human and computer categorization should be considered as a viable path forward.

T. Saini and S. Tripathi [7] considers that the concept of finding relevant tags in a forum is necessary for reasons such as, reducing search space for duplicate questions, which can be helpful in finding questions which are relatively similar to the one being asked which in all probability might be already answered in the same forum. The same could also be used to find groups and clusters of similar topics which can help the user to quickly browse through the same. The multilabel classification can be implemented by using one-vs-all approach, or by creating multiple individual classifiers and using it with boosting methods or like a random forest classifier. They used one-vs-all approach in which SVM with linear kernels were used. Apart from that, 2 other implementations like Naive Bayes

approach and Feature Extraction method were applied to various sections of the data to increase the performance of the model. Various features in the given code were extracted by using “lexical and syntactical analysers using methods such as parsing and tokenizing” the input to improve the efficiency. They used a dataset which was a large corpus of around 6 million questions asked by the client in the forum. The same was classified into around fifty thousand different topics or tags by the same user. Each entry in the dataset contains a question id, the heading of the question (the title) and the explanation part which can be considered as the body, which consists of the user’s question in normal language and/or code snippets if uploaded. The dataset can be found at - <https://archive.org/details/stackexchange>. Some of the classifiers used for the tag prediction methods are “Bayes Analysis on Indexed Dump” which used classical naive-bayes method and used the same to generate tags using the heading only, “SVM Analysis of List of Titles”, “Unique Feature Extraction Approach” in which the given question is converted to a vector, and the alignment of the same with the labels can be used to predict the tags. The results of the same showed that in a general case, the SVM outperforms Bernoulli and Multinomial Naive Bayes. The improvement could be anywhere between 30 - 50% in the given paper.

X. Zhang, L. Shao and J. Zheng [8] found out that currently, using the existing methods, the usual way of detecting vulnerabilities is done by either analysing the source code using some static method, or it will be by using the execution of the available final executables. The issue arises when some proprietary software vendors refuse to provide the code base to testing for external audit for obvious reasons, so the method of using executable files is often used. The given paper will explore of applying fuzzy techniques for detecting buffer overflow attacks the various ways in which the detection of vulnerability could be either Source-code Audit or Fuzzing technique. Source-code audit technique reads the given piece of code entirely to find and analyse possible vulnerabilities of any kind. This is somewhat suitable for relatively low-level languages like C/C++ since usually the vulnerabilities are standalone and entirely dependent on some other behaviour. This has a deficiency related to the number of possible genres of vulnerabilities that could be detected since it is usually along the lines of basic issues like string overflow or memory leaks and assignment issues since those are usually vulnerable regardless of use case of the same. Fuzzing technique is a way to automatically pass a series of semi-valid test data to the executable program and check the behaviour of the program for undefined/wrong behaviour. This might even lead to crash of the program depending on the reliability of the software. IDA-based assembling code audit is based on reverse engineering and decomposing the executable and checking the vulnerability search on the same. Since the assembly languages isn’t built in an easily human readable format and hundreds of possible assembly instructions, this method is pretty advanced to construct and needs a good experience with reverse

engineering and assembly programming. The term malformed packet means the set of data which adhere to requirements of the given grammar but are not accepted as input to the given program. Few of the easy methods to generate the same are:

- Buffer overflow vulnerabilities: This could be as simple as overloading the input or buffer with large string which might exceed the boundaries of storage allocated
- The easiest method to check the presence of “integer overflow and underflow vulnerabilities” is to pass border values, which could cause unintended effects like integer wrap around. For example, values like “-1, 0, 1, 0xff, 0xffff and 0xffffffff” could be used

The execution of sending of data into the program is found to be of high importance. Most of the known vulnerabilities need a specific constraint to be triggered, that could be a specific input or data, without which the program might function properly and not break down, malform or crash. The main challenge isn't creating a fuzzer which can create and send testing data, the main issue arises when trying to analyse the output/response to find the presence of the vulnerability. This is hard without an external debugger. Upon detection of an exception, the testing software will end the testing process and save the state of the program under consideration along with the additional details like which input caused the trigger etc. Some of the issues which could arise might be because of some proprietary software vendors use some anti-debugging tools while releasing their software for the public to avoid decompiling etc by the general public and competitions. Such software will cause issues whenever we use a debugger while testing the software via fuzzy techniques. The main goal in the future is to find a new algorithm to create test cases for the given software under consideration without including a human tester who could create hidden bias in the test data. Also, an approach to use stacktrace etc of the software to find and track the functions and procedural calls done by the software under consideration with an attempt to find the vulnerability more effectively and not be affected by anti-debug.

4. DATA

This section describes the datasets used for the different machine learning models of the system.

4.1 Question Tagging Dataset

This dataset contains data to be used for the Question Tag Prediction Model and Domain Expert Searching Model.

This is organized into 3 tables:

- Questions.csv containing the title, body, creation date, closed date (if closed), score, and owner ID.
- Answers.csv containing the body, creation date, score, and owner ID.
- Tags.csv containing the tags.

4.2 Software Vulnerability Detection Dataset

This dataset contains data to be used for the Software Vulnerability Detection Model.

This contains the code snippet/functions written in C/C++ along with the software vulnerabilities present in them.

The dataset is of a type called HDF5 (The Hierarchical Data Format version 5), which is basically an open-source file format that supports large, complex, heterogeneous data. HDF5 uses a "file directory" like structure that allows you to organize data within the file in many different structured ways, as you might do with files on your computer.

The data contains a field called 'functionSource' which is basically the raw code snippets themselves along with top 4 vulnerabilities and other fields which are Boolean fields designed to show whether each of the vulnerability is present in that particular snippet of code.

Example:

	0	CWE-119	CWE-120	CWE-469	CWE-476	CWE-other
0	b'default_event_handler(\n GuiWidget *widg...	False	False	False	False	False
1	b'krb5_krbhst_init_flags(krb5_context context,...	False	False	False	False	False
2	b'swap_info_get(swp_entry_t entry)\n{\n\tstruc...	False	False	False	False	False
3	b'parseattrs4(char *&c, const Vec4 &ival = V...	False	False	False	False	False
4	b'generateExecCode(CompileState* comp)\n{\n ...	False	False	False	False	False

