UNIVERSITÀ DEGLI STUDI DI SALERNO

Dipartimento di Informatica

Corso di Laurea Triennale in Informatica

TESI DI LAUREA

Identificazione di Bloated Dependency in Python

RELATORI

Prof. Carmine Gravino

Prof. Fabio Palomba

Dott. Emanuele Iannone

Dott. Giammaria Giordano

Università degli Studi di Salerno

CANDIDATO

Daniele Fabiano

Matricola: 0512110373

Anno Accademico 2022-2023

Questa tesi è stata realizzata nel

Job's not finished, job finished? I don't think so

Kobe Bryant

Abstract

All'interno dei progetti software sono diffusi diversi fenomeni che ne danneggiano la qualità, alcuni dei quali sono causati dalla cattiva gestione delle dipendenze e dei relativi file di configurazione. Questi fenomeni sono definiti come dependency smell. In questa tesi sarà analizzato un particolare caso di dependency smell detto bloated dependency. Le bloated dependency sono dipendenze inutilizzate che non sono necessarie alla build del progetto e causano l'aumento delle dimensioni dell'artefatto software, lasciando scoperta una maggiore superficie di attacco e introducendo ulteriori problemi relativi alla sicurezza, rendendo le dipendenze vulnerabili. Attraverso l'utilizzo combinato di vari strumenti, è stato realizzato BloatWeak, un tool capace di automatizzare l'analisi di un progetto software in Python, per restituire il listato delle bloated dependency affette da vulnerabilità note. Lo studio si concentrerà sull'analisi di progetti open-source realizzati con Python, per verificare la diffusione di questi fenomeni e fornire dei risultati utili, con l'obiettivo di cercare di colmare il gap presente in letteratura. I risultati ci mostrano che il 19% dei progetti analizzabili sono affetti dal fenomeno delle bloated dependency e il 38% dei progetti già affetti dal fenomeno delle bloated dependency, sono contemporaneamente affetti anche dal fenomeno delle dipendenze vulnerabili.

Indice

Elenco delle Figure **iii**

Elenco delle Tabelle **iv**

1 Introduzione **1**

1.1 Contesto applicativo 1

1.2 Motivazioni e obiettivi 2

1.3 Risultati ottenuti 3

1.4 Struttura della tesi 3

2 Background e stato dell'arte **5**

2.1 Background 5

2.1.1 Dead Code 5

2.1.2 Bloated Dependency 6

2.1.3 Dipendenze vulnerabili 6

2.1.4 Dipendenze in Python 7

2.1.5 PyVENV - Python Virtual ENVironment 9

2.2 Stato dell'arte 10

2.2.1 DEAP-CLEAN 10

2.2.2 PRÄZI 10

2.2.3 PyCD 11

2.2.4 Alcuni utili tool 12

3 Metodologia **15**

3.1 Un approccio base 15

3.2 BloatWeak 18

4 Validazione Manuale di BloatWeak **23**

4.1 Dataset dei 10 progetti 23

4.2 Metodo di validazione e risultati 25

5 Sperimentazione **28**

5.1 Research Question 28

5.2 Dataset 29

5.3 Procedura di Analisi 31

5.4 Risultati 34

5.4.1 RQ1: La diffusione delle bloated dependency 34

5.4.2 RQ2: La diffusione delle dipendenze vulnerabili. 35

5.4.3 RQ3: Il legame tra bloated dependency e dipendenze vulnerabili 38

6 Conclusioni **40**

6.1 Limitazioni attuali 40

6.2 Sviluppi Futuri 41

6.3 Considerazioni Finali 42

Bibliografia **43**

Elenco delle figure

2.1 Snippet dell'output che si ottiene dal comando di fawltydeps. 13
2.2 Snippet dell'output che si ottiene dal comando di safety. 14

3.1 La struttura dell'approccio base. 16
3.2 La struttura di BloatWeak. 18
3.3 L'output sulla shell di BloatWeak per il progetto mycli. 22

5.1 Il grafico a barre della diffusione delle bloated dependency. 35
5.2 Il grafico a barre della diffusione delle dipendenze vulnerabili. 37
5.3 Il grafico a barre della diffusione delle dipendenze bloated e vulnerabili. 39

Elenco delle tabelle

2.1 Uno snippet dell'output di GetDep_ast.py per il progetto voltron. . . 11
2.2 Uno snippet dell'output di Inference.py per il progetto voltron. . . . 12

3.1 Uno snippet della tabella delle dipendenze per il progetto mycli. . . 20
3.2 Uno snippet della tabella del file safteyDB_out.csv. 21

4.1 Il contenuto del file validation_report.csv. 25
4.2 Tabella di errata classificazione per la validazione di BloatWeak. . . . 26

5.1 Uno snippet di report.csv. 33

CAPITOLO 1

Introduzione

1.1 Contesto applicativo

L'Intelligenza Artificiale è sempre stata una disciplina a cavallo tra la scienza e la filosofia, considerata per molto tempo come qualcosa di pioneristico e inarrivabile per la commercializzazione nel mercato della tecnologia. Nel corso degli ultimi anni ha invece iniziato sempre di più a farsi spazio all'interno dei sistemi informativi, a partire da un semplice modulo applicativo per passare ad intere applicazioni realizzate fin dall'inizio con idee di soluzioni intelligenti. Grazie all'evoluzione della tecnologia, l'Intelligenza Artificiale è arrivata ad essere a portata di mano di qualunque utente che ne possa far uso sia come semplice utilizzatore che come sviluppatore all'interno di infrastrutture software ben più articolate. Ad oggi, è possibile automatizzare e velocizzare task sempre più complessi sfruttando le informazioni già esistenti come un fattore importante per raggiungere in maniera più rapida degli utili risultati. Conversational Agent, Image Processing, Sound Processing e molti altri, sono i casi d'uso in cui si possono introdurre soluzioni intelligenti. Affinchè però queste strategie possano competere nel realizzare dei prodotti concreti, è necessario che entri in gioco il mondo dei Software. Attraverso il design e ingegnerizzazione di sistemi informatici, è possibile integrare l'Intelligenza Artificiale all'interno di applicativi

desktop, applicazioni web, applicazioni mobile. L'introduzione di queste soluzioni intelligenti introduce ancora maggiore complessità nel ciclo di vita e sviluppo di un software, lasciando spazio ad ulteriori difetti e alla diffusione di pericolosi fenomeni che vanno ad alterare l'affidabilità e la sicurezza di un software. Bisogna perciò essere in grado di individuare e monitorare questi fenomeni, cercando di minimizzare i danni e capire come arginarli. Sono diversi i linguaggi di programmazione che permettono la realizzazione di soluzioni intelligenti. Tra i più famosi, linguaggi come Java e C++ permettono l'utilizzo di queste soluzioni, in particolare con C++ si riescono ad ottenere anche prestazioni elevate. A volte però, è la semplicità a farla da padrone e nel corso del tempo il linguaggio Python si è imposto come lo standard nel campo dell'Intelligenza Artificiale grazie alla sua struttura snella e leggera, priva di fronzoli e che garantisce una implicita pulizia nella formattazione del codice.

1.2 Motivazioni e obiettivi

Sono sempre di più le librerie in Python che permettono di creare soluzioni intelligenti e che continuano a facilitare la vita degli sviluppatori. Se da un lato le implementazioni diventano più facili, dall'altro lato le operazioni di manutenzione diventano più difficili. La vita di un software non termina con il suo rilascio, ma sono previste ulteriori fasi come la già citata manutenzione, il rilascio di nuove release e l'esecuzione di nuovi processi di testing. Data la complessità e la fretta di lanciare sul mercato software intelligenti, in molti casi la manutenzione può diventare complicata per colpa di cattive decisioni prese durante le fasi di progettazione e design che abbassano la qualità del software. L'insieme dei fenomeni che evidenziano difetti del software e difficoltà nella manutenzione, sono denominati come Software Smell. Esistono diverse tipologie di Software Smell, in particolare con l'importazione delle librerie all'interno di un software, è necessario effettuare la dichiarazione dei vincoli di dipendenza tra una particolare versione di una libreria e il software che si sta realizzando, mediante l'utilizzo dei file di configurazione. In questa tesi sarà esplorato un particolare fenomeno di Software Smell applicato all'analisi dei file di configurazione e appartenente alla categoria dei dependency smell, detto Bloated Dependency. L'obiettivo è quello di analizzare la diffusione di questo fenomeno e

considerare le criticità a cui un software è esposto quando è affetto dal fenomeno delle Bloated Dependency. Per questo motivo, nel corso di questo studio è stato realizzato lo strumento BloatWeak che unisce tool già esistenti e combina le loro funzionalità, per fornire allo sviluppatore l'elenco delle bloated dependency presenti nel codice sorgente di un progetto Python analizzato e per evidenziarne le eventuali vulnerabilità presenti, così da portare alla luce ancor di più le problematiche di questo fenomeno con l'individuazione delle dipendenze vulnerabili.

