UNIVERSITÀ DEGLI STUDI DI SALERNO

Dipartimento di Informatica

Corso di Laurea in Informatica

TESI DI LAUREA

Identificazione a Grana Fine di Vulnerabilità Software in Codice Python

RELATORE

Prof. Fabio Palomba

Dott. Emanuele Iannone

Università degli Studi di Salerno

CANDIDATO

Raffaele Aviello

Matricola: 0512110529

Anno Accademico 2022-2023

Questa tesi è stata realizzata nel

Alla mia famiglia

Abstract

L'ampia diffusione e utilizzo dei software hanno portato alla luce quanto siano importanti gli aspetti relativi alla loro sicurezza e per garantire ciò è indispensabile usare tecniche di identificazione di vulnerabilità. Esistono diversi approcci a questo problema, ma tra gli esempi più interessanti spiccano le tecniche di identificazione di vulnerabilità a grana fine che sfruttano modelli di machine learning. In questo contesto le tecniche basate su criteri a grana fine permettono l'identificazione delle righe di codice vulnerabile, semplificando di molto il compito di localizzare le vulnerabilità del codice. Esistono numerosi esempi in letteratura di approcci che analizzano codice scritto in linguaggio C, tuttavia non sono altrettanto numerosi quelli che permettono l'identificazione di codice Python vulnerabile. L'obiettivo di questo studio è di modificare due modelli già esistenti: LineVul e LineVd. I due modelli, che analizzano codice sorgente scritto in C, sono stati modificati per permettere l'identificazione di codice Python vulnerabile. Per addestrare e testare i due modelli, modificati a partire da quelli originali, sono stati creati due dataset di funzioni Python, compatibili con la logica di funzionamento di ogni modello, contenenti diverse informazioni relative alle righe di codice vulnerabili, come la posizione di una riga vulnerabile all'interno di una funzione o le differenze tra i due commit relativi alla risoluzione di una vulnerabilità. I due nuovi modelli sono stati valutati su due dataset, che constano rispettivamente di millenovecentosessantanove e milleduecentodieci funzioni, esibendo prestazioni piuttosto distanti da quelle dei due modelli originali. I modelli PyLineVul e PyLineVd hanno ottenuto rispetto alla metrica F1 score, per l'identificazione di vulnerabilità a livello di funzione, i seguenti punteggi: 0.42 e 0.19. Inoltre il modello PyLineVd riesce ad essere più efficiente rispetto al modello originale, in fase di predizione a livello di riga, avendo ottenuto una Top-5 Accuracy pari a 1. Tenendo conto dei risultati ottenuti, questo studio dimostra la possibilità di poter convertire approcci per l'identificazione di vulnerabilità a grana fine per codice C, in approcci in grado di operare anche su codice Python.

Indice

Elenco delle figure

Elenco delle tabelle

CAPITOLO 1

Introduzione

1.1 Contesto Applicativo

I software attualmente sono presenti in numerosi aspetti della vita quotidiana e risultano essenziali anche in contesti più delicati: ospedali, cliniche, banche e simili usano quotidianamente software di vario genere. In questo contesto emerge la necessità di garantire sicurezza agli utenti che utilizzano software, identificandone le vulnerabilità. Una vulnerabilità software è una debolezza presente nel codice sorgente di un software, che può essere sfruttata da parte di un attaccante per arrecare danni a un sistema informatico [3], tipicamente ottenendo accesso illecito a informazioni riservate, cambiando privilegi di accesso a oppure interferendo con le operazioni di un sistema. Come indicato da Li et al. [4], per mitigare la presenza di vulnerabilità, esistono principalmente due approcci al problema: tool di analisi del codice e tecniche di identificazione di vulnerabilità basate su modelli di machine learning. Entrambi gli approcci al problema possono usare due criteri, che riguardano la portata dell'identificazione di vulnerabilità: a livello di funzione o grana grossa (coarse grained) e a livello di riga o grana fine (fine grained). Le tecniche basate sull'identificazione di vulnerabilità a livello di funzione indicano se una funzione sia vulnerabile o meno, mentre quelle basate su criteri a livello di riga indicano quale riga di codice presenta

delle vulnerabilità. Lo studio è incentrato sull'identificazione di vulnerabilità software e dati i diversi approcci esistenti per svolgere questo compito, si è scelto di concentrarsi su uno tra quelli che risultano più interessanti rispetto a, come indicato da Cass [5], uno dei linguaggi di programmazione più utilizzati: l'identificazione di vulnerabilità software con modelli di machine learning, in relazione agli approcci a grana fine per codice scritto in Python.

1.2 Motivazioni e Obiettivi

Lo studio nasce con la necessità di esplorare gli approcci di identificazione vulnerabilità di codice scritto in Python laddove non ne esistono numerosi esempi, infatti la maggior parte dei framework utilizzati per l'identificazione di vulnerabilità software analizzano codice scritto generalmente in C o C++, come BVDetector [6] che opera a livello di program slice o l'approccio presentato da Jimenez et al. [7], che opera a livello di file. Con questa motivazione si definiscono conseguentemente gli obiettivi dello studio: verificare se sia possibile convertire due approcci già esistenti, LineVul [1] e LineVd [2], in approcci in grado di operare su codice Python e valutarne i risultati.

1.3 Risultati Ottenuti

Dopo aver costruito i due dataset per addestrare e testare i modelli PyLineVul e PyLineVd, opportunamente modificati per andare ad operare su codice Python, sono stati esaminati i risultati del lavoro svolto. I risultati ottenuti evidenziano il fatto che sia senza dubbio possibile utilizzare approcci già esistenti per funzionare anche rispetto a codice Python, raggiungendo risultati inferiori nel caso di predizione a livello di funzione, con degli F1 score pari a 0.42 e 0.19. Nonostante l'inferiorità dei modelli PyLineVul e PyLineVd a livello di funzione, il modello PyLineVd riesce ad essere più efficace del modello originale a livello di riga, totalizzando una Top-5 Accuracy pari a 1.

1.4 Struttura della Tesi

La tesi è divisa nei seguenti capitoli: Background e stato dell'arte, Metodologia, Risultati e Conclusioni. Il capitolo Background e stato dell'arte, illustra i concetti che hanno costituito le fondamenta del lavoro svolto e descrive i lavori già presenti in letteratura. Il capitolo Metodologia descrive il lavoro svolto, illustrando nel dettaglio l'approccio e gli strumenti utilizzati per condurre la ricerca. Il capitolo Risultati descrive i risultati ottenuti con questo studio e in che modo si collegano agli obiettivi posti inizialmente. Infine nel capitolo Conclusioni viene fatto un riassunto del lavoro svolto e descritti i possibili sviluppi futuri di questo.

CAPITOLO 2

Background e Stato dell'Arte

2.1 Difetti e Vulnerabilità

Un bug in un sistema software, come indicato da Bruegge e Detoit [8], è la causa algoritmica di uno stato di errore, che ad ogni ulteriore elaborazione porta il sistema software ad un fallimento, tipicamente generando comportamenti inattesi o risultati errati. In questo contesto, come delineato da Dempsey et al. [3], una vulnerabilità software è una falla di sicurezza presente nel codice sorgente di un software, che viene generata da uno o più bug e che può essere sfruttata da un attaccante, un individuo o organizzazione che agisce in modo malevolo in relazione ad un sistema informatico. Quando si verifica un attacco a un sistema informatico, sfruttando una o più vulnerabilità di questo, generalmente si vanno a colpire uno o più dei tre componenti cardine della triade CIA: confidenzialità, integrità e disponibilità dei dati. Come illustrato da Irwin [9], per confidenzialità dei dati si intende mantenere al sicuro informazioni sensibili, mentre per quanto riguarda l'integrità dei dati si intende assicurare la loro correttezza e infine la disponibilità dei dati rappresenta la possibilità di accedere a questi quando necessario.

2.2 Identificazione di Vulnerabilità

L'identificazione di vulnerabilità software è fondamentale nel prevenire attacchi informatici e garantire la sicurezza di un sistema software ed è perciò un compito gravoso, che può essere facilitato grazie all'uso di diversi strumenti, come tool di analisi di codice sorgente. Tra gli strumenti a disposizione per svolgere questo compito, quelli più interessanti risultano essere gli approcci basati su modelli di machine learning / deep-learning. Questi approcci utilizzano degli snippet di codice, ottenuti in maggior parte da progetti open-source, da cui si possono ottenere diversi tipi di rappresentazioni del codice che, come affermano Nima et al. [10], possono basarsi su grafi, alberi oppure rappresentazioni vettoriali. Queste rappresentazioni vengono poi trasformate in embedding: rappresentazioni numeriche dei dati, che possono essere comprese dai modelli di machine learning / deep-learning per effettuare la classificazione.

