UNIVERSITÀ DEGLI STUDI DI SALERNO

Dipartimento di Informatica

Corso di Laurea Magistrale in Informatica LM-18

TESI DI LAUREA

BetterJuliet: Analisi e Miglioramento della Test Suite Juliet

RELATORE

Prof. **Fabio Palomba**

Dott. **Emanuele Iannone**

Università degli Studi di Salerno

CANDIDATO

Emanuele Fittipaldi

Matricola: 0522501260

Anno Accademico 2022-2023

Ringrazio la mia famiglia e chiunque abbia creduto in me durante questo percorso.

Sommario

Dateset per addestrare classificatori di vulnerabilità sono diventati sempre più richiesti. Per sopperire alla mancanza di dati vulnerabili, si è cercato un modo per costruire dataset sintetici. Questo ha permesso la creazione di alcuni dataset largamente utilizzati in letteratura, come la Test Suite Juliet. La rapidità con cui ciò è avvenuto non ha permesso la creazione di uno standard per la costruzione di tali dataset, causando problemi strutturali, come duplicati, campioni poco relistici e non labellati correttamente. In letteratura inoltre, è osservabile un approccio ancora timido verso la data augmentation applicata a sorgenti vulnerabili. Gli approcci maggiormente utilizzati in letteratura sono basati su iniezione di codice, o scraping da progetti Open Source.

Questa tesi, prendendo come caso di studio la Test Suite Juliet Java, mira a fornire gli elementi per descrivere dataset vulnerabili, per valutarne ed aumentarne la qualità e sperimentare la data augmentation tramite la conversione di sorgenti vulnerabili in immagini per generare nuovi campioni. A partire dalle classi Java contenute nel dataset, tramite Code2Vec sono state ottenute le rappresentazioni vettoriali di queste classi e delle funzioni ivi contenute, al fine di ottenere una duplice granularità dei dati. Tale dataset arricchito viene presentato in questa tesi come BetterJuliet. Infine sono stati condotti cinque esperimenti dove è stato testato come le performance di una rete neurale (NN) cambiassero nel task di classificazione di vulnerabilità, al variare della granularità dei campioni, definendo nuove feature ed etichette tramite l'impiego di K-Means. Negli utlimi due esperimenti, i sorgenti convertiti in precedenza, sono stati convertiti in matrici, e le matrici interpretate come immagini, e sono stati infine generati nuovi campioni tramite l'applicazione di un filtro luminosità con valori randomici. Sono state infine confrontate le performance di una NN ed una convolutional neural network (CNN) sul task di classificazione di vulnerabilità sul nuovo dataset creato tramite augmentation.

I risultati degli esperimenti mostrano un incremento del 18% in termini di precision, recall ed F1-Score, impiegando una NN e le nuove label generate tramite K-Means, rispetto all'esperimento in cui sono state impiegate le label originali, ed un peggioramento di circa il 5% nel caso in cui il dataset è stato bilanciato tramite tecniche di augmentation basate su immagini, impiegando una CNN.

Indice

Elenco delle figure

Elenco delle tabelle

CAPITOLO 1

Introduzione

1.1 Contesto applicativo

La pandemia di Covid-19 ha generato una accelerazione del processo di transizione digitale di circa quattro anni e, in media sette anni per la velocità con la quale le aziende stanno sviluppando prodotti e servizi digitali [1]. Questo ha fatto nascere sopratutto nei primi periodi di pandemia molte soluzioni temporanee per far fronte ad una emergenza imprevista, cogliendo del tutto impreparate le organizzazioni.

L'aspetto della sicurezza informatica, nel tentativo di fornire delle soluzioni digitali nel breve termine, ne ha risentito fortemente. Secondo uno studio condotto da Deloitte [2], la pandemia è stata percepita dai criminali informatici come una opportunità per intensificare le proprie attività criminali, sfruttando una serie di vulnerabilità introdotte dalla velocità con la quale è stato necessaria la trasformazione digitale. Questa velocità, secondo il report annuale di HackerOne [3], ha incrementato del 151% nel 2022 errori di configurazione, i quali hanno dato vita a diverse vulnerabilità sfruttabili da parte di criminali informatici.

Una panoramica di quali vulnerabilità hanno subito un incremento nella frequenza di apparizione nei report è possibile ottenerla analizzando la classifica delle 10 vulnerabilità più comuni legate al mondo delle web app stilata da OWASP [4]. possiamo osservare come dal 2017 al 2021 il tipo di vulnerabilità salito vertiginosamente alla ribalta è A01:2021-Broken Access Control, il quale è passato dalla quinta posizione alla prima.

Il controllo degli accessi è fondamentale per comprendere quali azioni sono permesse

ad ogni tipo di utente. Una mancanza di controlli di questo tipo potrebbe portare alla divulgazione o cancellazione di dati sensibili, costituendo un pericolo reale per i pilastri di confidenzialità ed integrità.

Questo trend è confermato anche dall'ultimo report annuale del 2022 fornito dalla piattaforma HackerOne [3] dove è attestato che le vulnerabilità attualmente più sfruttate nei settori governativi sono quelle di Improper Authentication e Improper Authorization le quali hanno subito dal 2021 una impennata rispettivamente del 33% e 76%. La vulnerabilità più sfruttata invece nel settore informatico nel 2022 è stata l'XSS (Cross-Site Scripting) la quale ha subito una impennata del 32%.

1.2 Motivazioni e obiettivi

La prevenzione per questi motivi sta diventando sempre di più importante, ma c'è un problema. Secondo lo studio condotto da X. Li et al [5], i metodi tradizionali di detection delle vulnerabilità i quali fanno affidamento sull'intervento umano, soffrono di un alto tasso di falsi positivi. Per queste ragioni si è incominciato a fare largamente uso di software in grado di rilevare e classificare vulnerabilità in maniera automatica. L'impiego di tali software ha permesso di alleggerire il carico umano, permettendo di rilevare e classificare vulnerabilità anche su grandi moli di dati.

Per l'addestramento dei modelli di intelligenza artificiale costituenti il motore dietro questi tool automatici per la detection e classificazione di vulnerabilità nasce la necessità di utilizzare dataset contenenti esempi di codice sorgente vulnerabile aventi come label il tipo di vulnerabilità di cui il sorgente è affetto. Purtroppo, secondo il lavoro svolto da X. Li et al [5] i metodi intelligenti di detection delle vulnerabilità, soffrono di una mancanza di esempi vulnerabili. Per queste ragioni è nata l'esigenza di raggruppare il maggior numero possibile di campioni e in alcuni casi, di sintetizzarli. Questa tesi mira in primo luogo ad esplorare innanzitutto quali sono le problematiche principali che affliggono i dataset sintetici di vulnerabilità, prendendo come modello la Test Suite Juliet. In secondo luogo viene analizzata la possibilità di applicare un approccio di data augmentation su campioni vulnerabili. Nello specifico, vengono confrontate le performance di una rete neurale e di una CCN addestrate su questo dataset aumentato, convertendo prima i sorgenti vulnerabili in immagini nel caso della CNN.

1.3 Risultati ottenuti

I risultati sperimentali indicano che l'addestramento di un classificatore di vulnerabilità utilizzando campioni a livello di funzione non ha portato a un miglioramento delle prestazioni generali del modello, bensì ad un degrado di circa l'8% rispetto l'utilizzo di una granularità a livello di classe . Questo potrebbe essere dovuto al basso numero di Lines of Code (LOC) utilizzabili, che potrebbe non aver permesso al modello di apprendere in modo sufficiente le caratteristiche delle vulnerabilità e il contesto in cui si verificano. Un'altra possibile spiegazione è che le funzioni siano molto simili tra di loro, dunque il compito di classificare vulnerabilità risulta più difficile.

D'altro canto, l'uso dell'algoritmo di clustering K-Means ha prodotto risultati interessanti, con un notevole miglioramento delle prestazioni di circa il 18% del modello neurale addestrato su un nuovo set di dati rilabelato tramite K-Means. Il re-labelling, consente alla rete neurale di apprendere relazioni tra le vulnerabilità che non sarebbero rilevabili utilizzando le CWE originali. Questo suggerisce che un approccio di rappresentazione vettoriale in uno spazio N-Dimensionale e successivo clustering, potrebbe essere una soluzione promettente per aiutare nella descrizione delle vulnerabilità.

Inoltre, l'esperimento che ha utilizzato l'approccio di data augmentation e ha confrontato le prestazioni di una rete neurale (NN) con una rete neurale convoluzionale (CNN) ha mostrato che la CNN ha prestazioni leggermente miglioriEntrambi gli esperimenti hanno Micro Avg F1-Score di 0.77 (CNN) e 0.74 (NN), indicando una migliore capacità di generalizzazione e adattamento ai dati rispetto alla NN.

Tuttavia, ci sono alcune limitazioni evidenziate nel corso di questa tesi. La principale limitazione è stata la difficoltà nel reperire dataset contenenti campioni vulnerabili, a causa della loro eterogeneità e della limitata disponibilità di dataset ben documentati. Inoltre, la scarsa documentazione e le configurazioni problematiche di alcuni strumenti di trasformazione del codice hanno causato problemi nell'esecuzione di tali strumenti.

Nonostante queste limitazioni, il lavoro ha contribuito in modo significativo alla comprensione dei problemi relativi ai dataset di vulnerabilità e ha proposto alcune metriche per valutare la loro qualità. Inoltre, sono stati generati nuovi dati che arricchiscono la Test Suite Juliet Java, denominata BetterJuliet, fornendo rappresentazioni vettoriali di classi e funzioni Java contenenti o non contenenti vulnerabilità. Questi dati rappresentano un contributo originale che potrebbe essere utilizzato da futuri lavori di ricerca per esplorare nuove idee e approcci nella rilevazione di vulnerabilità nel codice sorgente.

1.4 Struttura della tesi

Nel capitolo 2 si affronta l'esplorazione del panorama dell'arte circa quali sono i dataset di vulnerabilità maggiormente utilizzati e quali metriche possono aiutare a comprenderne la qualità e costruire dataset sintetici di vulnerabilità. Viene inoltre fornita una descrizione circa quali sono le limitazioni presenti attualmente nello stato dell'arte circa questi argomenti. Nel capitolo 3 vengono definiti gli obiettivi, le domande di ricerca e le metriche adottate al fine di accettare o riggettare le ipotesi mosse, seguendo il criterio del GQM (Goal Question Metric). Vengono qui presentati gli esperimenti condotti. Nel capitolo 4 vengono analizzati i risultati ottenuti dalla fase sperimentale, suddivisi per Domanda di Ricerca. Nel capitolo 5 viene effettuata una analisi circa quanto ciò che è stato riscontrato come risultato è dovuto alle azioni intraprese in questa tesi (Validità interna) e circa quanto i risultati ottenuti siano riproducibili anche in situazioni diverse da quelle di seguito presentate (validità esterna). Conclusioni.

CAPITOLO 2

Stato dell'arte

2.1 Background e lavori correlati

2.1.1 Tipologie di dataset di vulnerabilità

Il dataset oggetto di studio in questa tesi è la Test Suite Juliet (nello specifico Juliet Java). Questa Test Suite è stata sviluppata dal NIST (National Institute of Standards and Technology) [6] e fa parte del SARD (Software Assurance Reference Dataset) [7] creato sotto il progetto SAMATE (Software Assurance Metrics And Tool Evaluation) [8]. Questo progetto portato avanti dal NIST ha lo scopo di unire metodi e processi per mitigare o rimuovere le debolezze e vulnerabilità presenti nel software al fine da garantirne il funzionamento previsto. Oltre il SARD, il progetto SAMATE comprende altri due progetti, il SATE (Static Analysis Tool Exposition) [9] e il Bugs Framework [10]. Il primo progetto è legato all'esplorazione delle caratteristiche dei tool di analisi statica in termini di rilevanza degli avvertimenti per la sicurezza, la loro correttezza e priorità, Bugs Framework invece è una classificazione delle debolezze software, strutturata, completa, ortogonale e indipendente dal linguaggio. Ogni classe rappresenta una tassonomia completa di un certo tipo di bug.

Questi però non rappresentano gli unici tipi di dataset disponibili. In generale, i dataset in circolazione secondo lo studio condotto da L. Liu et al [11] sono generalmente classificabili in tre categorie:

- *Dataset artificiali*: tra questi dataset ricade la Test Suite Juliet e quelli proposti dal SARD,

creati come dataset di benchmark per tool di analisi statica del codice. I dati contenuti in questo tipo di dataset sono relativamente semplici e sono affetti da una singola vulnerabilità.

- *Dataset semi-realistici*: questi dataset vengono popolati a partire da dati collezionati dal mondo reale, i quali vengono manualmente modificati per annotare ed indicare le vulnerabilità, come l'NVD (National Vulnerability Dataset).
- *Dataset Open-Source*: questi tipi di dato coinvolgono un ampio range di vulnerabilità e strutture differenti, riflettendo le vulnerabilità presenti nel mondo reale.

2.1.2 Problemi dei dataset sintetici

Esplorando lo stato dell'arte si evince un sempre più largo utilizzo di dataset del primo tipo per l'addestramento di modelli di intelligenza artificiale. Queste Test Suite sono state originariamente create per fungere da benchmark a tool intelligenti di detection delle vulnerabilità, e per questa ragione non sono di ottima qualità per allenare un modello di intelligenza artificiale. Questo tipo di dataset nasconde una serie di problematiche. I campioni vengono creati a partire da pattern predefiniti e per questo motivo mancano della diversità che è possibile trovare nel software prelevato da contesti reali. Questo è possibile osservarlo anche dalla distribuzione delle classi di vulnerabilità contenute in tali dataset, spesso infatti ci troviamo di fronte a dataset fortemente sbilanciati i quali favoriscono esageratamente determinate classi di vulnerabilità a discapito di altre. Un dataset sbilanciato subisce negativamente il cambiamento di popolarità di vulnerabilità che classifiche come OWASP Top Ten ci hanno abituato ad osservare. Ciò che un tool automatico per la detection di vulnerabilità aveva perfettamente imparato a riconoscere per la presenza di molti campioni in una specifica vulnerabilità potrebbe diventare obsoleto di conseguenza. I dataset sintetici inoltre potrebbero contenere molti sorgenti duplicati o quasi duplicati, ovvero sorgenti molto simili se non per differenze banali. La presenza di sorgenti di questo tipo infatti può comportare il rischio di data leak il cui fenomeno può gonfiare sproporzionatamente le performance del modello. Nello studio condotto da R. Russel et al [12] è stato osservato che il dataset Juliet C/C++ prima di essere soggetto a pre-processing conteneva 121,353 funzioni, e dopo aver effettuato il processo di rimozione dei duplicati, il numero delle funzioni è sceso a 11,896. Tra gli altri problemi legati ai dataset sintetci troviamo la scarsità di campioni in esso presenti e la distribuzione non uniforme tra i diversi linguaggi di programmazione. È stato osservato infatti dallo studio condotto da Y. Lin et al [13] che la maggior parte di dataset

sintetici in circolazione contiene codice C/C++, e solo al secondo e terzo posto troviamo dataset contenente codice Java e PHP. Inoltre, anche se i dataset contenenti codice C/C++ sono quelli più popolati, a malapena riescono a coprire le cinque vulnerabilità più diffuse. Queste limitazioni frenano di molto la possibilità di progettare ed implementare un modello per la detection di vulnerabilità multilinguaggio.

Questo rappresenta un problema in quanto la quantità e la qualità dei dati di addestramento determinano l'efficacia dei metodi intelligenti per la detection di vulnerabilità [5].

Nella letteratura è possibile osservare come questo tipo di dataset vengono utilizzati così come sono, magari apportando qualche piccolo accorgimento, oppure si tenta di espanderli in qualche modo.