1.3 Risultati ottenuti

Dopo aver completato la realizzazione del tool BloatWeak, sono stati effettuati i processi di validazione del tool e sperimentazione su un dataset di progetti che fanno utilizzo di soluzioni AI/ML. In particolare per la validazione, sono stati analizzati in maniera approfondita 10 progetti e sono state calcolate diverse metriche. Rispetto al campione dei 10 progetti, BloatWeak ha una Precision pari a 0.5, una Recall pari a 1 e una Accuracy pari a 0.6. Le metriche di Precision e Accuracy sono migliorabili cercando nel futuro di affrontare e superare i limiti del tool, dovuti alle difficoltà nel valore semantico rispetto all'analisi di alcune particolari dichiarazioni delle dipendenze. Inoltre l'esecuzione di BloatWeak sul dataset di progetti NICHE opportunatamente filtrato, è stato in grado di fornire dei risultati iniziali i quali mostrano che 70 progetti su 365 sono affetti dal fenomeno delle bloated dependency e 355 progetti su 365 sono affetti dal fenomeno delle dipendenze vulnerabili. I due fenomeni si combinano insieme su 38 progetti.

1.4 Struttura della tesi

La seguente tesi è strutturata in diversi capitoli:

- Capitolo 1 - Introduzione: Il capitolo iniziale dove il lettore è introdotto agli aspetti trattati in questa tesi, attraverso una breve panoramica del contesto applicativo, del lavoro realizzato e dei risultati ottenuti;

- Capitolo 2 - Background e Stato dell'Arte: Il capitolo dove sono presentati e analizzati studi già presenti in letteratura, che trattano di aspetti simili con approcci anche in altri linguaggi. Inoltre sono presentati alcuni utili strumenti per la realizzazione del tool e l'avanzamento di questo studio;

- Capitolo 3 - Metodologia: Il capitolo dove sono descritti nel dettaglio i vari approcci provati per l'avanzamento di questo studio, mostrando alcuni esempi e descrivendo come è possibile utilizzare il tool di cui viene discusso il funzionamento;

- Capitolo 4 - Validazione Manuale di BloatWeak: Il capitolo dove sono presentati e discussi i processi di validazione del tool, attraverso l'analisi dei progetti e il calcolo delle metriche;

- Capitolo 5 - Sperimentazione: Il capitolo dove sono descritti i processi della sperimentazione e raccolti i risultati, attraverso la formulazione e la risposta a diverse Research Question;

- Capitolo 6 - Conclusioni: Il capitolo finale dove si riassume il lavoro svolto, prendendo in considerazione ulteriori aspetti per sviluppi futuri e miglioramento del lavoro attuale.

CAPITOLO 2

Background e stato dell'arte

In questo capitolo sono introdotti i concetti fondamentali per il lavoro di tesi e sono analizzati studi e strumenti già presenti in letteratura. Inoltre sono presentati alcuni tool open-source, utili per l'avanzamento della ricerca e che saranno utilizzati nelle fasi successive di questo studio.

2.1 Background

2.1.1 Dead Code

Le cattive decisioni che vengono prese nelle fasi di design ed implementazione di un software, danno vita ai cosidetti fenomeni dei code smell. I code smell identificano dei difetti di design che si manifestano all'interno del codice sorgente [1]. Questi difetti vanno a danneggiare la qualità del software, dal punto di vista della comprensibilità, efficienza, sicurezza e manutenibilità. Esistono varie tipologie di code smell, uno di questi è il fenomeno del dead code. Il termine "dead" si riferisce alle porzioni di codice che sono inutilizzate e irraggiungibili, a partire da una semplice variabile, per passare ad un metodo mai richiamato e finire in alcuni casi, per avere intere classi che non vengono mai utilizzate [2]. Gli sviluppatori consapevoli di questo fenomeno

sono favorevoli alla sua mitigazione quando risulta possibile mantenere bassi i costi, in termini di tempo e di rischi.

2.1.2 Bloated Dependency

Possiamo estendere il concetto di "inutilizzato" ad ulteriori aspetti, in particolare riguardo la pratica di riuso del codice e delle librerie. Quando si realizza un software, potrebbe essere necessario l'utilizzo di una libreria di terze parti per effettuare in maniera più semplice e immediata alcune particolari operazioni, riducendo drasticamente i costi di sviluppo. L'aggiunta delle librerie di terze parti all'interno di un progetto, introduce tra essi una relazione che viene detta dipendenza. Fare eccessivo affidamento alla pratica del riuso, è la principale causa che può dar vita a fenomeni analoghi a quelli dei code smell. Quando sono introdotte le dipendenze in un progetto, è necessario dichiararle all'interno di un file di configurazione. Oltre ad effettuare manutenzione sul codice sorgente per garantirne la qualità e il corretto funzionamento, sarà necessario farlo anche relativamente ai file di configurazione che a loro volta possono essere vittime dei fenomeni di smell, in questo caso denominati come dependency smell. Un particolare smell è quello delle bloated dependency. Le bloated dependency (dipendenze inutilizzate) fanno riferimento ad alcune dipendenze, le cui liberie associate formano una parte non necessaria all'interno di un software e la loro rimozione non comporta problemi nella fase di build del progetto [3]. Ci sono innumerevoli vantaggi nel mitigare il fenomeno, uno tra questi è la riduzione delle dimensioni dell'artefatto software ottenuto. Possiamo arrivare ad ottenere un importante risparmio nello spazio di archviazione della macchina, sulla quale deve essere effettuato il deploy o l'installazione del relativo software.

2.1.3 Dipendenze vulnerabili

Quando si pratica il riuso utilizzando librerie di terze parti, lo sviluppatore si affida al lavoro effettuato da altri. Se questo può velocizzare i tempi di sviluppo, fa anche ereditare tutte le criticità presenti nelle librerie. Oltre alla questione relativa allo spazio di archiviazione, le dipendenze possono portare con sé dei problemi relativi alla sicurezza, rendendosi quindi vulnerabili a possibili attacchi di malintenzionati.

Avendo una maggiore quantità di codice sorgente, la superficie di attacco è più ampia e permette agli attaccanti di avere ulteriori vie per alterare il sistema. Queste possibili vulnerabilità non si applicano esclusivamente alle bloated dependency ma anche a quelle particolari dipendenze, nelle quali la porzione di liberia sfruttata è minima rispetto alla sua grandezza reale, lasciando altra superficie scoperta a favore degli attaccanti. Riducendo le dipendenze inutilizzate e poco utilizzate, si aggiunge robustezza al software rendendolo meno esposto agli attacchi.

2.1.4 Dipendenze in Python

Python[1] è un linguaggio di programmazione ad "alto livello" riconosciuto per la sua sintassi compatta e per gli svariati ambiti di utilizzo, sia per lo scripting e sviluppo web che in campo scientifico e machine learning. PyPI[2] è un repository di software in Python dal quale è possibile recuperare librerie di terze parti da aggiungere ai progetti, attraverso il gestore di pacchetti pip[3]. Quando bisogna dichiarare le dipendenze in un progetto software, queste sono classificate in base alla loro visibilità nei file di configurazione [4]:

- **Dipendenze dirette** (esplicite): Le dipendenze sono esplicitamente dichiarate nei rispettivi file di configurazione;
- **Dipendenze transitive** (implicite): Le dipendenze dirette presentano al loro interno ulteriori file di configurazione, attraverso il quale sono recuperate altre dipendenze; Queste dipendenze non sono esplicitate nei principali file di configurazione del progetto, ma sono necessarie per garantire il corretto funzionamento delle dipendenze dirette;
- **Dipendenze ereditate**: Nel caso di un progetto costruito su più moduli, le dipendenze del modulo principale saranno ereditate dai moduli secondari.

Per dichiare una dipendenza in Python è necessario specificarne il nome e la versione (facoltativo) in tre possibili modalità [5]:

[1] `https://www.python.org/`
[2] `https://pypi.org/`
[3] `https://pypi.org/project/pip/`

- **Pinned Dependency**: Una specifica versione è inclusa nella dichiarazione della dipendenza (e.g. `scrapy==2.9.0`);
- **Constrained Dependency**: Un insieme di versioni viene specificato nella dichiarazione della dipendenza (e.g. `scrapy<=2.9.0`);
- **Uncostrained Dependency**: La specifica della versione non è presente nella dichiarazione della dipendenza (e.g. `scrapy`).