Come delineato da Koehrsen [11], gli embedding permettono di rappresentare variabili categoriche discrete tramite vettori continui, consentendo di ridurre la dimensionalità delle variabili senza perdere informazioni significative. Rispetto ad una pratica come il feature engineering, che rappresenta le caratteristiche dei dati solo a livello numerico, gli embedding permettono di mantenere informazioni strutturate e significative che, come illustrato da Dolphin [12], richiedono molte risorse a livello computazionale e possono essere di difficile interpretazione rispetto a delle feature tradizionali. Gli embedding, tipicamente, vengono generati grazie a modelli che effettuano task di NLP (Natural Language Processing), come ad esempio BERT [13]: un modello di machine learning basato su trasformatori, una architettura di deep-learning che sfrutta il meccanismo della self attention, che permette di pesare in maniera differente ogni parte di un input che viene ricevuto. Al modello BERT, basta aggiungere un layer finale, opportunamente adattato in base al task che si deve svolgere, per poter creare nuovi modelli che possono raggiungere prestazioni simili ai modelli state-of-the-art, proprio come CODEBERT [14]: un modello bimodale, che opera sia su linguaggio naturale che su linguaggi di programmazione e che viene utilizzato in diversi framework di identificazione di vulnerabilità software per generare embedding di codice. Come si evince dalla figura 2.1: una volta generati gli

embedding di codice, questi passando attraverso i layer del modello (che variano di tipologia in base alle logiche operative del framework) subiscono delle operazioni, che tipicamente mirano ad analizzare la relazione tra una funzione vulnerabile e quella delle righe vulnerabili, per poter classificare le righe di codice.

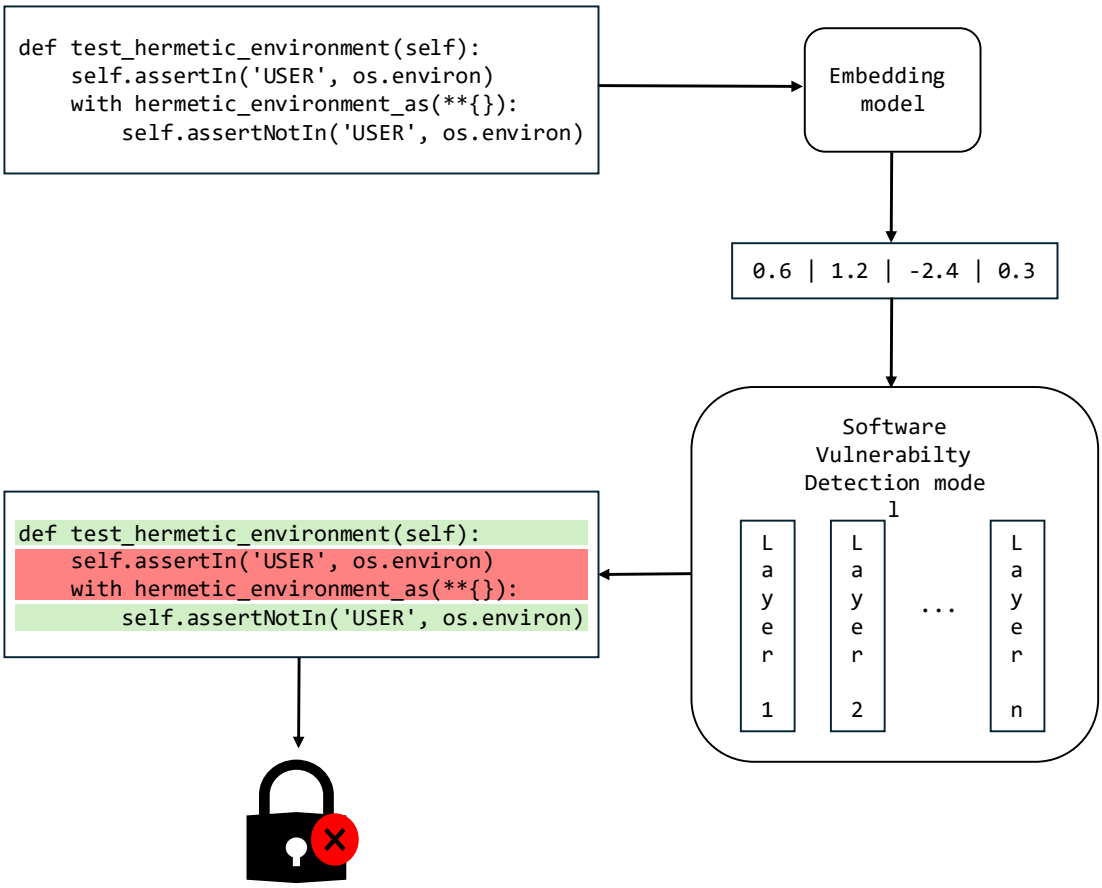

Figura 2.1: Processo di identificazione delle righe vulnerabili di una funzione

2.3 Lavori Correlati

Come mostrato da Croft et al. [15], oltre alle differenze rispetto alle logiche di funzionamento di ciascun modello, questi si differenziano soprattutto per il livello di granularità a cui operano: component, file, funzione, program slice, commit e riga. Nelle sottosezioni seguenti vengono discussi alcuni approcci che operano ai livelli di granularità citati precedentemente.

2.3.1 Tecniche a Grana Grossa

Come definito da Bruegge e Detoit [8], un component è un'istanza autonoma di una o più classi, che definiscono un insieme di operazioni correlate, per formare applicazioni complete. I modelli che operano a questo livello di granularità vanno quindi ad analizzare uno o più file. Scandariato et al. [16] hanno presentato uno studio in cui si utilizzavano diversi tipi di modelli di apprendimento in combinazione con tecniche di text mining. Il framework inizia ad operare rappresentando i dati, in questo caso un file sorgente Java, con la tecnica bag-of-words, in cui il file viene rappresentato come una serie di termini, a cui è associato il numero di volte che compare nel testo. I termini e le frequenze vengono utilizzati come feature, per poi essere sottoposti ad una discretizzazione, trasformando il range numerico delle frequenze in classi nominali. Nonostante si discutano le performance di diversi modelli, quelli che hanno ottenuto risultati migliori sono il Naive Bayes e il Random Forest. Il modello Naive Bayes si basa sull'idea che la presenza di una feature in una classe, non sia correlata alla presenza di altre feature. Il modello Random Forest, invece, utilizza degli alberi decisionali: dei grafi che rappresentano una decisione e i suoi possibili esiti. Il modello si basa sull'idea che che generando diversi alberi decisionali indipendenti tra di loro, il modello possa fornire predizioni più accurate se questi venissero combinati.

Un esempio di modello che opera a livello di file è quello presentato da Jimenez et al. [7]: una replica di un approccio precedentemente sviluppato, modificato per andare a ricercare vulnerabilità software rispetto ai file che compongono il kernel Linux. Nello specifico si va ad analizzare una variante tra quelle proposte, in cui l'approccio utilizzato è basato sull'intuizione che i file vulnerabili condividono degli elementi comuni: import e chiamate a funzione. Gli elementi citati precedentemente rappresentano le feature che vengono utilizzate dal modello, che vengono estratte dai file grazie agli Abstract Syntax Tree (AST), una struttura dati che rappresenta la struttura di un software in maniera gerarchica, generati dai file grazie al software Joern[1]. Una volta estratte le feature, queste vengono rappresentate tramite una matrice delle feature, le cui colonne rappresentano le feature e le cui righe rappresentano i

[1] https://github.com/joernio/joern

file da classificare. La classificazione viene effettuata da un modello Support Vector Machine che sfrutta un kernel lineare, il quale risulta adatto per il task in questione perché il dataset è linearmente separabile, ossia i dati possono essere divisi da una linea che divide l'insieme dei dati in esattamente in due sottoinsiemi.

Un approccio che opera al livello di funzione è quello proposto da Lin et al. [17]: il framework rappresenta il codice tramite embedding, generati a partire dagli AST delle funzioni, classificandolo grazie a una rete LSTM bidirezionale. Il framework opera su un dataset di funzioni, costruito manualmente dagli autori dello studio. Per prima cosa il framework genera le feature a partire da codice sorgente, opportunamente etichettato: data una funzione, si crea l'AST della funzione per poi serializzarlo in una frase che contiene i suoi elementi. Una volta ottenute tutte le rappresentazioni delle funzioni tramite AST, queste vengono rese della stessa lunghezza e poi ulteriormente convertite in embedding. Gli embedding a questo punto vengono usati per allenare una rete neurale bidirezionale Long short-term memory. Una volta allenata la rete neurale si può procedere con l'effettivo processo di classificazione. Dando in input al modello del codice sorgente non etichettato, questo viene convertito in embedding come illustrato precedentemente, per individuare le vulnerabilità tramite un classificatore, opportunamente allenato con un sottoinsieme delle rappresentazioni generate dal codice sorgente non etichettato.