Quest'ultimo è il caso di uno studio condotto da Russel et al [12] in cui è stato utilizzato il dataset Juliet C/C++.

Per espandere il dataset Juliet C/C++ è stato collezionato il maggior numero possibile di funzioni, facendo del mining da progetti Open-Source di GitHub e dalle distribuzioni di Debian. L'adozione di tecniche di mining è giustificata dalla natura sintetica dei dati presenti nel dataset Juliet C/C++ e dunque dalla volontà di fornire dei codici sorgenti provenienti da contesti reali. I sorgenti sintetici, da soli, non coprono a sufficienza lo spazio di codice naturale appropriato per un training set. Una volta effettuato il mining delle funzioni, bisogna capire quali di queste sono vulnerabili per poter attribuire una label indicante la vulnerabilità che contiene. Per fare ciò, tutte le funzioni non labellate sono state analizzate utilizzando tre tool di analisi statica, nello specifico: Clan, Cppcheck e Flawfinder. È stato poi compito di un team di ricercatori nel campo della sicurezza, analizzare i risultati ottenuti impiegando questi tool per poter dedurre la CWE di cui la funzione è affetta.

2.1.3 Metriche per descrivere dataset di vulnerabilità

Nel corso degli anni la ricerca nel campo dei dataset di vulnerabilità ha fatto uno sforzo per tentare di dare una descrizione sia qualitativa che quantitativa dei dataset esistenti. Più volte sono state definite diverse metriche e diverse feature per poter ottenere delle insight sui test case e sul codice contenuto in questi dataset.

Nel lavoro di L. Liu et al [11] viene data per la prima volta una descrizione oggettiva dei dataset utilizzati per la valutazione di tool intelligenti per la detection di vulnerabilità.

I dataset vengono descritti in termini di:

- Granularità: Definendo due livelli di granularità, a livello di funzione o a livello di singola linea di codice.

- Similarità: Definendo la similarità inter-classe e la similarità intra-classe per i campioni presenti nel dataset.

- Code features: Sono state definite cinque metriche: AvgCyclomatic, AvgEssential, AvgLine, AvgCountInput, AvgCountOutput per avere una descrizione oggettiva del codice sorgente e poterlo giudicare da diversi punti di vista. Il lavoro di Y. Lin et al [13] evidenzia come le feature sono essenziali per l'apprendimento di modelli di intelligenza artificiale. Avere un buon numero di feature a disposizione ci permette di descrivere la vulnerabilità più dettagliatamente. Tuttavia, la maggior parte dei dataset di vulnerabilità collezionano soltanto il codice sorgente di una vulnerabilità. È bene evidenziare che considerare troppo feature rappresenta anch'esso un problema in quanto va a degradare le performance del modello. Per queste reagioni, quali feature dovrebbe contenere un dataset di vulnerabilità è una domanda ancora aperta.

I risultati ottenuto da questo studio sono essenzialmente tre: (1) I campioni a grana fine aiutano a rilevare le vulnerabilità. (2) Dataset di vulnerabilità con più bassa similarità inter-classe, più alta similarità intra-classe e struttura semplice, aiutano il rilevamento di vulnerabilità nei dati contenuti nel test set. (3) Dataset di vulnerabilità con più alta similarità inter-classe, più bassa similarità intra-classe e struttura complessa possono aiutare nel rilevamento di vulnerabilità in altri dataset. Questo è dovuto al fatto che non è facile rilevare vulnerabilità in dati complessi, e la complessità delle features può allenare meglio l'abilità di rilevamento del detector di vulnerabilità basato su Deep Learning.

Come anticipato poc'anzi, le feature sono estremamente importanti per l'addestramento di vulnerability detector e per poter valutare e scegliere opportunamente un dataset di vulnerabilità. Nello studio condotto da R. Jain et al [14] Sono state definite delle caratteristiche per valutare la distribuzione del codice sorgente nei dataset analizzati, tra cui Juliet. Inoltre, queste feature sono state utilizzate per confrontare i dataset tra loro e per valutare la coerenza e l'uniformità dei dataset di vulnerabilità nella loro composizione. Le feature sono:

- *Dimensione del codice* (misurata in linee di codice)

- *if-conditions*: Rappresenta il costrutto base per il controllo e tramite esso vengono generati diversi branch nel codice.

- *Complessità ciclomatica*: La complessità ciclomatica è una metrica software che viene utilizzata per misurare la complessità di un programma. Questa metrica si basa sul grafo di controllo di flusso del programma e conta il numero di percorsi di esecuzione distinti attraverso il codice. In altre parole, la complessità ciclomatica indica il numero di decisioni prese durante l'esecuzione del programma. La complessità ciclomatica è spesso correlata alla qualità del codice, in quanto un valore elevato di questa metrica può indicare un codice difficile da leggere, testare e mantenere. Valori bassi della complessità ciclomatica, invece, possono indicare un codice più facile da comprendere e da mantenere.

- *Numero di chiamate a funzione*

- *Numero di Variabili locali e globali*

- *Chiamate alle funzioni della libreria libc*:La libreria libc (o glibc) è una libreria di sistema per i sistemi operativi Unix e Unix-like, che fornisce molte delle funzioni di basso livello necessarie per le applicazioni. La libreria libc è spesso associata alle vulnerabilità nel codice sorgente perché molte delle funzioni che essa fornisce sono vulnerabili ad attacchi informatici. Ad esempio, ci sono vulnerabilità legate alla gestione della memoria, come buffer overflow e heap overflow, che possono essere sfruttate dagli attaccanti per eseguire codice malevolo o per accedere a informazioni riservate. Inoltre, la libreria libc è ampiamente utilizzata nelle applicazioni di sistema e di basso livello, il che significa che una vulnerabilità in essa può avere un impatto significativo sulla sicurezza dell'intero sistema.

- *Presenza di strutture dati complesse come vettori e strutture*: Basti pensare come gli array, se non gestiti correttamente possono costituire un point of failure all'interno del codice. Dove c'è un array ci potrebbero essere per esempio problemi di buffer overflow.

Malgrado siano stati effettuati diversi tentativi nella definizione di code feature per la descrizione dei codici contenuti nei dataset, sia vulnerabili che non e malgrado la definizione di nuove metriche per descrivere i dataset di vulnerabilità, non sembra esserci ancora uno standard che metta tutti d'accordo.

Lo studio condotto da Y. Lin et al [13] ha evidenziato che non c'è un metodo di costruzione standardizzato unificato per soddisfare la valutazione standardizzata di diversi modelli di rilevamento delle vulnerabilità. Vengono elencate diverse sfide ancora aperte nella costruzione di un dataset di vulnerabilità tra cui (1) Affidabilità della fonte di dati: Nel campo della

sicurezza informatica, esistono molte organizzazioni che forniscono informazioni sulle vulnerabilità di sicurezza. Tuttavia, queste organizzazioni possono utilizzare specifiche diverse per descrivere le stesse vulnerabilità, causando una discrepanza di informazioni. Ciò può portare a errori o ridondanze nei dati delle vulnerabilità provenienti da diverse fonti. Per evitare questo problema, è importante sviluppare specifiche uniformi per descrivere e recuperare le informazioni da diverse fonti di dati. In questo modo, i dati possono essere utilizzati per l'apprendimento automatico senza la necessità di verifiche manuali, che richiedono molto tempo.

(2) Dati multi-sorgente mancanti: Quando si creano set di dati da diverse fonti, è possibile che ci siano parti mancanti o dati differenti tra le fonti. Questo può causare dati incompleti e difettosi per l'apprendimento automatico. Per esempio, se un'organizzazione fornisce informazioni sul codice sorgente di una vulnerabilità, ma un'altra organizzazione non lo fa, i dati non saranno completi e quindi non saranno utili per l'apprendimento automatico. Ciò può causare problemi nell'identificazione delle vulnerabilità e nella costruzione di modelli di apprendimento automatico.

(3) Feature: Le feature sono le caratteristiche utilizzate per descrivere le vulnerabilità nei dataset di apprendimento automatico. Tuttavia, non è ancora chiaro quali feature dovrebbero essere incluse nei dataset di vulnerabilità. Da un lato, è importante avere un numero sufficiente di feature per descrivere le vulnerabilità in modo dettagliato, dall'altro, troppi dati possono degradare le performance del modello. La scelta delle feature giuste è un'area di ricerca ancora aperta.

(4) Granularità: La granularità si riferisce al livello di dettaglio con cui le vulnerabilità sono descritte nei dataset. Ad esempio, un dataset può includere informazioni su singoli statement vulnerabili, ma anche su file, classi o funzioni. Una granularità adeguata è importante per individuare la radice del problema e le relazioni tra le funzioni. Tuttavia, una granularità eccessiva può causare problemi nella costruzione dei modelli di apprendimento automatico. Ad oggi, la granularità a livello di funzione è quella più utilizzata, ma ci sono ancora limiti nell'individuare tutte le vulnerabilità in un programma.

(5) Grandezza del dataset: I dataset di vulnerabilità sono generalmente limitati e potrebbero non coprire tutte le classi di vulnerabilità esistenti nel mondo reale. Per ovviare a questa mancanza, si è iniziato a creare dataset sintetici, iniettando vulnerabilità in programmi esistenti o costruendo programmi vulnerabili appositamente. Tuttavia, i dati sintetici possono essere meno affidabili rispetto ai dati reali, ma possono comunque essere utilizzati per migliorare l'accuratezza

2.2 Limitazioni e contributo

Attraverso l'analisi dei lavori emersi durante l'esplorazione dello stato dell'arte, è stato possibile evidenziare diverse limitazioni per tali studi. Queste limitazioni sono discusse di seguito, insieme a quale contributo originale questa tesi apporta.

- Nel lavoro di Russel et al [12], si è tentato di creare un dataset ibrido combinando codice sorgente proveniente da contesti reali. Tuttavia, si è riscontrato un problema nella definizione di una label univoca che potesse essere utilizzata dai tool di analisi statica per identificare eventuali vulnerabilità.

- Nel lavoro di L. Liu et al [11] si è cercato di descrivere i dataset di vulnerabilità in termini di granularità, similarità e feature del codice. Inoltre, è stato analizzato come queste componenti influenzino la capacità di un rilevatore di vulnerabilità di individuare le vulnerabilità. Tuttavia, lo studio si è concentrato solamente sul codice C/C++ e non sono state approfondite possibili strategie per migliorare la qualità dei dataset sintetici.

- Nel lavoro di R. Jain et al [14] non viene specificato come i dataset analizzati siano stati fusi insieme e ne quali tecniche siano state impiegate nella creazione del dataset finale. Inoltre questo studio si concentra solo sulla rilevazione di vulnerabilità nel codice C/C++.

In sintesi, nel campo della detection di vulnerabilità attraverso l'uso di strumenti intelligenti, la ricerca non si è ancora spinta verso alcuni campi. Il contributo di questa tesi è orientato:

- A dedurre dal panorama dell'arte quali sono le best practice per ottimizzare i dataset sintetici, utilizzando come caso di studio la Test Suite Juliet, ed in particolare Juliet Java.

- Ad esplorare la possibilità di utilizzare approcci di data augmentation nel contesto di dataset sintetici.

CAPITOLO 3

Metodo di Ricerca

3.1 Obiettivi di Ricerca

Questa tesi vuole essere un contributo nel definire quali metriche possono essere prese in considerazione all'atto della creazione di un dataset di vulnerabilità, in maniera tale da promuovere uno standard adottabile da ricercatori futuri. Tramite l'esplorazione di tecniche di augmentation applicate a sorgenti vulnerabili si vuole inoltre dare un'alternativa a tecniche basate su trasformatori o su iniezione di codice vulnerabile e valutare come l'impiego di tali tecniche ha effetto su modelli di classificazione addestrati su tali dati. Nello specifico questa tesi mira a raggiungere i seguenti obiettivi:

1. Investigare sui problemi principali che riguardano i dataset sintetici di vulnerabilità e qual è un sottoinsieme di metriche utilizzabile per descriverli.
2. Investigare sulle tecniche applicabili per fare data augmentation su codice sorgente.
3. Misurare l'impatto di alcune di queste tecniche applicate a modelli di detection di vulnerabilità.

In merito al punto uno, sono stati presi in considerazione un sotto-insieme dei problemi osservati sui dataset sintetici di vulnerabilità. Nello specifico:

- Il problema legato alla mis-classification delle debolezze all'interno di questi dataset, e dunque alla loro affidabilità

- Il problema della granularità dei campioni disponibili, definendo una nuova granularità a livello di funzione
- Il problema del feature engineering, e quindi la creazione di una nuova feature per aiutare il processo di addestramento di un classificatore

Le ipotesi su cui questo lavoro è basato sono le seguenti:

- Ipotesi I: una granularità più fine dei campioni aiuta i classificatori ad imparare meglio le caratteristiche di una vulnerabilità.
- Ipotesi II: la classificazione di sorgenti vulnerabili sotto una specifica CWE è un compito difficile. I Software Engineer ed i tool automatici di classificazione di vulnerabilità possono sbagliare ad attribuire una CWE ad un sorgente vulnerabile. Questo significa che c'è una percentuale di label associata agli elementi del dataset che è erroneamente attribuita
- Ipotesi III: l'applicazione di tecniche di augmentation applicate a sorgenti vulnerabili è possibile e permetterebbe di: velocizzare la creazione di un Dataset di vulnerabilità, controllarne le caratteristiche, generare un numero più o meno arbitrario di campioni, condurre analisi più complesse, costruire modelli di detection di vulnerabilità più accurati, aggiornati e generalizzabili.

Prima di avviare la fase sperimentale dunque, si attendevano dei risultati che mirassero a confermare la mis-classisfication dei sorgenti vulnerabili, mostrando magari come utilizzando un altro sistema di classificazione potesse generare delle label più accurate. Altri due risultati attesi erano che:

- utilizzando una granularità dei campioni a livello di funzione, si potesse aiutare un classificatore a porre un maggior focus sulle vulnerabilità, ottenendo una accuracy migliore nel processo di classificazione
- addestrando un modello di deep learning su un dataset su cui è stato fatto data augmentation, gli avrebbe permesso di essere generalmente migliore rispetto ad un modello allenato sul dataset originale.

Per la scelta degli esperimenti da condurre è stato preso in considerazione il dataset Juliet Java. La ragione dietro questa scelta è stata quella di condurre degli esperimenti utilizzando dei campioni scritti in un linguaggio di programmazione su cui non sembrano essere stati

condotti molti esperimenti di questo genere. Esplorando lo stato dell'arte infatti è stato riscontrato che la maggior parte degli studi ha fatto utilizzo della versione C della Test Suite Juliet. Non è stato preso in considerazione gli altri dataset accennati nella fase di Data Exploration in quanto, teoricamente, le azione qui intraprese sono riproducibili anche in altri linguaggi di programmazione e dunque utilizzando altri dataset. Per la scelta delle tecniche di data augmentation invece: (i) Non sono stati presi in considerazione studi nei quali si facesse riferimento a tecniche di augmentation basate sullo scraping di codice sorgente da progetti pubblici (ii) Sono stati invece presi in considerazione i principali studi adoperanti tecniche di augmentation basate sul concetto di *trasformazione* del codice sorgente e manipolazione dello stesso in forma vettoriale.

