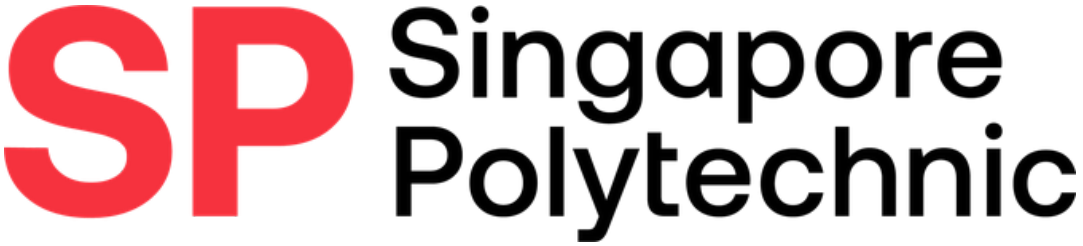
SINGAPORE POLYTECHNIC
SCHOOL OF COMPUTING



PROJECT NO. DISM/2023/3A67

APPLYING LARGE LANGUAGE MODELS TO SOURCE CODE BUG-FINDING

AUGUST 2023

PROJECT NO. DISM/2023/3A67

APPLYING LARGE LANGUAGE MODELS TO SOURCE CODE BUG-FINDING

SUBMITTED BY:

2104047	Isaac Choong Zhu En
2104162	Tay Kai Zer
2128524	Aldrich Tan Kai Rong
2128748	Edison Chan Whye Kit
2128834	Ryan Sng

A REPORT SUBMITTED IN PARTIAL FULFILMENT
OF THE SUBJECT ST2601 (ITSP)

Project Supervisor: Mr Calvin Siak

SCHOOL OF COMPUTING
SINGAPORE POLYTECHNIC
500 DOVER ROAD
SINGAPORE 139651

Acknowledgements

The team would like to thank our external sponsor, Defence Science Organisation (DSO) National Laboratories, for providing us with a meaningful project to work on. It was an enriching and rewarding experience for the team, allowing us to hone our skills in the field of Artificial Intelligence and Cyber Security. Additionally, the team would like to offer a special acknowledgement to Dr Khoo Wei Ming and Ms Poh Hui-Li Phyllis for dedicating their time to arrange bi-weekly meetings to catch up with the team's progress. Moreover, their deep knowledge and industry insights have helped aid the team's progress tremendously.

Moreover, the team would like to extend our sincere gratitude to our internal supervisor, Mr Calvin Siak for his guidance and mentorship. His invaluable feedback and expertise in Infosec Project Development and Management Project (ITSP) have propelled our team forward, enabling us to overcome numerous obstacles on our journey to the end of ITSP. Mr Siak's commitment to the team has presented numerous priceless learning opportunities for growth and development.

Lastly, the team would like to express our gratitude to Mr Mohamed Uvaise, the Module Coordinator of ITSP for ensuring a seamless and well-coordinated journey throughout ITSP. His prompt replies to emails ensured that the team's inquiries were swiftly addressed and resolved. Mr Mohamed Uvaise's project management resources have been instrumental in fostering excellent coordination in the team's project management efforts.

Once again, the team wishes to express our utmost gratitude to all individuals who have played a crucial role in our ITSP. All contributions have been vital to our progress, and the team is thankful for the guidance and support throughout the project. The team believes that the experiences and knowledge imparted will be invaluable.

Abstract

Purpose	:	The purpose of the project is to research the effectiveness of Large Language Models in supplementing deep learning-based Vulnerability Detection in source code.
Brief Description of Project	:	It aims to study the performance of Large Language Models for curating datasets of vulnerable functions by classifying vulnerability-fixing commits. It assesses Large Language Models trained on these curated datasets for Vulnerability Detection. This provides empirical evidence to demonstrate the effectiveness of Large Language Models.
Conclusion/ Recommendations	:	Through qualitative analysis, the team showed that Large Language Models can automatically curate accurate, diverse, and large datasets of vulnerable functions. The work in this paper can be used to streamline Vulnerability Detection research efforts and significantly reduce time spent on laborious labelling tasks.

Table of Contents

List of Abbreviations

Abbreviation	Definition
CNN	Convolutional Neural Network
CVE	Common Vulnerabilities and Exposure
DL	Deep Learning
GNN	Graph Neural Network
LLM	Large Language Model
MLM	Masked Language Modelling
NLP	Natural Language Processing
NVD	National Vulnerability Database
RNN	Recurrent Neural Networks
SOTA	state of the art
VD	Vulnerability Detection
VFC	Vulnerability-Fixing Commit

1 Introduction

As more corporations integrate digital technology into their business processes, their reliance on robust and secure software solutions becomes paramount. Tech giants such as Intel, Google, and Facebook employ their software engineers and security specialists to improve the security of not only their products but also open-source software, such as the Linux kernel [1]. Nevertheless, secure code reviews are prone to human error which can lead to drastic consequences.

The process of discovering security issues in software largely falls under static or dynamic analysis. For the former, traditional approaches involve manually combing through source code or using automated tools such as Cppcheck and Joern to identify vulnerabilities. However, these methods incur large amounts of man-hours, have high rates of false positives and negatives, or require advanced expertise. As such, many security researchers [2] - [4] have explored the use of **D**eep **L**earning (DL) techniques for automated source code **V**ulnerability **D**etection (VD).

According to Ahmed et al. [5], an obstacle in VD with DL is the difficulty of curating quality training and evaluation data or the lack thereof. Quality implies diversity and volume which directly influences the performance of DL models. However, curating such datasets requires advanced security knowledge, which even then is still susceptible to human error.

Moreover, a significant amount of time has to be invested. For instance, Zhou et al. [2] invested over 600 man-hours of several professional security experts to label 48,000 commits. Furthermore, in the realm of security, the freshness of datasets is crucial to stay relevant to current software vulnerability trends; one-off manually labelled datasets become outdated as new vulnerabilities are discovered.

Therefore, like the need for automated source code VD, there is an equal need for automated dataset curation of vulnerable source code.

1.1 Objectives

The purpose of this project is to research the application of **L**arge **L**anguage **M**odels (LLMs) for supplementing DL-based VD with source code. In particular, the team intends to explore the effectiveness of LLMs in classifying Git commits for dataset curation. Then, the team will enrich established ground-truth datasets by adding commits classified by these LLMs. Afterwards, the team will assess the VD performance of LLMs on these original and enriched datasets to demonstrate the effectiveness of LLMs.

1.2 Scope

The project will focus on C/C++-related source code and commits as these programming languages account for more than 50% of reported open-source

vulnerabilities [6]. Additionally, the team will mainly be extracting data from GitHub repositories as it is the largest Git repository provider. GitHub repositories that are specifically chosen by the team throughout the report are large repositories with well established contributor guidelines, which ensures that commits within these repositories are high quality.

1.3 Contributions

This paper makes the following contributions:

- The team releases a dataset of 3,500 **V**ulnerability-**F**ixing **C**ommits (VFCs), curated from 10 distinct C/C++ repositories on GitHub that took 40 man-hours of manual labelling. The team enriched this dataset by amassing 4 prominent VFC datasets: BigVul, Devign, CVEfixes, and Linux Kernel CVEs, totalling 36,625 commits. The team's compiled dataset is publicly available to help further research.[1]
- The team experimented with two **s**tate **o**f **t**he **a**rt (SOTA) LLMs for classifying VFCs: DistilBERT and StarEncoder. The best of which, StarEncoder, achieved an F1 score of 97.77% and 90.00%, and an accuracy of 97.70% and 89.79% on evaluation and test datasets. On average, StarEncoder scored 21.59% and 26.43% higher in accuracy and F1 when compared to other VFC classification models and methods.
- The team incorporated the fine-tuned StarEncoder model into a VFC classification tool, along with function extraction capabilities. This tool can be used to curate large datasets of vulnerable functions to aid researchers in DL-based VD.[2]
- The team applied the VFC classifier tool to double the size of Devign, a ground-truth dataset of vulnerable functions. This tool took 2 hours to classify 639,278 commits and extract 503,320 functions while the authors of Devign took 600 man-hours to manually label 25,872 commits and extract 27,318 functions. The team then trained StarEncoder on this enlarged dataset for VD, and observed an increase in the F1 and accuracy scores by 13% and 12% compared to the same model trained on the original dataset.
- The team's experiments show that by balancing vulnerable function classes, deduplicating functions by their label, and using suitable SOTA Encoders, the team managed to train a model that achieved an F1 score of 89%, which on average is 48% higher than existing models and VD tools.

[1] https://huggingface.co/datasets/neuralsentry/bigvul_devign_cvefixes_neuralsentry_commits

[2] https://github.com/neuralsentry/vulnfix-commit-llm-classifier

2 Background

2.1 Project Background

This project was sponsored by Defense Science Organisation (DSO) National Laboratories under the guidance of Mr Khoo Wei Ming and Ms Poh Hui-Li Phyllis.

The original objective of the project aimed to use OpenAI's GPT-3 to detect bugs in source code. However, after additional research and consideration, the team and sponsors decided to broaden the scope of the project. The team pivoted towards training and evaluating open-source LLMs for DL-based VD. Additionally, the team deployed the same LLMs to curate datasets of vulnerable source code to supplement these VD objectives.

The shift in objectives emerged during the first two weeks of project planning with the team's sponsors (Refer to Appendix C). The team examined the constraints of closed-source LLMs such as OpenAI's GPT-3, which included financial costs, inability to tweak model parameters, and lack of support for fine-tuning and transfer learning. This led to the decision of exploring open-source LLMs. Furthermore, the team identified that utilising open-source LLMs necessitated training datasets. However, suitable datasets were either scarce or, if publicly available, often had inherent limitations. It was the team's sponsors who suggested the idea of exploring DL-based **N**atural **L**anguage **P**rocessing (NLP) techniques using LLMs to classify Git commit messages. The rationale behind this was the potential of this approach to yield higher quality and quantity datasets compared to current methods.

2.2 Related Work

2.2.1 Deep Learning-based Vulnerability Detection

Convolutional and Recurrent Neural Networks

Convolutional **N**eural **N**etworks (CNNs) and **R**ecurrent **N**eural **N**etworks (RNNs) are widely adopted for DL. Although these models excel at computer vision and audio, the same can not be said for NLP.

The first issue is that CNNs and RNNs are not capable of handling long-range dependencies. Simply put, these neural networks have trouble understanding natural language.

The second issue is that these neural networks take longer to train because of their sequential processing nature. When Russel et al. [7] evaluated for VD, RNNs and CNNs achieved an F1 score of 53% and 54% respectively using real-world source code from GitHub.

Large Language Models

In recent works, the Transformer architecture pioneered by Vaswani et al. [8] has gained prominence in NLP, outperforming CNNs and RNNs. Transformers allow for increased training speed from parallelisation and better contextualisation and understanding of textual data through the self-attention mechanism. For these reasons, Transformers can be trained on a very large corpora of data, resulting in the emergence of LLMs. For instance, OpenAI's GPT-4, Meta's LLaMA, and Google's Bard, all utilise the Transformer architecture and are trained on dozens of gigabytes of data.

To the best of the team's knowledge, as of current, *DiverseVul*'s papers [9] has presented the most recent and comprehensive evaluation of LLMs for VD. The paper conducted a qualitative analysis of several LLM transformer-based architectures: Encoder, Decoder, and Encoder-decoder. Additionally, the researchers curated their own dataset to establish a baseline for comparing different model architectures.

Interestingly, however, their results showed that encoder-decoder models, CodeT5 and NatGen, outperformed a pure encoder model, CodeBERT, by scoring 48% and 43% respectively. These results are surprising as encoder models are typically more performant at text classification tasks while encoder-decoder models are usually associated with input-output-dependent tasks such as language translation. For decoder models from the GPT-2 family, they scored an F1 score of 40% which was expected as decoders were less effective at text classification compared to text generation.

None of the Encoder models used were pre-trained on C/C++ whereas CodeT5 and NatGen were. In the paper, they mentioned that pre-training on C/C++ has little impact on VD performance as the GPT models they evaluated which were pre-trained on C/C++ performed no better than Encoder models. However, comparing the impact of pre-training between GPT Decoder models and Encoder models may not be so straightforward.

Graph Neural Networks

Graph **N**eural **N**etworks (GNNs), despite being introduced in 2005, are also an emerging DL architecture for VD. This rise in popularity can be attributed to the fact that source code can be represented as graphs. A well-known graph representation is the **C**ode **P**roperty **G**raph (CPG) which is an amalgamation of the **A**bstract **S**yntax **T**ree (AST), **C**ontrol **F**low **G**raph (CFG), and **P**rogram **D**ependence **G**raph (PDG). However, *DiverseVul* [9] found that LLMs still outperform current GNNs for VD, achieving an F1 score of 49% and 29% respectively. As such, the focus of this paper will remain on LLMs, though these scores could change as GNNs continue to be researched and refined.

2.2.2 Dataset Curation for Vulnerability Detection

The *DiverseVul* paper conducted extensive research on VD using multiple DL models.

It identified the essential attributes of a quality dataset for VD: **diversity** and **volume**. By expanding the quantity of vulnerable and non-vulnerable functions (i.e., volume) from 168,089 to 552,783, and the number of source code repositories (i.e., diversity) from 564 to 945, the F1 score of an LLM saw an increase from 10.5% to 48.9%.