Tian et al. [6], propongono una tecnica che opera al livello di program slice: BVDetector. Gli autori dello studio definiscono una program slice come una sequenza di righe di codice assembly, che hanno la caratteristica di essere legate a dipendenze dei dati o di controllo. Il framework proposto opera in due fasi: fase di allenamento e fase di identificazione. Nella prima fase, vengono estratte delle program slices da programmi in formato binario, per poi aggiungere una label ad ogni program slice. Le label vengono aggiunte ad una program slice effettuando un confronto con una libreria. Questa libreria, costruita dagli autori dello studio, contiene informazioni riguardanti le vulnerabilità, ottenute da programmi C/C++ che sono notoriamente vulnerabili. Dopo essere state etichettate, le program slices vengono convertite in vettori per allenare il modello, nello specifico le program slices vengono divise, per poi rappresentare ogni parola con un token. Questi token vengono usati per allenare un modello word2vec, in modo da poter creare degli embedding. Gli embedding

vengono utilizzati per allenare un modello Gated Recurrent Unit bidirezionale, simile ad una rete neurale Long short-term memory, ma preferita rispetto a quest'ultima per la maggiore semplicità di configurazione. Infine, nella fase di identificazione il modello effettua la classificazione di programmi C/C++ in formato binario.

2.3.2 Tecniche a Grana Fine

Una tecnica che opera al livello di commit è quella presentata da Perl et al. [18]: VCCFinder, un tool per segnalare commit che contengono codice vulnerabile. I commit sono delle modifiche apportate ad un software, tramite un software di versioning: GIT, che permette di tenere traccia di queste modifiche. In virtù del fatto che, spesso, le modifiche di un software vengono impegate per risolvere delle vulnerabilità, alcune tecniche vanno ad operare al livello di granularità relativo ai commit effettuati. Gli autori hanno costruito un dataset contenete numerosi esempi di commit vulnerabili e non, ottenuti dal servizio di hosting di progetti software GITHUB, basato su GIT. Il tool si basa sul concetto di modifica di una riga: se una riga viene modificata, probabilmente questa contribuiva alla vulnerabilità del software. Le feature che vengono utilizzate dal tool, sono state scelte in modo da poter differenziare efficacemente ciascun commit e sono basate sui metadati del progetto GITHUB (numero di stelle del progetto, numero di commit effettuati, numero di modifiche e simili). Le feature vengono rappresentate tramite un modello che utilizza la generalized-bag-of-word representation: ogni commit viene fatto corrispondere ad una funzione. Questa funzione prende in input un commit e un token, che rappresenta una parola chiave, identificatore o testo contenuto in un commit. La funzione restituisce una variabile booleana che indica se il token appartiene al commit o meno. Dato l'alto numero delle feature, il modello scelto è il Support Vector Machine con un kernel lineare. Il modello classifica i commit in ordine di vulnerabilità, dopo aver calcolato dei punteggi per ogni commit.

Oltre alle tecniche LineVul [1] e LineVd [2], che vengono mostrate in dettaglio nella sezione 3.2 in quanto oggetto di studio, un'altra tecnica che effettua predizioni a livello di riga è quella presentata da Mosolygó et al. [1]: un modello che effettua predizioni a livello di riga su codice JavaScript. La tecnica si basa sulla similarità

tra embedding e il modello di machine learning utilizzato anziché lavorare sui dati, viene utilizzato per creare gli embedding di codice. Dopo aver costruito un dataset contenente una collezione di righe vulnerabili JavaScript, chiamato dagli autori vulnerable lines repository (VLR), vengono creati degli embedding a partire da questa collezione di righe e dalle righe che devono essere classificate. Gli embedding vengono creati a partire da un modello word2vec, che viene allenato con codice JavaScript tokenizzato. Dopo aver fatto ciò, si calcola la distanza tra gli embedding di una riga che deve essere classificata e quelle presenti nel VLR, in modo da capire se la riga da classificare è probabilmente vulnerabile. Dopo aver memorizzato le distanze minime (quindi somiglianza più alta ad una vulnerabilità), di ogni riga da classificare, si calcolano le probabilità di ogni riga di essere vulnerabile, con delle funzioni che tengono in considerazione le distanze minime precedentemente citate. Il VLR viene diviso in train, test e dev set, dove quest'ultimo viene utilizzato per trovare empiricamente la soglia per la quale una riga viene classificata come vulnerabile o meno, basandosi sulle probabilità calcolate precedentemente.

CAPITOLO 3

Metodologia

3.1 Obiettivo e Domande di Ricerca

L'obiettivo di questo studio è capire se sia possibile utilizzare tecniche sviluppate per analizzare codice vulnerabile in linguaggio C, anche per il linguaggio Python mediante opportune modifiche e valutarne i risultati. Per il raggiungimento di questo obiettivo, ci si aspetta di rispondere alle domande di ricerca proposte di seguito.

🔍 **RQ$_1$.** *Quali sono le differenze di prestazioni dei modelli PyLineVul e PyLineVd rispetto ai modelli originali?*

In virtù del fatto che i due modelli sono stati soggetti a delle modifiche a livello di funzionamento e al dataset sul quale sono stati allenati, risulta necessario il paragone con le tecniche originali per capire se l'obiettivo dello studio è stato raggiunto. Nello specifico si vuole capire quanto le modifiche effettuate possano influenzare la capacità operativa dei modelli, in modo da valutare se la modifica di un modello sia conveniente.

🔍 **RQ$_2$.** *Quali sono le differenze di prestazioni tra i modelli PyLineVul e PyLineVd?*

In forza delle modifiche apportate, oltre che valutare le effettive prestazioni dei modelli PyLineVul e PyLineVd, con questa domanda di ricerca si vuole capire se le

prestazioni di questi ultimi mantengano la stessa proporzionalità delle prestazioni dei modelli originali.

3.2 PyLineVul e PyLineVd

Prima di modificare i modelli per effettuare la classificazione di codice Python, sono stati replicati gli script presenti nei repository GitHub[1,2] relativi ai lavori. Di seguito viene illustrato il funzionamento originale dei due modelli e le modifiche apportate.

3.2.1 LineVul

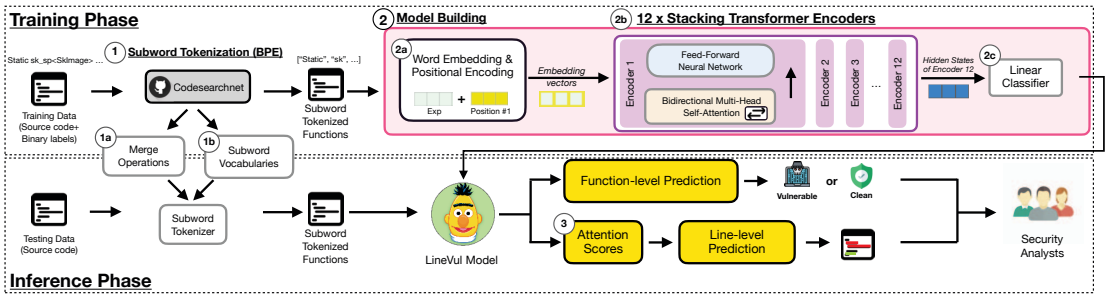

Figura 3.1: Funzionamento del modello LineVul [1], dal paper relativo al framework

LINEVUL [1] permette predizioni di vulnerabilità software grazie all'utilizzo di BERT, che viene usato per evidenziare le dipendenze tra le righe di codice vulnerabile e CODEBERT, che viene usato per ottenere delle rappresentazioni vettoriali del codice. Come illustrato nella figura 3.1, il framework opera in due fasi: la fase di allenamento e la fase di inferenza. Nella fase di allenamento viene costruito il modello, mentre in quella di inferenza, che sfrutta il modello precedentemente allenato, viene effettuata la classificazione. Il modello esegue la classificazione del codice sorgente in due step: una fase di predizione di vulnerabilità a livello di funzione e una fase di predizione di vulnerabilità a livello di riga. La fase di predizione di vulnerabilità a livello di funzione inizia con la tokenizzazione, che viene effettuata utilizzando un algoritmo

[1]https://github.com/awsm-research/LineVul
[2]https://github.com/davidhin/linevd

di compressione dati: BPE (Byte Pair Encoding), che per tokenizzare il corpus di codice, divide tutte le parole del corpus in sotto-parole, dopodiché identifica le coppie di caratteri più comuni e le sostituisce con un carattere non presente nel testo. Operando in questo modo l'algoritmo permette di rappresentare il corpus di codice in maniera compressa. In seguito viene costruito il modello: le parole tokenizzate vengono convertite in un vettore che ne descrive la posizione nella funzione, per poi dare in input a BERT i vettori ottenuti, da cui si ottiene un vettore risultante che viene dato in input ad un layer lineare che permette di classificare la funzione in vulnerabile o non vulnerabile. Nel caso in cui la funzione sia vulnerabile, si ricercano le righe di codice vulnerabili tramite i layer di self-attention di BERT, tenendo conto del fatto che i token più vulnerabili son quelli che sono stati più influenti nella predizione di vulnerabilità della funzione. Avendo ottenuto i punteggi di self attention rispetto ad ogni sottoparola tokenizzata, essi verrano combinati per ottenere i punteggi di ogni riga della funzione. Il modello utilizza la configurazione mostrata di seguito:

```
class RobertaClassificationHead(nn.Module):
    def __init__(self, config):
        super().__init__()
        self.dense = nn.Linear(config.hidden_size, config.hidden_size)
        self.dropout = nn.Dropout(config.hidden_dropout_prob)
        self.out_proj = nn.Linear(config.hidden_size, 2)
    ...
```

Il modello viene configurato inizializzando i layer che effettuano le trasformazioni sulle funzioni tokenizzate date in input al modello. Nello specifico i layer lineari prendono come input le dimensioni dei dati che attraversano i layer stessi, sia per l'input che per l'output. Mentre il layer di dropout prende in input la probabilità di ignorare un'istanza data in input al layer (azzerare o porre a zero un'istanza). Le altre informazioni riguardanti la configurazione del modello, sono presenti nello script di allenamento del modello, presente nella sezione 3.2.3. Il framework prende in input una serie di funzioni non etichettate, per poi restituire le righe in ordine decrescente di vulnerabilità.