Figure 1 - Multilabel classification of different Software Vulnerabilities

5. METHODOLOGY

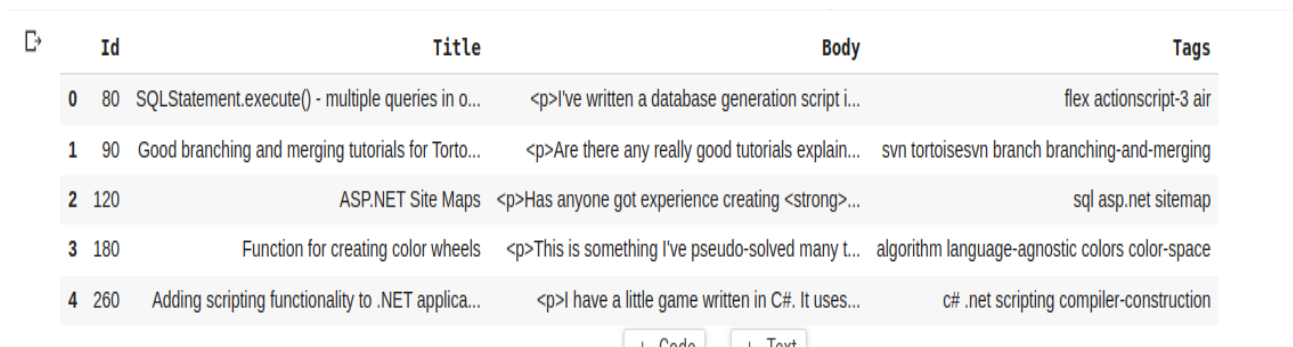
5.1 Question Tagging

The following steps were taken to build the Question Tagging module:

5.1.1 Data Processing and Preparation:

As mentioned in the data section of this report, the dataset was split into 3 csv files containing questions, answers and tags. These data frames were joined using an “Id” field. Irrelevant columns in the resulting data frame were dropped. A sample of the data frame after this step is shown below.

Questions with score less than 5 were not properly framed or were not tagged appropriately and hence cannot be used for further building of the model, hence these were dropped from the dataset.



	Id	Title	Body	Tags
0	80	SQLStatement.execute() - multiple queries in o...	<p>I've written a database generation script i...	flex actionscript-3 air
1	90	Good branching and merging tutorials for Torto...	<p>Are there any really good tutorials explain...	svn tortoiseshn branch branching-and-merging
2	120	ASP.NET Site Maps	<p>Has anyone got experience creating ...	sql asp.net sitemap
3	180	Function for creating color wheels	<p>This is something I've pseudo-solved many t...	algorithm language-agnostic colors color-space
4	260	Adding scripting functionality to .NET applica...	<p>I have a little game written in C#. It uses...	c# .net scripting compiler-construction

Figure 2 - Dataset of questions and their tags

Exploratory Data Analysis - A pictorial view of the dataset was to gain insights on how to further proceed with data cleaning and pre-processing.

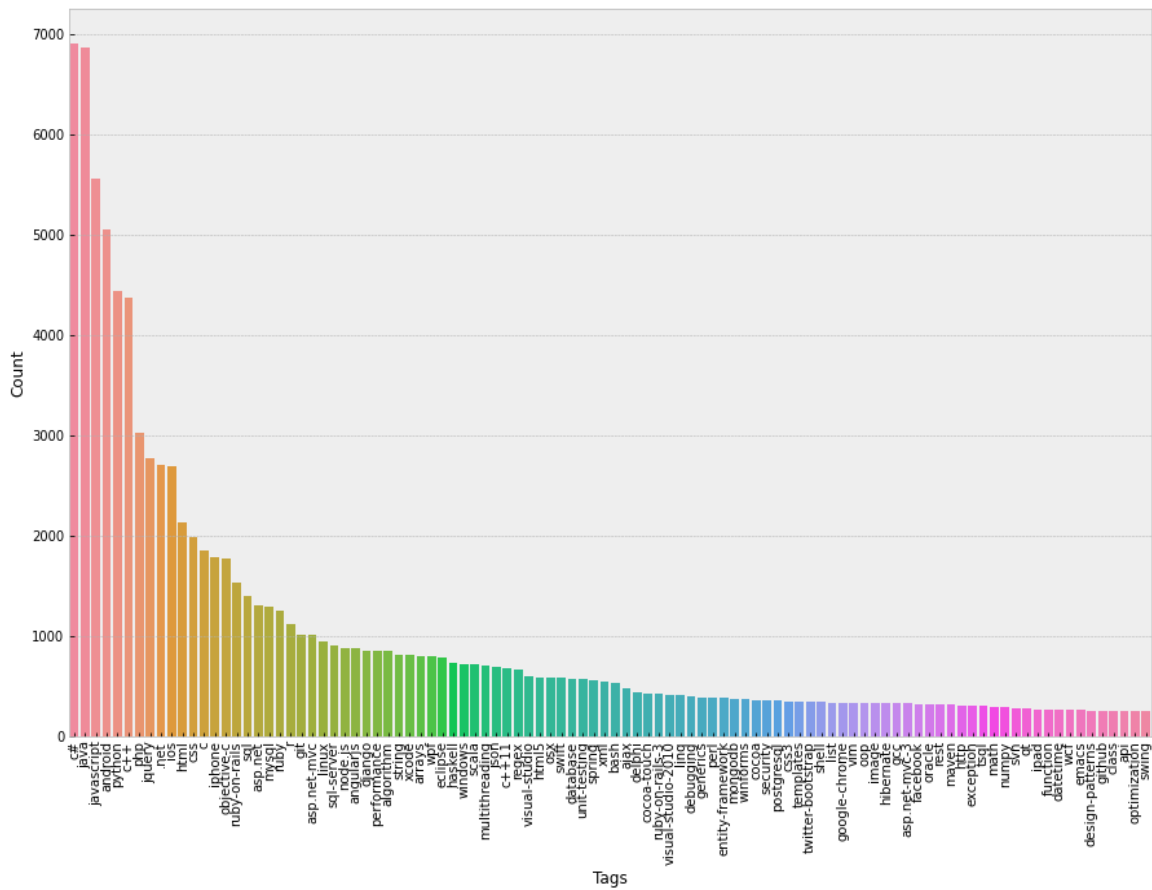
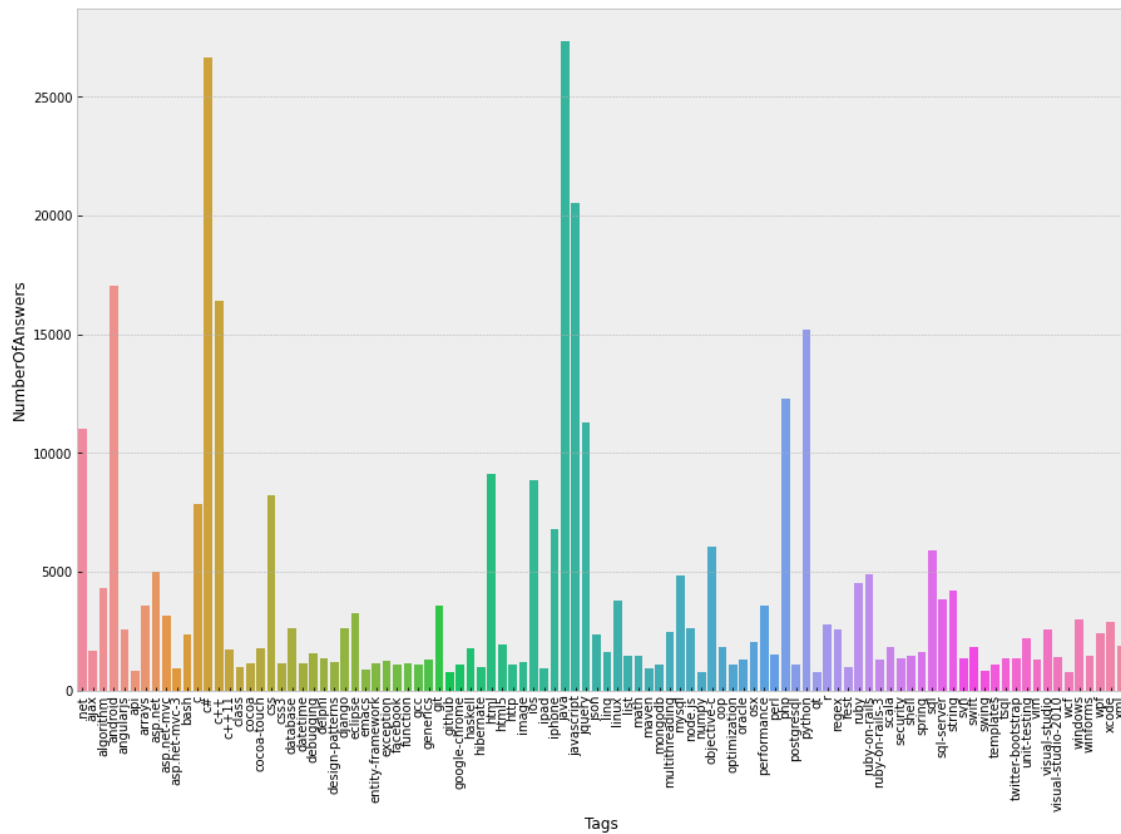