Another important characteristic is the **organicness** of a dataset. Numerous papers [2], [9], [10] highlighted that synthetically generated datasets do not portray real-world source code, thereby hindering VD performance.

Moreover, the **accuracy** of a dataset is without doubt important as inaccurately labelled data can lead to false positives and negatives.

Last but not least, the **freshness** of a dataset is crucial, especially in the context of security where vulnerability trends are constantly changing. As such, manually labelled datasets become outdated over time as they require a lot of effort to be updated regularly. Therefore, the ability to automate most, if not all, of the dataset curation process promotes freshness.

A notable factor that influences the accuracy of these datasets is known as granularity, which pertains to how data is classified. There are two commonly mentioned types of granularity in the context of VD. **Function granularity** involves classifying vulnerable source code by examining and extracting functions. Conversely, **commit granularity** involves classifying Git commit messages and retrieving the vulnerable source code from the parent commit. However, the latter approach faces label noise issues [11], which refers to errors in labelling. These errors are not introduced by the specific commit granularity method used, but by the inherent issue of knowing which functions in a commit are responsible for a vulnerability. For instance, In *Devign* [2], the researchers assumed all modified functions to be vulnerable, which is not always the case. In *DiverseVul*, all non-modified functions were considered non-vulnerable, which is also not always true.

In existing literature, a multitude of methods for curating datasets of vulnerable source code have been researched. While each method aims to address at least one or more the aforementioned dataset characteristics, none have addressed all five concurrently. These curation methods can generally be classified into four categories: synthesis, static analysis, security issue collection, and DL. Furthermore, each of these methods employ different granularity techniques. The following subsections will discuss these curation methods and their associated granularity types, analysing their merits and potential limitations.

Synthesis

The synthesis method is a function granularity method that involves handcrafting vulnerable source code. This approach diverges from other methods that typically rely on labelling existing code or commits. One clear advantage of synthetic datasets lies in their high accuracy.

However, there are many drawbacks. For one, although the size and diversity of synthetic datasets can be large, it relies heavily on a researcher's ability to create varied samples and the amount of time they are willing to dedicate. Furthermore, synthesis is a labour-intensive process, meaning that these datasets are often not updated frequently, impacting their freshness. On top of that, their artificial nature does not accurately represent the diversity and complexity of real-world software, causing them to be inherently inorganic.

One prominent example is the Juliet Test Suite, which is a collection of synthetically-made vulnerable C/C++ source code [12]. Its primary purpose was to evaluate security assurance tools such as Cppchecker [13], Flawfinder [14], and Joern [15]. However, several researchers have also used the test suite for training VD models, which was not its original intention [10]. In fact, Black [16], an author of Juliet, acknowledged that the test suite could be improved by making its data more organic. He suggested that injecting synthetic bugs into production source code would better emulate real-world software. For these reasons, it is becoming less common to see researchers use synthetic datasets for DL-based VD.

Static Analysis

When implemented at the function granularity level, static analysis involves running tools that examine source code based on a set of predefined rules and conditions. Once vulnerable lines of code are identified, their corresponding parent functions can be extracted. A significant advantage of static analysis is its capability for automation. As such, static analysis can yield fresh datasets and be kept up to date with an existing codebase. Additionally, automation allows static analysis to classify large quantities of functions. Furthermore, the datasets from static analysis are organic because these tools can be run on genuine source code.

However, there are some limitations. Like the synthesis method, function-level static analysis often results in less diverse datasets. This is because the variety of identifiable functions is limited by the researcher's ability to engineer diverse rules and conditions. Moreover, these tools are known for being inaccurate due to high rates of negatives and their inability to classify unknown vulnerabilities [17].

Draper [7] exemplifies the use of static analysis for curating datasets. This dataset contains a collection of vulnerable functions which were curated using three static analysis tools: Clang, Cppcheck, and Flawfinder. About 13 million vulnerable and non-vulnerable functions were extracted from public repositories on GitHub and the Debian Linux distribution. Despite being employed by other researchers for DL-based VD [18], the *Draper* dataset is criticised for its low label accuracy [9].

Static analysis can also operate at the commit granularity level, with most methods utilising regular expressions to classify commit messages based on specific keywords. Figure 1 provides an example of the filters used in LICA [19], a tool designed to extract VFCs from the Linux kernel repository. The use of keywords is essentially a rule-based method, therefore this approach shares the same advantages and disadvantages as function level static analysis.

Figure 1
LICA keyword filters

```
BASIC_FILTER = {
    "UAF": [ "use-after-free", "use after free", "UAF" ],
    "Double Free": [ "double-free", "double free" ],
    "Races": [ "race condition" ],
    "Heap Overflow": [ "heap overflow", "heap buffer overflow", "heap-based overflow", "heap-based buffer overflow" ],
    "Stack Overflow": [ "stack overflow", "stack buffer overflow", "stack-based overflow", "stack-based buffer overflow" ],
    "Memory Corruption": [ "out-of-bounds write", "oob write", "out of bounds write", "out-of-bound write", "out of bound write" ],
    "Info Leak": [ "info leak", "information leak", "uninitialised var",
                   "uinitialized var", "unitititialised stack", "uninitialized stack",
                   "out-of-bounds read", "oob read", "out of bounds read", "out-of-bound read", "out of bound read"
    ],
    "General": [ "exploit", "memory corruption", "lpe ", "privesc", "privilege escalation", "attacker", "code execution", "hijack" ]
}
```

Note. From *Lica*, by Sam4k, 2023.

For instance in the *Devign* paper [2], the researchers utilised regular expressions to extract data from five repositories on GitHub: Linux, Qemu, Wireshark, and FFmpeg. They used keywords that contained terminologies referenced in these different libraries (Refer to Appendix A) . This demonstrates the lack of diversity in static analysis methods as library-specific keywords had to be used in order to capture a broader range of commits. Moreover, the researchers acknowledged the inaccuracy of regular expressions and had to undergo an additional round of labelling manually after the initial data collection.

Security Issue Collection

Within the software domain, security issues are vulnerabilities reported by users. Many organisations maintain and track these issues in databases. For instance, the **N**ational **V**ulnerability **D**atabase (NVD) is the largest publicly available database of its kind [20]. The NVD is a live mirror of the **C**ommon **V**ulnerabilities and **E**xposure (CVE) database and also includes additional metadata such as the **C**ommon **V**ulnerability **S**coring **S**ystem (CVSS), **C**ommon **W**eakness **E**numeration (CWE), and most importantly, the Git commit hashes associated with vulnerability patches, which makes security issue collection a commit granularity approach.

The advantage of security issue collection is in its inherent accuracy as security experts vet each reported issue before they are included in the database. Next, the diversity of this approach is significant as security issues span a wide range of vulnerabilities, programming languages, operating systems, and et cetera. Furthermore, this method guarantees organicness since all security issues stem from real-world software. Finally, as mentioned before, security issue collection is an automatable process, thereby allowing the dataset to be continuously updated, and thereby contributing to its freshness.

Nonetheless, there exist several limitations to this approach. Although the NVD contains over 200,000 CVEs, a considerable amount of entries do not include the commit hash to their patches. Therefore, this limits the amount of vulnerabilities that can be extracted. Furthermore, the occurrence of silent patches is well documented

[21]. These refer to vulnerabilities or flaws that have been corrected without public disclosure. As such, security issue collection can not capture this data.

In 2023, Bhandari et al. [10] developed a tool called *CVEFixes*. It automatically collects commits and corresponding vulnerable source code from the NVD. It is built incrementally which means it can include new changes to the NVD to its existing dataset. However, the researchers mentioned that 20% of CVEs do not reference their patch fix, therefore limiting the volume of data that can be collected.

One last issue with security issues is that derivative datasets naturally only contain VFCs. For example, *BigVul* [22] crawled several CVE repositories and gathered 4,310 VFCs, but 0 non-VFCs.

Deep Learning

DL is the latest SOTA approach for curating VD datasets for DL. By leveraging the innate ability of neural networks to form patterns across multiple data points, using DL for processing large amounts of data is one of the most popular ways to utilise it.

With DL, any relevant raw data can be easily turned into meaningful information; for example, Git commits. A Git commit by itself isn't meaningful enough without any type of labelling. Without a human interpreter who has a vast knowledge of the domain, a layman would not be able to tell if a commit is a vulnerability fix accurately. By utilising a well-trained DL model, the labelling can be done with accuracy and efficiency, performing much faster than what manual labour can achieve. When compared to other methods, DL is able to accomplish all the advantages of each method to a certain extent.

Git commits are virtually infinite and diverse. Github, one of the largest code hosting platforms utilising the Git version control system, has over 100 million repositories, each packed full of commits by millions of unique users spanning decades. Popular open-source software used in production environments like the Linux Kernel is hosted on GitHub too.

By training a high-quality DL model focused on classifying Git commits, automating classification can be faster and more accurate than manual labour or static methods, which have the tendency to miss edge cases that otherwise could be crucial data points.

However, it is not a perfect solution.There are a multitude of methods to train deep-learning models. While there are benchmarks and papers that showcase which method is best for a use case similar to commit classification, countless new methods are being introduced over time, and something considered SOTA can be quickly replaced by newer research.

Not only that, one also needs to decide what data points to put in the training dataset. Putting too many types of data can confuse the DL algorithm, making it perform worse. With Git commits, there are a few data types for consideration; for example, commit message, code diff, commit author and so on.

3 Requirement Analysis

3.1 Problems

DL is the forefront for SOTA VD, which requires the availability of quality data infrastructure in order to train an effective model. Data infrastructures refers to existing high-quality datasets, and the tools used to curate them. However, current methods do not have the ability to build such quality datasets. They either lack volume, diversity, accuracy, organicness, or freshness. Table 2 summarises these methods used by past researchers to create datasets of vulnerable source code.

Manually engineered datasets are highly accurate and diverse when created by professional security researchers. However, they are often small because of the laborious process and its inability to be automated. Moreover, the data is not organic which affects a DL VD model's ability to generalise to real-world data.

Automated static analysis techniques at the function and commit level are able to scale to large volumes of data, and can be run on real-world source code or commits. However, they are rule-based approaches which restrict the diversity of data collected. Whatsmore, they have high rates of false positives and false negatives.

Conversely, manual static analysis is able to achieve higher accuracy when conducted by trained security professionals. However, this requires strenuous efforts which results in a curation of smaller datasets. This is why *Devign* combined both automated and manual to curate a large, accurate, and organic dataset for their research. Nonetheless, combining both methods does not address the automatability concerns.

Security issue collection methods can automatically curate diverse, accurate, and organic datasets. However, a significant amount of source code vulnerabilities are not publicly announced or tracked. Hence, this method misses out on a potentially larger amount of commits.

Lastly, DL techniques are the objectively the most effective solution for dataset curation. However, their performance is dependent on the dataset they are initially trained on. There are many methods to curate this preliminary training dataset. For instance, one could manually label a few thousand commits or use the other aforementioned methods. Nevertheless, how good the trained model is at the end is heavily influenced on the quality of the training dataset.

Table 2
Summary of Different Dataset Curation Methods

Method	Type	Granularity	Attributes
Hand-engineered Code E.g. Juliet Test Suite	Synthesis	Function	☐ Large ☑ Diverse ☑ Accurate ☐ Organic ☐ Automatable
Code Analysis Tools E.g. Joern, Cppcheck, Flawfinder	Static Analysis	Function	☑ Large ☐ Diverse ☐ Accurate ☑ Organic ☑ Automatable
Regular Expression E.g. LICA	Static Analysis	Commit	☑ Large ☐ Diverse ☐ Accurate ☑ Organic ☑ Automatable
Manual Labelling E.g. Devign	Static Analysis	Commit	☐ Large ☐ Diverse ☑ Accurate ☑ Organic ☐ Automatable
Proper Commit Linting E.g. SECOMLINT	Static Analysis	Commit	☐ Large ☐ Diverse ☑ Accurate ☑ Organic ☑ Automatable
Web Scraping E.g. CVEfixes, BigVul, Linux Kernel CVEs	Security Issue Collection	Commit	☐ Large ☑ Diverse ☑ Accurate ☑ Organic ☑ Automatable
Natural Language Processing	Deep Learning	Commit	☑ Large ☑ Diverse

Commit Classification E.g. PatchRNN, word2vec			☑ Accurate ☑ Organic ☑ Automatable

As shown above, curating datasets with DL techniques is the most effective solution. However, existing methods use older DL technologies to do so. This is evident by their low accuracy and F1 scores, illustrated in Table 3.

Table 3
DL Dataset Curation Performance

Method	Dataset	Accuracy	F1
Random Forest + Class Balancer [23]	NVD, C Language	54.75%	0.485
CodeBERT + RoBERTa [24]	900Repo Dataset, Mixed Language	87.4%	0.839
PatchRNN [25]	PatchDB, C and C++ Language	83.57%	0.747

3.2 Proposed Solution

The team proposed that by using SOTA Transformer architecture models, we can outperform existing commit classification techniques. The team uses Transformers as it is currently the most prominent and effective DL architecture for NLP. Thus, it will allow us to create highly performant models for dataset curation which will assist security researchers in creating powerful DL VD models.