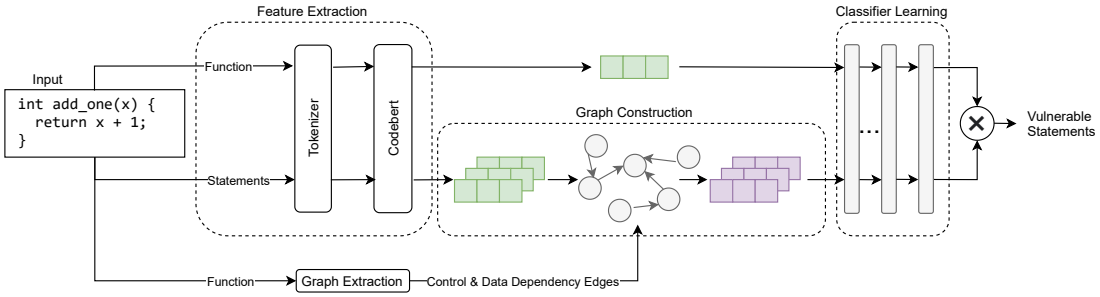

Figura 3.2: Funzionamento del modello LineVd [2], dal paper relativo al framework

3.2.2 LineVD

LineVd [2] è un framework che effettua predizioni di vulnerabilità software a livello di riga di codice, sfruttando una GNN (Graph Neural Network) e un modello transformer-based. Come si può notare nella figura 3.2, il framework è basato su 3 componenti principali: feature extraction, graph construction e classifier learning. La fase di feature extraction consiste nel dare in input al framework una funzione: questa viene divisa in statements che vengono tokenizzati usando l'algoritmo BPE con CodeBERT. La funzione e gli statements vengono dati in input a CodeBERT, in modo da avere una rappresentazione del codice sia a livello di funzione che di riga. CodeBERT a questo punto produrrà gli embeddings del codice. La fase di costruzione del grafo, che definisce i rapporti di dipendenza tra i vari statements, sfrutta un Graph Attention Network che impiega un metodo di diffusione delle informazioni, che aggiorna i nodi del CPG (code dependency graph) in maniera incrementale, per memorizzare le informazioni riguardanti la dipendenza degli statements. La fase di classifier learning punta ad allenare il modello per effettuare le predizioni. Grazie ad una serie di layer lineari e di dropout, vengono presi in input gli embeddings della funzione e degli statements, le cui classificazioni (1 = vulnerabile, 0 = non vulnerabile) verranno moltiplicate alla fine, in modo da tener conto delle predizioni a livello di funzione e di riga contemporaneamente. Il modello utilizza la configurazione mostrata di seguito:

```python
class LitGNN(pl.LightningModule):

    def __init__(
        self,
        hfeat: int = 512,
        embtype: str = "codebert",
        embfeat: int = -1,
        num_heads: int = 4,
        lr: float = 1e-3,
        hdropout: float = 0.2,
        mlpdropout: float = 0.2,
        gatdropout: float = 0.2,
        methodlevel: bool = False,
        nsampling: bool = False,
        model: str = "gat2layer",
        loss: str = "ce",
        multitask: str = "linemethod",
        stmtweight: int = 5,
        gnntype: str = "gat",
        random: bool = False,
        scea: float = 0.7,
    ):
    ...
```

Il modello viene configurato definendo il tipo di task da effettuare, il tipo di rete neurale da utilizzare, il tipo di metodo di embedding da impiegare e inizializzando i vari layer che compongono il modello. Similmente a quello precendente, il framework prende in input una serie di funzioni non etichettate, per poi restituire le righe vulnerabili in ordine decrescente di vulnerabilità.

3.2.3 Adattamenti

Per poter permettere l'identificazione di vulnerabilità per codice Python sono state necessarie delle modifiche agli script dei progetti, di seguito sono illustrati i cambiamenti per ciascun progetto.

Gli script relativi al framework LINEVUL sono stati modificati rispetto al tokenizer e allo script per l'allenamento e testing del modello. Per avere a disposizione un tokenizer che potesse creare dei token significativi per codice Python, si è scelto di utilizzare un tokenizer allenato da zero, in quanto quello del modello originale utilizzava un tokenizer capace di operare su linguaggio C. Nello script originale per

tokenizzare il codice viene usata la classe RobertaTokenizer, che sfrutta l'algoritmo di compressione dati BPE e necessita di due file: il primo, vocab.json, è un file JSON contenente tutti i vocaboli tokenizzati del corpus di codice, mentre il secondo, merges.txt, è un file di testo che contiene il mapping tra il testo e i tokens presenti nel file vocab.json. Per ottenere questi file è stato necessario prima costruire un dataset su cui allenare il tokenizer. Per costruire il dataset sono state utilizzate le funzioni presenti nel dataset CODESEARCHNET [19]. Il dataset di funzioni CODESEARCHNET contiene circa due milioni di coppie funzione / commenti, ottenute da progetti open-source presenti sulla piattaforma di hosting GitHub. Ogni riga del dataset contiene diverse informazioni, ma per la costruzione del dataset per allenare il tokenizer sono state prese in considerazioni solo le colonne `language` e `whole_func_string`, per ottenere funzioni scritte in linguaggio Python. Di seguito viene mostrato lo script per estrarre tutte le funzioni Python dal dataset, per poi salvare in un file di testo per ciascuna funzione.

```
import os
from datasets import load_dataset


def column_to_files(column, prefix, txt_files_dir):
    i = prefix

    for row in column.to_list():
        file_name = os.path.join(txt_files_dir, str(i) + '.txt')
        try:
            f = open(file_name, 'wb')
            f.write(row.encode('utf-8'))
            f.close()
        except Exception as e:
            print(row, e)
        i += 1

    return i


def main():
    dataset = load_dataset("code_search_net", "python")

    df_pandas = dataset["train"].to_pandas()

```

```
    data = df_pandas["whole_func_string"]

    data.replace("\n", " ")

    prefix = 0

    txt_files_dir = "/home/raffaele/Desktop/functions_as_text"

    prefix = column_to_files(data, prefix, txt_files_dir)

    print("\n\n\nLast prefix: " + str(prefix))


if name == "main":
    main()
```

Dopo aver ottenuto il dataset si è allenato il tokenizer, come illustrato di seguito:

```
from tokenizers.implementations import ByteLevelBPETokenizer
from pathlib import Path


def main():
    paths = [str(x) for x in Path("/home/raffaele/Desktop \
        /functions_as_text/").glob("*.txt")]

    tokenizer = ByteLevelBPETokenizer(lowercase=True)

    tokenizer.train(files=paths, vocab_size=50257, min_frequency=2,
                    show_progress=True,
                    special_tokens=[
                                    "<s>",
                                    "<pad>",
                                    "</s>",
                                    "<unk>",
                                    "<mask>",
                    ])

    tokenizer.save_model("/home/raffaele/Desktop/tokenizer",
        "bpe_tokenizer")
    tokenizer.save("/home/raffaele/Desktop/tokenizer/config.json")


if name == "main":
    main()
```

Il tokenizer originale utilizzava dei file merges.txt e vocab.json, che erano stati creati a partire da codice sorgente in linguaggio C e in virtù di ciò, si è scelto di allenare solo il tokenizer. Normalmente non sarebbe corretto allenare solo il tokenizer e non il modello che lo usa, però in questo caso è stato possibile perché il modello che utilizza effettivamente il tokenizer (codebert-base), è un modello pre-allenato su diversi linguaggi di programmazione (tra cui Python) e sullo stesso corpus di codice su cui è stato allenato il nuovo tokenizer, ossia il dataset di funzioni CODESEARCHNET. A questo punto grazie ai due file, merges.txt e vocab.json, si è potuto allenare il modello utilizzando la seguente configurazione:

```
python3 linevul_main.py \
--output_dir=./saved_models \
--model_type=roberta \
--tokenizer_name=microsoft/codebert-base \
--model_name_or_path=microsoft/codebert-base \
--use_non_pretrained_tokenizer \
--do_train \
--do_test \
--train_data_file=train.csv \
--eval_data_file=val.csv \
--test_data_file=test.csv \
--epochs 1 \
--block_size 512 \
--train_batch_size 16 \
--eval_batch_size 16 \
--learning_rate 2e-5 \
--max_grad_norm 1.0 \
--evaluate_during_training \
--seed 123456  2>&1 | tee train.log
```