3.2 Domande di Ricerca

Replicare la realtà è un problema che affligge anche il campo dell'ingegneria del software. Questa difficoltà è ancora più evidente nel campo della ricerca sulle vulnerabilità. Una vulnerabilità è un qualcosa di molto sottile, che può interessare anche una sola riga di codice su un totale di centinaia o forse migliaia di righe, inoltre il software presente in "natura" può arrivare ad essere molto complesso e contenere più di una vulnerabilità. Nel campo della ricerca riguardo la costruzione di tool di analisi statica e dinamica per la detection di vulnerabilità spesso ci si è imbattuti nel problema di dove reperire campioni vulnerabili e di conseguenza su quali campioni andare a testare le loro performance. Inoltre, non c'è una base condivisa di codice vulnerabile e non esiste uno standard da seguire in merito a come deve essere costruito tale dataset e quali caratteristiche dovrebbe essere conforme. Nel lavoro svolto da X. Li et al [15] sono stati individuati ben 26 database di vulnerabilità, ognuno con una propria struttura, obiettivi e caratteristiche differenti.

Nel corso degli anni l'utilizzo di dataset sintetici per allenare i tool per la detection di vulnerabilità ha incominciato a diffondersi. Molti di questi dataset tra cui la Test Suite Juliet non è stata pensata originariamente per essere utilizzata come fonte per l'addestramento di tali tool, ma bensì come benchmark per la valutazione delle loro performance. Questo tipo di dataset però è affetto da diversi problemi.

Digitando la "Juliet Test Suite" su Google Scholar, ci si rende facilmente conto del grado di diffusione di questo dataset. Questo ci fa intuire quanto la diffusione di un dataset fallace per costruzione possa rappresentare un problema per lo sviluppo di tool intelligenti per la detection di vulnerabilità.

Da qui nasce la prima Research Question:

> *RQ.1: Quali sono le best practice per migliorare un dataset sintetico di vulnerabilità, e nello specifico la Test Suite Juliet?*

Questi dataset di vulnerabilità inoltre soffrono di altre due problematiche importanti, Scarsità di campioni disponibili e dataset sbilanciati a favore di alcune CWE.

Uno dei principali metodi attualmente utilizzati per sopperire a questi problemi è quello di cercare di raccogliere più campioni possibile, da qualsiasi fonte disponibile. C'è una strada però poco esplorata, ovvero quella della data augmentation nel contesto di sorgenti vulnerabili. Da qui nasce la seconda Research Question:

> *RQ.2: Qual è l'impatto della data augmentation sulle prestazione di un modello di detection di vulnerabilità?*

3.3 Sperimentazione

Per validare o confutare le ipotesi avanzate, sono stati condotti dei test, nei quali sono stati addestrati dei classificatori di vulnerabilità. Le metriche adottate per valutare le performance di questi modelli sono Precision, Recall, F1-Score, Support.

Per condurre questi esperimenti, è stato necessario esplorare quali possibilità fossero disponibili per rappresentare del codice sorgente in una forma alternativa, e quali tecniche adottare per l'augmentation del dataset di partenza.

Secondo lo studio condotto da L. Liu et al [11] i metodi per rappresentare un codice sorgente in forma vettoriale ricadono in tre categorie:

- Rappresentazione basata su sequenza: Il codice viene trattato come una semplice sequenza testuale, senza badare alla sua struttura interna. Un tool che permette di realizzare ciò è per esempio Word2Vec.

- Rappresentazione basata su AST (Abstract Syntax Tree): È una rappresentazione ad albero della struttura sintattica astratta del codice sorgente. Per prima cosa, decompone il codice in una serie di cammini chiamati Abstract Syntax Tree, poi usa una rete neurale per apprendere la rappresentazione di ogni cammino e come integrare la rappresentazione di tutti i path. Un esempio di tool che permette di realizzare ciò è invece Code2Vec.

- Rappresentazione basata su grafi: È una rappresentazione che si basa sulla codifica delle dipendenze tra i dati di controllo sottoforma di grafi. Questo tipo di rappresentazione si concentra di più sul controlo di flusso e sul flusso dei dati. Per esempio Gated Graph Neural Networks (GGNN).

L'idea di utilizzare una rappresentazione alternativa dei codici sorgenti è emersa a partire dai precedenti individuati nella fase di esplorazione dello stato dell'arte. Nel lavoro di X. Li et al [5] per la generazione degli embedding è stato utilizzato Word2Vec fornito dalla libreria Gensim, mentre nel lavoro di R. Russel et al [12] è stato creato un Lexer C/C++ progettato per catturare i token ritenuti più significativi. La giustificazione dietro la creazione di un Lexer custom è derivata dal fatto che, secondo le loro osservazioni, i lexer progettati per la compilazione di codice vanno a catturare troppi dettagli che possono indurre in Overfitting. Il Lexer creato a partire da questo studio utilizza degli embeddings a 156 dimensioni. Ogni elemento dell'embedding rappresenta un token i quali comprendono tutte le keyword base di C/C++, operatori e separatori.

Per condurre gli esperimenti in questa tesi esperimenti la scelta è ricaduta sulla rappresentazione dei codici sorgenti in forma vettoriale utilizzando Code2Vec.

Code2Vec opera attarverso i passaggi seguenti:

- Tokenizzazione ed embedding: Inizialmente, il codice viene diviso in token, come parole chiave, identificatori di variabili e operatori. Ogni token viene poi mappato in uno spazio vettoriale continuo attraverso un processo chiamato "embedding". Questo cattura le relazioni semantiche tra i token.

- Creazione degli AST: Code2Vec costruisce alberi di sintassi astratta (AST) basati sui percorsi dei token nel codice. Gli alberi AST catturano la struttura e le relazioni tra le diverse parti del codice, come le dipendenze tra le variabili e le chiamate alle funzioni.

- Apprendimento delle rappresentazioni: Utilizzando una rete neurale profonda, il modello apprende a trasformare questi percorsi AST in rappresentazioni vettoriali significative. Queste rappresentazioni catturano sia le informazioni semantiche che quelle strutturali del codice.

In generale quindi, l'utilizzo di Code2Vec, a differenza di una rappresentazione testuale, ha permesso di conservare la struttura del codice, utilizzando gli alberi AST. In questo modo le dipendenze tra le parti del codice sono state preservate, cosa che non sarebbe accaduta se si fosse fatto uso di rappresentazioni puramente testuali. Inoltre l'utilizzo di vettori risulta

sicuramente più semplice da comprendere e maneggiare rispetto a forme di rappresentazione più complesse di codice sorgente come per esempio la rappresentazione mediante grafi.

Tuttavia la scelta di utilizzare una rappresentazione diversa potrebbe essere una interessante strada da percorrere come sviluppo futuro. Potrebbe essere interessante utilizzare delle rappresentazioni basate su grafi e confrontare come una queste rappresentazioni alternative impattano sui risultati finali.

Per quanto riguarda invece la data augmentation, essa è una delle principali tecniche adottate per migliorare le capacità di generalizzazione di un modello di intelligenza artificiale. L'essere generale nel contesto del Deep Learning sta ad indicare la capacità di un modello di performare bene su dati mai visti in precedenza, tanto quanto performa sui dati utilizzati per l'addestramento.

Per rendere un modello il più generale possibile bisogna far si che vengano forniti ad esso un grande numero di campioni, rappresentati sotto "diverse angolazioni" in modo da permettergli di imparare meglio quali siano le caratteristiche discriminanti. Uno dei metodi utilizzati per la data augmentation di codice sorgente è applicare delle trasformazioni al codice. Queste trasformazioni consistono nel manipolare gli statements in maniera tale da conservarne la semantica ma rendere il codice trasformato differente da quello originale. Questo approccio dovrebbe essere applicato con cautela. Applicando infatti delle trasformazioni molto banali si potrebbe incorrere nel rischio di generare dei *near duplicates* o peggio ancora dei *cloni* andando a favorire l'overfitting di un ipotetico modello che fa uso di questi dati.

Nello studio condotto da S. Yu et al [16] sono state proposte 18 trasformazioni comprovate essere semantic e natural preserving, ovvero capaci di conservare sia il significato del codice che la naturalezza con la quale è scritto. Per naturalezza si intende "quanto si avvicina ad un codice scritto da un essere umano".

Queste trasformazioni sono riportate nella Tabella 3.1

ID	Rule name	Description
3	ReverseIfElse	Switch the two code blocks in the if statement and the corresponding else statement.
4	SingleIf2ConditionalExp	Change a single if statement into a conditional expression statement.
5	ConditionalExp2SingleIF	Change a conditional expression statement into a single if statement.
8	InfixExpressionDividing	Divide a infix expression into two expressions whose values are stored in temporary variables.
9	IfDividing	Divide a if statement with a compound condition into two nested if statements.
10	StatementsOrderRearrangement	Switch the places of two adjacent statements in a code block, where the former statement has no shared variable with the latter statement.
11	LoopIfContinue2Else	Replace the if-continue statement in a loop block with if-else statement.
14	SwitchEqualSides	Switch the two expressions on both sides of the infix expression whose operator is =.
15	SwitchStringEqual	Switch the two string objects on both sides of the String.equals() function.
16	PrePostFixExpressionDividing	Divide a pre-fix or post-fix expression into two expressions whose values are stored in temporary variables.
17	Case2IfElse	Transform the "Switch-Case" statement into aht ecorresponding "If-Else" statement.
0	LocalVarRenaming	Replace the local variables' identifiers with new non-repeated identifiers.
1	For2While	Replace the for statements with an semantic-equivalent while statement.
2	While2For	Replace the while statement with an semantic-equivalent while statement.
6	PP2AddAssignment	Change the assignment x ++ into x+=1
7	AddAssignment2EqualAssignment	Change the assignment x+=1 into x+=x+1
12	VarDeclarationMerging	Merge the declaration statements into a single composite declaration statement.
13	VarDeclarationDividing	Divide the composite declaration statement into separated declaration statements.

Tabella 3.1: Trasformazioni proposte da Yu et al in "Data Augmentation by Program Transformation"

Una cosa molto importante da precisare è che Il concetto di equivalenza semantica è utilizzato principalmente per assicurare che degli snippet di codici su cui sono state applicate trasformazioni possano condividere la stessa label col codice di partenza. Questo requisito non è invece richiesto quando si parla di codice vulnerabile. Questo è un aspetto molto importante e poco stressato negli studi analizzati.

Due codici semanticamente diversi, infatti, possono essere affetti dallo stesso bug. Di conseguenza l'unica cosa che conta quando si effettuano trasformazioni su codice contenente bug è che dopo l'applicazione della trasformazione esso continui a presentare tale bug. Questo è un aspetto estremamente importante perché ci permette di rilassare il vincolo di utilizzare soltanto delle trasformazioni semantic preserving, aprendo uno scenario molto più ampio alle manipolazioni applicabili a sorgenti vulnerabili.

È bene precisare che il termine *trasformazione* non è legato soltanto alla manipolazione del codice sorgente attraverso parser o in plain text, ma è possibile adottare anche delle soluzioni più originali. Un codice sorgente può essere visto come una immagine (e quindi come una matrice). Su questa matrice possiamo effettuare tutte le operazioni che è possibile compiere su una matrice. Possiamo: Specchiarla, Ruotarla, etc.

Per avviare dunque la fase sperimentale, tramite Code2Vec, sono stati estratti un insieme di vettori, a due livelli di granularità 3.2:

- Granularità a livello di classe: Code2Vec è in grado di rappresentare in forma vettoriale del codice sorgente associato ad una funzione 3.1. Dato che in una classe ci sono più funzioni, ciò ha richiesto la manipolazione del sorgente originale di questo Tool affinchè, dopo l'estrazione dei vettori associati alle singole funzioni, fosse possibile ottenere in output un singolo vettore. Per accorpare i diversi vettori in un singolo vettore è stato applicata la media tra i vettori. Sarebbe interessante, valutare in futuro come metodi diversi di accorpamento di questi singoli vettori possa incidere sul risultato finale.

- Granularità a livello di singola funzione: a partire da ciascuna classe Java, ogni funzione in essa presente è stata convertita nella sua rappresentazione vettoriale e memorizzata.

Avendo tutto ciò di cui avevo bisogno è stato possibile pianificare e condurre i seguenti esperimenti:

1. Addestramento di una rete neurale sui vettori con granularità a livello di classe, utilizzando come label le CWE originali

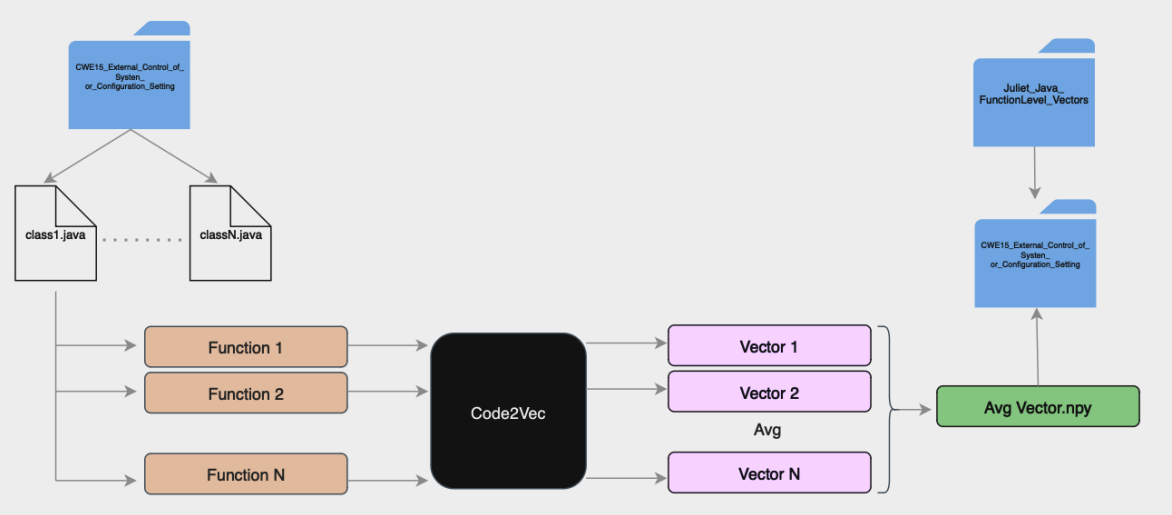

Figura 3.1: Pipeline di estrazione dei vettori dalle classi Java

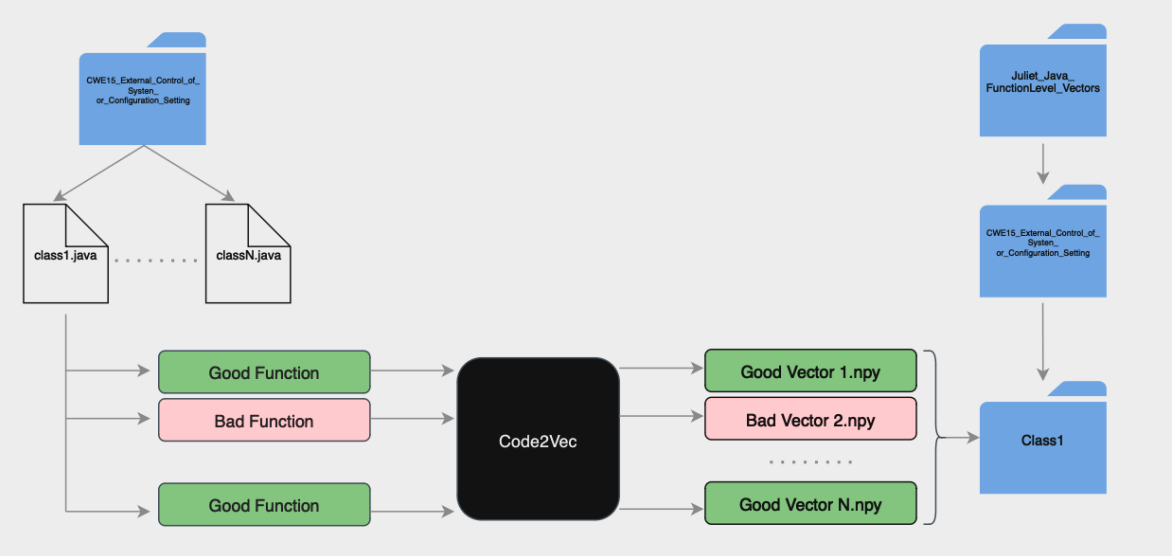

Figura 3.2: Pipeline di estrazione dei vettori dalle funzioni Java

2. Addestramento di una rete neurale su dei vettori con granularità a livello di classe, utilizzando come label le CWE originali, accorpando le CWE figlie nelle CWE più grandi

3. Addestramento di una rete neurale su vettori con granularità a livello di classe, utilizzando come label delle nuove label generate tramite K-Means

4. Addestramento di una rete neurale sui vettori con granularità a livello di funzione utilizzando come label le CWE originali

5. Addestramento di una rete neurale sui vettori con granularità a livello di funzione utilizzando come label le label originali, aggiungendo come nuova feature il cluster di appartenenza generato da K-Means

6. Addestramento di una CNN e Rete Neurale su dataset oggetto dell'augmentation.

3.3.1 Esperimento N.1 e N.2

Dall'applicazione di Code2Vec è stato possibile ricavare per ogni CWE un'unica matrice, la quale ha come righe la rappresentazione vettoriale delle classi vulnerabili. Tali vettori hanno 384 componenti. Prima di procedere con l'addestramento della rete neurale, I vettori sono stati tutti accorpati in una nuova matrice. Tale matrice è stata denominata "vectors". Parallelamente sono state create anche la matrice delle label, creando le etichetta a partire dal nome delle matrici le quali sono state nominate con la CWE corrispettiva.