To this end, the team will build a ground-truth dataset that combines several publicly available datasets of VFCs. Then, the team will train SOTA Transformer models on this dataset for VFC classification, and evaluate its performance relative to existing solutions. By combining the different datasets, it results in a more diverse training source for DL models.

Next, the team will incorporate the trained VFC classification model into a prototype tool. This tool will be used to create a very large and diverse dataset of vulnerable functions.

Lastly, the team will measure the difference in VD performance between two models. The first is trained on a ground-truth dataset, and the second is trained on the same ground-truth dataset and the curated dataset. If the team's experiments return

positive results, it thus demonstrates the effectiveness of the team's approach to DL-based dataset curation.

3.3 Research Questions

Apart from the team's proposed solution, the team also researched one question that emerged during the course of this project.

The team aims to determine if Encoder-based Transformers can outperform other architectures in VD. In *DiverseVul*, they demonstrated the Encoder-Decoder models outperformed Encoder models in VD. However, according to the team's research, Encoders should be the most effective architecture for classification tasks.

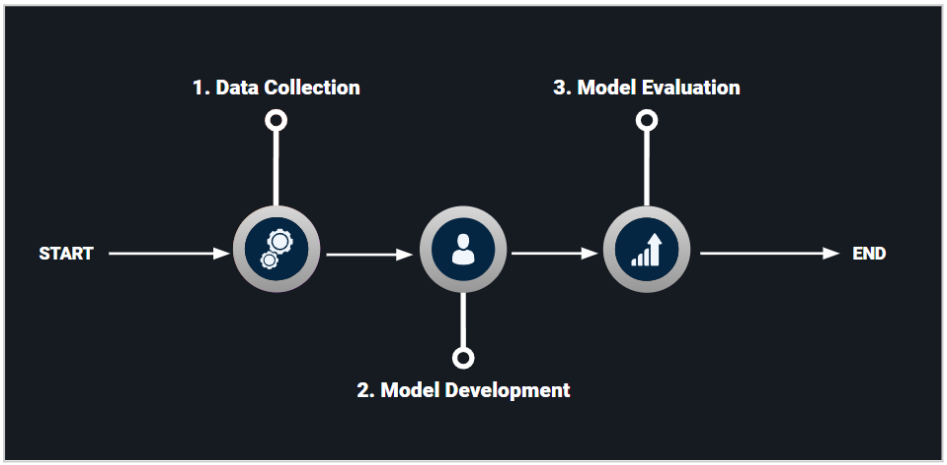
This discrepancy may be a result of their comparison between Encoder-decoders pre-trained on C/C++, and Encoders that were not. Additionally, the team found that *DiverseVul* uses heavily imbalanced datasets which negatively affects the performance of VD [26], [2].

Hence, the team will attempt to emulate the setup used by *DiverseVul*, whilst avoiding their setbacks to find out if SOTA Encoder models are effective at VD.

4 Investigation Stage

This section contains all prerequisite research, methodologies, and actions that lead to the team's experiments and findings in Section 5. Figure 2 gives an overview of the project's investigation workflow.

Figure 2
Project Objectives



4.1 Data Preparation

To explore the application of LLMs for automated dataset curation and VD, the team has collected two types of data: VFCs in Table 4 and vulnerable functions in Table 5.

4.1.1 Data Sources

In order to streamline the team's efforts, the team has combined pre-existing datasets curated by *CVEfixes* [10], *Devign* [2], *BigVul* [22], and *ReVeal* [26], chosen because of their diversity and accuracy. This combination of datasets is based on the work of *DiverseVul*,which also applied these datasets for their research. However, they have yet to release their dataset to the public, hence why the team will be collating them in this paper.

Although these datasets are of high quality, they do have some limitations. Specifically, *CVEfixes* and *BigVul* only contain VFCs as they source their data from security issues. As a result, all non-VFCs in the combined dataset come from *Devign*, as shown in Table 4. Therefore, the diversity for non-VFCs is significantly lower than VFCs because *Devign* contains only two projects.

To address this imbalance of diversity, the team contributed to the combined dataset by manually labelling an additional 3,500 commits from 10 distinct repositories on GitHub. Additionally, for ground-truth evaluation, the team extracted 1,827 VFCs

from the Linux repository using the Linux Kernel CVEs dataset[3] and manually labelled an additional 500 non-VFCs.

These 10 repositories: FFmpeg, Git, Apache HTTP, ImageMagick, OpenLDAP, OpenSSH, OpenSSL, Postgres, Redis, and Samba; were chosen because of their significance and widespread usage in the industry, meaning that these projects are well-established and frequently updated, hence providing a continual source of data. In addition, these projects cater to a wide range of applications, such as operating systems, web servers, image processing, and more. For the same reason, they suffer from diverse types of vulnerabilities. For instance, Apache HTTP is more likely to contain web-related vulnerabilities like path traversals, whilst OpenSSL is more susceptible to cryptographic bypasses and attacks.

The team collected and shuffled commits from these repositories made between 2015 and 2023, amounting to 177,279 commits. Then, the team's five members invested a total of 40 man-hours labelling these commits. Each member labelled approximately 175 VFCs and 175 non-VFCs to maintain a balanced dataset. Similar to *Devign*, the team makes use of keywords indicative of a VFC (Refer to Appendix A), using the regular expression search feature of Google Sheets, and manually labels these commits. Afterwards, the remaining unmatched commits are sequentially labelled until the team's target count is met. Additionally, the team include commits mentioned in the security board and issue tracker of each project where available, such as the Apache HTTP Server's security vulnerability page[4]. Furthermore, to create more precise labels, the team excluded any commits that were vague or provoked any degree of uncertainty regarding its label. Then, the team conducted a final cross-validation round where each member would validate the labels of another member.

In the team's dataset, the team made the distinction between vulnerability fixes and bug fixes. The former is a subset of bug fixes and will have to be labelled differently. For example, a commit that fixes a spelling mistake is considered a bug fix, but not a vulnerability fix. During the team's cross-validation stage, the team ensured that the dataset followed this principle.

4.1.2 Dataset Processing

The *CVEfixes*[5] dataset is an SQL dump and requires a few join queries to extract the data, and where clauses were specified to only extract C/C++ commits and functions. The *BigVul*[6], *Devign*[7], and *ReVeal*[8] datasets are available in CSV or JSON

[3] https://github.com/nluedtke/linux_kernel_cves

[4] https://httpd.apache.org/security/vulnerabilities_24.html

[5] https://doi.org/10.5281/zenodo.4476563

[6] https://github.com/ZeoVan/MSR_20_Code_Vulnerability_CSV_Dataset

[7] https://sites.google.com/view/devign

[8] https://drive.google.com/drive/folders/1KuIYgFcvWUXheDhT--cBALsfy1I4utOy

format and required no major preprocessing steps, apart from renaming columns. Then these datasets, along with the team's manually labelled dataset, were merged and deduplicated by their commit hashes, as shown in Table 4.

For more details on the steps used, see Appendix B for the data preprocessing source code.

Table 4

Dataset VFC information

Dataset	No. Projects	VFCs	Non-VFCs
CVEfixes	581	3,534	0
Devign	2	10,894	14,978
BigVul	247	4,310	0
Ours	10	1,737	1,763
Linux Kernel CVEs	1	1,827	500
Combined (deduplicated)	600	19,408	17,217

Note. The *ReVeal* dataset was not included as it does not provide commits, only functions.

4.1.3 Function Extraction

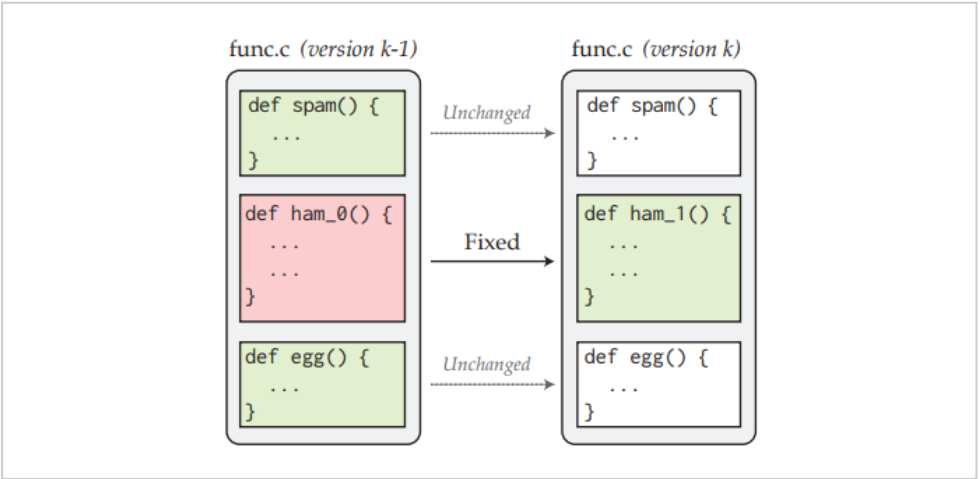
Devign, *CVEfixes*, *ReVeal*, and *BigVul* use commit granularity approaches to extract vulnerable functions as these datasets source their data from Git repositories. Therefore, they each use different methods for extracting vulnerable and non-vulnerable functions from commits.

Devign extracts modified functions from a commit. For a VFC, modified functions in the parent commit are considered vulnerable. Similarly, given a non-VFC, modified functions in the child commit are considered non-vulnerable. *CVEfixes* also applies this approach, but only for VFCs since *CVEfixes* does not contain non-VFCs.

In the case of *BigVul* and *ReVeal* , they extract both modified and unmodified functions. Modified functions in VFCs are considered vulnerable, however, because these datasets do not contain non-VFCs, they label all non-modified functions in the parent commit of VFCs as non-vulnerable. Figure 3 illustrates how *ReVeal* extracts functions.

Figure 3

Function Extraction Method of the *ReVeal* Dataset



Note. From *Deep Learning based Vulnerability Detection: Are the team There Yet?*, by Chakraborty et al., 2020. Green highlight = non-vulnerable, red highlight = vulnerable.

While similar, the second method results in datasets that are skewed towards non-vulnerable functions, apparent in Table 5 as *BigVul* and *ReVeal* are heavily imbalanced towards the latter. Class Imbalance is often not ideal in DL classification tasks and, as proven by Chakraborty et al. [26], balancing vulnerable function datasets leads to better VD performance.

Table 5

Vulnerable Functions

Dataset	No. Projects	Vuln Functions	Non-vuln Functions
CVEfixes	508	6,430	0
Devign	4	16,549	16,484
BigVul	310	10,900	177,736
ReVeal	2	2,240	20,283
Combined (deduplicated)	508	35,607	214,479

Note. The team received permission from DSO and an author of *Devign* to use the unreleased version of their vulnerable functions dataset which contains two additional projects.

4.2 Model Development

4.2.1 Model Architecture

In LLM Transformers, there are 3 main sub-architectures which are Encoders, Decoders, and Encoder-decoders.

In this project, the team will use the Encoder Transformer architecture. The team's project involves classification of source code and commit messages, which is the inherent strength of Encoders. This is because Encoders are proficient at mapping one input with another. On the other hand, architectures like Decoders are more adept at predicting or generating outputs from inputs. Additionally, Encoder models are more effective at classification tasks and require less parameters for similar performance. This means that the model size is smaller and less compute resources are required for training and inference. This is especially beneficial for the project as the team will be building a commit classification tool that should be lightweight enough for inference on consumer hardware.

For Encoder-decoder models, they are as capable as Encoder models for text classification and can reach similar performances [9]. However, Encoder-decoders are more complex and require greater resources to train and infer.

4.2.2 Model Selection

Before the team began researching which Encoder models to use, the team first consolidated several model criterias that would suit the team's needs of source code VD and VFC classification.

First, the team needed a model that is pre-trained. This is to streamline the team's efforts and maintain focus on the main objective as pre-training is a long and costly process. Furthermore, using pre-trained models avoids warranting unnecessary carbon emissions [27]. Second, the team requires the model to be open-source as it grants the most flexibility for training, and also means that the work can be freely distributed to other consumers. Third, the model should be pre-trained on C/C++ as the team will be training, evaluating, and comparing it with other C/C++ VD models. Lastly, the model should not be hardware-demanding. This means a model that is small in parameter size, but without sacrificing significant performance drops.

To discover suitable models, journals on arXiv, blogs on HuggingFace, and online articles were researched. Ultimately, two models were agreed upon: StarEncoder [28] and DistilBERT [29]. Two models were chosen because, for VFC classification, commits are a blend both of natural and programming languages, though more so for natural language. As such, it is uncertain pre-training on programming language will improve or negatively affect VFC classification performance. Hence, the team includes both models as DistilBERT is trained on solely natural language, while StarEncoder is trained on natural language and source code. The team aims to train, evaluate, and compare both, to find the best pre-training objective for VFC

classification. The best of which will be used for subsequent experiments. More information about these models are covered in the following sections.

DistilBERT
DistilBERT is an open-source “distilled” version of the BERT model. Distillation is an optimisation method that reduces the size of a language model, allowing for faster training and inference. DistilBERT used this technique to reduce the size of the BERT model by 40% while maintaining 97% of its original capacity, allowing for a 60% increase in speed.