Le dichiarazioni delle dipendenze possono essere posizionate in diverse tipologie di file di configurazione [6]:

- **setup.py**: Il principale file di configurazione per distribuire un applicativo software e permetterne la corretta installazione. All'interno di questo file possiamo dichiarare le dipendenze delle librerie in diversi modi a seconda dell'utilizzo che bisogna farne. Le dipendenze necessarie all'installazione del software sono dichiarate alla voce *install_requires*. Altre dipendenze possono essere dichiarate in ulteriori voci, ad esempio le dipendenze necessarie alla fase di testing sono dichiarate alla voce *test_requires*;
- **requirements.txt**: Un file di testo dove sono dichiarate le dipendenze e che viene usato per installare manualmente le dipendenze del progetto tramite pip. Per convenzione le dipendenze necessarie all'installazione del software sono dichiarate in un file denominato `requirements.txt`. A seconda dell'utilizzo di altre dipendenze, ulteriori file di configurazione sono denominati con suffissi. Ad esempio, le dipendenze necessarie alla fase di testing sono dichiarate in un file denominato `requirements-test.txt`;
- **pipfile**: A seguito delle proposte PEP517[4] e PEP518[5], nell'ecosistema di Python è stato introdotto un nuovo strumento detto Pipenv[6] che cerca di facilitare la gestione delle dipendenze e la distribuzione del software. Pipenv fa uso dei pipfile in formato toml per dichiarare le dipendenze. Le dipendenze necessarie all'installazione del software sono indicate alla voce *dependencies*.

[4] `https://peps.python.org/pep-0517/`
[5] `https://peps.python.org/pep-0518/`
[6] `https://pypi.org/project/pipenv/`

2.1.5 PyVENV - Python Virtual ENVironment

Quando si sviluppa un software in Python, potrebbe essere necessario l'utilizzo di più librerie di terze parti. Sulla stessa macchina si potrebbe star lavorando a diversi progetti che potrebbero richiedere la stessa libreria ma con differenti versioni, creando un conflitto su quale libreria debba rimanere installata a livello di sistema. A seguito della proposta PEP405[7], all'interno dell'ecosistema Python è stato introdotto un nuovo strumento in grado di risolvere questa problematica. Il modulo *venv*[8] introduce la possibilità di creare dei virtual environment che permettono l'utilizzo in maniera isolata di un particolare set di librerie di terze parti. Per utilizzare un virtual environment in Python, è necessario effettuarne la creazione attraverso il seguente comando: `python -m venv /path/to/new/virtual/environment`.
Una volta che il virtual environment è stato creato, bisogna attivarlo attraverso il seguente comando: `source <venv>/bin/activate`.
In questo modo sarà possibile utilizzare una sessione del terminale dove la variabile PATH utilizzata per individuare gli eseguibili presente nel sistema, punterà temporaneamente al virtual environment appena attivato. Per disattivare il virtual environment basterà eseguire il comando `deactivate`. I virtual environment sono ormai ampiamente utilizzati nello sviluppo dei progetti Python e anche i più famosi IDE come Pycharm[9], includono la creazione di un virtual environment[10] in maniera automatica al momento della creazione di un nuovo progetto[11]. Oltre al risolvere il conflitto tra versioni di liberia da installare, l'utilizzo di un virtual environment introduce ulteriori vantaggi per la facile riproducibilità dell'ambiente di esecuzione del progetto. Si evitano anche spiacevoli situazioni riguardo un eventuale aggiornamento di sistema, che possa aver compromesso delle librerie specificamente utilizzate in alcuni progetti.

[7] https://peps.python.org/pep-0405/
[8] https://docs.python.org/3/library/venv.html
[9] https://www.jetbrains.com/pycharm/
[10] https://www.jetbrains.com/help/pycharm/creating-virtual-environment.html
[11] https://www.jetbrains.com/help/pycharm/creating-and-running-your-first-python-pr
html#creating-simple-project

2.2 Stato dell'arte

2.2.1 DEAP-CLEAN

Soto-Valero et. al[4] hanno realizzato il tool *DEAP-CLEAN*[12] che analizza progetti Java[13] con Maven[14], per identificarne le bloated dependency. Attraverso questo studio hanno stabilito che la maggior parte delle bloated dependency nei progetti Java sono introdotte attraverso dipendenze transitive. Per giungere a questa conclusione hanno considerato due strutture dati denominate come alberi delle dipendenze. L'albero delle dipendenze di Maven contiene tutte le dipendenze dichiarate nel file di configurazione *pom.xml* del progetto e le relazioni che si presentano tra esse. L'albero delle dipendenze utilizzate contiene invece tutte le dipendenze realmente utilizzate nel codice sorgente e le relazioni che si presentano tra esse. Analizzando le differenze tra questi due alberi, sono state individuate le bloated dependency del relativo progetto. Tutti i processi sono stati automatizzati attraverso il sopracitato tool che è stato utilizzato anche in uno studio relativo all'identificazione delle vulnerabilità. Ponta et al.[7] hanno fatto uso di vari software di debloating per progetti Java tra cui *DEAP-CLEAN*. L'obiettivo era quello di effettuare la build di tutti i progetti analizzati e identificare eventuali vulnerabilità. Dopo aver eseguito i software di debloat, sono state effettuate le build di tutti i progetti per verificare nuovamente l'identificazione di relative vulnerabilità. Lo studio ha prodotto un caso in cui le operazioni di debloat sono riuscite a rimuovere delle vulnerabilità, rafforzando ancora di più l'esistenza di un legame tra bloated dependency e vulnerabilità del software.

2.2.2 PRÄZI

Hejderup et al.[8] hanno invece realizzato il tool *PRÄZI*[15] che lavora sui progetti in linguaggio RUST[16], effettuando un'analisi delle librerie associate alle dipendenze, ad un livello di granularità molto più fine rispetto al precedente tool. Questo tool

[12] `https://github.com/ASSERT-KTH/depclean.git`
[13] `https://www.oracle.com/it/java/`
[14] `https://maven.apache.org/`
[15] `https://github.com/praezi/rust`
[16] `https://www.rust-lang.org/`

costruisce un grafo tra le funzioni/metodi delle librerie e le funzioni/metodi che le utilizzano, in modo tale che i nodi siano rappresentati dalle funzioni/metodi della libreria e gli archi siano rappresentati come una coppia che identifica i due metodi/funzioni coinvolti nella chiamata. A questo livello di granularità risulta più semplice e immediato sia quantificare la reale percentuale di utilizzo della libreria, sia di poter constastare se si sta usando la porzione vulnerabile o meno della libreria chiamata. Potendo stabilire la quantità di libreria realmente utilizzata, è possibile ridurre ulteriormente la superficie scoperta di quanto non sia già stato fatto rimuovendo le bloated dependency.

2.2.3 PyCD

dep	version	filepath	type	condition	status
cursor	==*	/home/daniele/git/voltron/setup.py	install_requires	@(sys.platform == 'win32')@*	*
scruffington	>=0.3.6	/home/daniele/git/voltron/setup.py	install_requires	*@*	*
flask	==*	/home/daniele/git/voltron/setup.py	install_requires	*@*	*
flask_restful	==*	/home/daniele/git/voltron/setup.py	install_requires	*@*	*
blessed	==*	/home/daniele/git/voltron/setup.py	install_requires	*@*	*
pygments	==*	/home/daniele/git/voltron/setup.py	install_requires	*@*	*
requests	==*	/home/daniele/git/voltron/setup.py	install_requires	*@*	*
requests$_u$*nixsocket*	==*	/home/daniele/git/voltron/setup.py	install_requires	*@*	*
six	==*	/home/daniele/git/voltron/setup.py	install_requires	*@*	*
pysigset	==*	/home/daniele/git/voltron/setup.py	install_requires	*@*	*

Tabella 2.1: Uno snippet dell'output di GetDep_ast.py per il progetto voltron.

PyCD[17] è un tool realizzato da *Cao et al.*[6] in merito allo studio dei dependency smell. Il tool prevede l'utilizzo di due particolari script. Per mostrare gli output di esempio è stato analizzato *voltron*[18]. Il primo script denominato *GetDep_ast.py* permette di estrarre le dichiarazioni delle dipendenze presenti in tutti i file di configurazione, per ottenere un file in formato tabellare con estensione csv, il suo contenuto è mostrato nella Tabella 2.1. Lo script prevede di ricevere in input due argomenti dalla riga di comando, rispettivamente il path del progetto da analizzare e il path

[17] `https://github.com/NJUJisq/DS_Python`

[18] `https://github.com/snare/voltron`

dove salvare il csv di output. Il comando da eseguire è costruito come di seguito:
`python3 GetDep_ast.py /path/to/the/project/ /path/to/the/csv`

dep
Flask
cursor
Pygments
requests
requests_unixsocket
blessed
six
scruffy

Tabella 2.2: Uno snippet dell'output di Inference.py per il progetto voltron.

Il secondo script denominato *Inference.py* permette di estrarre le dichiarazioni delle librerie associate alle dipendenze che sono state importate all'interno del codice sorgente. L'output prodotto è mostrato nella Tabella 2.2 e i parametri da inserire sono speculari allo script precedente. Il comando da eseguire è costruito come di seguito:
`python3 Inference.py /path/to/the/project/ /path/to/the/csv`

2.2.4 Alcuni utili tool

In letteratura non sono presenti molti strumenti che lavorano sui progetti realizzati in Python, relativamente a queste tematiche. Il linguaggio diventa però sempre più utilizzato e risulta necessario fornire un nuovo approccio a queste problematiche, sfruttando come idee di base gli studi, gli strumenti e le tecniche che analizzano invece progetti realizzati in altri linguaggi di programmazione. Nei prossimi capitoli di questa tesi, si cercherà di colmare il gap che si è creato in letteratura, usando come casi di studio diversi progetti open-source realizzati in Python. In questa sezione sono invece presentati alcuni utili tool che lavorano su progetti Python. I repository di pip

presentano una moltitudine di pacchetti software dai quali è possibile recuperare utili strumenti. In seguito ad una ricerca su PyPI, sono stati individuati due strumenti open-source, ma anche di natura commerciale. Facendo uso dei vari tool, verrà realizzato un nuovo strumento in grado di automatizzare l'identificazione delle dipendenze inutilizzate e le relative vulnerabilità associate.