La Test Suite Juliet Java definisce 112 CWE diverse e di conseguenza abbiamo a disposizione altrettante Label.

Nell'esperimento N.2 È stata eseguita una fase di pre-processing sulle Label. Esaminando le CWE a disposizione mi sono reso conto che alcune CWE rappresentavano delle macro-categorie, le quali possono essere scisse in CWE più specifiche. Piuttosto che scindere queste macro-categorie si è preferito accorpare le CWE figlie all'interno della CWE genitore. È stato questo il caso per le seguenti CWE:

- CWE535 accorpata nella CWE209 Information Leak Error
- CWE600 accorpata nella CWE248 Uncaught Exception
- CWE759, 760, 319, 328, 614 accorpate nella CWE327 Use Broken Crypto
- CWE789 accorpata nella CWE400 Resource Exhaustion

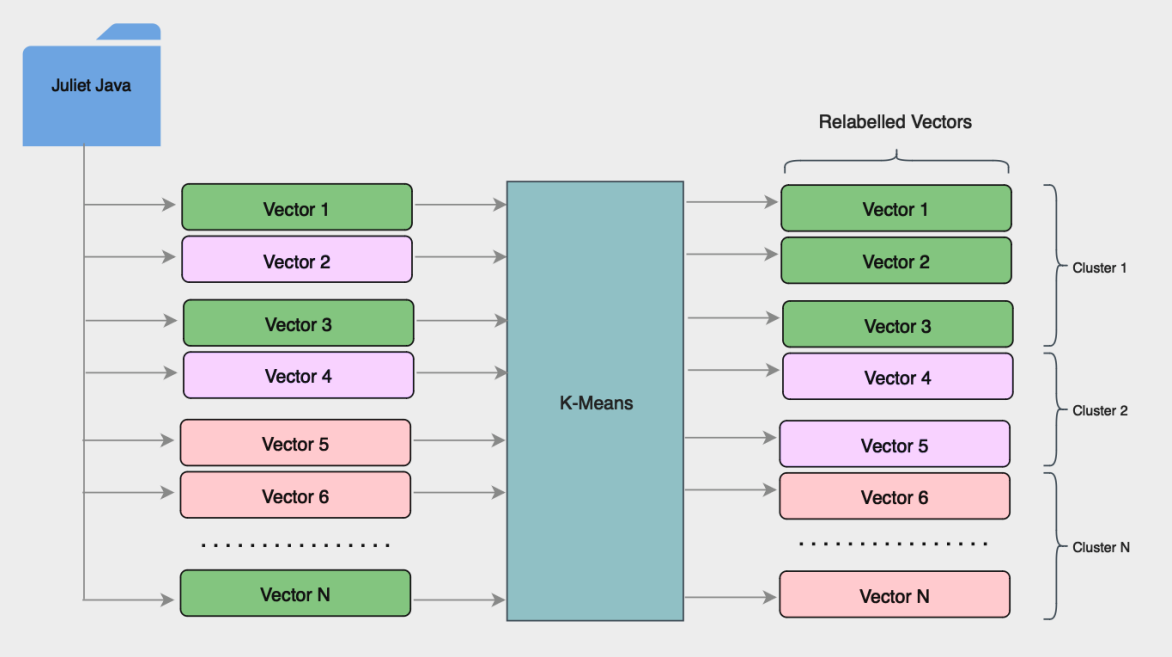

Figura 3.3: Re-labelling dei campioni mediante K-Means

- CWE568, 775, 772, 459, 226 accorpata nella CWE404 Improper Resource Shutdown
- CWE510, 511 accorpata nella CWE506 Embedded Malicious Code
- CWE609, 764, 765, 832, 833 accorpata nella CWE667 Improper Locking
- CWE197 accorpata nella CWE681 Incorrect Conversione Between Numeric Types

Dopo l'accorpamento, il numero delle label è sceso a 91.

Il dataset è stato diviso dunque in Training, Testing e Validation Set. Tale suddivisione è stata fatta in maniera casuale ma stratificata. Questo significa che si è tentato di mantenere la stessa distribuzione delle classi tra i diversi insiemi, in modo tale da non favorire nessuna classe in particolare. Le label prima di essere utilizzate sono state codificate tramite la tecnica One-Hot-Encoding la quale mi restituisce una rappresentazione binaria, comprensibile alla rete neurale.

3.3.2 Esperimento N.3

In questo esperimento si è voluto testare l'ipotesi che le label assegnate a ciascun vettore vulnerabile potessero non essere del tutto accurate. Tali label infatti sono state assegnate menualmente o tramite l'utilizzo di tool automatici. Per questo motivo è stato allenato un algoritmo di clustering, nello specifico si è fatto utilizzo di K-Means in maniera da "raggruppare" i vettori sulla base di quanto fossero più o meno vicini tra loro all'interno

di uno spazio N-dimensionale. Dopo aver concluso questo processo sono state ottenute delle nuove label le quali indicano i cluster a cui appartengono i vettori 3.3. L'ipotesi era quella che addestrando una rete neurale per la classificazione di debolezze utilizzando queste nuove label, avrei ottenuto un incremento nella capacità di classificazione, rispetto al modello allenato sulle label originali. Il motivo per cui si è deciso di combinare K-Means con le reti neurali è che K-Means mi permette di generare nuove label per i sorgenti al fine di addestrare una rete neurale utilizzando queste nuove label invece che quelle originali. Clusterizzando i dati otteniamo relazioni oggettive che ci fanno capire come le CWE sono legate tra loro. Potrebbe essere un approccio più veloce ed accurato rispetto ad una classificazione manuale. K-Means però non può essere usato da solo per la classificazione di un nuovo sorgente, sotto una specifica CWE in quanto: (1) Assume che i cluster siano sferici e che i confini tra loro siano lineari. Non sempre però è così. Le reti neurali possono aiutare a modellare relazioni non lineari tra le feature, aumentando l'accuratezza della classificazione. (2) K-Means è sensibile a dataset sbilanciati, dove ci sono cluster molto più grandi di altri. In definitiva, tramite K-Means otteniamo un primo partizionamento dei dati in cluster, e la rete neurale in seguito, apprende le relazioni più complesse tra le feature ed i cluster. Quando si fa uso di K-Means diventa fondamentale la scelta del numero di cluster come parametro dell'algoritmo. In sostanza è possibile compiere due scelte:

- Scegliere un numero di cluster uguale al numero di label: In origina abbiamo 112 label, 91 dopo la ridistribuzione, le quali rappresentano CWE diverse contenute nel dataset Juliet Java. Adoperando un numero di cluster pari al numero di CWE, conserviamo la granularità della classificazione, introducendo al tempo stesso una ridistribuzione dei campioni. Sorgenti che appartenevano a CWE diverse possono ora ritrovarsi sotto lo stesso cluster a prova che la classificazione di un sorgente sotto una CWE è un compito difficile e spessi ci si sbaglia. Di conseguenza il cluster di appartenenza diventano le nuove label dei campioni.

- Utilizzare un numero di cluster inferiore al numero di label disponibili: In questo caso per determinare il numero di cluster da utilizzare si è fatto utilizzo dell'elbow technique 3.4 tramite la quale è possibile determinare qual è il numero ottimale di cluster. È stato riscontrato essere circa 17. Perseguendo quesa strada si perde la granularità della classificazione di un sorgente vulnerabile. Adoperando un numero di cluster inferiore al numero di label stiamo producendo un accorpamento di diverse CWE. Possiamo ottenere in questo modo una visione di insieme più a largo spettro, mettendo

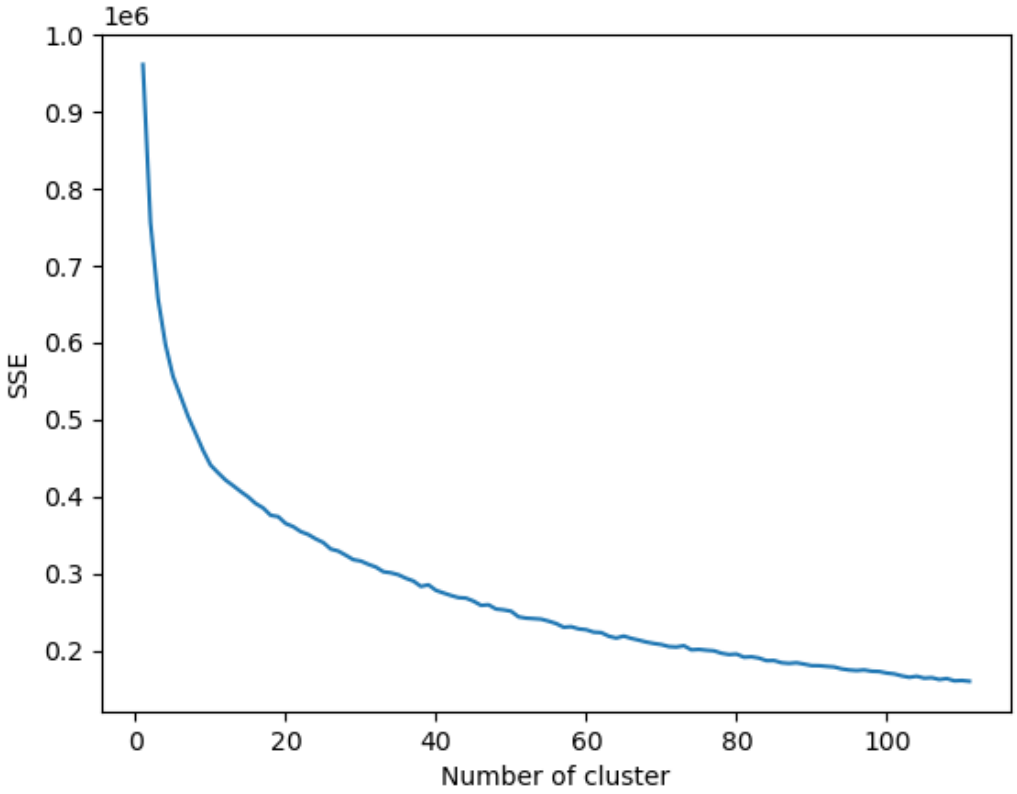

Figura 3.4: Risultati dell'applicazione dell'elbow technique

in relazione sotto un unico cluster CWE simili

In questo esperimento prima di eseguire K-Means è stata eseguita una riduzione dimensionale sui vettori di vulnerabilità. Code2Vec mi ha infatti restituito dei vettori a 384 dimensioni. Un tale numero di dimensioni potrebbe catturare anche dettagli delle vulnerabilità che non sono rappresentativi. Per questo motivo è stato fatto utilizzo di un algoritmo di PCA, per ridurre le dimensioni a 15. Questa pratica è stata suggerita dal lavoro condotto da L. Liu et al [11]. In questo studio infatti affermano che per trattenere le informazioni più importanti hanno ridotto le dimensioni dei vettori dapprima a 15 dimensioni utilizzando PCA, per poi utilizzare T-SNE per ridurre la dimensionalità dei vettori a due dimensioni.

3.3.3 Esperimento N.4

Nella conduzione di questo esperimento si è dunque deciso di abbassare la granularità dei campioni impiegati. Infatti, negli esperimenti condotti in precedenza si è fatto uso di vettori i quali rappresentavano intere classi Java. Queste classi però al loro interno contengono sia funzioni vulnerabili che non vulnerabili. Di conseguenza un vettore estrapolato da una

classe Java è il risultato della media dei vettori estratti da ogni funzione presente all'interno della classe. Questo, dai risultati degli studi analizzati, potrebbe rendere più difficile il compito di apprendere le caratteristiche peculiari di una vulnerabilità da perte di un modello di intelligenza artificiale. È stato osservato, che utilizzare una granularità più fine o una granularità mista, potrebbe aiutare tali modelli ad apprendere meglio le caratteristiche discriminanti. Di conseguenza è stato utilizzato Code2Vec per generare per ogni classe:

- Per ogni funzione vulnerabile, un vettore vulnerabile. L'insieme di questi vettori vulnerabili sono contenuti all'interno della matrice bad_npys.csv
- Per ogni funzione non vulnerabile, un vettore vulnerabile. L'insieme di questi vettori non vulnerabili sono contenuti all'interno della matrice good_npys.csv

Per migliorare la qualità dei dati e garantire che la rete neurale imparasse da esempi significativi, è stata eseguita un'operazione di pre-elaborazione. In particolare, sono state rimosse le etichette che avevano meno di 10 esempi nel dataset. Questo passaggio era importante per evitare che classi con pochi campioni influenzassero negativamente l'addestramento della rete neurale.

Successivamente, è stato calcolato nuovamente il numero di etichette uniche rimaste nel dataset ed è stato registrato questo valore nella variabile numberOfClasses.

3.3.4 Esperimento N.5

In questo esperimento si è voluto testare quanto la capacità di classificazione della rete neurale precedentemente allenata sulle singole funzioni vulnerabili, cambiasse, aggiungendo una nuova feature ai vettori vulnerabili. La feature candidata per l'aggiunta è risultata sin da subito essere il cluster di appartenenza. L'ipotesi era che aggiungendo questa nuova feature, essa poteva dare un contributo significativo nell'addestramento della rete neurale.

3.3.5 Esperimento N.6

La fase sperimentale per la data augmentation invece ha richiesto un ulteriore trattamento. Facendo riferimento alla Figura 3.5 ogni vettore di 383 componenti estratto da ogni classe vulnerabile è stato convertito in una matrice 16x24 e poi in una immagine. La conversione è stata realizzando mappando il valore di ogni singola componente dei vettori come un valore compreso tra 0 e 255.