Figure 3 - Popularity of different tags

Based on figure 3, the questions with the top 50 most common tags were chosen for further analysis. The questions which did not have any of the top 50 tags were dropped from the dataset.



The number of answers for questions with each tag is shown in figure 4. It can be seen that programming languages like C, JavaScript and Python have a relatively greater number of answers than some other tags like “performance”, “database” etc.

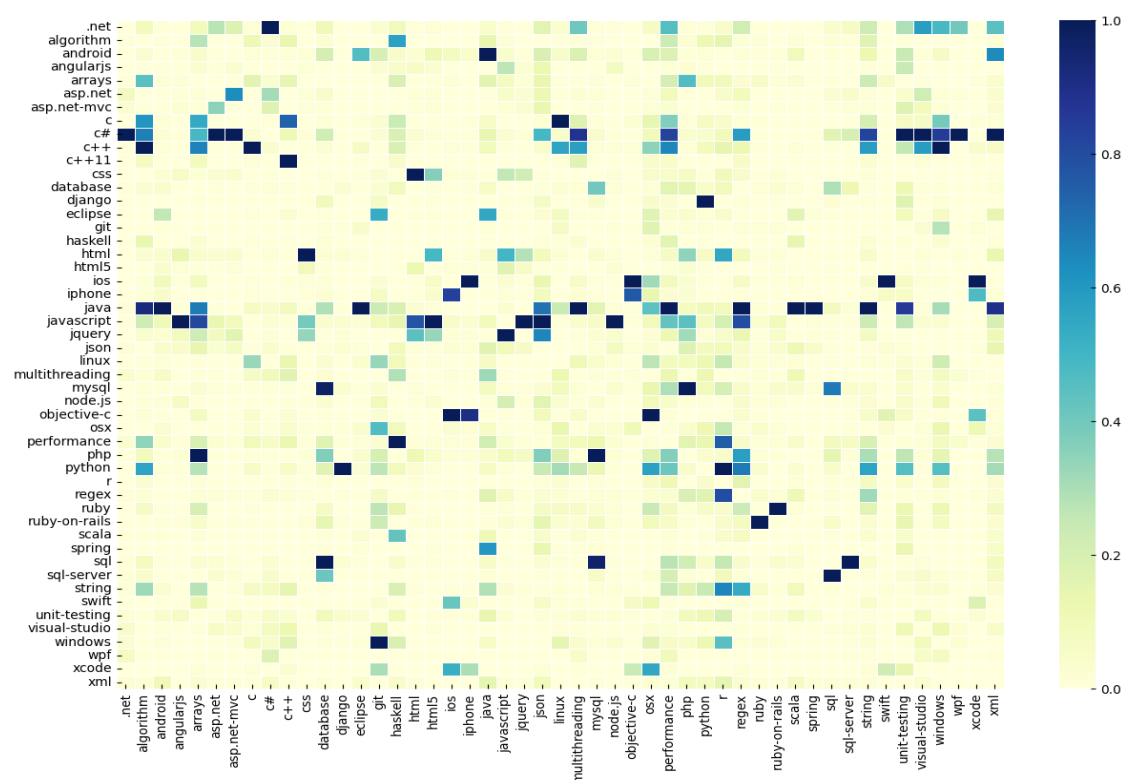


Figure 5 - Co-occurrence of tags

Co-occurrence of one tag with another is shown in figure 5. Here we can see how relatively often a question is tagged in a pairwise manner. For e.g., Java and Android seem to be paired quite often compared to others. This can be used to ensemble with the results of the deep learning model.

Data Cleaning - As part of data cleaning, the text portion of the dataset was pre-processed for further steps.

Since the dataset contains questions scraped from StackOverflow, the question title and body consisted of non ascii characters, html tags, email ids and URLs. These were removed using a pre-built pre-processing library. A sample of the dataset after this step is shown in figure 6.

	Tags	Question	Text
2	[sql, asp.net]	ASP.NET Site Maps <p>Has anyone got experience...	aspnet site maps has anyone got experience cre...
3	[algorithm]	Function for creating color wheels <p>This is ...	function for creating color wheels this is som...
4	[c#, .net]	Adding scripting functionality to .NET applica...	adding scripting functionality to net applicat...
5	[c++]	Should I use nested classes in this case? <p>I...	should i use nested classes in this case i am ...
6	[.net]	Homegrown consumption of web services <p>I've ...	homegrown consumption of web services i have b...

Figure 6 - Question tagging dataset after initial pre-processing

For the next steps of text pre-processing, the nltk library was used for lemmatization and stop word removal. Lemmatization in linguistics is the process of grouping together the inflected forms of a word so they can be analysed as a single item, identified by the word's lemma, or dictionary form.

Stopword are words such as “and”, “the”, “if” etc which do not add any significant meaning while tokenization. A sample of the dataset after this step is shown in figure 7.