Fawltydeps

Fawltydeps[19] è un tool[9] che individua e fornisce il listato delle dipendenze inutilizzate e non dichiarate, fornendo eventualmente i dettagli relativi al tipo di file di configurazione in cui sono state definite le dichiarazioni. Per mostrare gli output di esempio è stato analizzato *voltron*. Per garantire il corretto funzionamento del tool, è necessario installare le dipendenze del progetto analizzato, in questo modo nel caso il nome della dipendenza e il nome della libreria associata da importare differiscano, fawltydeps sarà in grado di recuperare tutti i metadati necessari per stabilire quali sono le dipendenze inutilizzate e non dichiarate. Per utilizzare fawltydeps ed ottenere il listato delle dipendenze inutilizzate come mostrato in Figura 2.1, bisogna posizionarsi nella directory del progetto da analizzare, oppure fornire il path della directory come argomento, ed eseguire il seguente comando:

```
fawltydeps (/path/to/the/project/) --check-unused (--detailed)
```

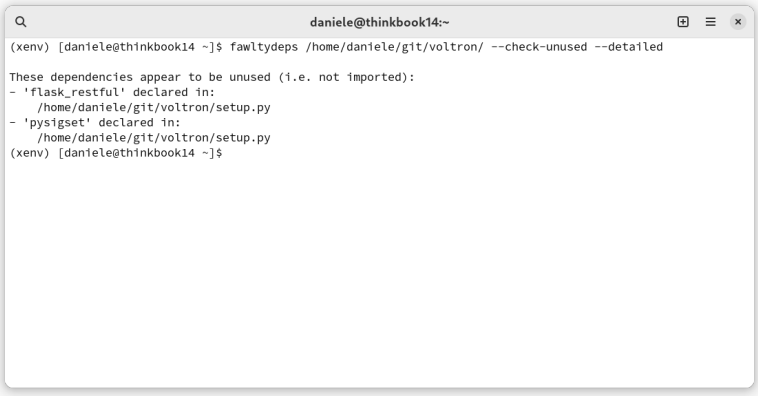

Figura 2.1: Snippet dell'output che si ottiene dal comando di fawltydeps.

[19] https://pypi.org/project/fawltydeps/

Safety

Safety[20] è un tool che analizza le dipendenze di un progetto scritto in Python e ne individua le vulnerabilità note presenti al loro interno. Per mostrare gli output di esempio è stato analizzato *mycli*[21]. Come parametro di input è necessario fornire il listato delle dipendenze formattato come un file di configurazione del tipo `requirements.txt`. Inoltre le dipendenze dichiarate devono obbligatoriamente specificare dei vincoli di versione, altrimenti saranno ignorate dal tool. Essendo un software di natura commerciale, il database[22] di vulnerabilità fornito gratuitamente è in una versione ridotta. Per utilizzare safety ed ottenere il listato delle vulnerabilità come mostrato in Figura 2.2, bisogna eseguire il seguente comando:

```
safety check -r requirements.txt
```

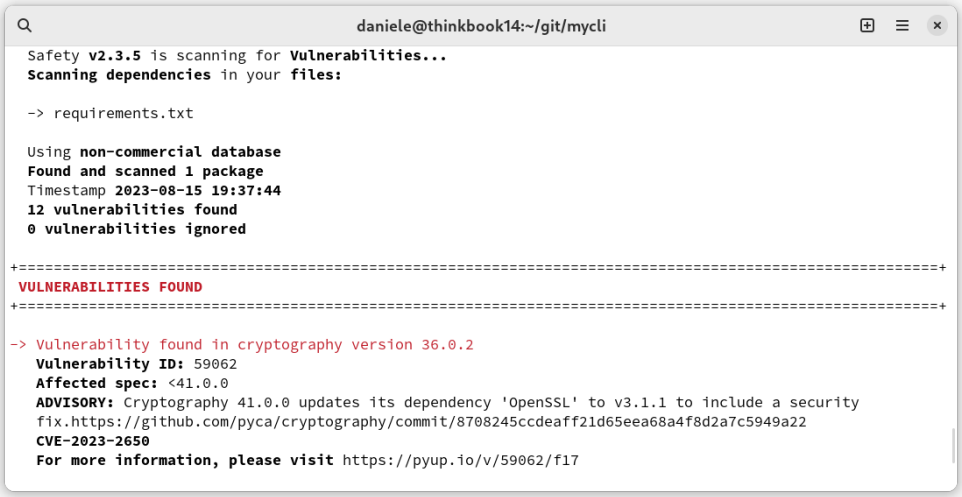

Figura 2.2: Snippet dell'output che si ottiene dal comando di safety.

[20] https://pypi.org/project/safety/
[21] https://github.com/dbcli/mycli
[22] https://github.com/pyupio/safety-db

CAPITOLO 3

Metodologia

In questo capitolo sono presentate le tecniche utilizzate per realizzare lo strumento necessario all'avanzamento del nostro studio. Gli output ottenuti dallo strumento sono mostrati attraverso l'analisi di un progetto di esempio. Inoltre sono descritti alcuni approcci iniziali che sono stati scartati data la loro inadeguatezza rispetto il problema da voler risolvere.

3.1 Un approccio base

Nel voler individuare le bloated dependency di un progetto Python, la prima fase di questo studio intende fornire un possibile approccio in grado di estrarre il listato delle dipendenze inutilizzate dai file di configurazione del progetto, utilizzando direttamente il tool PyCD. L'idea è quella di sfruttare la libreria pandas[1] per effettuare manipolazioni su entrambi i file di output ottenuti dagli script di PyCD. Il primo file ottenuto dallo script *GetDep_ast.py*, contiene le informazioni relative alle dipendenze, dichiarate nei file di configurazione del progetto che si sta analizzando. Il secondo file ottenuto dallo script *Inference.py*, contiene i nomi delle librerie associate alle dipendenze, che sono state importate nel codice sorgente del progetto che si sta

[1] `https://pandas.pydata.org/`

analizzando. Sapendo come sono strutturati i due file di output, si vuole effettuare una operazione di anti-join[2] tra la colonna del primo file, contentente gli identificativi delle dipendenze dichiarate nei file di configurazione e la colonna del secondo file, contenente gli identificativi delle librerie associate alle dipendenze che sono utilizzate nel codice. Considerando un qualunque progetto software P, l'operazione di anti-join si rifà al concetto della differenza insiemistica, formalmente:

$D = \{\ d \mid d$ è una dipendenza di P dichiarata nei file di configurazione $\}$

$I = \{\ i \mid i$ è una dipendenza di P la cui liberia associata è importata nel codice sorgente $\}$

$B = D \setminus I = \{\ b \mid b$ è una bloated dependency di P dichiarata nei file di configurazione $\}$

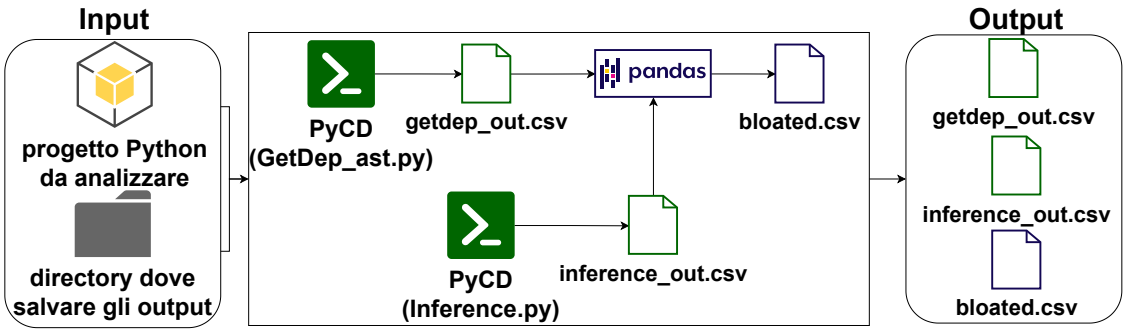

Figura 3.1: La struttura dell'approccio base.