These reasons make DistilBERT ideal for the team’s purposes as it will allow for quick training and experimentation with little costs. Moreover, DistilBERT can be easily run on consumer hardware due to its small size and low memory usage. In fact, even at high batch counts, DistilBERT can be trained without the need for advanced ML-driven GPUs.

StarEncoder
StarEncoder is an open-source model based on the BERT architecture. However unlike DistilBERT, it was pre-trained on a substantial dataset comprising over 100GB worth of code written in C/C++. This would enable the model to more effectively comprehend and classify C/C++ source code to detect vulnerabilities.

Starencoder is pre-trained on 74.93GB of markdown, 54.40GB of GitHub issues, and 64.00GB of Git commits. The language used in such files is a mix of both programming and natural language. Therefore,Starencoder is very capable of understanding code-related documentation (i.e., Git commits).

Other than C/C++, StarEncoder is also trained on 84 other popular programming languages which total over 800GB of source code. This broad language coverage will equip the model with the ability to understand very diverse codebases.

However, unlike DistilBERT, StarEncoder is a more resource-demanding model, requiring twice the memory size for inference. Additionally, this means that experiments involving training will be more slow and tedious without the use of expensive ML-driven GPUs. Nonetheless, the team will proceed on with StarEncoder because of its incredibly large and relevant pre-training objective.

4.2.3 Model Training

The team made use of the HuggingFace Transformers library for model training as it provides extensive Python APIs and libraries to easily interact with StarEncoder and DistilBERT, download public datasets, and publicise the work done by the team.

To train these models, the team rented GPUs on RunPod.io. Afterwards, the team utilised the Weights & Biases application to log and visualise model training progress.

The following hyperparameters are shared in all training instances conducted in this paper, unless stated otherwise.

- Precision: Half-precision Floating Point (FP16)
- Optimiser: AdamW
- Learning Rate Scheduling: Linear
- Weight Decay: 1e-2
- Seed: 420

The team used FP16 to speed up training and reduce computing costs. For the other hyperparameters, they are commonly used in most general applications and the team saw no need to modify them.

Vulnerability-fixing Commit Classification

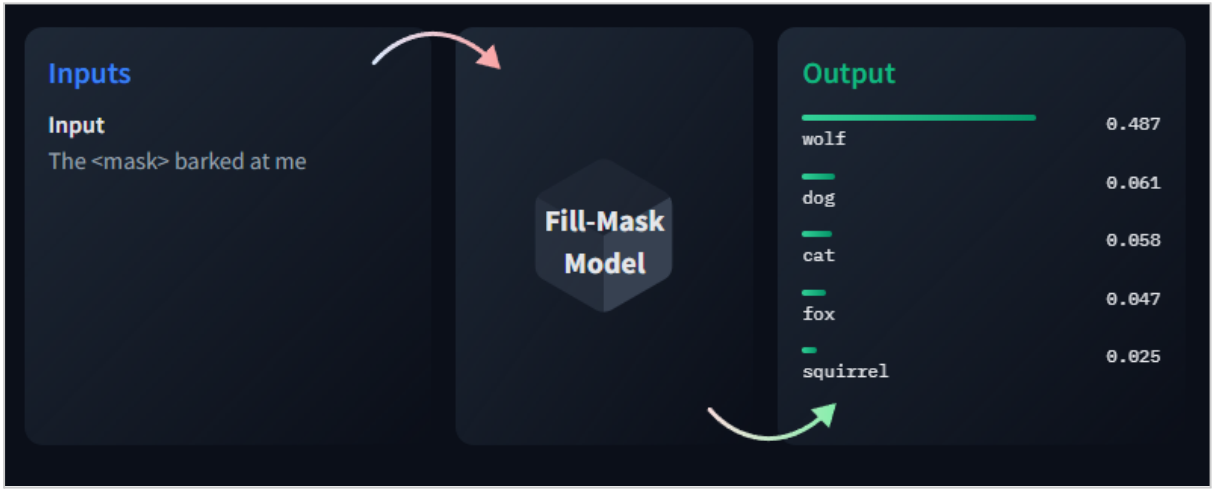
VFC Classification training will be broken down into two stages.

The first stage involves domain adaptation training with the **M**asked **L**anguage **M**odelling (MLM) objective. The purpose of domain adaptation is to teach a pre-trained model to learn the structure of a given domain, which in this case is commit messages.

As illustrated in Figure 4, MLM involves randomly masking words from a sentence, then having a model predict the possible words. MLM is often done using random sampling where a percentage of words in a sentence are masked at random. In this paper, the team randomly masks 15% of words, the same value used in StarEncoder and the original BERT model [30].

Figure 4

MLM Example



Note. From Hugging Face, n.d.

MLM is a self-supervised task, meaning that it can be done without manually labelled data. Hence, the team will train DistilBERT and StarEncoder for MLM using the 177,279 commits the team collected for manual labelling.

The following are the hyperparameters used in this project's MLM training, excluding the shared parameters covered previously:

- GPU: NVIDIA A100 40GB
- Number of Epochs: 20
- Learning Rate: 1e-4
- Batch Size: 256 (DistilBERT) and 128 (StarEncoder)
- Maximum Sequence Length: 256 Tokens
- Train Validation Split: 80/20
- Mask Probability: 15%

The team trains for 20 epochs as MLM usually takes a longer time to reach a perfect fit. The learning rate the team used is a common value used for MLM. The batch sizes used were the maximum batch size that would fit in the GPU. As DistilBERT is smaller, more batches can be fit for training.

The usage of 256 tokens for the maximum sequence length was arrived at after calculating the mean word count of commit messages in the dataset, which was 37.64. The standard deviation was 61.44 with the 75th percentile of commits being over 64 words long. However, the token count is usually larger than the word count. For instance, the word “fixing” can be divided into 2 tokens: “fix” and “ing”. Additionally, programming language tokens often contain more tokens than natural language. Therefore, a larger value of 256 should account for most of these inputs.

For the train validation split, the team used a higher split as 177,279 is a very large dataset.

Both models took a total of 2.2 hours for training and their train and evaluation loss are shown in Figure 5. The team can see that StarEncoder better understands the semantics of commit messages because it has a lower absolute loss value.

Figure 5
DistilBERT and StarEncoder MLM Loss Curves

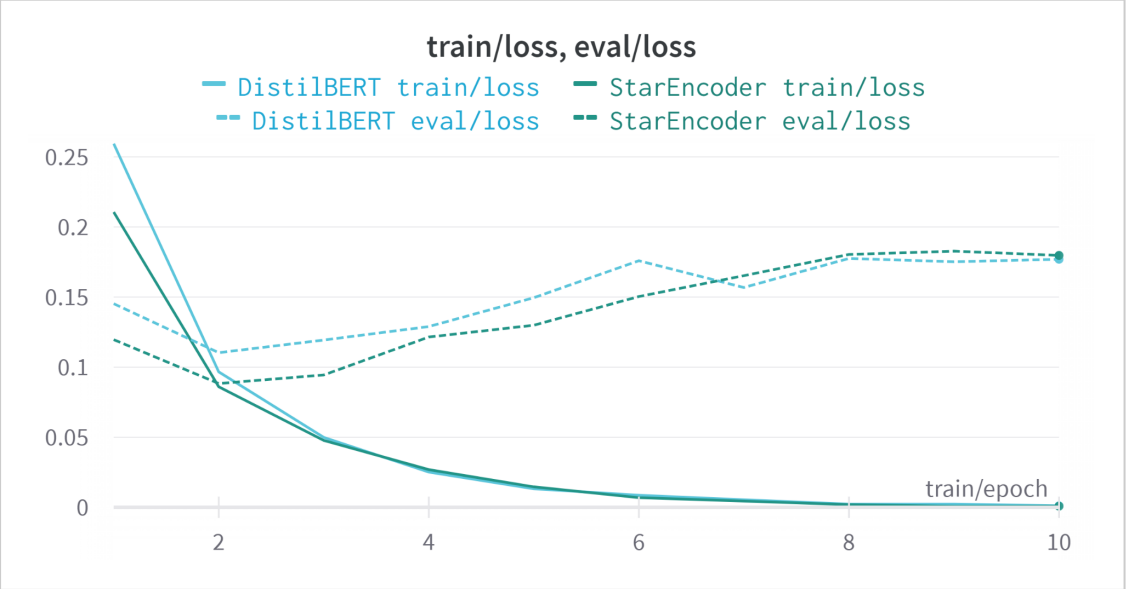
After MLM, the second and last stage will be binary classification of VFCs. The team uses the combined VFC dataset prescribed in Section 4.1.1.

The following are the hyperparameters used:

- GPU: RTX 3090
- Number of Epochs: 10
- Learning Rate: 1e-4
- Batch Size: 256 (DistilBERT) and 128 (StarEncoder)
- Maximum Sequence Length: 256 Tokens
- Train Validation Split: 90/10

Both models took a total of 40 minutes to train, and their train and evaluation loss are shown in Figure 6. Signs of overfitting started around the 3rd epoch.

Figure 6
DistilBERT and StarEncoder VFC Classification Loss Curves



Vulnerability Detection

VD is a task entirely centred around source code and programming language. Therefore, DistilBERT will not be trained for this purpose, only StarEncoder. Additionally, the over 100GB of C/C++ pre-training done for StarEncoder means that domain adaptation is not necessary. As such, this project will have a single stage training for VD.

There will be a total of 3 StarEncoder models trained for VD in this project.

The first model will be trained on the public *Devign* function dataset consisting of 27,318 functions.

The second model will be trained on the dataset curated by the team's VFC Classification Tool, covered in Section 5.3.

Finally, the third model will be trained on the combined vulnerable functions dataset from Table 5. Because this dataset is heavily imbalanced, random sampling is applied to non-vulnerable functions to obtain the same amount of vulnerable functions, totalling 71,214.

The following are the hyperparameters used:

- GPU: RTX 3090 Ti
- Number of Epochs: 10
- Learning Rate: 9e-6
- Batch Size: 45
- Maximum Sequence Length: 512 Tokens
- Train Validation Split: 90/10

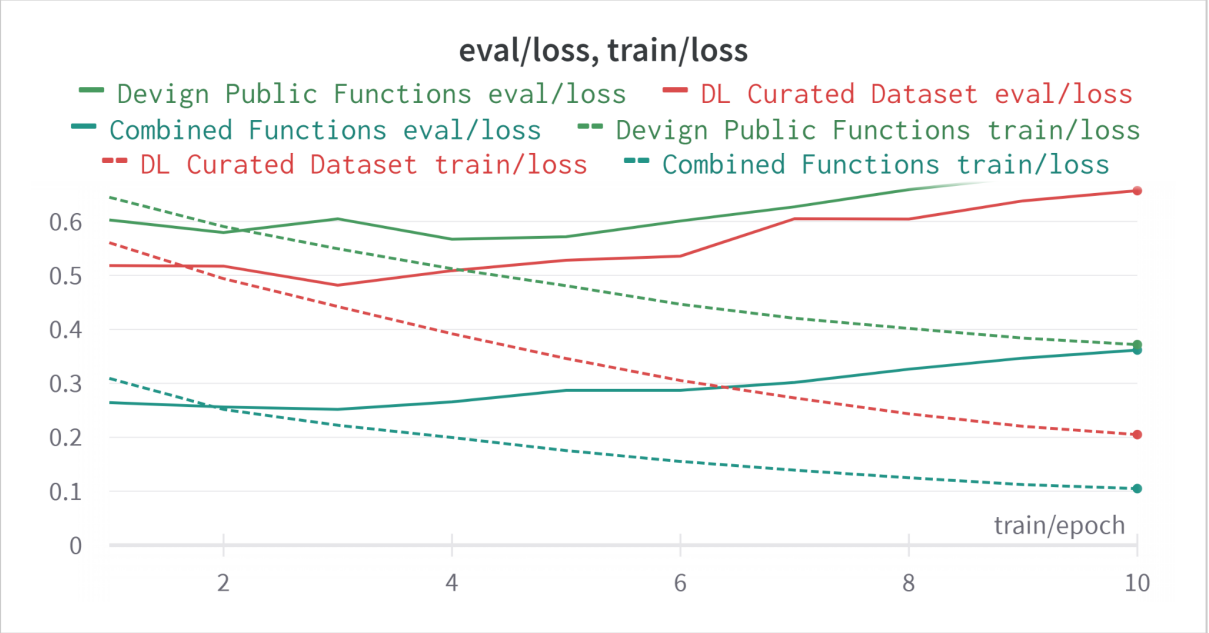
The team uses a learning rate much lower than the previous trained models as previous works have pointed out that higher learning rates can result in a degenerate model that only predicts one label.

A larger maximum sequence of 512 tokens was used as source code functions can reach very large lengths. StarEncoder's actual maximum is 1024 tokens, however, increasing the input size results in more memory consumption. In doubling the sequence length, the highest stable batch size we could use decreased to 45.

Figure 7 contains the loss curves of all 3 models. In total, these models were trained on 129,142 functions which took 3.5 hours.