	Tags	Text
0	[sql, asp.net]	aspnet site map ha anyone got experience creat...
1	[algorithm]	function creating color wheel something pseudo...
2	[c#, .net]	adding scripting functionality net application...
3	[c++]	use nested class case working collection class...
4	[.net]	homegrown consumption web service writing web ...

Figure 7 - Final pre-processed dataset for Question tagging

The 50 tags were represented using the binary format as show in figure 8

	Text	.net	algorithm	android	angularjs	arrays	asp.net	asp.net-mvc	c	c#
0	aspnet site map ha anyone got experience creat...	0	0	0	0	0	1	0	0	0
1	function creating color wheel something pseudo...	0	1	0	0	0	0	0	0	0
2	adding scripting functionality net application...	1	0	0	0	0	0	0	0	1
3	use nested class case working collection class...	0	0	0	0	0	0	0	0	0

Figure 8 - Questions with their binary representation of op 50 tags

Tokenization - For this step, the standard keras tokenizer was used with a threshold for maximum number of words as 5000 and maximum length of each question as 500. This is the last step in the data cleaning and pre-processing stage.

5.1.2 Model Building

This stage consists of building and training deep neural networks for the purpose of predicting tags for questions. The results obtained on testing will be described in the Results and Discussion Section.

3 major types of neural networks were built for this project:

Vanilla Artificial Neural Network: A simple Artificial Neural Network was built initially consisting of a single embedding layer followed by a single dense layer with a sigmoid activation function for the output. The loss function used is binary cross entropy with an Adam optimizer. The code for the model is shown in figure 9.

```
model = Sequential()
model.add(Embedding(max_words, 20, input_length=maxlen))
model.add(GlobalMaxPool1D())
model.add(Dense(num_classes, activation='sigmoid'))
model.compile(optimizer=Adam(0.015), loss='binary_crossentropy', metrics=[tf.keras.metrics.AUC(),tf.keras.metrics.Accuracy()],

callbacks = [
    ReduceLROnPlateau(),
    ModelCheckpoint(filepath='model-simple.h5', save_best_only=True)
]
```

Figure 9 - Question Tagging ANN

A batch size of 32 was used for training with 30 epochs.

Long Short-Term Memory Neural Network: A simple neural network with an embedding layer as input was followed by a single LSTM layer and then a single dense layer with sigmoid activation function as the output. The loss function used is binary cross entropy with an Adam optimizer. The code for the model is shown in figure 10.

```
deep_inputs = Input(shape=(maxlen,))
embedding_layer = Embedding(max_words, 100, weights=[embedding_matrix], trainable=False)(deep_inputs)
LSTM_Layer_1 = LSTM(128)(embedding_layer)
dense_layer_1 = Dense(50, activation='sigmoid')(LSTM_Layer_1)
model = Model(inputs=deep_inputs, outputs=dense_layer_1)

callbacks = [
    ReduceLROnPlateau(),
    ModelCheckpoint(filepath='model-conv1d.h5', save_best_only=True)
]

model.compile(loss='binary_crossentropy', optimizer='adam', metrics=[tf.keras.metrics.AUC(),tf.keras.metrics.Accuracy()])
```

Figure 10 - Question Tagging LSTM

A batch size of 32 was used for training with 30 epochs.

Convolutional Neural Network: A single embedding layer was used for consuming the input. This was followed by 3 pairs of convolutional and pooling layers. Each convolutional layer had different

dropout values to prevent overfitting with a Rectified Linear Unit activation function and a filter length of 300. This was followed by 2 dense layers of size 64 and 16 respectively with a Rectified Linear Unit activation function. The final dense layer was used as the output layer with a sigmoid activation function. The loss function used is binary cross entropy with an Adam optimizer. The code for the model is shown in figure 11.

```
model = Sequential()
model.add(Embedding(max_words, 20, input_length=maxlen))
model.add(Conv1D(filter_length, 3, padding='valid', activation='relu', strides=1))
model.add(MaxPooling1D())
model.add(Dropout(0.1))
model.add(Conv1D(filter_length, 3, padding='valid', activation='relu', strides=1))
model.add(MaxPooling1D())
model.add(Dropout(0.3))
model.add(Conv1D(filter_length, 3, padding='valid', activation='relu', strides=1))
model.add(GlobalMaxPool1D())
model.add(Dropout(0.2))
model.add(tf.keras.layers.Flatten())
model.add(tf.keras.layers.Dense(64, activation='relu'))
model.add(tf.keras.layers.Dense(16, activation='relu'))
model.add(Dense(num_classes))
model.add(Activation('sigmoid'))

model.compile(optimizer='adam', loss='binary_crossentropy', metrics=[tf.keras.metrics.AUC(), tf.keras.metrics.Accuracy()],
```

Figure 11 - Question Tagging CNN

A batch size of 32 was used for training with 30 epochs.

In an attempt to improve results and follow some of the state-of-the-art models, a static analysis method was ensembled with the current Convolutional Neural Network. The static analysis method uses a correlation matrix between the tags to predict which tags are related to other tags very closely and hence occur together very frequently. These highly correlated tags were combined with the originally predicted tags by the model and given as the final output. An example of the correlation matrix is shown in figure 11.

	.net	algorithm	android	angularjs	arrays	asp.net	asp.net-mvc	c	c#	c++
arrays	-0.016450	0.034403	-0.032067	-0.008214	1.000000	-0.015544	-0.013093	0.043364	-0.011371	0.018742
css	-0.040490	-0.022472	-0.052111	-0.006518	-0.019316	-0.016523	-0.015112	-0.033556	-0.063915	-0.052819
javascript	-0.067819	-0.029117	-0.086293	0.143470	0.019320	-0.018944	-0.029528	-0.057754	-0.110157	-0.089661
osx	-0.017008	-0.012036	-0.020807	-0.012259	-0.011661	-0.013860	-0.011836	-0.006148	-0.032161	-0.009295
algorithm	-0.020968	1.000000	-0.035373	-0.014778	0.034403	-0.018115	-0.014761	0.021851	-0.018131	0.013379
c++	-0.057800	0.013379	-0.083736	-0.034736	0.018742	-0.042578	-0.036793	0.126488	-0.088521	1.000000
string	0.000448	0.020070	-0.025342	-0.013314	0.035495	-0.015816	-0.013338	0.007119	0.019276	0.020916
c	-0.036679	0.021851	-0.051976	-0.022068	0.043364	-0.027050	-0.023692	1.000000	-0.061032	0.126488
git	-0.028414	-0.015939	-0.037741	-0.015153	-0.015442	-0.019899	-0.017429	-0.023050	-0.047834	-0.037463
c#	0.360423	-0.018131	-0.108872	-0.043419	-0.011371	0.140070	0.074691	-0.061032	1.000000	-0.088521

Figure 12 - Static analysis with a correlation matrix

This method of ensembling static analysis along with the model prediction improved the recall of the overall model.