Questo approccio è schematizzato nella Figura 3.1 ma si è dimostrato essere inadeguato per i seguenti motivi:

- **Differenze nella nomenclatura di dipendenze e librerie**: Quando nel codice si importa una libreria, il suo nome potrebbe non coincidere perfettamente con il nome della dipendenza dichiarata nel file di configurazione, alcuni caratteri potrebbero differire nell'essere in maiuscolo/minuscolo e potrebbero essere presenti delle abbreviazioni. In queste condizioni, l'operazione di anti-join non è in grado di ottenere i risultati attesi. Come ad esempio nella Tabella 2.1 e nella Tabella 2.2, i nomi delle dipendenze flusk e pygments differiscono dalla rispettiva libreria importata per il carattere iniziale maiuscolo/minuscolo (Flusk e Pygments). Invece il nome della dipendenza scruffington differisce dal nome della libreria importata (scruffy) per un'abbreviazione. Nel voler

[2] `https://www.statology.org/pandas-anti-join/`

effettuare corettamente l'anti-join, sarebbe necessario effettuare delle operazioni di normalizzazione del testo su entrambe le colonne dei rispettivi file;

Presenza di duplicati: *GetDep_ast.py* fornisce in output un file in formato tabellare che raccoglie le informazioni su tutte le dipendenze del progetto. In alcuni casi questo file, potrebbe presentare delle righe duplicate. Questo introduce ambiguità nei risultati ottenuti e sarebbe quindi necessario prima di effettuare ogni operazione, eliminare le righe duplicate;

- **Limiti progettuali di PyCD**: PyCD non è stato progettato per fornire in maniera diretta le informazioni relative a quali sono le bloated dependency del progetto. Volendosi affidare solamente a questo strumento, esso si dimostra essere inadeguato. Come descritto precedentemente, dovendo effettuare particolari modifiche per garantirne il funzionamento desiderato, si è preferito sfruttare PyCD per come è stato realmente ideato;

- **Presenza di altri tool**: Nella Sezione 2.2.4 - Alcuni utili tool, sono stati presentati i tool *fawltydeps* e *safety* che possono essere utili all'avanzamento di questo studio. Piuttosto che utilizzare un unico strumento, risulta vantaggioso unire e combinare il funzionamento dei vari strumenti, per crearne uno nuovo in grado di utilizzare al meglio le funzionalità degli strumenti già esistenti.

3.2 BloatWeak

Per favorire l'avanzamento di questo studio, è stato realizzato BloatWeak, uno strumento in grado di automatizzare l'identificazione delle bloated dependency e le relative vulnerabilità associate. Dovendo far uso dello script *GetDep_ast.py* di PyCD, si è deciso di effettuare il fork[3] del progetto iniziale. Una volta clonato il nuovo progetto, per garantire il corretto funzionamento di BloatWeak è necessario creare un virtual-environment e installare attraverso pip le seguenti librerie di terze parti:

- Richieste direttamente da BloatWeak:
  - **fawltydeps**, **pandas**, **packaging**;
- Richieste da *GetDep_ast.py*:
  - **astunparse**, **requests**, **toml**.

Lo script prevede di ricevere in input due argomenti dalla riga di comando, rispettivamente il path del progetto da analizzare e il path della directory dove salvare i file di output. Il comando da eseguire è costruito come di seguito:

```
python3 bloatWeak.py /path/to/the/project/ /path/to/save/dir/
```

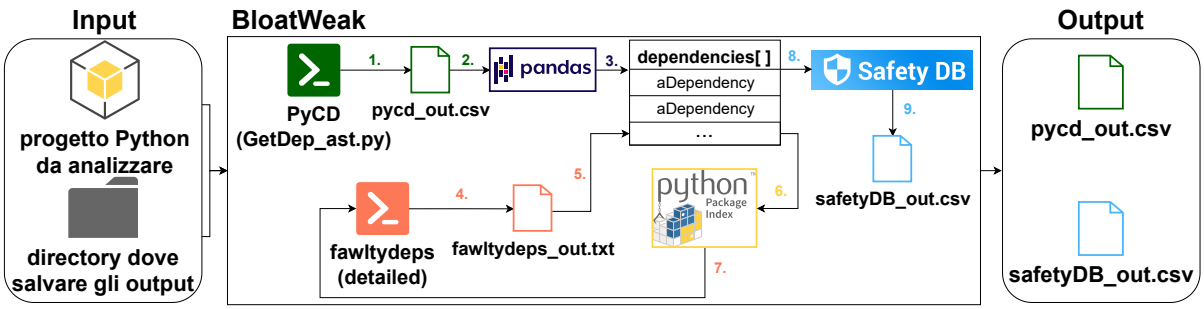

Figura 3.2: La struttura di BloatWeak.

La Figura 3.2 rappresenta graficamente la struttura dello strumento. La sua esecuzione può essere descritta nei seguenti punti:

[3] https://github.com/Tensa53/BloatWeak

1. Dati gli argomenti dalla riga di comando, esegue come sottoprocesso lo script *GetDep_ast.py* ed ottiene il **primo file di output** in formato tabellare denominato **`pycd_out.csv`**, dove sono presenti tutte le informazioni relative alle dipendenze del progetto analizzato, eventuali righe duplicate sono eliminate.

2. Per ogni dipendenza, attraverso l'utilizzo di *pandas* recupera il nome, la relativa versione e il path del rispettivo file di configurazione dove è stata dichiarata la dipendenza.

3. Memorizza le informazioni recuperate modellandole in una classe, per aggiungerle alla lista delle dipendenze;

4. Dati gli argomenti dalla riga di comando, esegue come sottoprocesso il tool *fawltydeps* in modalità detailed ed ottiene il suo output sottoforma di un file di testo denominato `fawltydeps_out.txt`, dove sono presenti i nomi di tutte le dipendenze inutilizzate e i path dei file di configurazione dove sono presenti le dichiarazioni delle rispettive dipendenze;

5. Legge il file `fawltydeps_out.txt` e per ogni riga:

    5.1 Estrae il nome della dipendenza dalla riga e fin quando non trova una nuova riga con il nome di un'altra dipendenza:

        5.1.1 Legge la riga successiva per estrarre il path del file di configurazione;

        5.1.2 Se il path letto corrisponde ad un file di configurazione contenente dipendenze necessarie all'installazione del software, allora attraverso il nome della dipendenza e al path individuato, ricerca la rispettiva dipendenza nella lista per identificarla come inutilizzata;

6. Visita tutti gli elementi della lista delle dipendenze e se una dipendenza è stata identificata come inutilizzata, recupera la relativa libreria da PyPI installandola attraverso pip;

7. Ripete i punti 4. e 5. avendo l'accortezza di identificare le dipendenze realmente inutilizzate;

8. Visita tutti gli elementi della lista delle dipendenze e attraverso il nome della dipendenza e la relativa versione, interroga il database del tool *safety* per verificare se la dipendenza è vulnerabile.

9. Recupera tutte le informazioni relative alle dipendenze e attraverso l'utilizzo di *pandas* costruisce il **secondo file di output** in formato tabellare denominato **`safteyDB_out.csv`** che presenta le seguenti colonne:

   - **dep**: Il nome della dipendenza;
   - **version**: La versione della relativa dipendenza;
   - **filepath**: Il path relativo al file di configurazione dove è stata dichiarata la dipendenza;
   - **bloated**: Un flag per identificare se la dipendenza è bloated o meno;
   - **cve**: L'identificativo CVE (Common Vulnerabilities and Exposures) che indica la vulnerabilità presente nella dipendenza;
   - **affected versions**: Tutte le versioni che presentano la vulnerabilità indicata;
   - **advisory**: Alcune informazioni che descrivono la vulnerabilità, in maniera discorsiva.

dep	version	filepath	type	condition	status
importlib_resources	>=5.0.0	/home/daniele/git/mycli/setup.py	install _requires	@(sys.version_info.minor < 9)@*	*
paramiko	==*	/home/daniele/git/mycli/setup.py	extras_require	*@*	*
click	>=7.0	/home/daniele/git/mycli/setup.py	install_requires	*@*	*
cryptography	==36.0.2	/home/daniele/git/mycli/setup.py	install_requires	*@*	*
Pygments	>=1.6	/home/daniele/git/mycli/setup.py	install_requires	*@*	*
prompt_toolkit	>=3.0.6,<4.0.0	/home/daniele/git/mycli/setup.py	install_requires	*@*	*
PyMySQL	>=0.9.2	/home/daniele/git/mycli/setup.py	install_requires	*@*	*
sqlparse	>=0.3.0,<0.5.0	/home/daniele/git/mycli/setup.py	install_requires	*@*	*
sqlglot	>=5.1.3	/home/daniele/git/mycli/setup.py	install_requires	*@*	*
configobj	>=5.0.5	/home/daniele/git/mycli/setup.py	install_requires	*@*	*

Tabella 3.1: Uno snippet della tabella delle dipendenze per il progetto mycli.