La fase di augmentation è stata realizzata utilizzando un filtro immagine, nello specifico un filtro luminosità. Tale filtro è stato applicato a campioni prelevati in maniera casuale

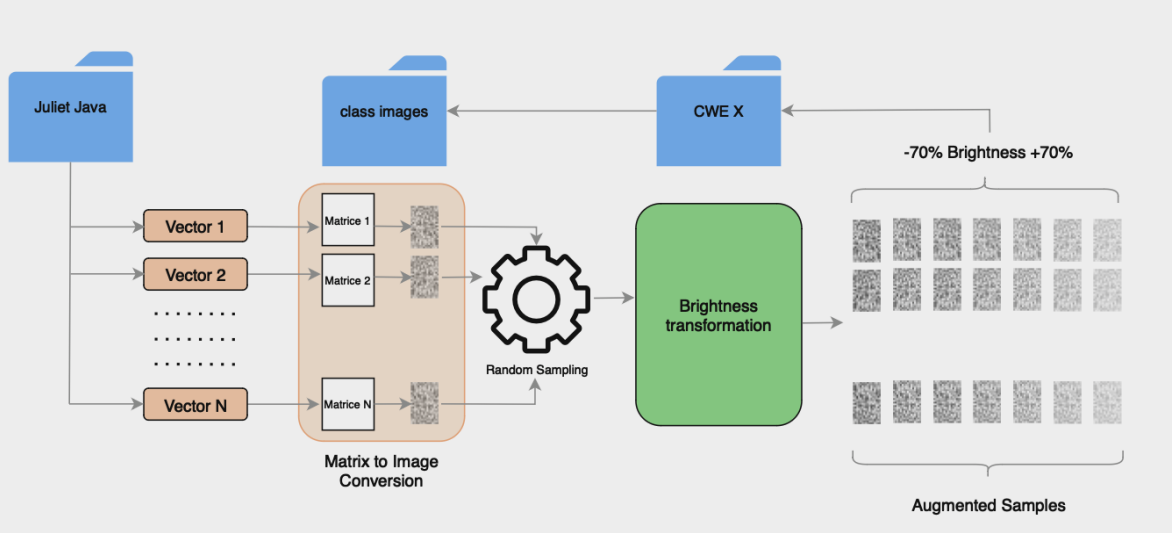

Figura 3.5: Pipeline di data augmentation

a partire da ciascuna cartella rappresentante una CWE. La percentuale di variazione di luminosità da apportare ad un campione è scelta a caso. Questa può variare dal -70% a +70%.

Sono state evitate percentuali più alte in quanto, ottenere campioni completamente bianchi o completamente neri, avrebbe significato introdurre dei cloni. Questo avrebbe apportato problematiche di Data Leak.

Applicando questa tecnica, ottengo dei campioni simili ma formalmente diversi, in quanto il filtro non ha modificato l'ordine dei pixel, ma ne ha in sostanza scalato i valori. Questo è solo uno dei possibili filtri che potrebbe essere applicato. È possibile combinare l'applicazione di filtri in maniera tale da aumentare il numero di campioni estraibile a partire da un singolo campione, rendendo il dataset più vario.

Durante il processo di augmentation è stato affrontato anche il problema dello sbilanciamento del dataset a favore di un numero ristretto di classi.

Per ottenere un dataset più bilanciato, sono stata eseguite le seguenti azioni:

- Scelta randomica dei campioni da rimuovere per ciascuna CWE. L'obiettivo è ottenere un dataset con 500 campioni per CWE
- Applicazione della tecnica di augmentation descritta, per aumentare il numero di campioni per quelle CWE con meno di 500 esemplari.

Questo metodo ovviamente va a scartare dei dati. Tale perdita è giustificata soltanto ai fini dell'esperimento per testare l'ipotesi che un dataset bilanciato potesse generalmente restituire risultati migliori.

Il dataset così creato, insieme alle label rappresentate in One-Hot Encoding, è stato fornito ad una CNN per eseguirne l'addestramento.

Successivamente è stato eseguito l'addestramento di una rete neurale in maniera tale da confrontare le prestazioni della stessa con quelle della CNN. Per eseguire l'addestramento della rete neurale sul dataset aumentato, è stato necessario condurre un passaggio in più prima della fase di training. Dato che l'output della fase di augmentation è rappresentato da immagini, queste sono state convertite di nuovo in vettori prima di poter essere usate dalla rete neurale.

CAPITOLO 4

Analisi dei Risultati

4.1 Risultati per Domanda di Ricerca RQ. 1

Data la diversità e quantità di dataset in circolazione è stato necessario ridefinire lo scope. Per questo motivo, questo lavoro di tesi mira all'esplorazione della sola Test Suite Juliet Java, fornendo però degli accenni anche su Juliet C++/C#/C ed il SateIV Java/C.

Juliet è stata rilasciata per la prima volta nel dicembre 2010, e da allora sono state rilasciate sucessivamente due versioni: la versione 1.2 rilasciata nel maggio 2013 e la versione 1.3 rilasciate nell'ottobre 2017, rispettivamente contenente codice Java e codice C/C++. La versione di Juliet contenente codice Java subisce una laggera modifica da parte di Paul E. Black il quale nel novembre dello stesso anno pubblica *Juliet Java 1.3 with extra support*. Anche la versione C/C++ subisce un update su 28 test case, realizzato questa volta da Paul E. Black e Yann Prono nell'agosto del 2022. Infine, nello stesso mese la Test Suite Juliet viene ampliata introducendo dei casi di test anche per il linguaggio di programmazione C#.

I cambiamenti apportati dalla versione 1.3 rispetto la 1.2 sono riportati all'interno del lavoro di P. E. Black et al [17]. Nella Figura 4.2 è possibile osservare come la Test Suite Juliet C/C++ sia la più popolata. Questo fa comprendere anche come mai la maggior parte dei lavori di ricerca ne fa utilizzo. Il numero di test case per la versione Java e C# della Suite invece sono quasi equivalenti.

I test case di Juliet sono organizzati in due parti: (1) test case e (2) file di supporto necessari per la compilazione dei casi di test. Ogni CWE ha la propria sotto-directory 4.1. Le CWE

con meno di mille test case contengono tutti i test case direttamente nella sotto-directory, mentre le CWE con più di mille files sono divise in sotto-directory chiamate s01, s02, etc. Ogni file di test case ha un nome univoco. Il nome file è costituito dal numero CWE e dal nome, due trattini bassi, seguiti da vari tipi identificativi, funzioni e alternative, quindi un numero di variante di controllo. Ogni test case ha una struttura diversa. Essi dispongono di una funzione nella quale è presente un bug e di altre funzioni con un comportamento simile ma senza bug.

La versione 1.3 della Test Suite Juliet malgrado sia un miglioramento rispetto la 1.2 contiene alcuni problemi ereditati dalla versione precedente che vanno dalla perdita di memoria all'accesso non sicuro alla memoria e bug intenzionali nel codice valido. Ci sono anche alcune questioni legate alla sicurezza che devono essere risolte.

Attualmente Il NIST non prevede la creazione di una nuova versione di Juliet, ma è orientato a generare test personalizzati per ogni utente invece di avere un set di test fisso. Questo ridurrà l'incentivo per gli sviluppatori di creare strumenti per un set di test che non cambia. Sebbene siano delle ragioni condivisibili, la Test Suite Juliet rimane uno strumento molto valido per esplorare nuovi orizzonti nel campo della ricerca sulla sintesi di esempi vulnerabili e in generale nel campo della cybersecurity.

Continuando il processo di data exploration possiamo osservare che le diverse versioni del dataset Juliet contengono CWE differenti ed in numero differente. Nella Figura 4.3 sono riportati il numero di CWE rappresentate per ogni dataset. Juliet C/C++ è la versione coprente più CWE mentre la versione C# ovvero quella più recentemente introdotta nella famiglia Juliet è quella contenente meno CWE.

La Figura 4.4 illustra l'intersezione delle CWE trattate da ciascun dataset. Da questa Figura è possibile trarre le seguenti considerazioni:

- Il dataset Juliet C/C++ copre ben 55 CWE non coperte da nessuno degli altri due dataset. È il dataset avente il maggior numero di CWE non coperte da nessun altro dataset.
- 49 CWE sono state trattate da tutti e tre i dataset.
- I dataset Juliet C# [1] e Juliet Java sono quelli che hanno il maggior numero di CWE in comune, 34, a seguire troviamo Juliet Java e Juliet C/C++ con otto ed infine Juliet C/C++ e Juliet C# con sei.

[1] In Juliet C# CWE400 è labellata come Uncontrolled Resource Consumption, mentre in Juliet C/C++ e Java è labellata CWE400 Resource Exhaustion.