5.2 Question Forwarding

As part of the end goal of the project, the objective of automating the tagging process is to forward these questions to the right demographic so that they can be answered accurately and in a timely manner. The question forwarding model uses the dataset to find out which users are the best choice to answer questions with particular tags. This is implemented using a heap to find the top-k users for the set of tags predicted by the question tagging model. It is implemented as shown in figure 13.

```

1  import heapq
2  class ExpertSearching:
3      def __init__(self,df,answers):
4          self.df = df
5          self.answers = answers
6          self.merged = None
7
8      def preprocess(self):
9          questions = self.df.copy()
10         questions.drop(columns = ["Title","Body"], inplace = True)
11         questions.rename(columns = {"Id" : "ParentId"},inplace = True)
12         self.merged = questions.merge(self.answers, on = 'ParentId')
13         self.merged.drop(columns = ["ParentId","Id","CreationDate","Body"],inplace = True)
14         self.merged = self.merged[self.merged["OwnerUserId"].notna()]
15         self.merged = self.merged.reset_index()
16
17     def createRatingMap(self):
18         ratingMap = {}
19         for index, row in self.merged.iterrows():
20             tags = row["Tags"]
21             id = int(row["OwnerUserId"])
22             score = row["Score"]
23             for tag in tags:
24                 if tag not in ratingMap:
25                     ratingMap[tag] = {}
26                 if id not in ratingMap[tag]:
27                     ratingMap[tag][id] = 0
28                 ratingMap[tag][id] += score
29         return ratingMap
30
31     def getTopK(self,tagsList,ratingMap,k):
32         outerHeap = []
33         topKList = []
34         for tag in tagsList:
35             innerHeap = []
36             innerMap = ratingMap[tag]
37             for key in innerMap:
38                 score = innerMap[key]
39                 heapq.heappush(innerHeap, (-score,key))
40             for _ in range(k):
41                 if innerHeap:
42                     heapq.heappush(outerHeap,heapq.heappop(innerHeap))
43         for _ in range(k):
44             if outerHeap:
45                 topKList.append(heapq.heappop(outerHeap))
46         return [element[1] for element in topKList]
47
48 obj = ExpertSearching(df,answers)
49 obj.preprocess()
50 ratingMap = obj.createRatingMap()
51 print(obj.getTopK(['python'],ratingMap,3))

```

Ln 52, Col 43 Spaces: 4 UTF-8

Figure 13 - Function to retrieve domain experts for a tag

5.3 Vulnerability Detection

5.3.1 Pre-processing and tokenizing of software vulnerability dataset

For vulnerability detection, the following pipeline is used. Firstly, for training the model itself, we used the dataset - <https://osf.io/d45bw/> which contains about a million code snippets in raw binary form, and a Boolean to show the existence of 4 vulnerabilities, i.e.,

- Out-of-bounds read and write
- NULL Pointer reference
- Buffer overflow and unsigned int wrap-around
- String size mismatch over input

in the hdf5 format. The example slice of dataset is as shown in figure 15.

	functionSource	CWE-119	CWE-120	CWE-469	CWE-476	CWE-other
0	b'default_event_handler(\n GuiWidget *widg...	False	False	False	False	False
1	b'krb5_krbhst_init_flags(krb5_context context,...	False	False	False	False	False
2	b'swap_info_get(swp_entry_t entry)\n\n\tstruc...	False	False	False	False	False
3	b'parseattrs4(char *&c, const Vec4 &ival = V...	False	False	False	False	False
4	b'generateExecCode(CompileState* comp)\n\n ...	False	False	False	False	False

Figure 14 - Software Vulnerability dataset

First thing to be done is to decode the binary strings to normal strings in UTF-8 format. After which we clean the given code to remove useless parts of the code like comments, variable names etc which take up a significant part of a human written real-life code, but doesn't matter for the existence of vulnerability at all. By using various counting and tokenization techniques, we found out that almost 40% of all tokens existing in our code are present for human validation rather than actual performance of the code in any form. We can clearly see that in figure 14 that some of these variables like "i", "n" etc and words like "the" come from either the user's name for the variable, or from comments (which exists in a significant part of the code as well). So firstly, we remove all these from the code itself, as follows.

```
[('if', 391391),
 ('0', 264234),
 ('return', 219332),
 ('i', 171609),
 ('1', 149455),
 ('int', 128049),
 ('null', 123197),
 ('the', 98336),
 ('t', 91601),
 ('n', 89007)]
```

Figure 15 - unimportant tokens present in code

Initially we remove all the single line comments using regex, and then we compare each token (word) with a list of all keywords in C/C++ main libraries, if

the token is not present in that, it will be replaced with the term “variable”, to remap all the variables to the same, after which we remove whitespaces (tabs, newlines etc) which do not affect the actual execution of the code in any form or factor since the language under consideration is only C/C++, finally we remove multiline comments as well, which currently will exist as a single line comment since the whitespaces are removed. The final pre-processed text is ready to be trained on. We can now see in the token count that remaining tokens are all relevant to the code.

```
Number of tokens: 83573
[('variable', 56632241),
 ('if', 2650444),
 ('return', 1564071),
 ('int', 886956),
 ('struct', 600907),
 ('char', 539741),
 ('const', 528651),
 ('else', 431035),
 ('case', 392080),
 ('for', 384375)]
```

Figure 16 - Relevant tokens remaining in the data

After all this pre-processing for text, we convert the Boolean fields into integers (0 or 1) since we are expecting output as a number, figure 17 shows the final usable data.

	functionSource	CWE-119	CWE-120	CWE-469	CWE-476	CWE-other
0	variable(int variable, int variable, int varia...	0	0	0	0	0
1	variable(variable* variable){variable* variabl...	0	0	0	0	0
2	variable(void){ if(variable) variable(variable...	0	0	0	0	0
3	variable(struct variable *variable){struct var...	0	0	0	0	0
4	variable(void){char *variable = variable("vari...	1	1	0	0	1

Figure 17 - Software Vulnerability pre-processed data

5.3.2 Training model to detect software vulnerability

For the training process, we had tried multiple models which were decided to be suitable for our use case after extensive literature review. Some of the models experimented by us are

- ANN
- LSTM
- CNN etc

The initial simple model is shown in figure 18.

```
11 model = Sequential()
12 model.add(Embedding(WORDS_SIZE, 20, input_length=INPUT_SIZE))
13 model.add(Dropout(0.5))
14 model.add(GlobalMaxPool1D())
15 model.add(Dense(5))
16 model.add(Activation('sigmoid'))
17
18 model.compile(optimizer='adam', loss='binary_crossentropy', metrics=[
19     tf.keras.metrics.AUC(),
20     tf.keras.metrics.Accuracy(),
21     tf.keras.metrics.Precision(),
22     tf.keras.metrics.Recall()
23 ])
24
25 callbacks = [
26     ReduceLROnPlateau(),
27     ModelCheckpoint(filepath='model-conv1d.h5', save_best_only=True)
28 ]
29
30 history = model.fit(x_train, y_train,
31                     epochs=30,
32                     batch_size=32,
33                     validation_split=0.3,
34                     callbacks=callbacks)
```

Figure 18 - SW Vuln. ANN model

The model suffered from being too simple for the large and complex data, it was found that since most of the data was not-vulnerable, the model was simply classifying everything to be False, but still getting very high accuracy because of said skewed data. The model was simply too weak for our use case. Checking the metrics showed us that the mode achieved as high as 90% accuracy, but less than 5% F1-score because of very high False - negatives (almost all) but very few True-Positives, and the usage of accuracy for multi-label classification is flawed as well, since the metric only considers correct results without considering the data and its skewness as well as the nature of multilabel being as many correct classifications as possible in the set of labels.