La configurazione prevede di allenare il modello con solo una sola epoch, con un nuovo dataset, la cui costruzione viene illustrata nella sezione successiva. Mentre per effettuare la predizione a livello di riga, è stata usata questa configurazione:

```
      python3 linevul_main.py \
      --model_name=model.bin \
      --output_dir=./saved_models \
      --model_type=roberta \
      --tokenizer_name=microsoft/codebert-base \
      --model_name_or_path=microsoft/codebert-base \
      --use_non_pretrained_tokenizer \
      --do_test \
      --do_local_explanation \
      --top_k_constant=10 \
      --reasoning_method=all \
      --train_data_file=train.csv \
      --eval_data_file=val.csv \
      --test_data_file=test.csv \
      --block_size 512 \
      --eval_batch_size 512
```

Gli script relativi al framework LineVd sono stati modificati per poter permettere il caricamento del nuovo dataset e avere la possibilità di creare grafi a partire dalle funzioni Python presenti nel nuovo dataset. La maggior parte delle modifiche effettuate sono state fatte per permettere l'effettivo funzionamento del framework, anche in relazione all'ambiente di sviluppo in cui sono stati eseguiti gli script. Per prima cosa è stata modificata la funzione che permette il caricamento del dataset Python, nello specifico la funzione `bigvul`, la cui versione modificata viene mostrata di seguito:

```
def bigvul(minimal=True, sample=False, return_raw=False, splits="default"):
    savedir = svd.get_dir(svd.external_dir())

    if minimal:
        try:
          df = pd.read_csv(savedir / "dataset.csv")
          return df
        except Exception as E:
          print(E)
          pass

    df = pd.read_csv(savedir / "dataset_before.csv")
    df['dataset'] = 'bigvul'

    # pre-processing

```

```
    dfv = df[df.vul == 1]
    dfv = dfv[~dfv.apply(lambda x: len(x.added) == 0 and \
        len(x.removed) == 0, axis=1)]
    dfv = dfv[dfv.apply(lambda x: \
        len(x.before.splitlines()) > 5, axis=1)]
    dfn = df[df.vul == 0].sample(len(dfv) * 10, replace=True)

    df = pd.concat([dfv, dfn], axis=0)
    df["id"] = range(0, len(df))

    # split

    df = train_val_test_split_df(df, "id", "vul")

    # training-set balancing

    train_vdf = df[(df['label'] == 'train') & (df['vul'] == 1)]
    train_nvdf = df[(df['label'] == 'train') & \
        (df['vul'] == 0)].sample(len(train_vdf), replace = True)
    train_df = pd.concat([train_vdf, train_nvdf])

    df = pd.concat([train_df, df[df['label'] != 'train']])
    df["id"] = range(0, len(df))

    df.to_csv(savedir / "dataset.csv")

    return df
```

Questa modifica è stata necessaria perché l'implementazione precedente presentava degli errori sorti grazie alle nuove versioni delle librerie utilizzate, inoltre dato che nel modello originale era proprio in questa funzione che veniva effettuato lo split, è in questa funzione che viene effettuato il bilanciamento del test set. In seguito sono stati modificati gli script per la creazione dei grafi, su cui il modello opera per trovare le dipendenze del codice. Gli script in questione inizialmente creavano dei file .c, per poi creare i grafi grazie ad uno script in linguaggio Scala, per questo si è modificato lo script `getgraphs.py` per creare dei file .py delle versioni `before` e `after` di ogni riga del dataset, per poi utilizzare lo script Scala, modificato come mostrato di seguito, per creare le rappresentazioni del codice mediante grafi.

```
@main def exec(filename: String) = {
   importCode.python(filename)
   run.ossdataflow
   cpg.graph.E.map(node=>List(node.inNode.id, node.outNode.id,
        node.label,
        node.propertiesMap.get("VARIABLE"))).toJson |>
        filename + ".edges.json"
   cpg.graph.V.map(node=>node).toJson |> filename + ".nodes.json"
   delete
}
```

Lo script è stato modificato perché nel modello originale permetteva esclusivamente l'utilizzo di codice C per creare i grafi relativi alle funzioni. Dopo aver effettuato queste modifiche è stato possibile allenare il modello.

3.3 Preparazione dei Dataset

Non esistono esempi di dataset di funzioni vulnerabili e non: in virtù di ciò ne sono stati creati due ad hoc per allenare i modelli PyLineVul e PyLineVd, creati a partire da quelli originali. Come si può notare dalla figura 3.3, i due dataset sono stati creati a partire da due corpus di codice preesistenti: il corpus di codice vulnerabile relativo al framework VUDENC [20] e il dataset di funzioni CODESEARCHNET [19]. Il corpus di codice vulnerabile relativo al framework VUDENC, è costituito da numerosi file JSON: ciascun file corrisponde a un determinato tipo di vulnerabilità software, come SQL Injection, Cross-site scripting e così via. Il file contiene dei commit ottenuti a partire da progetti open-surce, scritti in codice Python, presenti sul servizio di hosting di progetti software GitHub. Ogni riga del file contiene un commit, relativo al progetto da cui il commit è tratto, in cui viene risolta la particolare vulnerabilità a cui il file fa riferimento, difatti il codice viene considerato vulnerabile, in base al fatto che è stato modificato. I file JSON sono reperibili dalla repository GitHub del progetto e di seguito si mostra un esempio dei campi utilizzati per costruire il dataset:

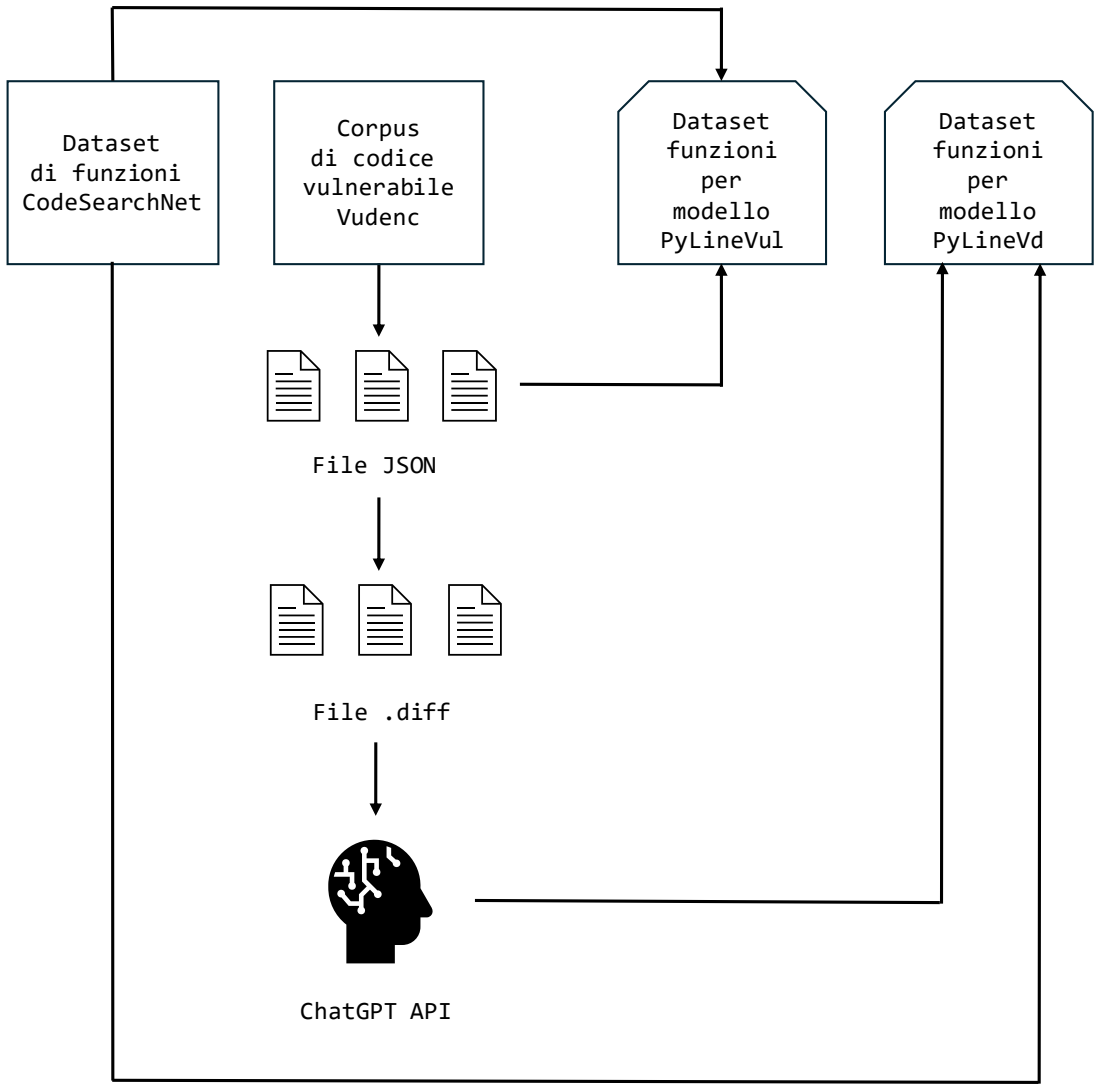

Figura 3.3: Creazione dei due dataset relativi ai modelli PyLineVul e PyLineVd

```
"changes": [{
    "diff": "\n \n     SELECT id, child_branch_id, ...",
    "add": 1,
    "remove": 1,
    "filename": "/base/models/group_element_year.py",
    "badparts": [
        "    WHERE parent_id IN ({list_root_ids})"],
    "goodparts": [
        "    WHERE parent_id IN (%s)"]
}],
"source": "..."
```

Ogni entry di un file JSON relativo ad una vulnerabilità, contiene diverse informazioni oltre che al contenuto effettivo del commit, come l'url del progetto, il messaggio relativo al commit e simili. Le informazioni utilizzate per la costruzione del dataset sono quelle relative al codice sorgente (campo `source`), alla differenza relativa alle modifiche effettuate in un commit (campo `diff`) e le righe di codice cancellate nel commit (campo `badparts`). Dal dataset di funzioni CODESEARCHNET per creare le parti non vulnerabili di entrambi i dataset, sono state estratte solo le funzioni presenti nel campo `whole_func_string`. Da questi due corpus di codice si sono ottenuti due dataset, uno per modello.

Per costruire il dataset necessario ad allenare il modello PyLineVul, inizialmente si sono osservate le colonne del dataset su cui il modello originale andava a lavorare. Le colonne necessarie per la costruzione del dataset sono quattro: `processed_func`, `target`, `flaw_line` e `flaw_line_index`. La colonna `processed_func` contiene la funzione da classificare, la colonna `target` indica se una funzione è vulnerabile o meno, la colonna `flaw_line` contiene le righe vulnerabili e la colonna `flaw_line_index` contiene l'indice delle righe vulnerabili all'interno della funzione. In un primo momento sono state prese le funzioni vulnerabili presenti nel corpus di codice relativo al framework VUDENC: per ogni file JSON, corrispondente ad un particolare tipo di vulnerabilità, sono state estratte tutte le funzioni che contenevano differenze in un commit, in quanto questo implica la presenza di codice vulnerabile, per poi ottenere le righe di codice vulnerabile tramite il campo `badparts` del file JSON. Popolate le colonne `processed_func` e `flaw_line`, si è popolata la colonna `flaw_line_index` con il numero di riga delle righe di codice vulnerabile per poi inserire nella colonna `target` il valore 1, che indica la vulnerabilità della funzione. A questo punto sono state eliminate le righe relative alle istanze vulnerabili che contenevano dei valori nulli, ottenendo di fatto la parte vulnerabile del dataset che consta di centodieci funzioni. Dopodiché si è popolata la parte non vulnerabile del dataset con le funzioni presenti nel corpus di codice CODESEARCHNET. Si sono inserite le funzioni nella colonna `processed_func`, lasciando vuote le colonne `flaw_line` e `flaw_line_index` in quanto non sono presenti righe vulnerabili e per concludere si è inserito nella colonna `target` il valore 0. Per finire si sono scelte casualmente millecento funzioni tra quelle non vulnerabili. È stato scelto un numero 10 volte più

grande rispetto a quello delle funzioni vulnerabili, per mantenere una proporzione tra funzioni vulnerabili e non simile a quella del dataset originale. Il dataset originale è stato costruito in questo per evitare di andare a modellare un problema irrealistico, dato che le funzioni vulnerabili sono molto più rare di quelle non vulnerabili. Di seguito, nella figura 3.1 viene mostrato un esempio di due righe del dataset ottenuto:

index	processed_func	flaw_line	flaw_line_index	target
12	def get_empty_instance(table): ...			0
96	def _main(): ...	data ...	16,17,22,26	1

Tabella 3.1: Righe di esempio del dataset relativo al modello PyLineVul

Come per il dataset necessario ad ad allenare il modello PyLineVul, anche per il dataset relativo all'allenamento del modello PyLineVd si sono osservate le colonne del dataset su cui operava il modello originale. Sono state individuate sei colonne necessarie per il funzionamento del framework: `before`, `after`, `vul`, `added`, `removed` e `diff`. La colonna `before` contiene le funzioni vulnerabili prima della modifica (e quindi prima del commit che ha portato alla risoluzione della vulnerabilità), con le righe che sono state aggiunte dopo la modifica in dei commenti. Di seguito viene mostrato un esempio:

```
def get(x):
# def get():
     return x * 2
```

La colonna `after` contiene le funzioni vulnerabili dopo la modifica (e quindi dopo il commit che ha portato alla risoluzione della vulnerabilità) con le righe che sono state rimosse dopo la modifica in dei commenti. Di seguito viene mostrato un esempio:

```
{
# def get(x):
def get():
     return x * 2
```

La colonna `added` contiene i numeri di riga delle righe di codice aggiunte (e quindi commentate) relativamente alla funzione presente nel campo `before`, quindi tenedo a mente l'esempio precedentemente esposto la colonna `added`, conterrà il seguente valore: [2]. La colonna `removed` contiene i numeri di riga delle righe di codice rimosse (e quindi commentate) in relazione alla funzione presente nella colonna `after`, quindi seguendo l'esempio precedentemente esposto la colonna `removed`, conterrà il seguente valore: [1]. La colonna `vul` indica se una funzione è vulnerabile o meno, mentre la colonna `diff` contiene tutte le modifiche effettuate durante un commit, come si può notare nell'esempio seguente:

```
---

+++

@@ -1,2 +1,2 @@

-def get(x):
+def get():
     return x * 2
```

La costruzione del dataset, inizia cominciando a popolarlo con le istanze vulnerabili, ottenibili dai file JSON usati anche per la costruzione del dataset precedente. Per popolare le colonne del dataset relativo al modello PyLineVd, sono state prese le modifiche effettuate rispetto al commit effettuato, presenti nel campo `diff` di ogni entry, di ogni file JSON. Ciascuna modifica è stata memorizzata in un file .diff. A questo punto dopo aver ottenuto diversi file .diff, grazie all'API fornita da OpenAI, sono state create due funzioni per ciascuna funzione presente in ogni file .diff. Le due funzioni richieste mediante un prompt per l'API hanno la seguente struttura: la prima funzione doveva contenere le righe rimosse e quelle rimaste invariate, mentre la seconda funzione doveva contenere le righe aggiunte e quelle rimaste invariate. L'utilizzo dell'API è stata necessaria perché in molti file .diff spesso si riuscivano ad ottenere solo una delle due tipologie di funzioni, rendendo impossibile la creazione di entry del dataset complete. Avendo a disposizione le due tipologie di funzioni è stato possibile popolare le colonne `before` e `after`, trovando le differenze tra queste due funzioni mediante la libreria Python difflib, che genera una stringa di caratteri

contenenti la differenza tra la due funzioni: tramite questa stringa di caratteri è stato possibile generare delle funzioni che rispettassero la struttura di quelle presenti nelle colonne `before` e `after` del dataset originale. Questo passo extra è stato necessario perché l'API di OpenAI non è riuscita a creare direttamente, delle funzioni da inserire nelle colonne `before` e `after`, che rispettassero i criteri del dataset originale. Una volta popolati le colonne `before` e `after` è stato possibile popolare anche le altre: nelle colonne `removed` e `added` sono stati inseriti rispettivamente gli indici delle righe di codice rimosse nella funzione `after` e gli indici delle righe aggiunte nella funzione `before`. La colonna `vul`, è stata popolata con il valore 1, mentre nella colonna `diff` si è inserita la stringa generata grazie alla libreria difflib. Dopo aver eliminato le righe relative alle istanze vulnerabili che contenevano valori nulli, il numero di queste è di centosettantanove. Per quanto riguarda le istanze non vulnerabili, ottenute dalle funzioni presenti nel corpus di codice CODESEARCHNET, si sono popolate le colonne `before` e `after` con lo stesso valore: le funzioni Python presenti nel dataset CODESEARCHNET, in quanto non vulnerabili e quindi senza modifiche. Per lo stesso motivo le colonne `added`, `removed` e `diff` contengono valori nulli, mentre la colonna `vul` contiene il valore 0. Per finire, analogamente alle costruzione del dataset precedente, si sono scelte casualmente millesettecentonovanta funzioni tra quelle non vulnerabili. Dopo aver creato il dataset sono state eliminate le righe le cui funzioni erano lunghe meno di cinque righe o che non avevano righe vulnerabili, come nel dataset usato dal modello originale. Nella figura 3.2 viene mostrato un esempio di due righe del dataset ottenuto:

id	before	after	added	removed	diff	vul
2	def disable(...	def disable(...	[]	[]	""	0
354	def add_artic ...	#def add_artic ...	[]	[4]	— +++ ...	1

Tabella 3.2: Righe di esempio del dataset relativo al modello PyLineVd

Entrambi i dataset dopo le operazioni di pre-processing e split in training, validation e test set, hanno subito dei bilanciamenti rispetto al training set, come eseguito

dai modelli originali, per cercare di avere un confronto quanto più equo possibile.