Grazie alla creazione di questo script è possibile ottenere un primo file in formato tabellare che recupera l'output di *GetDep_ast.py*, mentre il secondo file di output contiene le informazioni delle dipendenze, le eventuali vulnerabilità associate e quali

sono le bloated dependency. Dovendo analizzare più versioni della stessa dipendenza e non potendo vincolare tutte le versioni da analizare in una sola dichiarazione, si è scelto di non usare direttamente il tool *safety*. Nel voler costruire "una sorta di" file di configurazione con tutte le dichiarazioni delle dipendenze da analizzare, il tool avrebbe scartato le successive dichiarazioni che presentavano lo stesso nome di dipendenza ma diversa versione. Per questo motivo, è conveniente interrogare direttamente il database del tool e recuperare i dati necessari. Ogni dipendenza è identificata attraverso una tripla formata dal nome della dipendenza, la versione e il path del file di configurazione dove è presente questa dichiarazione. In `safteyDB_out.csv` possono essere presenti più righe che contengono la stessa tripla, questo vorrà dire che quella particolare dipendenza presenta più di una vulnerablità. Quando invece una dipendenza non presenta alcuna vulnerabilità, le colonne relative alle informazioni sulle vulnerabilità nella sua riga, assumeranno il valore testuale "no one".

dep	version	filepath	bloated	cve	affected versions	advisory
click	>= 7.0	/home/daniele/git/mycli/setup.py	False	PVE-2022-47833	<8.0.0	Click 8.0.0 uses 'mkstemp()' instead of the deprecated & insecure 'mktemp()'.
cryptography	==36.0.2	/home/daniele/git/mycli/setup.py	True	CVE-2023-0215	<39.0.1	Cryptography 39.0.1 updates its dependency 'OpenSSL' to v3.0.8 to include security fixes.
cryptography	==36.0.2	/home/daniele/git/mycli/setup.py	True	CVE-2023-0401	<39.0.1	Cryptography 39.0.1 updates its dependency 'OpenSSL' to v3.0.8 to include security fixes.
importlib_resources	>=5.0.0	/home/daniele/git/mycli/setup.py	False	no one	no one	no one
paramiko	==*	/home/daniele/git/mycli/setup.py	False	CVE-2008-0299	<1.7.2	Paramiko 1.7.2 includes a fix for CVE-2008-0299
paramiko	==*	/home/daniele/git/mycli/setup.py	False	CVE-2022-24302	<2.10.1	In Paramiko before 2.10.1, a race condition could allow unauthorized information disclosure.
prompt_toolkit	>=3.0.6	/home/daniele/git/mycli/setup.py	False	no one	no one	no one
Pygments	>=1.6	/home/daniele/git/mycli/setup.py	False	CVE-2022-40896	<2.15.0	Pygments 2.15.0 includes a fix for CVE-2022-40896
Pygments	>=1.6	/home/daniele/git/mycli/setup.py	False	CVE-2021-27291	>=1.1,<2.7.4	Pygments 2.7.4 includes a fix for CVE-2021-27291
PyMySQL	>=0.9.2	/home/daniele/git/mycli/setup.py	False	no one	no one	no one

Tabella 3.2: Uno snippet della tabella del file safteyDB_out.csv.

Per mostrare un esempio di funzionamento dello script, è stato selezionato come progetto di prova *mycli*. Considerando l'output relativamente al progetto analizzato, nella Tabella 3.1 viene mostrato il contenuto del file `pycd_out.csv`, mentre nella Tabella 3.2 viene mostrato il contenuto del file `safteyDB_out.csv`. Il tool prevede anche un output sulla shell come mostrato in Figura 3.3 che riassume quali sono le bloated dependency e ne specifica il path del file di configurazione dove sono dichiarate, insieme ad un eventuale avviso per indicare che la dipendenza è vulnerabile. L'utilizzo del tool ideato sarà fondamentale nella fase di sperimentazione per raccogliere i risultati di questo studio.

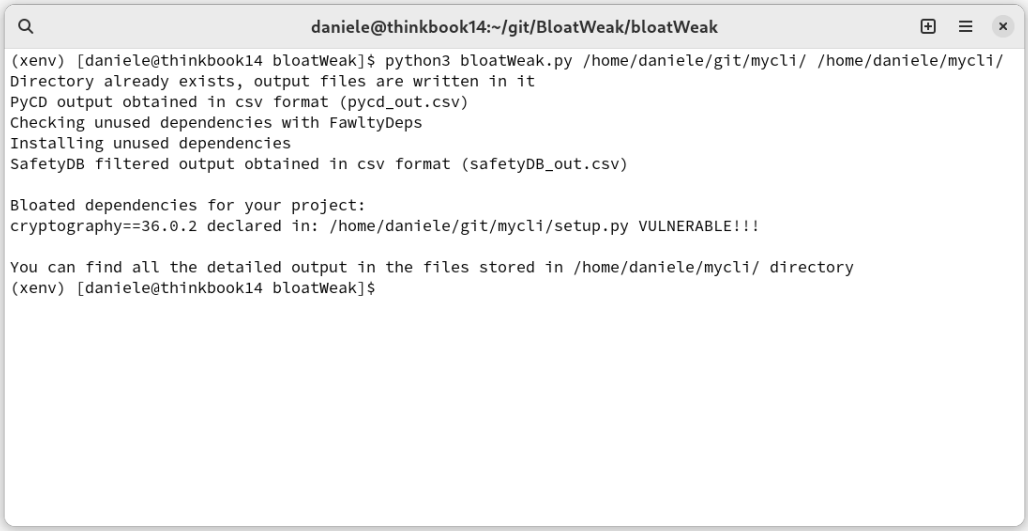

Figura 3.3: L'output sulla shell di BloatWeak per il progetto mycli.

CAPITOLO 4

Validazione Manuale di BloatWeak

In questo capitolo sono descritte le operazioni di validazione del tool realizzato, effettuando un'analisi approfondita di un ristretto campione di progetti attraverso il calcolo di alcune metriche e controllando il codice sorgente.

4.1 Dataset dei 10 progetti

Prima di passare alla raccolta dei risultati, bisogna verificare e validare il tool realizzato. Per questo motivo, sono stati scelti 10 progetti sui quali provare il tool e calcolare le metriche necessarie alla validazione. Per ognuno dei progetti selezionato è stato analizzato il codice sorgente, così da poter confermare il corretto funzionamento di BloatWeak. Dei 10 progetti, 5 sono stati scelti dal dataset NICHE[1]. Il motivo di questa scelta è dovuto al voler confrontare come si comporta il tool con progetti di natura diversa da quelli che sono parte del dataset. Il dataset sarà ancora oggetto di questo studio nella successiva fase di sperimentazione. Inoltre 2 di questi progetti sono già stati usati per mostrare gli esempi nei capitoli precedenti. Di seguito sono elencati i 10 progetti:

[1] `https://github.com/soarsmu/NICHE`

- Scelti dal dataset NICHE:
  - **deepmedic**[2], **fancyimpute**[3], **flashtorch**[4], **lucent**[5], **mcfly**[6];
- Scelti al di fuori del dataset NICHE:
  - **mycli**, **python-fire**[7], **rembg**[8], **voltron**, **wifiphisher**[9].

Tutte le informazioni relative alla validazione sono state raccolte in un unico file tabellare denominato `validation_report.csv` che viene mostrato nella Tabella 4.1. Il file presenta le seguenti colonne:

- **Github Repo**: Il link al repository del codice sorgente del progetto;
- **detectedBloated**: Un flag che indica se il tool ha identificato il progetto come affetto da bloated dependency;
- **realBloated**: Un flag che indica se in seguito all'analisi del codice sorgente il progetto risulta ancora essere affetto da bloated dependency;
- **#modules**: Il numero di moduli del codice sorgente analizzato;
- **comment**: Alcuni commenti che dettagliano i valori delle precedenti colonne.

[2] `https://github.com/deepmedic/deepmedic`
[3] `https://github.com/iskandr/fancyimpute`
[4] `https://github.com/MisaOgura/flashtorch`
[5] `https://github.com/greentfrapp/lucent`
[6] `https://github.com/NLeSC/mcfly`
[7] `https://github.com/google/python-fire`
[8] `https://github.com/danielgatis/rembg`
[9] `https://github.com/wifiphisher/wifiphisher`

Github Repo	detectedBloated	realBloated	#modules	comment
deepmedic/deepmedic	False	False	42	correctly detected as a non bloated one
iskandr/fancyimpute	True	False	12	cvxopt dependency is an optional transitive for cvxpy
MisaOgura/flashtorch	False	False	8	corretcly detected as a non bloated one
greentfrapp/lucent	True	False	26	future dependency is not bloated, the import name differs
NleSC/mcfly	True	True	11	correctly detected as a bloated one
dbcli/mycli	True	False	28	cryptography dependency is a situational transitive for paramiko
google/python-fire	True	False	23	enum34 is a situational dependency for python < 3.4
danielgatis/rembg	True	True	21	correctly detected as a bloated one
snare/voltron	True	True	43	correctly detected as a bloated one
wifiphisher/wifiphisher	True	True	25	correctly detected as a bloated one

Tabella 4.1: Il contenuto del file validation_report.csv.