The LSTM Model is shown in figure 19:

```

1 main_input = Input(shape=(MAXLEN,), dtype='int32', name='main_input')
2 x = Embedding(vocabulary_size, 50, input_length=MAXLEN)(main_input)
3 x = Dropout(0.3)(x)
4 x = Conv1D(64, 5, activation='relu')(x)
5 x = MaxPooling1D(pool_size=4)(x)
6 x = LSTM(100)(x)
7 x = Dropout(0.3)(x)

```

Figure 19 - SW Vuln. LSTM model

With summary shown in figure 20 we can clearly see that this had a main input layer and only one conv layer with various dropouts at each layer to avoid overfitting

1 model.summary()

Model: "model"

Layer (type)	Output Shape	Param #	Connected to
main_input (InputLayer)	(None, 50)	0	[]
embedding_1 (Embedding)	(None, 50, 50)	1000000	['main_input[0][0]']
dropout_2 (Dropout)	(None, 50, 50)	0	['embedding_1[0][0]']
conv1d_1 (Conv1D)	(None, 46, 64)	16064	['dropout_2[0][0]']
max_pooling1d_1 (MaxPooling1D)	(None, 11, 64)	0	['conv1d_1[0][0]']
lstm_1 (LSTM)	(None, 100)	66000	['max_pooling1d_1[0][0]']
dropout_3 (Dropout)	(None, 100)	0	['lstm_1[0][0]']
binary_output_CWE-119 (Dense)	(None, 1)	101	['dropout_3[0][0]']
binary_output_CWE-120 (Dense)	(None, 1)	101	['dropout_3[0][0]']
binary_output_CWE-469 (Dense)	(None, 1)	101	['dropout_3[0][0]']
binary_output_CWE-476 (Dense)	(None, 1)	101	['dropout_3[0][0]']
binary_output_CWE-other (Dense)	(None, 1)	101	['dropout_3[0][0]']

=====
Total params: 1,082,569
Trainable params: 1,082,569
Non-trainable params: 0
=====

Figure 20 - Summary of the LSTM model

This was run with 15 epochs with the same data. But while testing the data it was realized that the model was simply too weak to be used. It was found that since most of the data was not-vulnerable, the model was simply classifying everything to be False, but still getting very high accuracy because of said skewed data.

The last model built was a CNN model and after comparison between the same, it was found that CNN performed best for our use case.

Among these models as well, the results saw a considerable increase while using CNN relative to other models.

MathWorks defines CNN as “A convolutional neural network (CNN or ConvNet), is a network architecture for deep learning which learns directly from data, eliminating the need for manual feature extraction.”. This was perfect for our case as well. The architecture of the final CNN model used is as follows:

- The input layer which takes in the tokenized strings of code snippets
- 2 Convolution layers on 1 Dimensions with various dropout rate
- 3 Hidden dense layers.
- All the above layers use “ReLU” as the activation function.
- The final output layer, which gives 5 floating point numbers as output which determine the probability that, whether the model predicts that particular vulnerability to exist in the code snippet or not

```

10
11 model = Sequential()
12 model.add(Embedding(max_words, 20, input_length=maxlen))
13 model.add(Conv1D(filter_length, 3, padding='valid', activation='relu'))
14 model.add(tf.keras.layers.MaxPool1D(pool_size=5))
15 model.add(tf.keras.layers.Dropout(0.2))
16 model.add(Conv1D(filter_length, 3, padding='valid', activation='relu'))
17 model.add(tf.keras.layers.MaxPool1D(pool_size=5))
18 model.add(tf.keras.layers.Dropout(0.4))
19 model.add(tf.keras.layers.Flatten())
20 model.add(tf.keras.layers.Dense(64, activation='relu'))
21 model.add(tf.keras.layers.Dense(32, activation='relu'))
22 model.add(tf.keras.layers.Dense(16, activation='relu'))
23 model.add(Dense(num_classes))
24 model.add(Activation('sigmoid'))
25
26 model.compile([
27     optimizer='adam',
28     loss='binary_crossentropy',
29     metrics=[
30         tf.keras.metrics.AUC(),
31         tf.keras.metrics.Accuracy(),
32         tf.keras.metrics.Precision(),
33         tf.keras.metrics.Recall()
34     ]
35 ])

```

Figure 21 - Highly optimized CNN model for detecting SW vulnerabilities

We have used Adam Optimizer for its benefits like fast training, simple implementation and verifiable results.

The model was trained over 30 epochs, with a 70:30 split between training and testing and using 20% for validation during training itself.

The results of the model are shown in figure 22:

```

3231/3231 [=====]
auc_1: 0.9101868271827698
accuracy: 0.012662732973694801
precision_1: 0.7843882441520691
recall_1: 0.5439600944519043

```

Figure 22 - Results of the CNN model for SW vulnerability

Here we can clearly see that our model predicts with a precision of 78% which is good considering that this is a multi-label classification, and the metric of calculating is inherently flawed in some

perspectives, especially when the dataset is skewed towards one label more than the other and both are independent factors.