Figure 7

StarEncoder VD Loss Curves



4.3 Evaluation

4.3.1 Performance Metrics

In order to reliably assess and compare the classification performance of different models, methods, or techniques, the team employed the use of the following set of metrics that are standard in ML research:

Accuracy calculates the percentage of correct predictions made out of all predictions made.

$$Accuracy = \frac{True\ Positive + True\ Negative}{True\ Positive + True\ Negative + False\ Positive + False\ Negative}$$

Precision calculates the percentage of data with the label “1” that were correctly predicted.

$$Precision = \frac{True\ Positive}{True\ Positive + False\ Positive}$$

Recall calculates the percentage of actual data with the label “1” that were correctly predicted.

$$Recall = \frac{True\ Positive}{True\ Positive + False\ Negative}$$

F1 calculates the harmonic means between precision and recall.

$$F1 = \frac{2 \times Precision \times Recall}{Precision + Recall}$$

Weighted Average calculates the average in proportion to the labels.

$$Weighted\ Average = \frac{\sum_{i=1}^{n} x_i w_i}{\sum_{i=1}^{n} w_i}$$

Cosine Similarity calculates the similarity between two vectors.

$$similarity\ (A, B) = cos(\theta) = \frac{A \cdot B}{||A||\,||B||} = \frac{\sum_{i=1}^{n} A_i B_i}{\sqrt{\sum_{i=1}^{n} A_i^2} \cdot \sqrt{\sum_{i=1}^{n} B_i^2}}$$

4.3.2 Train Test Split

In ML, datasets are split into train and test sets to determine the ability of a model to generalise on unseen data. In this paper, the team uses different splits for the VFC and vulnerable function datasets gathered in Section 4.1. The team uses the Linux Kernel CVEs data in the VFC dataset as the unseen test dataset. For the vulnerable functions dataset, the team uses a 90-10 split.

4.3.3 Other Comparisons

The team will compare the results of the team's fine-tuned models against other models and methods to assess the relative performance of the team's methodologies.

Deep Learning Vulnerability-Fix Commit Classification
For VFC classification models, the team will compare them against three similar models, PatchRNN [25], Random Forest Class Balancer [23], CodeBERT + RoBERTa with distinct sequences [24].

Firstly, PatchRNN is a DL-based security patch identifier that utilises both text, in the form of commit messages, and code, in the form of commit source-code diff. PatchRNN uses a twin RNN model to generate code vector representations of commit code diffs and uses NLP to process commit messages and TextRNN to obtain vector representations of the commit messages. By using vectors, PatchRNN is able to aggregate both code and natural language to the decision-making of the DL model.

Secondly, Wang et al. created a DL-based vulnerability multi-class classifier, using a large-scale dataset of CVEs from NVD. The classifier uses a machine learning technique known as Random Forest, which utilises multiple decision trees to get a majority vote from each individual decision tree, allowing for more accurate predictions.

Lastly, Lee and Chieu [24] created a DL-based commit classifier that utilises Co-Training, which is similar to Wang's PatchRNN model. They separated the responsibility of classifying commit messages and commit source-code diffs into 2 distinct sequences, CodeBERT for the code diffs and RoBERTa for the commit messages. This leverages the abilities of both NLP Models and CodeBERT is pre-trained with mostly code whereas RoBERTa is pre-trained mostly with natural language.

Conventional Vulnerability-Fix Commit Classification
Additionally, the team's models will be compared against two distinct classification methods, Regular Expressions and Machine Learning embeddings.

The first is LICA [19], an open-source project that analyses Linux kernel commit messages for vulnerability fix commits using regular expression. It implements a rule-based approach for text classification, meaning that it uses predefined rules or keywords. The current iteration of LICA only has a few rules, hence, the team forked the project and modified it with additional keyword filters. Reason being, LICA's rules were developed for the Linux Kernel repository, and as different repositories have disparate types of vulnerabilities, these rules do not generalise well to repositories. Other than that, the team also modified LICA to output performance metrics from the test datasets. The team named the original version LICA v1 and the modified version LICA v2.

The second method is using semantic search with SOTA DL embeddings [31]. Semantic search is used to query against a corpus and retrieve the top k most similar results using cosine similarity. This approach involves converting a corpus of texts into word embeddings. Word embeddings is a method to represent and associate words in a high dimensional vector that encapsulates its semantic information. At a basic level, they are labels given to each word in a sentence, derived from its meaning in the context of its sentence. This allows for text classification using semantic search queries.

To create a semantic search tool that is suited for comparison, the team utilised Chroma, a vector database used for storing, searching and retrieving high dimensional vectors alongside 2 different word embedding models all-MiniLM-L6-v2 and text-embedding-ada-002 from SBERT and OpenAI respectively. These embedding models are chosen primarily because they generate embeddings at sentence level which can better understand the semantic of the commit messages and they also have great balance in terms of performance and size.

Vulnerability Detection

The team's StarEncoder VD models will be compared with the models trained in *DiverseVul*. This paper has emulated most of the crucial elements in their research: datasets used, training hyperparameters, and function extraction method.

Additionally, the team will compare the StarEncoder VD models with Callisto, a rule-based code analysis tool.

5 Experimentation & Findings

5.1 Vulnerability-fix Commit Classification

The 3rd and 10th epoch checkpoints of the StarEncoder and DistilBERT VFC classification models were selected as, according to Figure 6, both models showed signs of overfitting due to the divergence of train and evaluation loss curves.

Table 6 shows the results of these model checkpoints on their validation dataset split during training.

Table 6
Results for DistilBERT and StarEncoder, Evaluated on the Validation Split Dataset

Model	Acc	Prec	Recall	F1
		Weighted Avg	Weighted Avg	Weighted Avg
DistilBert (3 epochs)	96.40	96.79	96.26	96.53
StarEncoder (3 epochs)	97.29	97.76	97.00	97.38
DistilBert (10 epochs)	97.13	97.78	96.67	97.22
StarEncoder (10 epochs)	**97.70**	**98.41**	**97.14**	**97.77**

Table 7 shows the results of these model checkpoints on the Linux Kernel CVEs test split from Section 4.3.2.

Table 7
Results for DistilBERT and StarEncoder, Evaluated on Linux Kernel CVEs Dataset

Model	Conf. Matrix	Acc	Prec	Recall	F1
			Weighted Avg	Weighted Avg	Weighted Avg
DistilBert (3 epochs)	TP 1,783 FP 44 FN 309 TN 194	84.85	84.00	85.00	83.00

StarEncoder (3 epochs)	TP 1,715 FP 112 FN 126 TN 377	**89.79**	**90.00**	**90.00**	**90.00**
DistilBert (10 epochs)	TP 1,773 FP 54 FN 259 TN 244	86.57	86.00	87.00	85.00
StarEncoder (10 epochs)	TP 1,668 FP 159 FN 110 TN 393	88.45	89.00	88.00	89.00

Based on Table 6 and Table 7, the team can make the conclusion that pre-training on source code increases VFC classification performance as StarEncoder, trained on 64 GB of source code, consistently outperforms DistilBERT in all categories, with the most extreme being StarEncoder (3 epochs) that scored an F1 score 7% higher than DistilBERT (3 epochs).

Additionally, on the test dataset, the team can observe that the StarEncoder (3 epochs) scores higher overall than the 10th epoch StarEncoder model. This means further facilitates that at epoch 10, the model is overfitted. Hence, the following sections will use the 3rd epoch of StarEncoder for comparison with other models and methods.

5.1.1 Comparison with Other Models

The team compared StarEncoder with similar models, described in Section 4.3.3, for VFC classification, shown in Table 8.

Table 8
VFC Classification Comparison with Other Models

Model	Training Dataset	Acc	F1
Random Forest Class Balancer	NVD, C Language	54.75	48.50

CodeBERT + RoBERTa	900Repo Dataset, Mixed Language	87.40	83.90
PatchRNN	PatchDB, C and C++ Language	83.57	74.70
StarEncoder (3 epochs)	BigVul, CVEfixes, Devign, and the team's dataset	**89.79**	**90.00**

The results above in Table 8 shows that StarEncoder performs significantly better overall compared to 3 other models for VFC classification. On average, StarEncoder scores 14.55% accuracy and 20.97% F1 higher than other models.

5.1.2 Comparison with Other Methods

The team compared the trained StarEncoder model with other methods, described in Section 4.3.3, used for VFC classification. Table 9 contains the evaluation results for the validation split, and the results for the Linux Kernel CVEs dataset in Table 10.

Table 9
VFC Classification Evaluation Results with Other Methods On Validation Split Dataset

Model	Conf. Matrix	Acc	Prec	Recall	F1
			Weighted Avg	Weighted Avg	Weighted Avg
LICA v1	TP 62 FP 1 FN 3,493 TN 3,442	50.07	74.00	50.00	34.00
LICA v2	TP 953 FP 168 FN 2,602 TN 3,275	60.41	71.00	60.00	55.00
Semantic Search (all-MiniLM-	TP 2,793 FP 919	76.05	78.00	76.00	76.00

L6-v2)	FN 757 TN 2,524				
Semantic Search (text-embedding-ada-002)	TP 2,888 FP 890 FN 667 TN 2,553	77.75	78.00	78.00	78.00
StarEncoder (3 epochs)	-	**97.29**	**97.76**	**97.00**	**97.38**

Table 10
VFC Classification Evaluation Results with Other Methods On Linux Kernel CVEs Dataset

Model	Conf. Matrix	Acc	Prec	Recall	F1
			Weighted Avg	Weighted Avg	Weighted Avg
LICA v1	TP 136 FP 1 FN 1,691 TN 502	27.38	83.00	27.00	19.00
LICA v2	TP 918 FP 42 FN 909 TN 461	59.18	82.00	59.00	62.00
Semantic Search (all-MiniLM-L6-v2)	TP 1,636 FP 197 FN 191 TN 306	83.34	83.00	83.00	83.00

Semantic Search (text-embedding-ada-002)	TP 1,688 FP 188 FN 139 TN 315	85.96	86.00	86.00	86.00
StarEncoder (3 epochs)	TP 1,715 FP 112 FN 126 TN 377	**89.79**	**90.00**	**90.00**	**90.00**

As shown in Table 9 and 10, LICA v1 is able to achieve subpar accuracy levels on the Combined VFC Dataset, however, its accuracy experiences a notable drop when evaluated against the Linux Kernel CVEs Dataset. This is unusual as the default filters of LICA should be apt towards the Linux Kernel. Therefore, this may be an indication that the default filters on LICA are not thorough enough to cover the entire Linux Kernel. Furthermore, its high precision with extremely low F1 score on both datasets is an indication of class bias as it makes more correct predictions towards a certain class compared to another. Whatsmore, its confusion matrix illustrated that a significantly larger number of non-VFCs were predicted over VFCs in both datasets.

For LICA v2, the team's updated keyword filter has resulted in an overall improvement on both datasets. Despite these improvements made thus far, LICA can still achieve far better results given more time spent on refining its filters.

As for the semantic search methods, both embedding models yield comparable results, with the text-embedding-ada-002 embedding model slightly outperforming the all-MiniLM-L6-v2 model. This is an indication that the type of general embedding model, or the capability of performing classification tasks, does not noticeably affect the overall results for VFC classification.

Lastly, the StarEncoder model is shown to outperform all the other classification techniques with a higher average accuracy and F1 of 28.63% and 31.88% respectively. This large difference in scores between the team's model and others methods provides empirical evidence that the team's model is the most optimal solution for VFC classification.

5.2 VFC Classifier Prototype

The team developed this classifier tool with 2 features: VFC classification and function extraction. The tool can be downloaded or cloned using the team's GitHub

repository (Refer to Section 1.3 footnotes). Additionally, the installation, setup, and instructions are all included in the README.md file.

5.2.1 Classification

Running the Command

Figure 8

VFC Classifier Prototype Classification

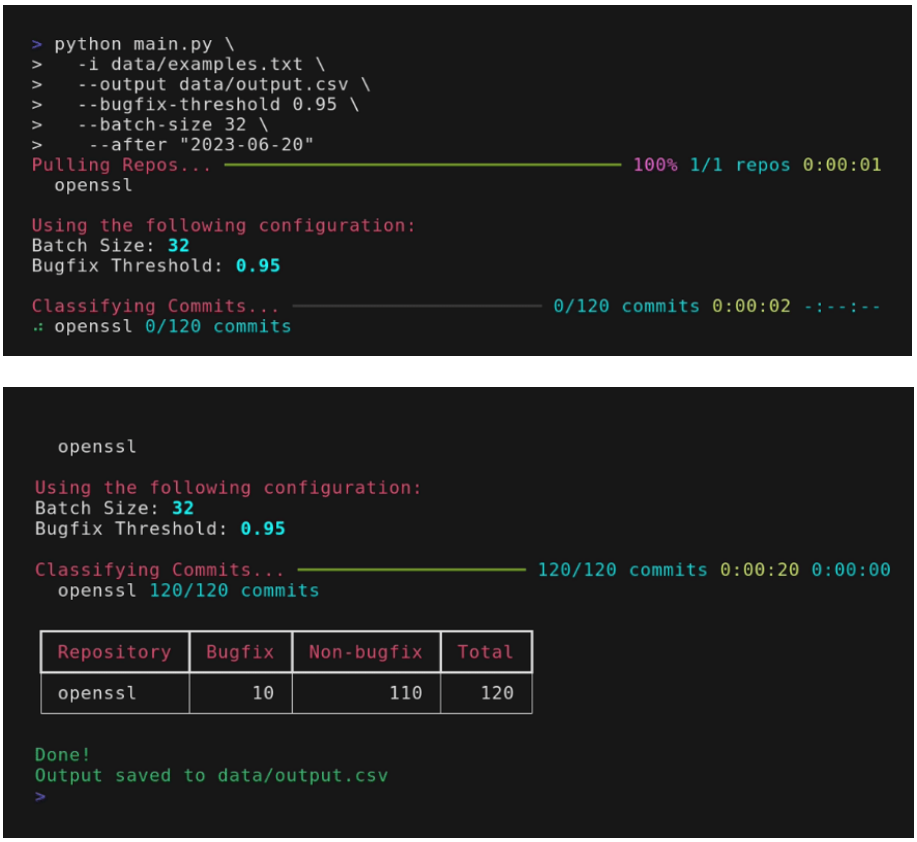


Figure 8 demonstrates how to classify commits using the team's tool on the command-line:

1. The program takes in either a line-separated file of GitHub repository URLs, or a single URL string ("--input | -i"). This option can be specified multiple times.
2. The program will clone the repository(s) locally into a configurable directory. If the repository already exists, any changes will be pulled instead.
3. The program then initialises the classifier model with HuggingFace Transformers. The program will use either the machine's GPU, if configured with NVIDIA CUDA, or CPU.
4. The program then starts reading the commit history from the local Git repositories and feeds it to the model to classify VFCs. The user can specify the classification threshold as a float between 0 and 1, which is 0.5 by default.