3.4 Setup Sperimentale

Il task di identificazione di vulnerabilità a livello di funzione è un task di classificazione binaria, in quanto il modello deve classificare degli elementi (in questo caso le funzioni vulnerabili), in una di due classi (in questo caso "vulnerabile" o "non vulnerabile"). Mentre il task di identificazione di vulnerabilità a livello di riga è un task di ranking, in quanto il modello deve ordinare degli elementi (in questo caso le righe vulnerabili), dal più rilevante a quello meno rilevante (in questo caso dalla riga più vulnerabile a quella meno vulnerabile).

Per avere un confronto equo con i modelli originali, i dataset dei modelli PyLineVul e PyLineVd sono stati sottoposti allo stesso tipo di split e balancing. Entrambi i dataset sono stati soggetti ad uno split, uguale a quello proposto dagli autori dei framework originali: 80% del dataset per il training set, 10% del dataset per il validation set e il restante 10% per il test set. Dopo aver effettuato lo split, è stato effettuato un balancing del training set riducendo il numero di istanze non vulnerabili, presenti in numero assai maggiori rispetto a quelle non vulnerabili, per evitare che i modelli avessero un bias rispetto a una delle due classi.

Per effettuare gli esperimenti, in un primo momento gli script sono stati eseguiti su una macchina virtuale, che sfruttava come sistema operativo Ubuntu 22.04 LTS. In seguito, data la necessità di maggiori risorse computazionali, gli script relativi all'allenamento dei modelli sono stati eseguiti sulla piattaforma GOOGLE COLABORATORY: una piattaforma che tramite un notebook Jupyter, ha permesso l'esecuzione degli script dei modelli. Per entrambi i progetti sono state installate le dipendenze relative ai progetti originali e dopo averli modificati opportunamente, come evidenziato nella sezione 3.2.3, è stato svolto l'allenamento (utilizzando una GPU V4) e successivamente il testing. L'allenamento e il testing di entrambi i modelli hanno richiesto tempi simili, circa 1 ora per l'allenamento e circa 15 minuti per il testing. Tempi del genere sono dovuti al fatto che l'allenamento dei modelli constava di una sola epoch: i modelli hanno iterato le operazioni di apprendimento e testing sull'intero dataset una volta sola.

3.5 Confronto dei Modelli

Il confronto dei modelli viene fatto rispetto alle due granularità a cui operano i modelli: a livello di funzione e a livello di riga.

Per valutare i modelli a livello di funzione, è stata posta attenzione alle metriche che più si prestano ai task di classificazione binaria:

- Precision, che rappresenta il rapporto tra le istanze positive e tutte le istanze classificate come positive;
- Recall, che rappresenta il rapporto tra le istanze positive e tutte le istanze effettivamente positive;
- F1 score, che rappresenta la media armonica tra Precision e Recall.

Di seguito vengono mostrate le formule per calcolare queste metriche:

$$Precision = \frac{TP}{TP + FP}$$

$$Recall = \frac{TP}{TP + FN}$$

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

Dove per TP si intende Veri Positivi (True Positives, istanze positive classificate come positive), FP si intende Falsi Positivi (False Positives, istanze negative classificate come positive), FN si intende Falsi Negativi (False Negatives, istanze positive classificate come negative) e TN si intende Veri Negativi (True Negatives, istanze negative classificate come negative). Oltre che per la motivazione citata precedentemente, si è scelto di concentrarsi su queste metriche perché, come definito da Kalouptsoglou et al. [21], la metrica F1 score risulta essere più usata rispetto alla metrica Accuracy, perché nonostante un'alta Accuracy un modello potrebbe avere un bias rispetto ai sample non vulnerabili, dato che i dataset relativi al task di identificazione di vulnerabilità tipicamente presentano un numero assai maggiore di istanze non vulnerabili. Nello specifico l'Accuracy, che rappresenta il rapporto tra predizioni corrette e tutte le predizioni effettuate, viene definita come segue:

$$Accuracy = \frac{TP + TN}{TP + TN + FP + FN}$$

La scelta di Precision e Recall, viene fatta per cercare di capire se un modello sia in grado di riconoscere falsi positivi e falsi negativi.

Invece per valutare i modelli a livello di riga si osservano le seguenti metriche:

- Top-5 Accuracy (acc@5), per capire se la risposta attesa è tra le prime cinque risposte fornite dal modello;
- Top-10 Accuracy (acc@10), per capire se la risposta attesa è tra le prime dieci risposte fornite dal modello.

Si è scelto di osservare queste metriche, quasi identiche, per contestualizzare l'operato di ciascuno dei due modelli, PyLineVul e PyLineVd, sia rispetto alle tecniche originali, che tra di loro.

I modelli PyLineVul e PyLineVd sono stati valutati anche rispetto ad altre metriche, come MCC e AUC-ROC, che non vengono considerate nel confronto dei risultati, perché non sono state utilizzate perla valutazione di uno dei modelli originali.

CAPITOLO 4

Risultati

L'obiettivo dello studio era capire se fosse possibile utilizzare tecniche per l'identificazione di vulnerabilità sviluppate per operare su codice scritto in C, anche in Python. Dai risultati ottenuti si deduce che ciò è senza dubbio possibile e di seguito si valutano i risultati dell'operato, rispondendo alle due domande di ricerca introdotte nella sezione 3.1.

4.1 Differenze con Modelli Originali

Prima di iniziare a confrontare i risultati è da precisare che le prestazioni del modello LineVd [2] originale, sono state ottenute replicando lo studio: il modello originale (senza subire modifiche) è stato allenato e testato su una porzione ridotta del dataset originale. Ciò è stato necessario in quanto i punteggi delle metriche, a livello di funzione, non sono riportate nello studio originale. Nella figura 4.1, vengono mostrate le prestazioni dei modelli originali, mentre nella figura 4.2, vengono mostrate le prestazioni dei modelli modificati.

Modello	Precision	Recall	F1 Score	Top-5 Accuracy	Top-10 Accuracy
LineVul	0.97	0.86	0.91		0.65
LineVd	0.88	0.92	0.89	0.90	

Tabella 4.1: Prestazioni dei modelli originali

Modello	Precision	Recall	F1 Score	Top-5 Accuracy	Top-10 Accuracy
PyLineVul	0.27	0.91	0.42	0.20	0.30
PyLineVd	0.10	0.90	0.19	1.00	1.00

Tabella 4.2: Prestazioni dei modelli modificati

Il modello PyLineVul, come mostrato dall'alto Recall, è in grado di distinguere in maniera ottimale le funzioni effettivamente vulnerabili nel dataset, proprio come il modello originale, tuttavia a differenza di quest'ultimo ha una Precision molto bassa, che sta a indicare che il modello PyLineVul predice erronamente molte istanze non vulnerabili come vulnerabili. In generale il modello PyLineVul risulta meno accurato, nella predizione a livello di funzione, rispetto a quello originale, come evidenziato dal maggiore F1 score. Il modello PyLineVul risulta meno accurato anche a livello di predizione di riga: solo circa il 30% delle risposte fornite dal modello, sono effettivamente le 10 righe più vulnerabili. Invece il modello PyLineVd, risulta molto meno accurato in fase di predizione a livello di funzione, ma non ha sbagliato nemmeno una predizione a livello di riga, risultando quindi più accurato del modello originale.

👉 **Answer to RQ_1.** Il modello PyLineVul, rispetto all'originale, risulta meno accurato in fase di predizione sia a livello di funzione, che a livello di riga. Il modello PyLineVd, rispetto al modello originale, risulta meno accurato in fase di predizione a livello di funzione, ma più accurato a livello di riga.