The usage methodology are as follows, firstly we have a ready API which can take up any StackOverflow question in the similar format of text part and code, in the form of a file which can be uploaded by the user, where code part is considered only if it's within the scope of the project (C/C++ code), in which cases, the data will be parsed and pre-processed in the similar to the pre-processing done for the dataset. The API follows a schema as shown in figure 23:

```
@app.post("/main")
def main(question: str = Body(example_input, media_type='text/plain'), file: Union[UploadFile, None] = None) -> Response:
    if not file:
        code = None
        return {"message": "No upload file sent"}
    else:
        code = "\n".join(file.file.read().decode("utf-8").split("\n"))
        return call(question, code)
```

Figure 23 - API Schema

The function being called takes in question and code as strings (which has been read from file if provided). The SVD detection class firstly will be initialized with required models, libraries etc at start time itself as shown in figure 24

```
class PreprocessSVD:
    def __init__(self):
        save_option = tf.saved_model.LoadOptions(experimental_io_device='/job:localhost')

        self.tokenizer = pickle.load(open("./vuln_tokenizer.sav", 'rb'))
        self.cnn_model = load_model("./vuln_model", options=save_option)
        self.keywords = set(open("./keywords.csv", "r").read().split("\n"))
```

Figure 24 - Initialization of SVD model

The function which will be detecting actual vulnerability is implemented as shown in figure 25.

```
def getVuln(self, code_part, threshold):
    l = []
    # since clean_code accepts df
    code_part = pd.DataFrame([code_part])
    code_part = self.clean_code(code_part)[0].iloc[0]
    print(code_part)
    l.append(code_part)
    self.tokenizer.fit_on_texts(l)
    sequences = self.tokenizer.texts_to_sequences(l)
    x_inp = pad_sequences(sequences, maxlen=500)
    pred = self.cnn_model.predict(x_inp)
    for i in range(5):
        if pred[0][i] < threshold:
            pred[0][i] = 0
        else:
            pred[0][i] = 1
    return pred.tolist()
```

Figure 25 - Function to detect vulnerability

The cleaned code will be sent into the loaded CNN Model which will predict the probability of any given vulnerability among the code which will be converted to a list of 5 Boolean numbers, which will be casted to actual vulnerability names and returned back to the user the API. The API can be used like any other REST API with no limit with any sort of client issues

6. RESULTS AND DISCUSSIONS

6.1 Question Tagging

Since question tagging is a multi-class classification problem, the major metrics used for assessing the performance of the question tagging model is precision, recall and F1 score.

A comparison of the various models built during the course of this project is shown below. It can be seen that the Convolutional Neural Network performs better in this case when compared to an Artificial Neural Network and a Long Short Term Memory Neural Network.

Model	Recall	Precision
ANN	0.51	0.72
CNN	0.61	0.73
LSTM	0.49	0.75
Enhanced CNN	0.78	0.81

Figure 26 - Results of different models trained to detect question tags

A comparison of our best model with the state-of-the-art results is shown below.

Paper	Recall@5	Precision@5
SOTagRec	0.817	0.343
TagCombine	0.595	0.221

TagMulRec	0.680	0.284
EnTagRec	0.805	0.346
Enhanced CNN (our model)	0.844	0.239

Figure 27 - Comparison of our best model with the state-of-the-art results

The metric used in this case for the purpose of comparison is Precision @5 and Recall@5.

The code snippet used to test our model after training is shown in figure 28.

```
# cnn_model = model
metrics = cnn_model.evaluate(X_test, y_test)
print("{}: {}".format(cnn_model.metrics_names[1], metrics[1]))
print("{}: {}".format(cnn_model.metrics_names[2], metrics[2]))
print("{}: {}".format(cnn_model.metrics_names[3], metrics[3]))
print("{}: {}".format(cnn_model.metrics_names[4], metrics[4]))
```

```
552/552 [=====] - 75s 134ms/step - loss: 0.0435 - auc_
auc_1: 0.9685834646224976
accuracy: 0.003608877770602703
precision_1: 0.8169436454772949
recall_1: 0.7830877900123596
```

Figure 28 - Evaluation metrics for models

Some insights and conclusions can also be made from the precision and recall of each tag for figure 29.

```
print(classification_report(y_true, y_pred, target_names=label_names))
```

	precision	recall	f1-score	support
.net	0.69	0.50	0.58	789
algorithm	0.71	0.75	0.73	245
android	0.96	0.92	0.94	1484
angularjs	0.97	0.91	0.94	275
arrays	0.71	0.69	0.70	223
asp.net	0.82	0.66	0.73	391
asp.net-mvc	0.87	0.77	0.82	303
c	0.77	0.72	0.75	568
c#	0.81	0.76	0.78	2053
c++	0.85	0.80	0.82	1288
c++11	0.73	0.69	0.71	173
css	0.86	0.86	0.86	604
database	0.68	0.39	0.49	187
django	0.95	0.90	0.93	231
eclipse	0.88	0.87	0.87	242
git	0.95	0.93	0.94	276
haskell	0.90	0.84	0.87	209
html	0.75	0.65	0.70	651
html5	0.85	0.61	0.71	190
ios	0.79	0.74	0.76	824
iphone	0.76	0.64	0.70	529
java	0.88	0.82	0.85	2009
javascript	0.83	0.74	0.79	1720
jquery	0.89	0.81	0.85	861
json	0.82	0.75	0.78	203
linux	0.80	0.69	0.74	292
multithreading	0.77	0.76	0.76	226
mysql	0.89	0.83	0.86	388
node.js	0.90	0.82	0.85	277
objective-c	0.75	0.63	0.69	556
osx	0.84	0.64	0.73	187
performance	0.77	0.35	0.48	241
php	0.94	0.82	0.88	935
python	0.93	0.88	0.90	1339
r	0.92	0.89	0.91	297
regex	0.89	0.82	0.85	195
ruby	0.84	0.75	0.79	383
ruby-on-rails	0.92	0.86	0.89	485
scala	0.95	0.90	0.93	235
spring	0.93	0.88	0.90	155
sql	0.75	0.69	0.72	427
sql-server	0.79	0.81	0.80	272
string	0.66	0.51	0.58	255
swift	0.92	0.82	0.87	163
unit-testing	0.75	0.75	0.75	161
visual-studio	0.76	0.58	0.66	171
windows	0.73	0.45	0.56	231
wpf	0.96	0.83	0.89	223
xcode	0.79	0.67	0.72	247
xml	0.83	0.64	0.72	178

Figure 29 - Precision and Recall of each tag

It can be seen that common programming languages like java, c++, python, AngularJS, Scala etc have high f1 scores because of clear distinguishing features and their high support. On the other hand, tags like “database”, “performance” and “windows” have relatively lower f1 scores due to lesser distinguishing features and support.

An example of a question picked up from StackOverflow and passed to our model is shown in figure 30.

```
[45] tags = getTags("How do I read / convert an InputStream into a String in Java?: \
If you have a java.io.InputStream object,\
how should you process that object and produce a String? Suppose I have an InputStream \
that contains text data, and I want to convert it to a String, \
so for example I can write that to a log file. What is the easiest way to take the \
InputStream and convert it to a String?", 0.6)
print(tags)

1/1 [=====] - 0s 83ms/step
[('java', 'string')]
```

Figure 30 - Example of question tagging model

6.2 Vulnerability Detection

Model	Recall	F1
ANN	0.51	0.61
LSTM	0.00	1.00
CNN	0.55	0.78

Figure 31 - Results for SVD models

A comparison of our best model with the state-of-the-art results is shown below.