5. The program uses Python Pandas to write the result, along with other useful commit metadata, to the output file. The file format is described in Table 11.

Table 11
VFC Classifier Prototype Commit Output Format

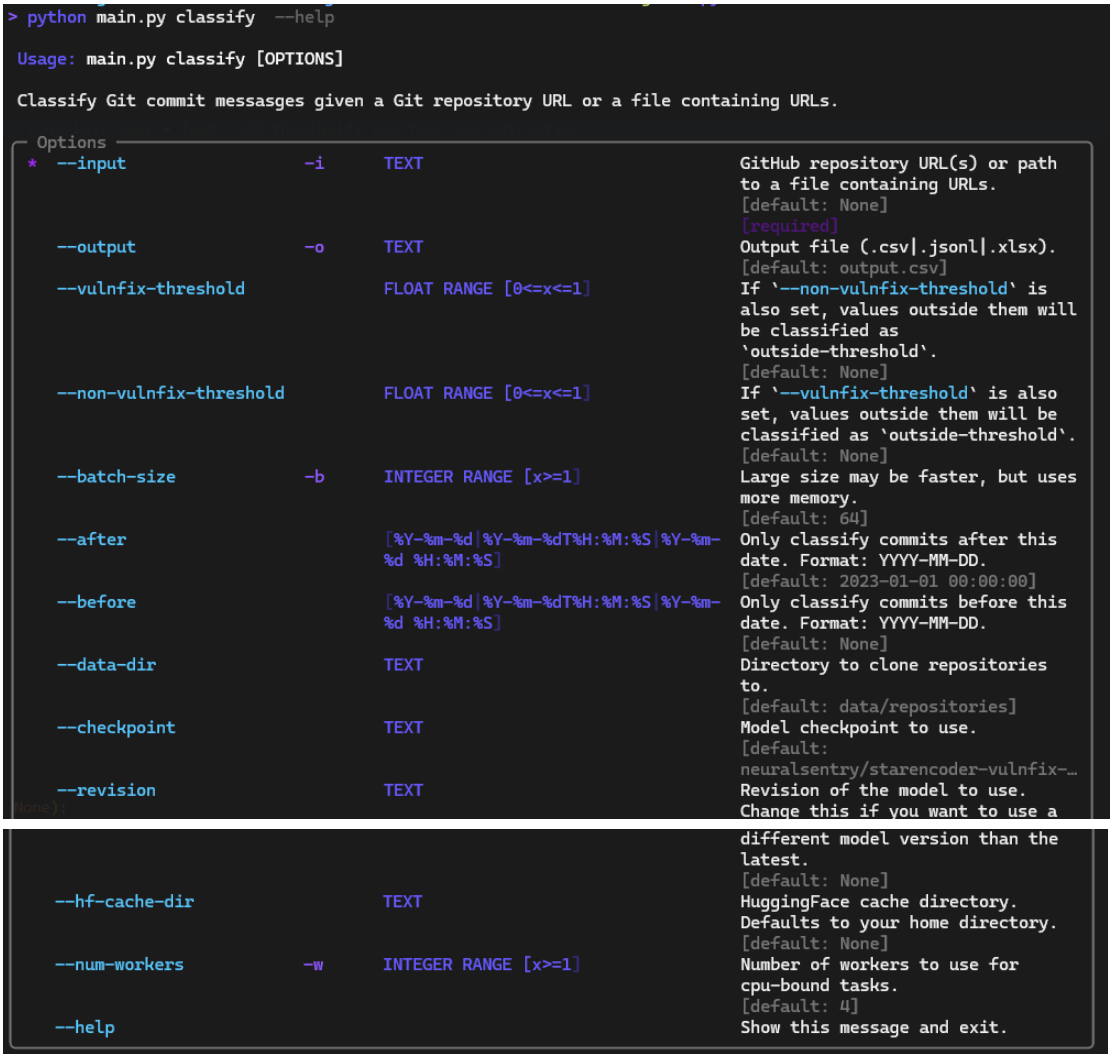
Column Name	Type	Description
is_merge	Boolean	Whether commit is a merge
commit_msg	String	The commit message
commit_hash	String	The commit hash
commit_url	String	The commit URL on Github
repo_url	String	The commit's repository URL on GitHub
date	String	The date the commit was written
labels	Integer	1 = VFC 0=non-VFC
vulnfix	Integer	The softmax value of the model's vulnerability-fix prediction
non-vulnfix	Integer	The softmax value of the model's non-vulnerability-fix prediction

Options
The team designed the prototype so that any VFC classification model can be used by specifying the "--checkpoint" flag, which accepts a HuggingFace model ID. By default, it uses the team's fine-tuned StarEncoder (3 epochs) model.

All classification options can be viewed using the "--help" option, as shown in Figure 9.

Figure 9

VFC Classifier Prototype Classification Help



Notable Features

The StarEncoder model is 500mb in size, however, the team managed to reduce the in-memory size to around 200mb after optimization with the HuggingFace Optimum library. This allows the classifier to run at very fast speeds. For reference, on an RTX 2060, the team classified 600,000 commits from 50 different repositories in just over an hour.

Additionally, the team implemented batching to prevent over-consumption of system memory when the program is run on very large repositories. Also, dates can be specified to reduce the number of commits collected.

5.2.2 Function Extraction

Running the Command

Figure 10

VFC Classifier Prototype Function Extraction

```
$ python main.py extract -i data/cvefixes50ReposSince2020.csv  -o data/cvefixes50ReposSince2020DCMB_functi
ons_nonModifiedAsNonVuln.csv --shuffle --numeric-labels --seed 420 --num-workers 10 --extract-nonvuln-from
-vulnfix
Extracting Functions from Commits... ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 22163/22163 commits 1:15:53 0:00:00
[+] Successfully extracted from commits     (11041/22163 commits)
[-] Ignored commits with no functions found (11122/22163 commits)
[+] Extracted functions (vuln)              (15305/0 functions)
[+] Extracted functions (non-vuln)          (488015/0 functions)
```

Repository	Vuln	Non-vuln
tensorflow	0	2
linux	8330	291666
src	1007	37967
openssl	100	2041
libgit2	11	224
njs	55	2275
qemu	231	9112
FFmpeg	206	3531
php-src	213	8768
ImageMagick6	27	492
libvncserver	0	2
LibRaw	1	18
openexr	37	373
WavPack	1	44
htmldoc	0	0
libmobi	0	0
Total	15305	488015

Figure 10 shows how to extract functions from classified commits using the “extract” command:

1. The command accepts a file that follows the format in Table 11, i.e., a file of classified commits.
2. The command will fetch source code, method name, and line change data to extract functions in commit
3. Python Pandas is then used to write the functions into a file. The file’s format is shown in Table 12.

Table 12

VFC Classifier Prototype Extracted Functions Format

Column Name	Type	Description
labels	Integer	1 = VF 0 = Non-VF

preds	Integer	1 = VF 0 = Non-VF
name	String	Function name
symbol	String	Function symbol
parameters	String	Function parameters
start_line	Integer	Start line of the function
end_line	Integer	End line of the function
function	String	Function source code
filename	String	The filename containing the function
path	String	The path to the file in the repository
token_count	Integer	The amount of individual components of a function, e.g. variables, operators, etc
repo_url	String	The commit's repository URL on GitHub
commit_msg	String	The commit message
commit_hash	String	The commit hash
commit_url	String	The commit URL on GitHub
date	String	The date the commit was written
file_url	String	The commit file URL on GitHub

Note. VF = Vulnerable Function

Options

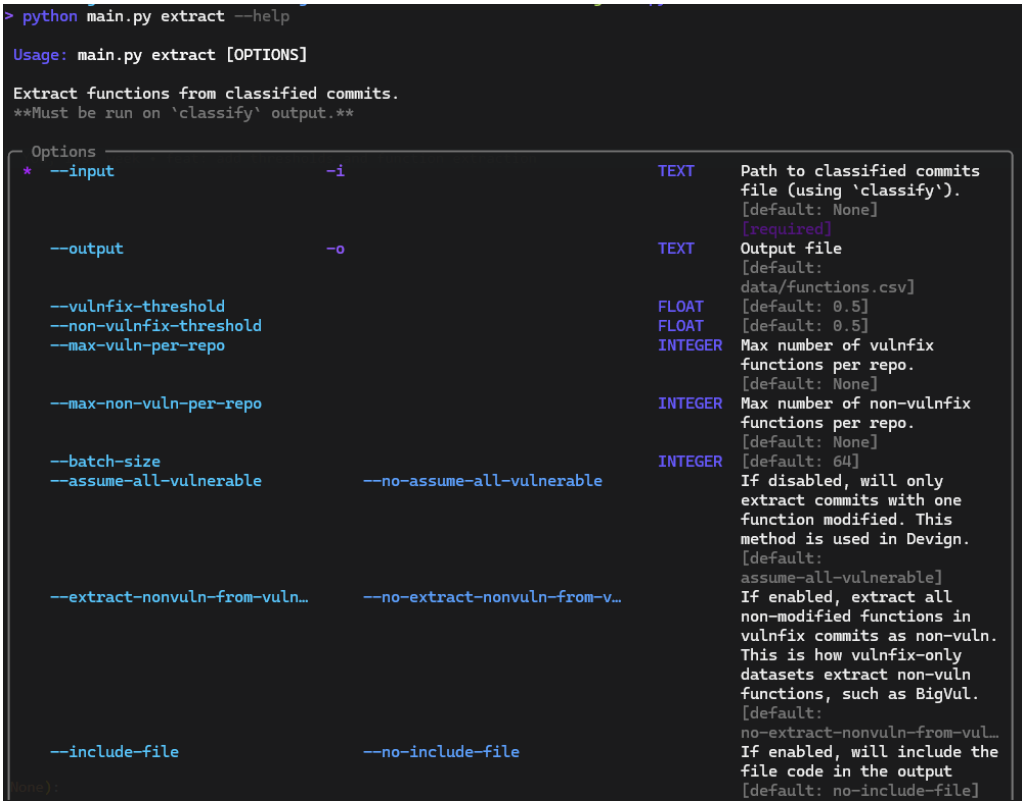
The "extract" command allows for 2 different function extraction methods. By default, it uses the same approach as *Devign* which only extracts modified functions in a commit. Using the "--extract-nonvuln-from-vulnfix' option, the *BigVul* function method

is used which extracts all non-modified functions in a VFC and labels them as non-vulnerable functions.

For more detailed instructions and options on running the function extraction command, the “--help” flag can be used, as shown in Figure 9.

Figure 11

VFC Classifier Prototype Function Extraction Help



Notable Features

This command uses the PyDriller library to extract and parse diffs, source code, and metadata from a commit. This is the same library used in *CVEfixes*.

5.3 Dataset Curation

To demonstrate the effectiveness of dataset curation using LLMs for VD, the team has curated a vulnerable functions dataset using the VFC Classifier Prototype.

More specifically, the team created a list of the top 50 C/C++ repositories sorted by their CVE count using the *CVEfixes* dataset (Refer to Appendix B). Then, this list was fed into the Classifier program and collected 639,278 commits.

Since the team only needed enough functions to double the size of the *Devign* dataset, 24,000 commits were randomly sampled. Then, the *BigVul* function extraction method was run, collecting a total of 503,320 functions. This data is summarised in Table 13.

Table 13
VFCs Curated by Classifier Tool From 50 C/C++ Repositories

No. Projects	VFCs	Non-VFCs	Vuln Funcs	Non-vuln Funcs
50	232,288	406,690	15,305	488,015

VFC classification and function extraction took slightly over 2 hours, and managed to curate a significantly large dataset size compared to *Devign*, which took 600 man-hours of labelling.