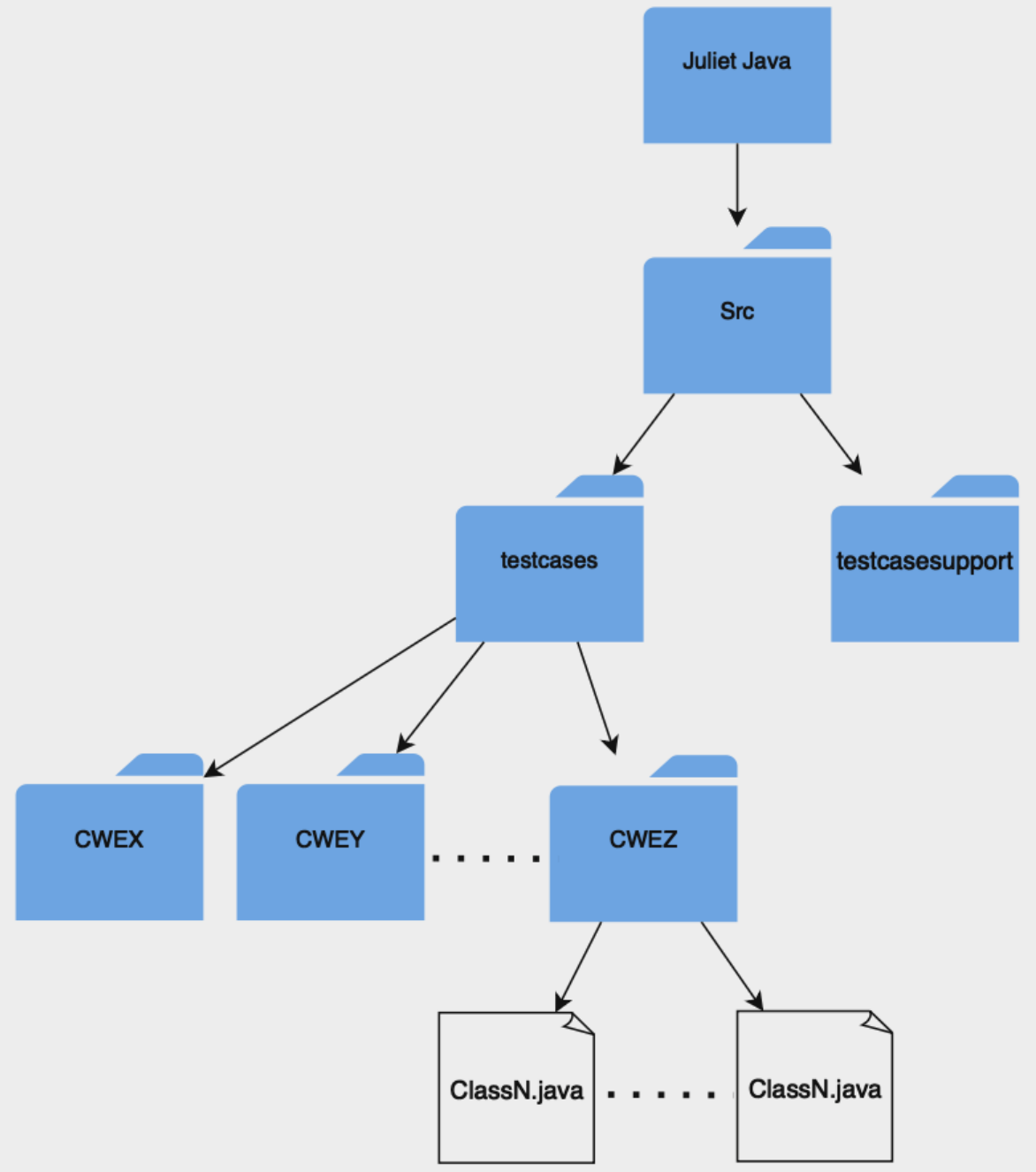

Figura 4.1: Struttura della Test Suite Juliet

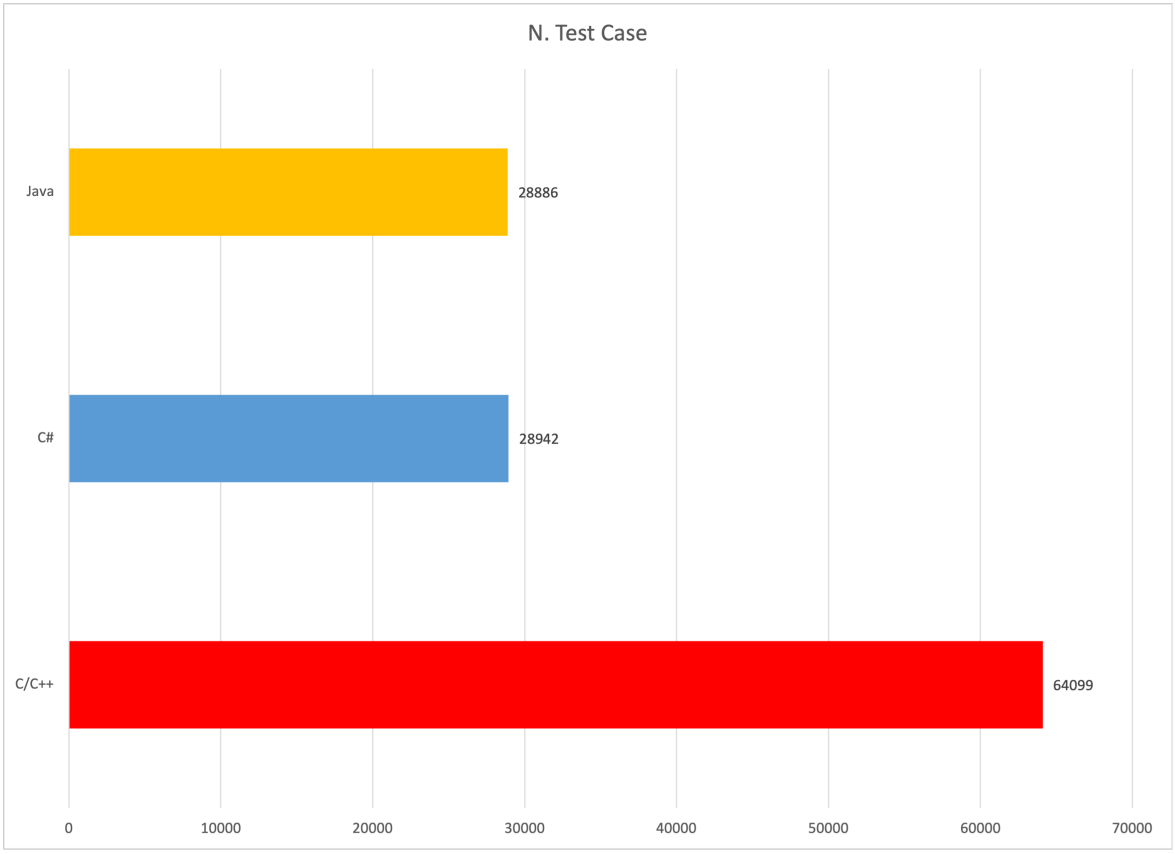

Figura 4.2: Numero di test case per Juliet Java, C# e C/C++

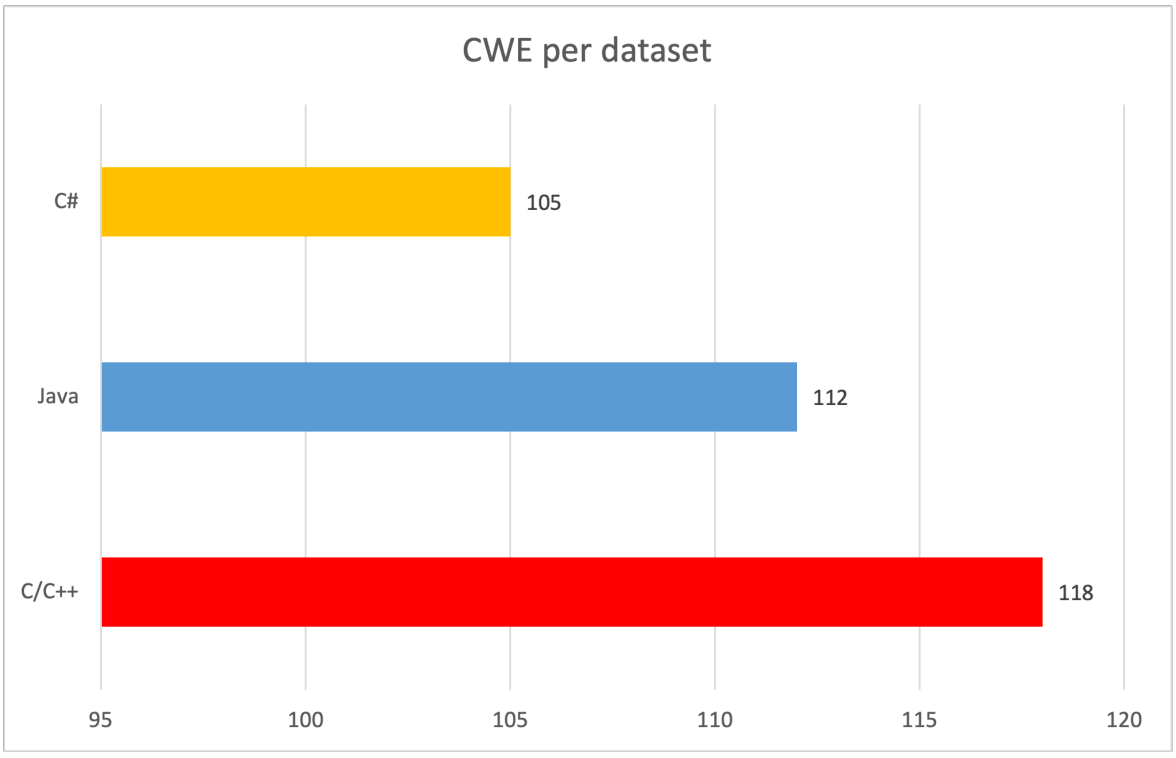

Figura 4.3: Numero di CWE coperte in Juliet C#, Java e C/C++

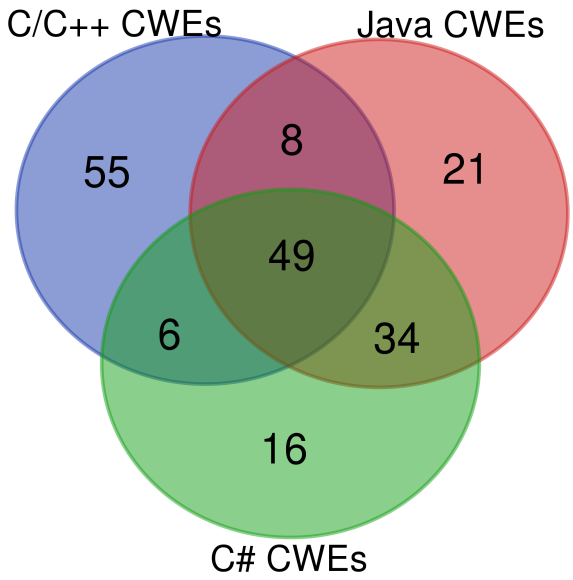

Figura 4.4: Intersezione CWE coperte dalla Test Suite Juliet

La Tabella 4.1 mostra nello specifico quali CWE sono incluse in ciascuna porzione del diagramma di Venn. Se fondessimo questi tre dataset riusciremmo a coprire 189 CWE che sono comunque poche rispetto alle 933 identificate sino ad oggi ottobre 2023. Questo fa già intuire come non è fattibile per un dataset sintetico mirare a coprire tutte le debolezze esistenti. È preferibile concentrarsi sulle CWE Top 25 [18] in quanto sono più rappresentative del trend corrente. Al momento della stesura di questa tesi è disponibile la versione della classifica aggiornata al 2022. Le Top 25 CWE sono riportate nella Tabella 4.2. Incrociando questa classifica con le CWE coperte dalle tre versioni del dataset Juliet è possibile osservare come la Test Suite ha subito fortemente il passare degli anni, in quanto non riesce più ad essere particolarmente rappresentativa della situazione corrente. Soltanto nove CWE delle 25 proposte dalla classifica sono coperte dall'unione della Test Suite Juliet e SATE IV.

Se vogliamo considerare anche i dataset forniti sotto il progetto SATE IV risalenti al 2010 possiamo osservare dalla Figura 4.5 che essi permetterebbero di coprire ulteriori 76 CWE portando la copertura totale nel caso fondessimo tutti i dataset a 265 CWE.

La Figura 4.6 invece mostra il numero di CWE in comune tra i dataset riportati e tramite l'indice di Jaccard, fornisce una metrica circa la similarità delle CWE contenute in questi dataset. Possiamo osservare come Juliet Java e Juliet C# hanno in comune molte CWE, mentre Sate IV Java e Juliet C oppure Juliet Java e SateIV C sono abbastanza disgiunti.

Verificato dunque quanti e quali CWE sono coperte da ciascun dataset, si è proceduto a esaminare quanti test case sono stati forniti per ciascuna CWE in ogni dataset. Questo permet-

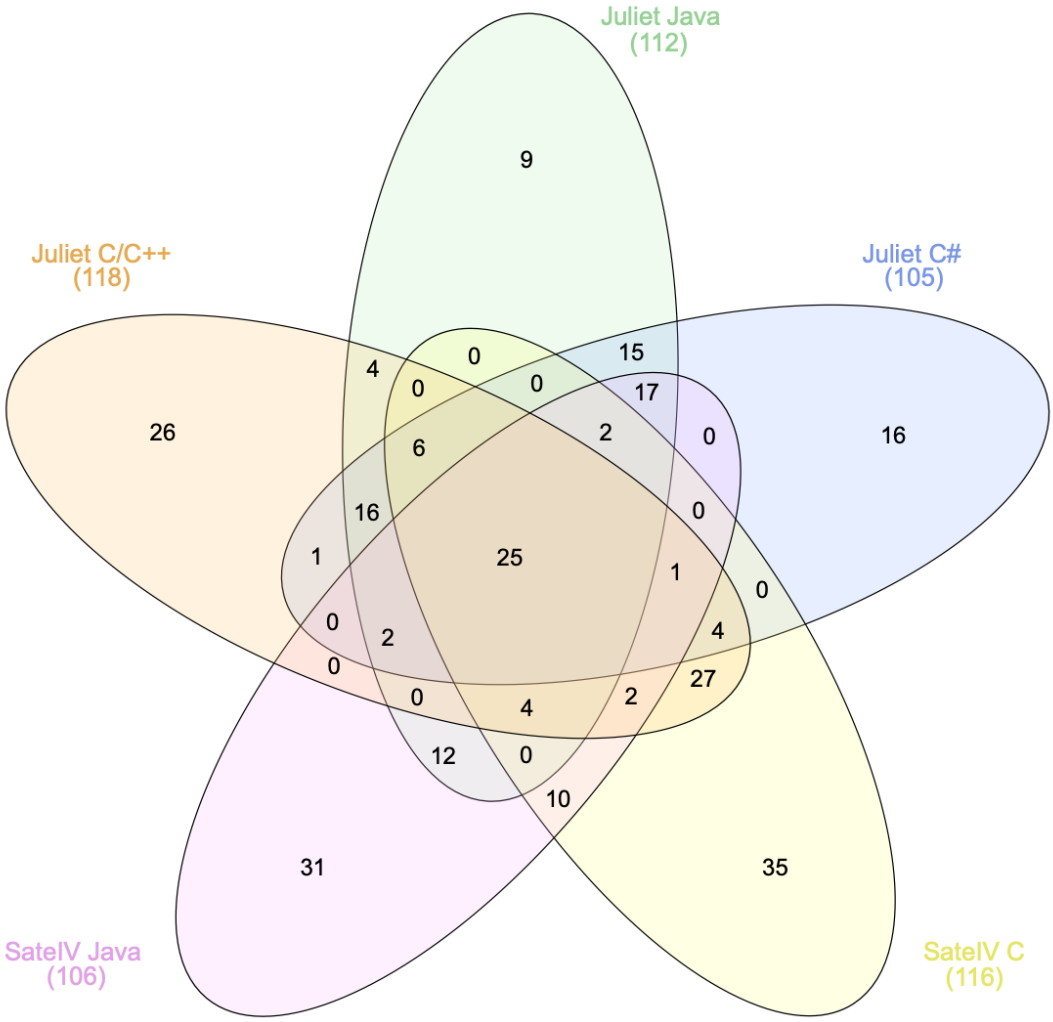

Figura 4.5: Sovrapposizione delle CWE coperte dalla Test Suite Juliet e SATE IV

Dataset	**Totale**	**CWE**
$< C\#\ ,\ C/C++\ , Java >$	49	114, 15, 190, 191, 197, 226, 23, 252, 253, 259, 319 321, 325, 327, 328, 338, 369, 36, 390, 396, 404, 459, 476, 478, 481, 482, 483, 506, 510 511, 526, 535, 546, 561, 570, 571, 605, 606, 615, 617, 667, 674, 681, 690, 775, 789, 78, 835, 90
$< C/C++\ ,\ Java >$	8	134, 256, 398, 400, 484, 500, 534, 563
$< C\#\ ,\ C/C++ >$	6	284, 397, 426, 427, 440, 675
$< C\#\ ,\ Java >$	34	113, 129, 193, 209, 248, 329, 336, 378, 379, 395, 470, 477, 486, 523, 539, 549, 566, 598, 601, 609, 613, 614, 643, 759, 760, 764, 765, 772, 80, 81, 832, 833, 83, 89
C/C++	55	121, 122, 123, 124, 126, 127, 176, 188, 194, 195, 196, 222, 223, 242, 244, 247, 272, 273, 364, 366, 367, 377, 391, 401, 415, 416, 457, 464, 467, 468, 469, 475, 479, 480, 562, 587, 588, 590, 591, 620, 665, 666, 672, 676, 680, 685, 688, 758, 761, 762, 773, 780, 785, 832, 843
Java	21	111, 315, 382, 383, 397, 491, 499, 533, 568, 572, 579, 580, 581, 582, 584, 585, 586, 597, 600, 607, 698
C#	16	117, 134, 256, 261, 313, 314, 315, 350, 366, 398, 400, 532, 563, 582, 698, 94

Tabella 4.1: CWE contenute in ogni porzione del diagramma di Venn relative alle CWE coperte dalla Test Suite Juliet

CWE ID	Nome	Juliet
787	Out-of-bounds Write	/
79	Improper Neutralization of Input During Web Page Generation ('Cross-site Scripting')	/
89	Improper Neutralization of Special Elements used in an SQL Command ('SQL Injection')	C#,Java,SATE IV Java
20	Improper Input Validation	/
125	Out-of-bounds Read	/
78	Improper Neutralization of Special Elements used in an OS Command ('OS Command Injection')	C#,C/C++,Java, SATE IV Java, SATE IV C
416	Use After Free	C/C++, SATE IV C
22	Improper Limitation of a Pathname to a Restricted Directory ('Path Traversal')	/
352	Cross-Site Request Forgery (CSRF)	SATE IV Java
434	Unrestricted Upload of File with Dangerous Type	/
476	NULL Pointer Dereference	C#,C/C++,Java
502	Deserialization of Untrusted Data	/
190	Integer Overflow or Wraparound	C#,C/C++,Java SATE IV C, SATE IV Java
287	Improper Authentication	/
798	Use of Hard-coded Credentials	C#,C/C++,Java, SATE IV C, SATE IV Java
862	Missing Authorization	/
77	Improper Neutralization of Special Elements used in a Command ('Command Injection')	/
306	Missing Authentication for Critical Function	/
119	Improper Restriction of Operations within the Bounds of a Memory Buffer	/
276	Incorrect Default Permissions	/
918	Server-Side Request Forgery (SSRF)	/
362	Concurrent Execution using Shared Resource with Improper Synchronization ('Race Condition')	/
400	Uncontrolled Resource Consumption	C#,C/C++,Java, SATE IV C, SATE IV Java
611	Improper Restriction of XML External Entity Reference	/
94	Improper Control of Generation of Code ('Code Injection')	C#

Tabella 4.2: Copertura delle Top 25 CWE da parte della Test Suite Juliet e SATE IV C/Java

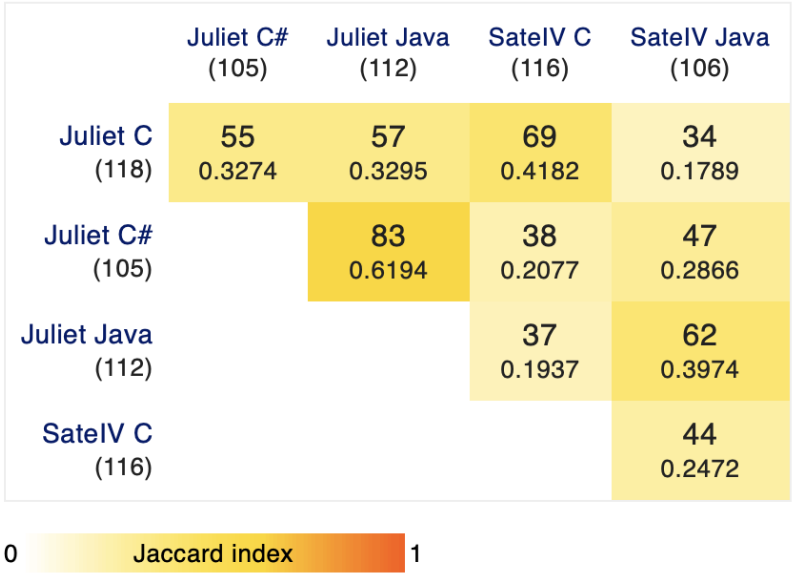

Figura 4.6: Similarità CWE trattate nella Test Suite Juliet e SATE IV

te di ottenere un'idea del numero potenziale di campioni disponibili per l'addestramento di un classificatore di vulnerabilità. Dall'analisi effettuata, è emerso una non equa distribuzione dei campioni, tra le CWE disponibili.

- Per il dataset Juliet C++ le CWE più popolate (le prime 3) risultano essere CWE122 Heap Based Buffer Overflow, CWE78 OS Command Injection e CWE121 Stack Based Buffer Overflow.
- Per il dataset Juliet C# le CWE più popolate risultano essere CWE197 Numeric Truncation Error, CWE190 Integer Overflow e CWE191 Integer Underflow.
- Per il dataset Juliet Java le CWE più popolate risultano essere CWE190 Integer Overflow, CWE191 Integer Underflow e CWE129 Improper Validation of Array Index.
- Per il dataset SateIV Java le CWE più popolate risultano essere CWE89 SQL Injection, CCWE113 HTTP Response Splitting e CWE190 Integer Overflow.
- Per il dataset SateIV C le CWE più popolate risultano essere CWE78 Command Injection, CWE122 Heap Based Buffer Overflow e CWE121 Stack Based Buffer Overflow.