Paper	Recall	F1
Automated Vulnerability Detection in Source Code Using Deep Representation Learning		0.56

Clang (static test)	< 0.50	
CNN (our model)	0.55	0.78

Figure 32 - comparison of State-of-the-art models with our best SVD model

Since Vulnerability Detection is a multi-class classification problem, the major metrics used for assessing the performance of the vulnerability model is precision, recall and F1 score.

```

2 cnn_model = tf.keras.models.load_model('/content/gdrive/My Drive/vuln_model')
3 metrics = cnn_model.evaluate(X_test, y_test)
4 print("{}: {}".format(cnn_model.metrics_names[1], metrics[1]))
5 print("{}: {}".format(cnn_model.metrics_names[2], metrics[2]))
6 print("{}: {}".format(cnn_model.metrics_names[3], metrics[3]))
7 print("{}: {}".format(cnn_model.metrics_names[4], metrics[4]))
8 #print("{}: {}".format(cnn_model.metrics_names[5], metrics[5]))

3231/3231 [=====] - 24s 7ms/step - loss: 0.2137 - auc_1: 0.9102 - accuracy: 0.0127 - precision_1: 0.7844 - recall_1: 0.5440
auc_1: 0.9101868271827698
accuracy: 0.012662732973694801
precision_1: 0.7843882441520691
recall_1: 0.5439600944519043

```

Figure 33 - Evaluation metrics for SVD model

Using the test dataset, the derived metrics are as follows, this clearly shows the performance detection of individual vulnerability:

	precision	recall	f1-score	support
119	0.63	0.67	0.65	8584
120	0.61	0.64	0.63	15699
469	0.93	0.98	0.95	6926
476	0.76	0.57	0.65	9409
other	0.60	0.48	0.53	12120
micro avg	0.68	0.64	0.66	52738
macro avg	0.71	0.67	0.68	52738
weighted avg	0.68	0.64	0.66	52738
samples avg	0.20	0.20	0.19	52738

Figure 34 - Derived metrics of test dataset

The first column refers to the vulnerabilities in the first part and the average metric of all of those in the second part. Even in testing data we can clearly see that the f1-score is almost always above 60% for any of the vulnerabilities, except “other” which is a very broad category with little data available in the dataset, causing for some predictability.

7. PLAN OF WORK FOR CAPSTONE PROJECT PHASE-3

7.1 Deliverable

After the implementation of the project in phase-2, the next part is to create a document which specifies and explains our project from scratch to the logic / idea behind the models. Only SCOPUS / Web of Science indexed technical conference/ Journal paper or Patent application is considered for evaluation. We create the deliverable which follows the guidelines required by the above-mentioned ways to present our project paper. The paper should be devoid of any plagiarised contents.

This paper also specifies on how the project we made is better than the current existing ones and made better than the ones used as references. The drawbacks of other papers implemented in our project are specified explicitly.

7.2 Expected Result

We expect our deliverable to be accepted / presented by the time of announcement of 8th Semester's Results.

REFERENCES/BIBLIOGRAPHY

- [1] R. Russell et al., "Automated Vulnerability Detection in Source Code Using Deep Representation Learning," 2018 17th IEEE International Conference on Machine Learning and Applications (ICMLA), 2018, pp. 757-762, doi: 10.1109/ICMLA.2018.00120.
- [2] J. R. Cedeño González, J. J. Flores Romero, M. G. Guerrero and F. Calderón, "Multi-class multi-tag classifier system for StackOverflow questions," 2015 IEEE International Autumn Meeting on Power, Electronics and Computing (ROPEC), 2015, pp. 1-6, doi: 10.1109/ROPEC.2015.7395121.
- [3] L. Ruggahakotuwa, L. Rupasinghe and P. Abeygunawardana, "Code Vulnerability Identification and Code Improvement using Advanced Machine Learning," 2019 International Conference on Advancements in Computing (ICAC), 2019, pp. 186-191, doi: 10.1109/ICAC49085.2019.9103400.
- [4] K. Nishizono, S. Morisaki, R. Vivanco, and K. Matsumoto, "Source code comprehension strategies and metrics to predict comprehension effort in software maintenance and evolution tasks," an empirical study with industry practitioners, in 2011 27th IEEE International Conference on Software Maintenance (ICSM), 2011, pp.473481.
- [5] Q. Zoubi, I. Alsmadi, and B. Abul-Huda, "Study the impact of improving source code on software metrics", in 2012 International Conference on Computer, Information and Telecommunication Systems (CITS), 2012
- [6] J. Dietrich, M. Luczak-Roesch and E. Dalefield, "Man vs Machine – A Study into Language Identification of Stack Overflow Code Snippets," 2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR), 2019, pp. 205-209, doi: 10.1109/MSR.2019.00041.
- [7] T. Saini and S. Tripathi, "Predicting tags for stack overflow questions using different classifiers," 2018 4th International Conference on Recent Advances in Information Technology (RAIT), 2018, pp. 1-5, doi: 10.1109/RAIT.2018.8389059.
- [8] X. Zhang, L. Shao and J. Zheng, "A Novel Method of Software Vulnerability Detection based on Fuzzing Technique," 2008 International Conference on Apperceiving Computing and Intelligence Analysis, 2008, pp. 270-273, doi: 10.1109/ICACIA.2008.4770021.
- [9] V. Jain and J. Lodhavia, "Automatic Question Tagging using k-Nearest Neighbors and Random Forest," 2020 International Conference on Intelligent Systems and Computer Vision (ISCV), 2020, pp. 1-4, doi: 10.1109/ISCV49265.2020.9204309.
- [10] T. Wang, T. Wei, G. Gu and W. Zou, "TaintScope: A Checksum-Aware Directed Fuzzing Tool for Automatic Software Vulnerability Detection," 2010 IEEE Symposium on Security and Privacy, 2010, pp. 497-512, doi: 10.1109/SP.2010.37.
- [11] Boris Chernis and Rakesh Verma. 2018. Machine Learning Methods for Software Vulnerability Detection. In Proceedings of the Fourth ACM International Workshop on Security and Privacy Analytics (IWSPA '18). Association for Computing Machinery, New York, NY, USA, 31–39.

APPENDIX: DEFINITIONS, ACRONYMS, AND ABBREVIATIONS

- CNN - Convolutional Neural Network
 - RNN - Recurrent Neural Network
 - TF-IDF - Term Frequency Inverse Document Frequency
 - NLP - Natural Language Processing
 - SVM - Support Vector Machine
 - SQL - Structured Query Language
 - LTS - Long Term Support
 - ER Diagram - Entity Relationship Diagram
 - LSTM – Long Short-Term Memory
 - Vuln. – Vulnerability
 - SVD – Software Vulnerability Detection
- ⇒ Questionnaire Forums - a medium platform where ideas and views on a particular issue can be exchanged.
- ⇒ Software Vulnerabilities - A software vulnerability is a defect in software that could allow an attacker to gain control of a system. These defects can be because of the way the software is designed, or because of a flaw in the way that it's coded.