4.2 Metodo di validazione e risultati

Facendo uso delle informazioni della Tabella 4.1, possiamo classificare i progetti secondo i criteri stabiliti nella Tabella 4.2. Dei 10 progetti manualmente analizzati, 4 risultano essere veri positivi, 2 risultano essere veri negativi, 4 risultano essere falsi positivi. Nessun progetto risulta essere un falso negativo. Inoltre considerando i due gruppi da 5 progetti rispettivamente parte del dataset NICHE e non parte del dataset, in entrambi i casi 3/5 progetti sono correttamente identificati come veri positivi o veri negativi. Questo dimostra che la natura del progetto e il caso d'uso per il quale viene realizzato non sono fattori che influenzano il corretto funzionamento del tool, ma i fattori da considerare sono altri. Uno di questi è la problematica relativa al valore semantico di alcune dichiarazioni di dipendenze, che dal punto di vista sintattico sono considerabili come bloated ma che nella realtà del progetto hanno un utilizzo situazionale. I casi dell'analisi dei progetti mycli e fancyimpute evidenziano questa problematica. In alcuni casi è possibile aggiungere un valore semantico alla dichiarazione di una dipendenza aggiungendo un ulteriore vincolo come ad esempio la versione di pyhton che richiede sia necessario installare questa dipendenza. Il caso dell'analisi di python-fire ha evidenziato questa problematica. Inoltre sono da considerare anche i costi di tempo nell'effettuare l'analisi manuale. Bisogna perciò individuare il giusto compromesso tra quantità di tempo da alloccare alle operazioni

	realBloated=True	**realBloated=False**
detecedtBloated=True	**Veri Positivi (VP)**	**Falsi Positivi (FP)**
detectedBloated=False	**Falsi Negativi (FN)**	**Veri Negativi (VN)**

Tabella 4.2: Tabella di errata classificazione per la validazione di BloatWeak.

di analisi manuale e fedeltà dei risultati ottenibili. Ci sono quindi alcuni limiti ancora presenti nel tool ed altri legati ai costi di tempo, che possono comunque in parte influenzare i risultati di questo studio. Per concludere la validazione e passare alla raccolta dei risultati, sono state calcolate alcune metriche per rafforzare le precedenti considerazioni. La Precision è una metrica che viene calcolata effettuando la divisione tra il numero di veri positivi, per la somma del numero di veri positivi e falsi positivi. Attraverso questa metrica è possibile capire quanto bene lavora il tool nell'effettuare la principale attività per il quale è stato ideato, ovvero individuare i progetti affetti dal fenomeno delle bloated dependency. In questo caso la precision vale 0.5. Questo vuol dire che sulla base di un dataset da analizzare, BloatWeak è attualmente in grado di identificare correttamente come progetti affetti da bloated dependency, solo la metà di quelli presenti nel dataset. Questa metrica dipende molto dagli attuali limiti del tool legati al valore semantico delle dichiarazioni. Per migliorare la precisione di BloatWeak, sarà quindi necessario capire come sorpassare questi limiti.

$$Precision = \frac{VP}{(VP + FP)} = \frac{4}{4+4} = \frac{4}{8} = 0.5$$

La Recall è una metrica che viene calcolata effettuando la divisione tra il numero di veri positivi, per la somma del numero di veri positivi e falsi negativi. Attraverso questa metrica è possibile capire quanto bene lavora il tool nell'identificare i progetti non affetti dal fenomeno delle bloated dependency. In questo caso la precision vale 1 perchè tutti i progetti inizialmente identificati come non bloated sono risultati esserlo per davvero. Questa metrica dipende molto dal dataset analizzato, in questo caso vale 1, ma potrebbero esserci altri dataset dove il suo valore è del tutto diverso. La Recall va quindi interpretata ponendo molta attenzione al dataset analizzato.

$$Recall = \frac{VP}{VP + FN} = \frac{4}{4+0} = \frac{1}{1} = 1$$

L'Accuracy è una metrica che viene calcolata effettuando la divisione tra la somma del numero di veri positivi e veri negativi, per il numero totale di progetti analizzati. Attraverso questa metrica è possibile capire in generale quanto bene lavora il tool nella corretta identificazione dei progetti, che siano o non siano affetti dal fenomeno delle bloated dependency. In questo caso l'accuracy vale 0.6. Questa metrica dipende sia dal numero di veri positivi che veri negativi, potrebbero quindi esserci casi in cui l'Accuracy sia anche più alta dello 0.6 ma nonostante ciò il tool potrebbe comunque non lavorare bene sull'identificazione dei progetti affetti da bloated dependency. Come per la Recall, anche l'Accuracy dipende dal dataset analizzato. Potremmo ad esempio trovarci in un caso in cui il dataset analizzato non presenta veri positivi ma sono invece presenti i veri negativi, andando ad azzerare le altre due metriche e ottenendo comunque un valore positivo rispetto al calcolo dell'Accuracy. Per questo motivo, è quindi importante calcolare tutte e tre le metriche mostrate, per comprendere al meglio i risultati della valdiazione.

$$Accuracy = \frac{(VP + VN)}{(VP + VN + FP + FN)} = \frac{4+2}{4+4+2+0} = \frac{6}{10} = 0.6$$

CAPITOLO 5

Sperimentazione

In questo capitolo sono descritte le Research Question individuate e successivamente il dataset di progetti chiamato NICHE, opportunatamente filtrato, sul quale sarà effettuato il processo di sperimentazione ai fini di raccogliere le informazioni necessarie per fornire i risultati di questo studio. Inoltre viene descritta la procedura di analisi adottata e poi i risultati ottenuti sulla base dei dati raccolti per i progetti misurabili del dataset NICHE.

5.1 Research Question

Grazie alla realizzazione di BloatWeak, possiamo combinare l'utilizzo di diversi tool per individuare le bloated dependency e le vulnerabilità associate. I risultati che intende fornire questa tesi sono preliminari e non esaustivi, per questo motivo è stato selezionato un ristretto campione di esempio di progetti per verificare e validare il funzionamento di Bloatweak. Potendo però eseguire il tool sul campione più ampio dei 365 progetti misurabili, è possibile comunque recuperare degli utili risultati, tenendo sempre conto delle considerazioni evidenziate precedentemente. L'obiettivo di questa tesi è quello di presentare il fenomeno delle bloated dependency e osservarne la diffusione all'interno di un particolare dataset di progetti.

Le bloated dependency oltre ad aumentare le dimensioni dell'artefatto software, rendono più ampia la superficie di attacco. Questa ulteriore supericie può lasciare spazio ad alcune vulnerabilità che semplificano il lavoro degli attaccanti. Una dipendenza potrebbe non solo essere bloated ma essere anche vulnerabile. Questi due fenomeni delle bloated dependency e delle dipendenze vulnerabili, vanno analizzati sia in maniera separata e anche quando i due fenomeni si possano combinare sullo stesso progetto, specialmente se il fenomeno è ulteriormente combinato sulle stesse dipendenze. Per facilitare la raccolta dei risultati e analizzare al meglio questi fenomeni, sono stati creati dei grafici grazie all'utilizzo di *matplotlib*[10] e sono state formulate tre diverse Research Question che saranno discusse nella sezione 5.4 - Risultati. Per ogni Research Question sarà analizzato il relativo fenomeno considerato per capire la sua diffusione e saranno calcolate misure di base come media, moda, mediana. Infine saranno effettuate alcune considerazioni rispetto ai risultati raccolti. Di seguito sono elencate le formulazioni delle rispettive RQ:

🔍 **RQ$_1$.** *Quanto è diffuso il fenomeno delle bloated dependency, all'interno di progetti Python che fanno uso di soluzioni ML?*

🔍 **RQ$_2$.** *Quanto è diffuso il fenomeno delle dipendenze vulnerabili, all'intero di progetti Python che fanno uso di soluzioni ML?*

🔍 **RQ$_3$.** *Come sono legate tra loro le diffusioni dei fenomeni delle bloated dependency e delle dipendenze vulnerabili, all'interno di progetti Python che fanno uso di soluzioni ML?*

5.2 Dataset

Widyasari et al.[11] hanno costruito NICHE[1], un dataset contenente una serie di progetti software che fanno uso di moduli che integrano soluzioni intelligenti basandosi su modelli di AI/ML. Affinchè un progetto potesse far parte di questo dataset sono stati considerati diversi criteri. Alcuni criteri sono relativi alla qualità del progetto:

[1] `https://github.com/soarsmu/NICHE`

- **Liberie di ML usate**: Un progetto deve utilizzare almeno una delle tre principali e più popolari librerie utilizzate nel campo del ML: **Theano**[2], **PyTorch**[3], **Tensorflow**[4].
- **Popolarità**: Un progetto deve avere almeno 100 star su GitHub, semplificando il processo della distinzione tra progetti ingegnerizzati e non ingegnerizzati;
- **Attività**: Un progetto deve avere almeno un commit effettuato di recente nei mesi scorsi, così da poter analizzare progetti ancora attivi;
- **Disponibilità del progetto**: Un progetto deve essere pubblicamente accessibile attraverso le GitHub API, per essere clonato su una nuova macchina;
- **Storia dei Commit**: Un progetto deve avere almeno 100 commit, distinguendo progetti creati da poco tempo rispetto a progetti creati da maggior tempo;