I probemi principali individuati esplorando lo stato dell'arte sono i seguenti:

Semplicità: I campioni presenti all'interno dei dataset sintetici sono creati a partire da pattern predefiniti e per questo motivo mancano della naturalezza, complessità e diversità che è presente in software prelavato da contesti reali.

Accuratezza: L'accuratezza delle etichette dipende dal metodo di etichettatura utilizzato. In generale, gli strumenti di etichettatura automatici sono più soggetti ad errori di tipo falso positivo o falso negativo rispetto all'etichettatura manuale. Inoltre, la precisione dell'etichettatura manuale dipende dalle competenze dell'annotatore.

Skewness: I dataset sono fortemente sbilanciati, favorendo esageratamente determinate classi di vulnerabilità a discapito di altre. Un dataset sbilanciato subisce negativamente il cambiamento di popolarità di vulnerabilità che classifiche come OWASP Top Ten ci hanno abituato ad osservare. Ciò che un tool automatico per la detection di vulnerabilità aveva perfettamente imparato a riconoscere per la presenza di molti campioni in una specifica vulnerabilità potrebbe diventare obsoleto di conseguenza.

Duplicati: I dataset sintetici inoltre potrebbero contenere molti sorgenti duplicati o quasi duplicati, ovvero sorgenti molto simili se non per differenze banali. La presenza di sorgenti di questo tipo infatti pò comportare il rischio di data leak il cui fenomeno può gonfiare sproporzionatamente le performance del modello. Nello studio condotto da Russel et al [12] è stato osservato che il dataset Juliet C/C++ prima di essere soggetto a pre-processing conteneva 121,353 funzioni, e dopo aver effettuato il processo di rimozione dei duplicati, il numero delle funzioni è sceso a 11,896.

Numero dei campioni: I dataset di vulnerabilità non contengono molti campioni. Anche se ce ne fossero molti, è davvero poco probabile che riescano a coprire tutte le classi di vulnerabilità esistenti nel mondo reale.

Linguaggio di programmazione: È stato osservato nello studio condotto da Y. Lin et al [13] che la maggior parte di dataset sintetici in circolazione contiene codice C/C++, e solo al secondo e terzo posto troviamo dataset contenente codice Java e PHP. Inoltre, anche se i dataset contenenti codice C/C++ sono quelli più popolati, a malapena riescono a coprire le cinque vulnerabilità più diffuse. Questo limita molto la possibilità di progettare ed implementare un modello per la detection di vulnerabilità multilinguaggio.

Granularità: Un dataset di alta qualità dovrebbe contenere diversi livelli di granularità, come file, classi, funzioni, etc. Dato che il codice inerente ad una vulnerabilità rappresenta una piccolissima parte di un progetto, imparare troppo codice non-vulnerabile per un modello non lo rende particolarmente capace di individuare vulnerabilità in programmi reali. La granularità a livello di funzione risolve efficacemente questo problema, infatti la maggior parte dei modelli utilizza le funzioni come unità base del modello. Tuttavia, la granularità a livello di funzione non è ancora sufficiente. In questo modo si riesce soltanto ad indicare se una funzione è vulnerabile oppure no, ma non riusciamo a risalire alla radice del problema e

nemmeno a cogliere le relazioni che possono sussistere tra le funzioni. La soluzione sarebbe quella di considerare come livello di granularità i singoli statements vulnerabili, non soltanto le funzioni vulnerabili.
Informazioni superflue: la presenza di informazioni non rilevanti come file di header e commenti, mitiga la possibilità per un modello di intelligenza artificiale di imparare bene quali sono le caratteristiche di una vulnerabilità.
Copertura CWE: Se c'è una cosa che accomuna tutti i dataset esplorati è la mancanza di copertura totale sulle CWE. È facile però, allo stesso tempo, intuire come una richiesta di questo genere non sia fattibile. Coprire però il maggior numero di CWE che hanno dato prova di essere sfruttate estensivamente, permetterebbe ad un tool automatico di poter essere al passo con i tempi e di poter rilevare e distinguere un numero più alto di CWE. Questo si traduce in una azione più specifica sul problema da risolvere.

Negli studi di Y. Lin et al [13] e R. Jain et al [14] così come in altri lavori analizzati i quali condividono a grandi linee gli stessi principi, sono state proposte alcune possibili soluzioni:

- *Combinare datasets*: Combinare diversi dataset per creare un dataset più grande. Questo può aiutare ad aumentare la diversità del dataset e a ridurre il bias.
- *Rimuovere il rumore*: Rimuovere il rumore dal dataset, come funzioni vuote o con singole chiamate ad altre funzioni.
- *Bilanciamento del dataset*: Bilanciare il dataset assicurandosi che il numero di esempi vulnerabili e non vulnerabili sia a grandi linee uguale. Questo aiuta a prevenire un ipotetico modello dall'avere un bias verso una sola classe.
- *Feature engineering*: Estrarre le feature dal codice che sono rilevanti per la vulnerability detection. Per esempio analizzare la dimensione del codice, la presenza di strutture dati sensibili alla sicurezza e istruzioni condizionali.
- *Granularità*: Creare un dataset con diversi livelli di granularità
- *Multi-sorgente*: Creare un dataset che contenga sia un piccolo numero di dati sintetici che un grande numero di campioni estratti da contesti reali.
- *Updated*: Mantenere il dataset sempre aggiornato in modo che le performance dei modelli addestrati su di esso siano mantenuti aggiornati di conseguenza.

- *Distribuzione vulnerabilità*: Creare un dataset che abbia la distribuzione delle vulnerabilità che sia la più uniforme possibile in modo da non creare bias.

		Precision	Recall	F1-Score	Support
Esperimento 1	micro avg	0.7932	0.7932	0.7932	13719
	macro avg	0.7303	0.6437	0.6708	13719
	weighted avg	0.8149	0.7932	0.7936	13719
	samples avg	0.7932	0.7932	0.7932	13719
Esperimento 2	micro avg	0.7729	0.7729	0.7729	13719
	macro avg	0.7351	0.6676	0.6834	13719
	weighted avg	0.8069	0.7729	0.7769	13719
	samples avg	0.7729	0.7729	0.7729	13719
Esperimento 3	micro avg	0.9761	0.9761	0.9761	13719
	macro avg	0.9871	0.9860	0.9863	13719
	weighted avg	0.9771	0.9761	0.9761	13719
	samples avg	0.9761	0.9761	0.9761	13719
Esperimento 4	micro avg	0.7063	0.7063	0.7063	9260
	macro avg	0.8779	0.8111	0.8268	9260
	weighted avg	0.7892	0.7063	0.7063	9260
	samples avg	0.7063	0.7063	0.7063	9260
Esperimento 5	micro avg	0.6975	0.6975	0.6975	9260
	macro avg	0.8915	0.8123	0.8309	9260
	weighted avg	0.7728	0.6975	0.6988	9260
	samples avg	0.6975	0.6975	0.6975	9260
Esperimento 6 - CNN	micro avg	0.77	0.77	0.77	11200
	macro avg	0.87	0.77	0.79	11200
	weighted avg	0.87	0.77	0.79	11200
	samples avg	0.77	0.77	0.77	11200
Esperimento 6 - NN	micro avg	0.74	0.74	0.74	11200
	macro avg	0.85	0.74	0.77	11200
	weighted avg	0.85	0.74	0.77	11200
	samples avg	0.74	0.74	0.74	11200

Tabella 4.3: Risultati degli esperimenti

Comparando i risultati degli esperimenti N.1 e N.2 con quelli dell'esperimento N.3 vediamo dalla Tabella 4.3 come siano peggiori. Questo potrebbe essere imputabile a diversi fattori, quali il numero di classi sul quale sto classificando, che potrebbe essere alto ed alla dimensione dei vettori anch'essa notevole. Tale differenza di prestazioni potrebbe avvalorare l'ipotesi di partenza per la quale un accorpamento di sorgenti vulnerabili tramite algoritmi di clustering è efficace come strumento di classificazione, o per lo meno, ha indicato una direzione interessante da percorrere. È interessante notare come nell'esperimento N.1 dove non è stato fatto l'accorpamento delle label, i risultati sembrano essere leggermente migliori.

Tramite questi esperimenti è stato possibile individuare quali sono i problemi principali che riguardano i dataset sintetici di vulnerabilità ed è stato misurato l'impatto che alcune delle soluzioni proposte hanno avuto sulla Test Suite Juliet. Per le ipotesi avanzate all'inizio si è potuto trarre le seguenti considerazioni:

- Ipotesi 1: Una granularità più fine non sembra aiutare o fare molta differenza in termini di classificazione, com è possibile osservare anche dall'esperimento N.4 e N.5. La causa di ciò potrebbe essere imputabile ad un numero significativamente basso di LOC (Lines of Code) utilizzabili. Di conseguenza il modello potrebbe non disporre di materiale a sufficienza per apprendere le caratteristiche della vulnerabilità.

- Ipotesi 2: È stato osservato un incremento significativo nel complesso delle prestazioni della rete neurale addestrata sui sorgenti vulnerabili nel caso in cui si adoperi un nuovo set di label ricavato tramite l'utilizzo dell'algoritmo di clustering K-Means. Questo ha avvalorato l'ipotesi che seguire un approccio di classificazione più oggettivo come la rappresentazione vettoriale di un sorgente in uno spazio N-dimensionale, potrebbe essere una soluzione interessante e che merita ulteriori indagini.

Per la *RQ.1: "Quali sono le best practice per migliorare un dataset sintetico di vulnerabilità, e nello specifico la Test Suite Juliet?"* è possibile concludere che tra le best practice adottabili nella creazione di un nuovo dataset di vulnerabilità, utilizzare una granularità a livello di classe rende il compito della classificazione di sorgenti vulnerabili più semplice. Inoltre la definizione di nuove label o la definizione di nuove feature tramite un processo di feature engineering basato su k-means permette di ottenere un incremento generale delle perfomance fino al 18% nel caso di studio in esame.

4.2 Risultati per Domanda di Ricerca RQ. 2

È possibile osservare che impiegando un approccio di data augmentation basato su immagini e successivamente fare uso di un modello di NN convoluzionale su un dataset bilanciato, ha portato diversi benefici rispetto l'impiego di una NN tradizionale sul dataset originale e sul dataset aumentato. Nel caso della CNN, è possibile osservare un incremento del 14% nella macro avg rispetto l'esperimento 1 e un incremento del 2% rispetto l'esperimento 6 con le NN. Questo ci informa che la CNN ha dimostrato delle prestazioni leggermente migliori rispetto la NN, senza considerare le dimensioni di ciascuna classe. Questo rappresenta il terzo risultato migliore dopo l'approccio facente uso di K-Means e della feature aggiuntiva *cluster di appartenenza*. Un trend simile è osservabile per la metrica micro avg. Le CNN hanno apportato un leggero incremento del 2% rispetto l'esperimento 1 e del 3% rispetto l'esperimento 6 con le NN, tuttavia c'è stato un calo del 10% rispetto la metrica macro avg. Questo può essere dovuto a diversi fattori, tra cui performance non uniformi. Anche se il dataset su cui è stato effettuato il test è stato bilanciato, il modello è stato in grado di classificare più efficacemente alcune classi rispetto ad altre. Questo è riconducibile alla struttura stessa dei casi di test, i quali, nella Test Suite Juliet Java possono essere più o meno complessi, facendo riferimento a metriche come AvgCyclomatic, etc. Per la metrica weighted avg invece il risultato, e dunque il vantaggio, si arresta ai valori riscontrati per macro avg.
Per la *RQ.2: "Qual è l'impatto della data augmentation sulle prestazione di un modello di detection di vulnerabilità?"* è possibile concludere che impiegare una CNN su un dataset pre-bilanciato apporta notevoli benefici. Tramite la conversione dei vettori in matrici in seguito trattate come immagini, e l'applicazione sel solo filtro luminosità in maniera casuale a queste immagini, è stato possibile generare un incremento delle prestazioni di circa il 13% rispetto l'impiego di classificatori consolidati nel tempo come le reti neurali. la CNN ha dimostrato una maggiore capacità di generalizzazione e adattamento ai dati rispetto alla Rete Neurale. Questi risultati suggeriscono che l'approccio basato su CNN potrebbe essere più adatto per il compito di classificazione considerato nella tesi.

CAPITOLO 5

Threats to validity

In questo capitolo vengono affrontati i possibili bias dei risultati

5.1 Validità del costrutto

Metriche utilizzate: una possibile minaccia alla validità del costrutto individuata, potrebbe sorgere dalla definizione e dall'uso delle metriche di valutazione delle prestazioni del modello. È importante assicurarsi che le metriche impiegate siano adeguate per valutare l'efficacia dei classificatori di vulnerabilità presentati in questa tesi. Per mitigare questa minaccia, sono state selezionae metriche ampiamente accettate nel capo dell'intelligenza artificiale, come l'F1-Score, precision, recall e support.

Quantità di campioni: inoltre, un'altra possibile minaccia alla validità del costrutto potrebbe derivare dalla definizione stessa delle vulnerabilità nel dataset. È possibile che alcune vulnerabilità non siano ben rappresentate nei dati, o che vi siano ambiguità nella classificazione delle stesse. Per affrontare questa minaccia, è stata fatta una analisi accurata del dataset, cercando di ridurre al minimo le ambiguità nella classificazione. Nello specifico, in alcuni degli esperimenti condotti è stato eseguito un processo di re-labelling dei campioni utilizzando l'algoritmo k-means, ed un accorpamento delle CWE più piccole nelle corrispettive macro-categorie.

5.2 Validità interna

Data leak: in letteratura è stato più volte affermato come nei dataset sintetici di codice sorgente, rispetto a dataset creati a partire da contesti reali, è spesso osservabile una struttura più semplice del codice. Una vulnerabilità nel codice sorgente inoltre, può interessare poche righe di codice, anche una soltanto, rendendo il compito della classificazione di vulnerabilità ancora più difficile. La sempliità dei casi di test e la dimensione in termini di Line Of Code (LOC) di una vulnerabilità costituisce una minaccia alla validità interna. Una semplicità eccessiva delle classi vulnerabili rende il dataset poco diversificato. In casi estremi, la similitudine eccessiva di classi vulnerabili, se appartenenti rispettivamente al training set ed al test set, può generare problematiche di data leak. Tale problema viene favorito dalla similitudine eccessiva tra classi, le quali vengono percepite come *cloni*. Una possibile soluzione potrebbe essere quella di calcolare la similarità tra tutti i campioni presenti nel dataset e stabilire una soglia di similitudine sopra la quale scartare determinate classi Java. Un'altra minaccia alla validità interna potrebbe provenire dalla tecniche di augmentation utilizzata per generare nuovi campioni al fine da bilanciare il dataset. Non è chiaro infatti attribuire quanto dell'incremento nelle prestazioni ottenuto dalla CNN sia effettivamente dovuto alla creazione del nuovo dato e quanto invece sia dovuto al rumore introdotto dai nuovi campioni aggiunti.

5.3 Validità esterna

Linguaggio di programmazione: in letteratura i dataset di vulnerabilità su cui sono stati condotti gli studi erano principalmente scritti in C. per questa ragione, in questa tesi, si è voluto fare utilzzo di un dataset scritto in un altro linguaggio noto, Java. Questa scelta allo stesso tempo costituisce una minaccia alla validità esterna, in quanto non sono stati esplorate le prestazioni dei classificatori di vulnerabilità allenati durante gli esperimenti condotti, anche su dataset scritti in linguaggi differenti.