To balance the dataset, 15,305 non-vulnerable functions were randomly sampled. Then, these 30,610 functions were then combined with the *Devign* functions dataset.

5.4 Vulnerability Detection

This section will conduct two experiments. The first experiment will aim to demonstrate the effectiveness of datasets curated using DL-based classification of VFCs. The second experiment will evaluate the performance of StarEncoder, an Encoder model, for VD, as well as its performance relative to other similar models and methods.

5.4.1 Evaluation of Dataset Curated using DL

Table 14 contains the evaluation results of the base StarEncoder model trained on the *Devign* dataset. Then, another StarCoder model is trained on the dataset curated by the VFC Classifier Too in the previous section. These models were evaluated on the 10th epoch using the combined functions 10% test split dataset.

These results show that increasing the *Devign* dataset size with the VFC Classifier leads to an overall increase in performance. Therefore, this proves that the team's approach to DL-based dataset curation is effective at improving VD performance.

Table 14
StarEncoder VD Evaluation Results Trained on Curated Dataset

Dataset	Conf. Matrix	Acc	Prec	Recall	F1
			Weighted Avg	Weighted Avg	Weighted Avg
Devign	TP 2,521 FP 1,060 FN 2,442 TN 1,099	50.83	51.00	51.00	49.00
Devign and Curated Dataset	TP 1,782 FP 1,799 FN 826 TN 2,715	**63.14**	**64.00**	**63.00**	**62.00**

5.4.2 Evaluation of StarEncoder

Comparison with Other Models
Table 15 contains the evaluation results of the base StarEncoder model trained on the team's combined functions dataset. Several other models trained by *DiverseVul* are included for comparison. All models use a 80-10-10 train, validation and test split. The results shown are on said test split.

Table 15
StarEncoder VD Evaluation Results Compared with *DiverseVul* Models

Model	Training Dataset	Acc	Prec	Recall	F1
			Weighted Avg	Weighted Avg	Weighted Avg
GPT-2 Base	BigVul, CVEfixes, Devign, ReVeal, CrossVul	91.21	47.14	20.30	28.38
CodeT5 Base		**91.49**	50.53	37.02	42.74
CodeBERT		90.49	42.69	28.15	33.93
StarEncoder	BigVul, CVEfixes, Devign,	89.40	**89.00**	**89.00**	**89.00**

	ReVeal				

Note. Adapted from *DiverseVul* [9].

The results in Table 15 show that although the *DiverseVul* models use very similar training datasets, there is a stark difference in performance as the team's trained model managed to significantly outperform theirs.

This is likely attributed to 3 reasons, the first being that *DiverseVul* uses an extremely imbalanced dataset. The second being that, although not specified in their paper, they might not have deduplicated functions by their labels, which causes vulnerability-introducing non-VFC functions to be present in the dataset. The last is their usage of less SOTA Encoders that were not trained on C/C++

Comparison with Other Methods

Table 16 contains the evaluation results of the team's StarEncoder VD model and Callisto. Both were evaluated on the 10% test split of the combined vulnerable functions dataset.

Table 16

StarEncoder VD Evaluation Results Compared with VD Methods

Model	Conf. Matrix	Acc	Prec	Recall	F1
			Weighted Avg	Weighted Avg	Weighted Avg
Callisto	TP 1545 FP 1025 FN 2036 TN 2516	57.02	58.00	57.00	56.00
StarEncoder	TP 3,209 FP 372 FN 372 TN 3,158	**89.40**	**89.00**	**89.00**	**89.00**

The results shown above further prove the effectiveness of the team's training methodology. In addition, it demonstrates the proficiency of StarEncoder, or SOTA Encoders as a whole, in effective VD.

6 Discussion

6.1 Issues

There were two major issues faced during the project. The first was an issue with the PyDriller library for function extraction.

When randomly sampling functions extracted by PyDriller, the team noticed that some functions did not start or end on the correct lines. Using Pandas to filter, the team discovered that approximately 2% of the functions in the team's dataset curated in Section 5.3 suffered from this error.

Thankfully, the team managed to catch on to this anomalous behaviour before committing to an hour long training session. These data points were removed from the dataset.

The second issue we noticed was the VFC Classification models' inability to discern bug fixes from vulnerability fixes. For instance, the commit message "Fix spelling error in CVE-2022-1038 security patch notes" would be misclassified as a VFC. However, this should be taken with a grain of salt as this commit was one hand crafted to test the VFC models. Additionally, the occurrence of such commits in real-world scenarios is likely very low. With that said, the team's StarEncoder VFC model is still able to discern less nuanced bug fixes and VFCs, such as "Fix spelling error" which is correctly classified as a non-VFC.

6.2 Future Work

6.2.1 Future Plans and Goals

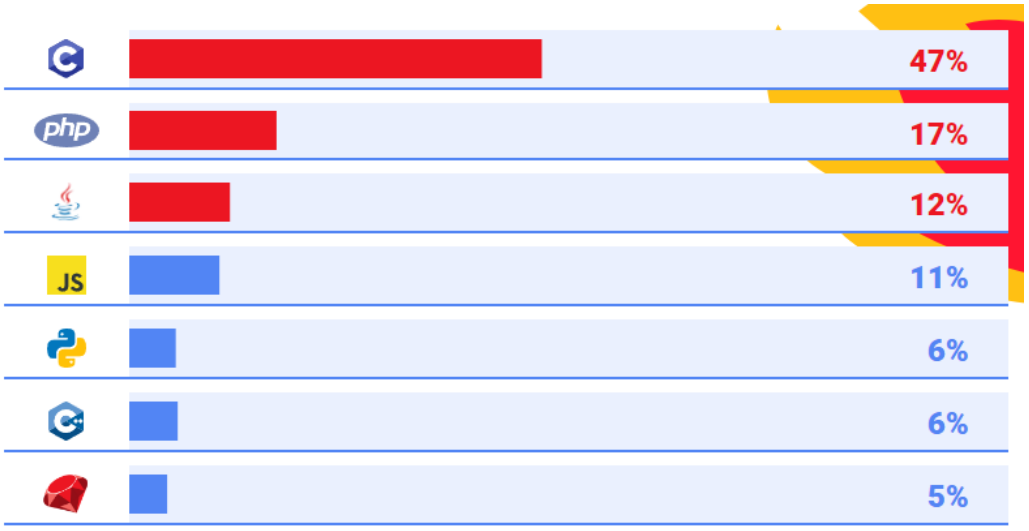
Implement classifier tool for line granularity for VD
The current state of the team's VD model is only able to perform binary classification at function level. By solely providing the prediction of whether the function is vulnerable or not does not offer substantial assistance in identifying the underlying cause of the vulnerability. Thus, further analysis and understanding on the function deemed vulnerable is required by developers to truly identify the root cause to the vulnerability. A solution to eliminate this burden is to enable line granularity on the VD to pinpoint the exact line of code that the model classifies as vulnerable. This can be achieved by modifying the team's existing VFC model to extract code changes made in a vulnerability fix commit. With these commit messages, the team can use it to further train on existing line granularity VD models such as LineVD.

6.2.2 Areas of Improvement

Training VD on more programming languages
Software development today involves a wide variety of programming languages, each with unique characteristics, syntax, and usage scenarios. While C/C++ is about

53% of programming language CVE statistics, the other 47% of programming languages include Python, PHP, Javascript, Java, among others as seen in figure 17. All of these programming languages include its own set of common bugs, and vulnerabilities which can significantly differ from each other.

Figure 17
CVE Count By Language



By expanding the dataset of vulnerable functions, the team's VD models can understand more programming languages, making it more applicable and extensive to more software projects.

Furthermore, diversifying the languages that the tool can handle would enable richer insights and comparisons across the multitude of languages, possibly revealing language specific trends of how bugs are introduced and fixed, leading to a better understanding of software vulnerabilities, aiding in their prevention and resolution.

Real Time Commit Classification

Currently, the team's git commit classification tool needs to run manually in a command line interface where the user can input a list of git repositories URL through a text file or a single URL directly as input. This method, although useful, may not be efficient. It would be more efficient to transition into a real-time system that classifies each commit as it is being pushed into each repository. With this system in place, developers could receive notifications regarding potential bugs or vulnerabilities within their code immediately after making the commit, facilitating prompt rectification.

Integration into development platforms

Numerous developers utilise **I**ntegrated **D**evelopment **E**nvironments (IDEs) such as Visual Studio Code or Eclipse IDE or source control management platforms like GitHub. The further development of plugins or extensions that could be integrated directly onto these platforms would enhance the accessibility, usability and ease of use of the team's tools.

For example, a plugin could be designed to automatically activate the bug finder on the currently opened code file in the IDE. Alternatively, a GitHub integration might be developed to automatically classify each new commit as it is pushed into the repository. Such integrations would provide a more convenient and timely means of detecting bugs early in the development process.

6.3 Applications

The ground-truth VFC and vulnerable function datasets the team had manually labelled, and collated from past works, can be built upon by future researchers or provide a strong foundation for security-related works.

DL-based commit classification allows for higher accuracy, faster labelling, larger dataset sizes, and increased diversity. Therefore, the commit classifier the team has developed can supplement a wide range of security-related research, not limited to Vulnerability Detection (VD). For instance, training vulnerability patching models, creating a security benchmark for evaluating VD models, summarisation and explanation of vulnerable code, and more. In our case, the team applied DL-based commit classification to curate a dataset of vulnerable functions, and improved the accuracy of a VD model.

Whilst the VD models created by the team or from past works may not be effective now, as the performance of DL models improve, using them for VD becomes more and more viable. This will be revolutionary for software development security as it requires lower costs, no human training, and can quickly conduct secure code reviews. Additionally, the trend of ML democratisation will make these models widely available, globally improving software security assurance.

Through the use of a git commit classifier, there are many potential use cases that can be applied. The following include some of the possible use cases of software developers, release managers and other users when using the git commit classifier.

Software developers are able to carry out impact analysis. This enables developers to understand the areas of the codebase that have undergone modification to address bugs or security threats. By analysing the impact of bug fix commits, teams can better assess potential implications on related functionalities and allocate resources for regression testing or further development.

Release managers can utilise this bug fix commit classifier to automatically identify bug fix commits, allowing for the generation of accurate, comprehensive and detailed release notes for the potential users of the software. Allowing users and stakeholders to understand the improvements and fixes introduced in each version release.

Project managers are able to make metrics and conduct performance analysis of the software developer team that are related to bug fixes. Through the automated classification of bug fix and non bug fix commits, the classifier enables tracking and

monitoring of metrics related to bug fix commits. Many types of metrics can be curated, such as number of bug fixes or stability of the software over time, providing insights into the effectiveness of the bug fixing process and help drive continuous improvement efforts.

After the use of the bug fix commit classifier, quality assurance and testing efforts can be directed to verifying the identified bug fixes, ensuring the identified issues have been resolved and minimising the potential for regression.

If an organisation were to carry out transfer learning on the git commit classifier model to not only classify bug fixes but other types of git commits, there would be a larger variety of potential use cases that can be applied.

7 Conclusion

This research project has shown that LLMs are incredibly versatile. Not only can they be used for VD, but they can also be used to curate datasets to train themselves.

Using the VFC Classifier, thousands of accurately labelled VFCs and vulnerable functions can be extracted within a few minutes.

The team demonstrated that the use of SOTA Encoder models for VD, combined with class balancing and deduplication allow for performance that far exceeds existing implementations.

The findings in this project can be used to supplement VD-related research by streamlining dataset curation.

References

[1] A. Das, “Guess Who Contributed the Most to Linux Kernel 5.10 Development? It’s Huawei (and Intel),” It’s FOSS News, Jan. 06, 2021. Accessed: Aug. 16, 2023. [Online]. Available: https://news.itsfoss.com/huawei-kernel-contribution/

[2] Y. Zhou, S. Liu, J. Siow, X. Du, and Y. Liu, “Devign: Effective Vulnerability Identification by Learning Comprehensive Program Semantics via Graph Neural Networks.” arXiv, 2019. doi: 10.48550/ARXIV.1909.03496.

[3] D. Hin, A. Kan, H. Chen, and M. A. Babar, “LineVD: Statement-level Vulnerability Detection using Graph Neural Networks.” arXiv, 2022. doi: 10.48550/ARXIV.2203.05181.

[4] X.-C. Wen, Y. Chen, C. Gao, H. Zhang, J. M. Zhang, and Q. Liao, “Vulnerability Detection with Graph Simplification and Enhanced Graph Representation Learning.” arXiv, 2023. doi: 10.48550/ARXIV.2302.04675.

[5] A. Ahmed, A. Said, M. Shabbir, and X. Koutsoukos, “Sequential Graph Neural Networks for Source Code Vulnerability Identification.” arXiv, 2023. doi: 10.48550/ARXIV.2306.05375.

[6] “Most secure programming languages,” Mend, https://www.mend.io/most-secure-programming-languages/ (accessed Aug. 16, 2023).

[7] R. L. Russell et al., “Automated Vulnerability Detection in Source Code Using Deep Representation Learning.” arXiv, 2018. doi: 10.48550/ARXIV.1807.04320.

[8] A. Vaswani et al., “Attention Is All You Need.” arXiv, 2017. doi: 10.48550/ARXIV.1706.03762.