4.2 Confronto tra PyLineVul e PyLineVd

Entrambi i modelli modificati, a livello di funzione, riescono entrambi a predire efficacemente le istanze vulnerabili del dataset come mostrato dall'alto recall, dimostrando una forte tolleranza rispetto ai falsi negativi. Il modello PyLineVul risulta leggermente più tollerante rispetto ai falsi positivi e in generale più accurato, rispetto al modello PyLineVd. Questa situazione è dovuta allo sbilanciamento del dataset e al basso numero di funzioni vulnerabili nel dataset. Se il modello PyLineVul risulta più accurato in fase di predizione a livello di funzione, invece a livello di riga risulta molto più efficace il modello PyLineVd, che risulta più accurato del 233% rispetto alla Top-10 Accuracy e del 400% rispetto alla Top-5 Accuracy. Inoltre si dimostra una certa proporzionalità rispetto ai modelli originali: il modello PyLineVul opera più efficacemente a livello di funzione che a livello di riga, rispetto al modello PyLineVd, proprio come i modelli originali.

👉 **Answer to RQ_2.** Il modello PyLineVul, rispetto al modello PyLineVd, risulta più efficiente in fase di predizione a livello di funzione e meno efficace in fase di predizione a livello di riga.

4.3 Osservazioni e Limitazioni

In virtù dei risultati si evince che la conversione di tecniche di identificazione di vulnerabilità a grana fine che operano su codice C, in tecniche in grado di operare su codice Python è possibile, tuttavia è doveroso contestualizzare i risultati ottenuti per comprendere le limitazioni dei modelli PyLineVul e PyLineVd.

Considerando che i dataset utilizzati per l'addestramento dei modelli constano di un numero di istanze assai inferiore, rispetto a quelli usati normalmente nell'addestramento di modelli che effettuano identificazione di vulnerabilità, si possono comprendere meglio i risultati ottenuti. Nello specifico entrambi i modelli hanno ottenuto dei punteggi bassi nella classificazione a livello di funzione: ciò è dovuto sia al fatto che i modelli sono stati allenati con una sola epoch, sia perché il numero di istanze vulnerabili era molto ridotto. Inoltre tra le funzioni vulnerabili, erano poche quelle che avevano delle informazioni dettagliate sulle righe vulnerabili, ciò permette

di spiegare anche il punteggio eccellente nella classificazione a livello di riga del modello PyLineVd: nel test set le funzioni con informazioni sulle righe vulnerabili sono presenti in numero esiguo e questo tipicamente porta a un'accuratezza del 100%.

Avendo contestualizzato al meglio i risultati ottenuti, un modo per poter valutare ancora più approfonditamente l'operato dei due modelli sarebbe quello di utilizzare dataset ad hoc, con a disposizione un numero maggiore di funzioni vulnerabili che al contempo abbiano a disposizione maggiori informazioni sulle righe vulnerabili.

CAPITOLO 5

Conclusioni

Questo studio ha cercato di verificare se fosse possibile utilizzare approcci di identificazione di vulnerabilità software a grana fine, sviluppati per operare su codice scritto in C, anche per codice Python. Con questo obiettivo in mente, mediante la costruzione di un dataset ad hoc e la conseguente modifica di due approcci già esistenti in letteratura, si è stabilito che ciò è possibile, anche se sono stati ottenuti risultati inferiori rispetto ai lavori originali. Nonostante l'esito dello studio, risulterebbe interessante considerare, come sviluppo futuro di questo, l'idea di replicare questo studio su dataset di codice Python, costruito appositamente per task di identificazione di vulnerabilità a livello di riga e anchedi creare un approccio di identificazione di vulnerabilità software a grana fine, sostituendo i modelli modificati con un modello creato da zero in grado di operare direttamente su codice Python.

Bibliografia

[1] M. Fu and C. Tantithamthavorn, "Linevul: A transformer-based line-level vulnerability prediction," *2022 IEEE/ACM 19th International Conference on Mining Software Repositories (MSR)*, 2022. (Citato alle pagine iii, 2, 9 e 12)

[2] D. Hin, A. Kan, H. Chen, and M. A. Babar, "Linevd: Statement-level vulnerability detection using graph neural networks," *2022 Mining Software Repositories Conference*, 2022. (Citato alle pagine iii, 2, 9, 14 e 30)

[3] K. Dempsey, P. Eavy, G. Moore, and E. Takamura, "Automation support for security control assessments," *NIST Special Publication 800-163*, p. 5, 2019. (Citato alle pagine 1 e 4)

[4] Y. Li, S. Wang, and T. N. Nguyen, "Vulnerability detection with fine-grained interpretations," *ACM*, p. 292, 2021. (Citato a pagina 1)

[5] S. Cass, "Top programming languages 2022," 2022, http://tinyurl.com/5cd2yteu [Accessed: (23/01/2024)]. (Citato a pagina 2)

[6] J. Tian, W. Xing, and Z. Li, "Bvdetector: A program slice-based binary code vulnerability intelligent detection system," *Information and Software Technology*, vol. 123, p. 106289, 02 2020. (Citato alle pagine 2 e 8)

[7] M. Jimenez, M. Papadakis, and Y. Le Traon, "Vulnerability prediction models: A case study on the linux kernel," *2016 IEEE 16th International Working Conference on Source Code Analysis and Manipulation*, 2016. (Citato alle pagine 2 e 7)

[8] B. Bruegge and A. H. Dutoit, "Object–oriented software engineering. using uml, patterns, and java," *Learning*, p. 444, 2009. (Citato alle pagine 4 e 7)

[9] L. Irwin, "Demystifying the cia triad: Why it's crucial for cyber security," 2023, http://tinyurl.com/3drfx46c [Accessed: (25/01/2024)]. (Citato a pagina 4)

[10] N. Shiri Harzevili, A. B. Belle, J. Wang, S. Wang, Z. M. Jiang, and N. Nagappan, "A survey on automated software vulnerability detection using machine learning and deep learning," *NIST Special Publication 800-163*, pp. 17–20, 2018. (Citato a pagina 5)

[11] W. Koehrsen, "Neural network embeddings explained," 2018, http://tinyurl.com/28xs2xuz [Accessed: (26/01/2024)]. (Citato a pagina 5)

[12] R. Dolphin, "The power of embeddings in machine learning," 2022, http://tinyurl.com/p4kbkcdz [Accessed: (26/01/2024)]. (Citato a pagina 5)

[13] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, "Bert: Pre-training of deep bidirectional transformers for language understanding," *arXiv:1810.04805v2*, 2019. (Citato a pagina 5)

[14] Z. Feng, D. Guo, D. Tang, N. Duan, X. Feng, M. Gong, L. Shou, B. Qin, T. Liu, D. Jiang, and M. Zhou, "Codebert: A pre-trained model for programming and natural languages," *arXiv:2002.08155v4*, 2020. (Citato a pagina 5)

[15] R. Croft, Y. Xie, and M. A. Babar, "Data preparation for software vulnerability prediction: A systematic literature review," *arXiv:2109.05740v2*, 2022. (Citato a pagina 6)

[16] R. Scandariato, J. Walden, A. Hovsepyan, and W. Joosen, "Predicting vulnerable software components via text mining," *IEEE Transactions on Software Engineering, VOL. 40*, 2014. (Citato a pagina 7)

[17] G. Lin, J. Zhang, W. Luo, L. Pan, Y. Xiang, O. De Vel, and P. Montague, "Cross-project transfer representation learning for vulnerable function discovery," *IEEE Transactions on Industrial Informatics, Vol. 14, No. 7*, 2018. (Citato a pagina 8)

[18] H. Perl, S. Dechand, M. Smith, D. Arp, F. Yamaguchi, K. Rieck, S. Fahl, and Y. Acar, "Vccfinder: Finding potential vulnerabilities in open-source projects to assist code audits," *ACM*, 2015. (Citato a pagina 9)

[19] H. Husain, H.-H. Wu, T. Gazit, M. Allamanis, and M. Brockschmidt, "CodeSearchNet challenge: Evaluating the state of semantic code search," *arXiv preprint arXiv:1909.09436*, 2019. (Citato alle pagine 16 e 21)

[20] L. Wartschinski, Y. Noller, T. Vogel, T. Kehrera, and L. Grunske, "Vudenc: Vulnerability detection with deep learning on a natural codebase for python," *arXiv:2201.08441v1*, 2022. (Citato a pagina 21)

[21] I. Kalouptsoglou, A. Ampatzoglou, M. Siavvas, and A. Chatzigeorgiou, "Software vulnerability prediction: A systematic mapping study," *Information and Software Technology*, 2023. (Citato a pagina 28)

Ringraziamenti

Giunto alla fine di questo elaborato mi è doveroso dedicare questo spazio alle persone, che con il loro supporto, hanno contribuito alla sua creazione.

Vorrei ringraziare di cuore il mio relatore, il prof. Palomba Fabio, per l'immensa disponibilità, i numerosi consigli e per avermi guidato durante tutto il mio percorso di tirocinio e tesi.

Un sentito ringraziamento va anche al mio tutor, il dott. Iannone Emanuele, per la straordinaria pazienza e le numerose dritte fornite durante il mio percorso.

Infine ringrazio sinceramente i miei genitori e mia sorella Valeria, per il loro sostegno, ottimismo e per aver sempre creduto in me.