Tecnica di augmentation: altre due minacce alla validità sono riconducibili alla tecnica di augmentation utilizzata ed alla rapprestatività del dataset di partenza. L'utilizzo di un filtro luminosità, potrebbe non riprodurre delle variazioni reali nel codice sorgente. Questo potrebbe limitare la capacità dei classificatori di generalizzare su campioni di codice sorgente più complessi. Infine, la distribuzione delle debolezze presenti nel dataset potrebbe non rappresentare le debolezze presenti nei programmi, e ciò potrebbe limitare la capacità di un

classificatore di vulnerabilità di individuare debolezze anche in altri dati, provenienti da fonti sintetiche e non, che non siano la Test Suite Juliet Java.

5.4 Validità delle conclusioni

Disponibilità dei dati: nel contesto delle conclusioni di questa tesi, emergono diverse sfide che richiedono un'attenta considerazione in quanto possono influire sulla robustezza e sull'applicabilità dei risultati ottenuti. Una delle sfide principali riguarda la disponibilità e la rappresentatività dei dataset contenenti campioni vulnerabili. Questi dataset sono spesso legati a progetti specifici o reperibili solo in contesti privati, il che solleva interrogativi sulla loro validità e generalizzabilità. La diversità tra i dataset disponibili può rallentare notevolmente il lavoro e limitare l'opportunità di esplorare approcci basati su un dataset multi-sorgente o di creare un dataset più ampio e diversificato.

Tecniche di conversione dei dati: un'altra sfida affrontata riguarda la selezione delle tecniche di conversione e data augmentation nei metodi di addestramento dei modelli. La scelta specifica di queste tecniche e delle relative configurazioni è stata un aspetto critico degli esperimenti, e potrebbe avere un impatto significativo sui risultati. La variabilità nelle configurazioni delle tecniche di conversione e augmentation solleva dubbi sulla generalizzabilità delle conclusioni a diverse situazioni e contesti, richiedendo un'attenta valutazione della validità delle tecniche utilizzate.

Metriche di valutazione dei modelli: un terzo aspetto critico riguarda le metriche di valutazione dei modelli. Sebbene siano state selezionate metriche ampiamente accettate, come Precision, Recall, F1-Score e Support, è importante riconoscere che potrebbero non catturare completamente l'efficacia dei modelli o escludere altre metriche rilevanti. Questo solleva domande sulla validità delle conclusioni in quanto le metriche utilizzate potrebbero non essere esaustive e potrebbero non riflettere pienamente le prestazioni dei modelli.

Applicabilità dei modelli: l'applicabilità dei modelli e delle tecniche proposte a diversi linguaggi di programmazione e contesti è stata una preoccupazione chiave. Poiché la tesi si è concentrata su dataset di vulnerabilità in linguaggio Java, la validità delle conclusioni potrebbe essere limitata in contesti diversi. Questo suggerisce la necessità di ulteriori ricerche per valutare e adattare le conclusioni a situazioni più ampie.

CAPITOLO 6

Conclusioni

In questa tesi, ci si è concentrati su due obiettivi principali: prima, l'analisi dei dataset sintetici di vulnerabilità attraverso una selezione di metriche rilevanti. Secondo, l'indagine sulle tecniche di data augmentation per il codice sorgente vulnerabile e la valutazione dell'effetto su modelli di detection di vulnerabilità.

Prima degli esperimenti, sono state formulate diverse ipotesi: (1) L'uso di campioni a grana fine per addestrare un classificatore di vulnerabilità potrebbe migliorarne le prestazioni. (2) La classificazione delle vulnerabilità in base a CWE è una sfida dovuta a falsi positivi e possibili errori umani nelle etichette. (3) L'applicazione di tecniche di data augmentation migliorerà le metriche di Precision, Recall, F1-Score e Support nei modelli di detection di vulnerabilità.

I risultati sperimentali hanno mostrato che l'uso di campioni a livello di funzione non ha portato a un miglioramento significativo delle prestazioni dei modelli di classificazione di vulnerabilità. Questo potrebbe essere dovuto alla limitata quantità di codice disponibile per l'apprendimento. D'altra parte, l'uso di clustering tramite K-Means ha migliorato notevolmente le prestazioni del modello. Questo suggerisce che un approccio di classificazione più oggettivo basato su rappresentazioni vettoriali in uno spazio N-dimensionale potrebbe essere una strada promettente da esplorare. Infine, nell'esperimento di data augmentation, i modelli di Convolutional Neural Network (CNN) hanno dimostrato prestazioni promettenti, con Precision più elevata, nonché valori superiori di F1-Score e Recall rispetto ai modelli di Neural Network (NN). Ciò indica una migliore capacità di generalizzazione e adattamento ai

dati da parte delle CNN.

6.1 Limitazioni e contributi

Durante la stesura di questa tesi sono state riscontrate diverse limitazioni. È da menzionare la difficoltà incontrata nel reperire dei dataset contenenti campioni vulnerabili. Spesso questi dataset sono stati rilasciati sotto dei progetti specifici (come lo è anche la Test Suite Juliet) oppure sono stati costruiti a partire da progetti privati, e pubblicati su piattaforme di Versioning come GitHub. L'eterogeneità, inoltre, dei dataset a disposizione, ha rallentato molto il lavoro e la possibilità di una esplorazione più in profondità circa l'unione di diversi dataset in un dataset più grande o la realizzazione di un dataset multi-sorgente. La causa alla base di queste difficoltà può essere imputabile ad uno zelo particolare da parte dei creatori di tali dataset nel cercare di limitare gli accessi a tali dataset a malintenzionati. In questo modo si limiterebbe la possibilità che codice vulnerabile circoli liberamente in rete e possa essere utilizzato come esempio per condurre attacchi verso terzi. Un'altra limitazione riscontrata è la scarsità di progetti Open Source ben documentati in grado di applicare trasformazioni a codice sorgente. Spesso questi tool sono risultati limitati a casi d'uso specifici o utilizzabili solo per un numero molto ristretto di linguaggi di programmazione. Le configurazioni di questi tool, inoltre, hanno apportato spesso problematiche, le quali hanno compromesso l'esecuzione stessa del tool.

I contributi originali apportati da questo lavoro di tesi sono molteplici. Si è delineato in maniera organica i problemi che affliggono i dataset di vulnerabilità e nello specifico la Test Suite Juliet Java, così come un sottoinsieme di metriche per definire la qualità dei dataset di vulnerabilità. In questo modo è stato possibile definire una base da cui partire per la definizione di nuovi dataset o per il miglioramento di quelli esistenti. Inoltre, dalle attività condotte nella fase sperimentale sono stati prodotti nuovi dati che arricchiscono la Test Suite Juliet Java, nella forma di:

- Vettori rappresentanti le classi Java contenenti vulnerabilità
- Vettori rappresentanti le funzioni Java contenenti vulnerabilità
- Vettori rappresentanti le funzioni Java non contenenti vulnerabilità

Tali dati contribuiscono a creare una versione "Aumentata" di Juliet Java che in questa tesi viene presentata come BetterJuliet.

Tali dati possono essere utilizzati anche da lavori futuri i quali possono fare uso di queste nuove rappresentazioni dei codici sorgenti per esplorare nuove strade e nuove idee.

6.2 Sviluppi Futuri

Prima di condurre gli esperimenti proposti in precedenza, spesso ci si è imbattuti nel prendere delle decisioni circa per esempio quale metodo adottare per la conversione di un sorgente in una rappresentazione alternativa, oppure quale dataset di vulnerabilità tra quelli disponibili prendere in considerazione per le analisi. Queste decisioni sono state necessarie per porre dei vincoli e stabilire degli obiettivi mirati, ma hanno dato la possibilità, anche alla luce dei risultati ottenuti, a molteplici sviluppi futuri. Un lavoro che potrebbe essere interessante condurre è quelli di esplorare le similitudini che sussistono tra una stessa vulnerabilità ma presenta in due sorgenti semanticamente equivalenti ma appartenenti a linguaggi di programmazione diversi. I risultati di tale analisi potrebbero essere utilizzati per capire se è possibile accorpare dataset di vulnerabilità scritti in linguaggi diversi, al fine di creare un dataset multisorgente. Sarebbe interessante comprendere anche, nel caso i risultati mostrassero che non ci sono somiglianze, che impatto apporterebbe sulle performance di un classificatore di vulnerabilità un dataset così costruito.

In questa tesi si è voluto inoltre esplorare una tecnica innovativa di data augmentation convertendo dapprima un sorgente in una sua rappresentazione vettoriale e poi in immagine. Su tale immagine poi sono state condotte delle trasformazioni, sottoforma di filtri luminosità. Non sarebbe però meno interessate esplorare la combinazione di tutte le principali tecniche di augmentation presenti correntemente nello stato dell'arte. Quale combinazione di tecniche permette di generare un dataset che si avvicini di più ad un dataset creato a partire da sorgenti reali e non sintetici?

Come è stato anticipato all'interno del lavoro di tesi, il fenomeno del data leak è un problema serio. Per evitare near duplicates sarebbe interessante utilizzare una metrica come il coseno di similarità tramite cui misurare quanto il campione generato è simile ai campioni disponibili per una data debolezza nel dataset. Impostando una soglia di similitudine è possibile definire un criterio di inclusione esclusione, tramite cui se il campione risulta troppo simile viene scartato, altrimenti viene aggiunto. In questa maniera si potrebbe limitare la fuoriuscita di campioni troppo simili dal test e validation set, nel training set e dunque ridurre l'overfitting.

CAPITOLO 7

Ringraziamenti

Faccio fatica a realizzare che cinque anni sono passati così in fretta. Sono cresciuto senza rendermene conto. Non sono più la persona che ero cinque anni fa. Sembra ormai lontanissimo il tempo in cui passeggiavo con mia mamma intorno al parco del mercatello pensando a cosa ne sarebbe stato del mio futuro. Essere adesso "nel futuro" mi ha fatto capire che non abbiamo la possibilità di prevederlo, e non per un limite di preveggenza, ma perché non sappiamo come cambieremo noi stessi. L'università, questo cammino durato 5 anni l'ho amato per davvero, e non me lo sarei mai aspettato. Adesso ringrazierò un po' di persone e non in ordine di importanza, perché tutte sono state elementi di un puzzle più grande che è stato il mio percorso di vita in questi cinque anni. Ringrazio la mia famiglia:mamma, Carmen e papà, sono il mio porto sicuro, la mia identità. Sono consapevole che sono le persone migliori che potrò mai conoscere. Sono grato loro per avermi dato la possibilità di esplorare me stesso, non imponendomi mai una decisione e supportandomi sempre. Spero che il mio impegno possa ripagare in parte quello che mi hanno donato. Ringrazio Alessia Parisi, la mia compagna di vita. Sei l'unica persona che ha sempre avuto la forza di entrare nelle mie complessità e mi ha aiutato a diventare mano a mano una persona più sicura di me. Sono grato di poter condividere anche quest'altro traguardo con te, e tu, insieme ai miei genitori e mia sorella, sarete gli unici sguardi che ricercherò sempre tra la gente. Ringrazio i miei amici. In primis Paolo Plomitallo. Sei stato e sei un amico fondamentale. Quando è capitato che fossi solo in UNISA, mi sono reso conto che l'università sei tu. La cosa che rendeva bello venire a lezione, prendere il caffè alle macchinette, chiacchierare di qualsiasi argomento, studiare,

sbaglaire, imparare, era ed è la tua amicizia. Ci siamo sostenuti a vicenda e siamo rimasti compatti ed uniti fino all'ultimo. Ringrazio Emanuele e Giuseppe che anche se a distanza, ho continuato a sentirli vicino, segno che quello che abbiamo vissuto alla triennale è qualcosa di autentico che spero non si spezzi mai. Voi tre siete stati particolarmente il sale di questo percorso. A voi tutti dico grazie di cuore.

Bibliografia

[1] McKinsey&Company. (2020) How covid 19 has pushed companies over the technology tipping point and transformed business forever. [Online]. Available: https://www.mckinsey.com/capabilities/strategy-and-corporate-finance/our-insights/how-covid-19-has-pushed-companies-over-the-technology-tipping-point-and-transformed-business-forever (Citato a pagina 1)

[2] Deloitte. (2020) Impact of covid-19 on cybersecurity. [Online]. Available: https://www2.deloitte.com/ch/en/pages/risk/articles/impact-covid-cybersecurity.html (Citato a pagina 1)

[3] HackerOne. (2022) 6th annual hacker-powered security report. [Online]. Available: https://www.hackerone.com/reports/6th-annual-hacker-powered-security-report (Citato alle pagine 1 e 2)

[4] OWASP Top Ten. (2021) Owasp top ten. [Online]. Available: https://owasp.org/www-project-top-ten/ (Citato a pagina 1)

[5] X. Li, L. Wang, Y. Xin, Y. Yang, and Y. Chen, "Automated vulnerability detection in source code using minimum intermediate representation learning," *Applied Sciences*, vol. 10, no. 5, p. 1692, 2020. (Citato alle pagine 2, 7 e 16)

[6] NIST. Nist. [Online]. Available: https://www.nist.gov (Citato a pagina 5)

[7] SARD. Sard. [Online]. Available: https://samate.nist.gov/SARD/ (Citato a pagina 5)

[8] SAMATE. Samate. [Online]. Available: https://www.nist.gov/itl/ssd/software-quality-group/samate (Citato a pagina 5)

[9] SATE IV. Sate. [Online]. Available: https://www.nist.gov/itl/ssd/software-quality-group/static-analysis-tool-exposition-sate-iv (Citato a pagina 5)

[10] Bugs Framework. Bugs framework. [Online]. Available: https://samate.nist.gov/BF/ (Citato a pagina 5)

[11] L. Liu, Z. Li, Y. Wen, and P. Chen, "Investigating the impact of vulnerability datasets on deep learning-based vulnerability detectors," *PeerJ Computer Science*, vol. 8, p. e975, 2022. (Citato alle pagine 5, 7, 11, 15 e 24)

[12] R. Russell, L. Kim, L. Hamilton, T. Lazovich, J. Harer, O. Ozdemir, P. Ellingwood, and M. McConley, "Automated vulnerability detection in source code using deep representation learning," in *2018 17th IEEE international conference on machine learning and applications (ICMLA)*. IEEE, 2018, pp. 757–762. (Citato alle pagine 6, 7, 11, 16 e 37)

[13] Y. Lin, Y. Li, M. Gu, H. Sun, Q. Yue, J. Hu, C. Cao, and Y. Zhang, "Vulnerability dataset construction methods applied to vulnerability detection: A survey," in *2022 52nd Annual IEEE/IFIP International Conference on Dependable Systems and Networks Workshops (DSN-W)*. IEEE, 2022, pp. 141–146. (Citato alle pagine 6, 8, 9, 37 e 38)

[14] R. Jain, N. Gervasoni, M. Ndhlovu, and S. Rawat, "A code centric evaluation of c/c++ vulnerability datasets for deep learning based vulnerability detection techniques," in *Proceedings of the 16th Innovations in Software Engineering Conference*, 2023, pp. 1–10. (Citato alle pagine 8, 11 e 38)

[15] X. Li, S. Moreschini, Z. Zhang, F. Palomba, and D. Taibi, "The anatomy of a vulnerability database: A systematic mapping study," *Journal of Systems and Software*, p. 111679, 2023. (Citato a pagina 14)

[16] S. Yu, T. Wang, and J. Wang, "Data augmentation by program transformation," *Journal of Systems and Software*, vol. 190, p. 111304, 2022. (Citato a pagina 17)

[17] P. E. Black and P. E. Black, *Juliet 1.3 test suite: Changes from 1.2*. US Department of Commerce, National Institute of Standards and Technology, 2018. (Citato a pagina 28)

[18] CWE Top 25. Cwe top 25. [Online]. Available: https://cwe.mitre.org/data/definitions/1387.html (Citato a pagina 32)