[9] Y. Chen, Z. Ding, L. Alowain, X. Chen, and D. Wagner, “DiverseVul: A New Vulnerable Source Code Dataset for Deep Learning Based Vulnerability Detection.” arXiv, 2023. doi: 10.48550/ARXIV.2304.00409.

[10] G. P. Bhandari, A. Naseer, and L. Moonen, “CVEfixes: Automated Collection of Vulnerabilities and Their Fixes from Open-Source Software,” arXiv, 2021, doi: 10.48550/ARXIV.2107.08760.

[11] L. Jiang and W. Yang, “Understanding deep learning on controlled noisy labels,” Google Research Blog, https://ai.googleblog.com/2020/08/understanding-deep-learning-on.html (accessed Aug. 16, 2023).

[12] T. Boland and P. E. Black, “Juliet 1.1 C/C++ and Java Test Suite,” Computer, vol. 45, no. 10. Institute of Electrical and Electronics Engineers (IEEE), pp. 88–90, Oct. 2012. doi: 10.1109/mc.2012.345.

[13] A. Wagner and J. Sametinger, “Using the Juliet Test Suite to Compare Static Security Scanners,” Proceedings of the 11th International Conference on Security and Cryptography. SCITEPRESS - Science and and Technology Publications, 2014. doi: 10.5220/0005032902440252.

[14] A. Kaur and R. Nayyar, “A Comparative Study of Static Code Analysis tools for Vulnerability Detection in C/C++ and JAVA Source Code,” Procedia Computer Science, vol. 171. Elsevier BV, pp. 2023–2029, 2020. doi: 10.1016/j.procs.2020.04.217.

[15] X. Nie, H. Wei, L. Chen, Z. Zhang, Y. Zhang, and G. Shi, “MVDetecter: Vulnerability Primitive-based General Memory Vulnerability Detection,” 2022 IEEE Intl Conf on Parallel & Distributed Processing with Applications, Big Data & Cloud Computing, Sustainable Computing & Communications, Social Computing & Networking (ISPA/BDCloud/SocialCom/SustainCom). IEEE, Dec. 2022. doi: 10.1109/ispa-bdcloud-socialcom-sustaincom57177.2022.00056.

[16] P. E. Black, “Juliet 1.3 test suite: changes from 1.2,” National Institute of Standards and Technology, Jun. 2018. doi: 10.6028/nist.tn.1995.

[17] Y. Zhou and A. Sharma, “Automated identification of security issues from commit messages and bug reports,” Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering. ACM, Aug. 21, 2017. doi: 10.1145/3106237.3117771.

[18] A. Tanwar, K. Sundaresan, P. Ashwath, P. Ganesan, S. K. Chandrasekaran, and S. Ravi, “Predicting Vulnerability In Large Codebases With Deep Code Representation.” arXiv, 2020. doi: 10.48550/ARXIV.2004.12783.

[19] S. Page, “Analysing linux kernel commits,” sam4k, https://sam4k.com/analysing-linux-kernel-commits/ (accessed Aug. 16, 2023).

[20] A. Murray, “The National Vulnerability Database explained,” Mend, https://www.mend.io/blog/the-national-vulnerability-database-explained/ (accessed Aug. 16, 2023).

[21] T. Beardsley, “The hidden harm of silent patches,” Rapid7 Blog, https://www.rapid7.com/blog/post/2022/06/06/the-hidden-harm-of-silent-patches/ (accessed Aug. 16, 2023).

[22] J. Fan, Y. Li, S. Wang, and T. N. Nguyen, “A C/C++ Code Vulnerability Dataset with Code Changes and CVE Summaries,” Proceedings of the 17th International Conference on Mining Software Repositories. ACM, Jun. 29, 2020. doi: 10.1145/3379597.3387501.

[23] X. Wang, S. Wang, K. Sun, A. Batcheller, and S. Jajodia, “A Machine Learning Approach to Classify Security Patches into Vulnerability Types.” GMU, 2020. https://csis.gmu.edu/ksun/publications/CNS20_PatchByType.pdf

[24] J. Y. D. Lee and H. L. Chieu, “Co-training for Commit Classification,” Proceedings of the Seventh Workshop on Noisy User-generated Text (W-NUT 2021). Association for Computational Linguistics, 2021. doi: 10.18653/v1/2021.wnut-1.43.

[25] X. Wang et al., “PatchRNN: A Deep Learning-Based System for Security Patch Identification,” arXiv, 2021, doi: 10.48550/ARXIV.2108.03358.

[26] S. Chakraborty, R. Krishna, Y. Ding, and B. Ray, “Deep Learning based Vulnerability Detection: Are We There Yet?” arXiv, 2020. doi: 10.48550/ARXIV.2009.07235.

[27] E. Strubell, A. Ganesh, and A. McCallum, “Energy and Policy Considerations for Deep Learning in NLP.” arXiv, 2019. doi: 10.48550/ARXIV.1906.02243.

[28] R. Li et al., “StarCoder: may the source be with you!” arXiv, 2023. doi: 10.48550/ARXIV.2305.06161.

[29] V. Sanh, L. Debut, J. Chaumond, and T. Wolf, “DistilBERT, a distilled version of BERT: smaller, faster, cheaper and lighter.” arXiv, 2019. doi: 10.48550/ARXIV.1910.01108.

[30] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, “BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding.” arXiv, 2018. doi: 10.48550/ARXIV.1810.04805.

[31] T. Schopf, D. Braun, and F. Matthes, “Evaluating Unsupervised Text Classification: Zero-shot and Similarity-based Approaches,” arXiv, 2022, doi: 10.48550/ARXIV.2211.16285.

Appendix

Appendix A

VFC Keywords

Figure A1
Keywords used in Devign

Types	Keywords List
General	*out of bound, use after free, double free, divide by zero, overflow, illegal, leak, disclosure, improper, unexpected, sanity check, uninitialize, fail, null pointer dereference, null function pointer, crash, corrupt, deadlock, race condition, denial of service, CVE, exploit, attack, vulnerable, fuzz, verify, security issue/problem/fix, privilege, malicious, undefined behavior, exposure, remote code execution, open redirect, OSVDB, ReDoS, NVD, clickjack, man-in-the-middle, hijack, advisory, insecure, cross-origin, unauthorized, infinite loop, authentication, brute force, bypass, crack, credential, hack, harden, injection, lockout, password, proof of concept, poison, privilege, spoof, compromise, valid, out of array, exhaust, off-by-one, privesc, bugzilla, limit, craft, overrun, overread, override, replay, constant time, mishandle, underflow, violation, recursion, snprintf, initialize, prevent, guard, protect, cross site scripting, directory traversal, cross site request forgery, XML External Entity, session fixation.*
Library-Specific	*KASAN, general protection fault (GPF), oops, panic, syzkaller, trinity, grsecurity, vsecurity, oss-security*

Note. From *Devign*, Y. Zhou., 2019.

Figure A2

Keywords Used in This Project

```
keywords = [
    re.compile(r"stack(?:-based)?[\s-](?:overflow|buffer overflow)", re.IGNORECASE),
    re.compile(r"heap(?:-based)?[\s-](?:overflow|buffer overflow)", re.IGNORECASE),
    re.compile(r"(?:buffer|integer)(?:-based)?[\s-]?overflow", re.IGNORECASE),
    re.compile(
        r"\b(?:oob|uaf|mitm|xss|csrf|rce|ddos|dos|sqli|lfi|rfi|ssrf|xxe|cve|cwe|cpe|cvss|leak|lpe)\b",
        re.IGNORECASE,
    ),
    re.compile(r"use[\s-]after[\s-]free", re.IGNORECASE),
    re.compile(r"out[\s-]of[\s-]bounds?[\s-]?(?:read|write)?", re.IGNORECASE),
    re.compile(r"double[\s-]free", re.IGNORECASE),
    re.compile(r"uninitiali(?:z|s)ed[\s-]?(?:bytes?)?", re.IGNORECASE),
    re.compile(
        r"uninitiali(?:z|s)ed[\s-](?:bytes?|var|stack|buffer|memory)", re.IGNORECASE
    ),
    re.compile(r"race[\s-]condition", re.IGNORECASE),
    re.compile(r"\bvulnerable\b", re.IGNORECASE),
    re.compile(r"\bexploit\b", re.IGNORECASE),
    re.compile(r"\bcorrupt(?:ion)?\b", re.IGNORECASE),
    re.compile(r"privilege[\s-]escalation", re.IGNORECASE),
    re.compile(r"\bprivesc\b", re.IGNORECASE),
    re.compile(r"\battack(?:er)?\b", re.IGNORECASE),
    re.compile(r"\bhacker\b", re.IGNORECASE),
    re.compile(r"\bhijack\b", re.IGNORECASE),
    re.compile(r"divide[\s-]by[\s-]zero", re.IGNORECASE),
    re.compile(r"\billegal\b", re.IGNORECASE),
    re.compile(r"\bimproper\b", re.IGNORECASE),
    re.compile(r"\bunexpected\b", re.IGNORECASE),
    re.compile(r"\bfail\b", re.IGNORECASE),
    re.compile(r"\bcorrupt\b", re.IGNORECASE),
    re.compile(r"\bcrash\b", re.IGNORECASE),
    re.compile(r"\bdeadlock\b", re.IGNORECASE),
    re.compile(r"denial[\s-]of[\s-]service", re.IGNORECASE),
    re.compile(r"null[\s-]pointer", re.IGNORECASE),
    re.compile(
        r"\b(?:improper|unexpected|fail|null[\s-]pointer[\s-]dereference|null[\s-]function[\s-]pointer|
        re.IGNORECASE,
    ),
    re.compile(r"sanity[\s-]check", re.IGNORECASE),
]
```

Appendix B

Dataset Processing Details

CVEfixes SQL Queries

Figure B1

Extract commits from CVEfixes

```
SELECT *
FROM commits AS c
JOIN repository as r ON c.repo_url = r.repo_url
JOIN file_change as fc ON c.hash = fc.hash
WHERE (fc.programming_language = 'C' OR fc.programming_language =
'C++')
```

Figure B2

Extract functions from CVEfixes

```
SELECT *
FROM commits AS c
JOIN file_change AS fc ON c.hash = fc.hash
JOIN method_change AS mc on fc.file_change_id = mc.file_change_id
WHERE (fc.programming_language = 'C' OR fc.programming_language =
'C++')
```

Figure B3

Extract C/C++ repositories and their CVE count

```
SELECT r.repo_url,
r.repo_language,
COUNT(DISTINCT c.hash) AS num_cves
FROM repository AS r
JOIN commits AS c on r.repo_url = c.repo_url
JOIN fixes AS f ON c.hash = f.hash
WHERE r.repo_language IN ('C', 'C++')
GROUP BY r.repo_url, r.repo_language
ORDER BY num_cves DESC;
```

Dataset Merging

Figure B4 and B5 shows the preprocessing steps done for *BigVul*: renaming columns and cleaning data. The same steps were repeated for all other datasets used.

Figure B4

BigVul Preprocessing Steps for Commits

```
bigvul_commits = pd.read_csv("data/all_c_cpp_release2.0.csv")
bigvul_commits  =  bigvul_commits[["commit_message",  "commit_id",
"project"]]
bigvul_commits["source"] = "bigvul"
bigvul_commits["labels"] = 1
bigvul_commits = bigvul_commits.rename(
    columns={
        "commit_id": "commit_hash",
        "commit_message": "commit_msg",
    }
)
bigvul_commits["project"] = bigvul_commits["project"].str.lower()
bigvul_commits = bigvul_commits.dropna(subset=["commit_msg"])
```

Figure B5

BigVul Preprocessing Steps for Functions

```
bigvul_functions = df[
    [
        "func_before",
        "commit_id",
        "codeLink",
        "project",
        "vul",
    ]
]
bigvul_functions["source"] = "bigvul"
bigvul_functions = bigvul_functions.rename(
    columns={
        "func_before": "function",
```

```
        "commit_id": "commit_hash",
        "codeLink": "commit_url",
        "vul": "labels",
    }
)
bigvul_functions["project"] =
bigvul_functions["project"].str.lower()
bigvul_functions["repo_url"] = None
bigvul_functions["commit_date"] = None
bigvul_functions=bigvul_functions.dropna(subset=["function",
"commit_hash"])
```

After merging, the team deduplicate the commit datasets by their commit hash. For functions, the team deduplicate according to the function code and commit hash.

Figure B6

Dataset Merging and Deduplication

```
combined_commits = pd.concat([bigvul_commits, devign_commits,
cvefixes_commits, our_commits])
combined_commits =
combined_commits.drop_duplicates(subset=["commit_hash"])

combined_functions = pd.concat([bigvul_functions,
devign_functions, cvefixes_functions, reveal_functions])
combined_functions =
combined_functions.drop_duplicates(subset=["function",
"commit_hash"])
```

Additionally, the team remove functions which have two different labels. This is because some feature-adding commits may have added vulnerabilities, which are then patched by a VFC. This results in duplicate functions with different labels.

Figure B7

Remove duplicate functions with different labels

```
def remove_duplicates(x):
    return not x['labels'].nunique() > 1

combined =
combined.groupby('function').filter(filter_duplicates).shape
```