In seguito all'applicazione di questo primo gruppo di criteri, sono stati individuati 572 progetti. Altri criteri sono relativi al livello di ingegnerizzazione del software, questi criteri sono stati ripresi da un precedente lavoro di Munaiah et al.[12] che hanno realizzato reaper[5], un tool capace di analizzare un progetto e definirne i seguenti criteri:

- **Architettura**: Un progetto deve prevedere un inisieme di relazioni ben definite tra le sue componenti;
- **Community**: Un progetto deve prevedere una comunità di sviluppo, così da mostrare la presenza di una forma di collaborazione;
- **Continuous Integration (CI)**: Un progetto deve prevedere una pipeline di CI;
- **Documentazione**: Un progetto deve avere una sufficiente documentazione da poter permettere una corretta manutenzione;
- **Storico**: Un progetto deve avere uno storico della attività, indice che il progetto viene migliorato per garantirne la sua funzionalità;

[2] `http://deeplearning.net/software/theano/`
[3] `https://pytorch.org/`
[4] `https://www.tensorflow.org/`
[5] `https://github.com/reporeapers/reaper`

- **Difetti**: Un progetto deve far uso della sezione Issues di GitHub, per poter rapidamente capire se la gestione del progetto sta avvenendo in maniera corretta;
- **Licenze**: Un progetto deve avere una licenza che stabilisce i vincoli sulla riproducibilità e distrubizione di sorgenti e binari;
- **Testing di unità**: Un progetto deve avere dei testing di unità, così da assicurare il corretto funzionamento del software;

In seguito all'applicazione di questo secondo gruppo di criteri, dei 572 progetti individuati, 441 sono progetti ingegnerizzati e 131 sono progetti non ingegnerizzati. Questo dataset sarà il punto di partenza della fase di sperimentazione. Sono presi in considerazione esclusivamente i progetti ingegnerizzati.

5.3 Procedura di Analisi

I file in formato tabellare sono molto utili per essere processati dai software di analisi statistica, così da poter calcolare in maniera più rapida misure e metriche ai fini di voler fornire i risultati di questo studio. Il campione selezionato da analizzare è parte del dataset NICHE. Per ogni progetto da analizzare, il processo di sperimentazione prevede i seguenti passi:

1. **Clonazione del progetto**: Attraverso il link del repository, sono scaricati i sorgenti del progetto;
2. **Esecuzione di BloatWeak**: Il tool è eseguito sul progetto per ottenere i file di output;
3. **Esecuzione dello script delle misure**: Per raggruppare le informazioni più utili del progetto, è stato realizzato lo script *measures_proj.py* che estrae dal file `safetyDB_out.csv` le seguenti misure:
    - **Numero di dipendenze totali**
    - **Numero di bloated dependency**

- **Numero di dipendenze vulnerabili**
- **Numero di bloated dependency vulnerabili**
- **Numero di vulnerabilità totali**

Ogni punto di questo processo sarà effettuato in modalità batch sul campione di progetti analizzati. Lo script delle misure produrrà un output finale in forma tabellare contenente le misure dei progetti. Attraverso queste misure, sarà possibile calcolare tutte le metriche che saranno ritenute adeguate a validare la sperimentazione e i risultati di questo studio. Come però evidenziato dagli stessi autori di NICHE, nel corso del tempo alcuni progetti presenti nel dataset potrebbero aver subito variazioni tali da doverli successivamente escludere. Inoltre bisogna verificare il corretto funzionamento di BloatWeak su ogni progetto del dataset. Per questi motivi sono state effettuate delle operazioni di filtraggio del dataset, a seguito di ognuno dei passi indicati nella descrizione del processo di sperimentazione. I progetti che sono stati tenuti in considerazione sono esclusivamente quelli ingegnerizzati e nel filtrare il dataset rimuovendo i progetti non ingegnerizzati, sono stati in realtà individuati 440 progetti ingegnerizzati contro i 441 indicati. Dopo aver clonato i progetti, non è stato possibile recuperare 1 dei 440 progetti ingegnerizzati, poichè il criterio di disponibilità del progetto è cambiato. Non risultando più pubblicamente accessibile essendo stato reso privato dagli autori del progetto, si è deciso di scartarlo. Dopo aver eseguito BloatWeak su tutti i progetti, non è stato possibile raccogliere gli output di 25 dei 439 progetti pubblici. In particolare 8 di questi sono progetti realizzati con linguaggi di programmazione diversi da Python, mentre per i restanti 17 viene restituito un output vuoto o nullo in seguito all'esecuzione come sottoprocesso dello script *GetDep_ast.py* di PyCD. Non potendo raccogliere gli output per questi progetti, si è deciso di scartarli. Dopo aver eseguito lo script delle misure, non è stato possibile raccogliere un ouptut completo per 49 dei 414 progetti compatibili. In particolare, 5 di questi mandano in errrore l'esecuzione come sottoprocesso del tool *fawltydeps*, mentre per i 44 restanti viene restituito un output incompleto in seguito all'esecuzione come sottoprocesso dello script *GetDep_ast.py* di PyCD. Non potendo raccogliere gli output completi di questi progetti, si è deciso di scartarli. In totale, ci sono a dispozione 365 progetti misurabili per verificare e validare il corretto funzionamento di BloatWeak e

dei tool interni. Tutte le informazioni del filtraggio sono state raccolte in un unico file tabellare denominato `report.csv`, di cui viene mostrato uno snippet nella Tabella 5.1. Il file presenta le seguenti colonne:

- **GiHub Repo**: Il link al repository del codice sorgente del progetto;
- **Cloneable**: Un flag che indica se è possibile clonare il repository del progetto;
- **Compatible**: Un flag che indica se su questo progetto BloatWeak viene eseguito senza errori;
- **Measurable**: Un flag che indica se su questo progetto BloatWeak riesce a fornire un output completo;
- **Implementation**: Indica il linguaggio in cui è stato realizzato il progetto o la piattaforma per cui è stato progettato;
- **Comment**: Alcuni commenti che dettagliano i valori delle precedenti colonne.

GitHub Repo	Cloneable	Compatible	Measurable	Implementation	Comment
apache/beam	True	False	False	Java	This project is not written in Python but in Java
apache/submarine	True	True	True	Python	This project is correctly measurable by measure_proj.py script
apachecn/Interview	True	True	True	Python	This project has no bloated dependencies
apple/turicreate	True	True	False	Python	This project has incomplete output. FawltyDeps returning some errors
arita37/mlmodels	False	False	False	Python	This project has not a public repo anymore
ARM-software/CMSIS_5	True	True	True	Python	This project has no bloated dependencies
arogozhnikov/einops	True	False	False	Python	This project has no output. PyCD returning an empty/no file
arogozhnikov/hep_ml	True	True	True	Python	This project is correctly measurable by measure_proj.py script

Tabella 5.1: Uno snippet di report.csv.

5.4 Risultati

5.4.1 RQ1: La diffusione delle bloated dependency

🔍 **RQ_1.** *Quanto è diffuso il fenomeno delle bloated dependency, all'interno di progetti Python che fanno uso di soluzioni ML?*

Dopo aver effettuato il processo di sperimentazione e aver raccolto i risultati, considerando i 365 progetti misurabili, 70 progetti sono affetti dal fenomeno delle bloated dependency. In media il numero di bloated dependency nei progetti è circa 8. Attraverso la diffusione delle bloated dependency, è possibile osservare che 26 progetti hanno un numero di bloated dependency pari a 1 (valore moda). In totale, 59 progetti, 14 più della metà dei 70, hanno un numero di bloated dependency minore di 5 (valore mediana). Tra gli 11 progetti rimanenti, 10 di questi hanno un numero di bloated dependency comunque minore di 10. L'ultimo progetto rimanente (*vaex*), ha invece il numero più alto di bloated dependency pari a 23 e ha complessivamente un numero di dipendenze pari a 120. All'interno dei 70 progetti, è presente anche un altro progetto (*PARL*) che ha un numero di dipendenze pari a 120, ma in questo caso il suo numero di bloated dependency è pari a 1. Per questo motivo, non è sempre vero che all'aumentare delle dipendenze aumenti anche il numero delle bloated dependency. Tutti i valori descritti in questa sezione, sono stati rappresentati attraverso un grafico a barre nella Figura 5.1. Sull'asse x, sono presenti i valori legati ai possibili numeri di bloated dependency (#bloated), mentre sull'asse y sono presenti i valori legati al numero di progetti (#project) che hanno il corrispondente numero di bloated dependency.

👉 **Per rispondere alla RQ_1.** 70 progetti su 365 sono affetti dal fenomeno delle bloated dependency. Le ragioni per cui una dipendenza è identificata come bloated sono condizioni non intriseche della dipendenza stessa, ma condizioni strettamente legate al relativo progetto in cui è dichiarata e quanto bene è stato organizzato quel progetto. Con il migliorare dell'organizzazione dei progetti software, diminuiscono le bloated dependency.

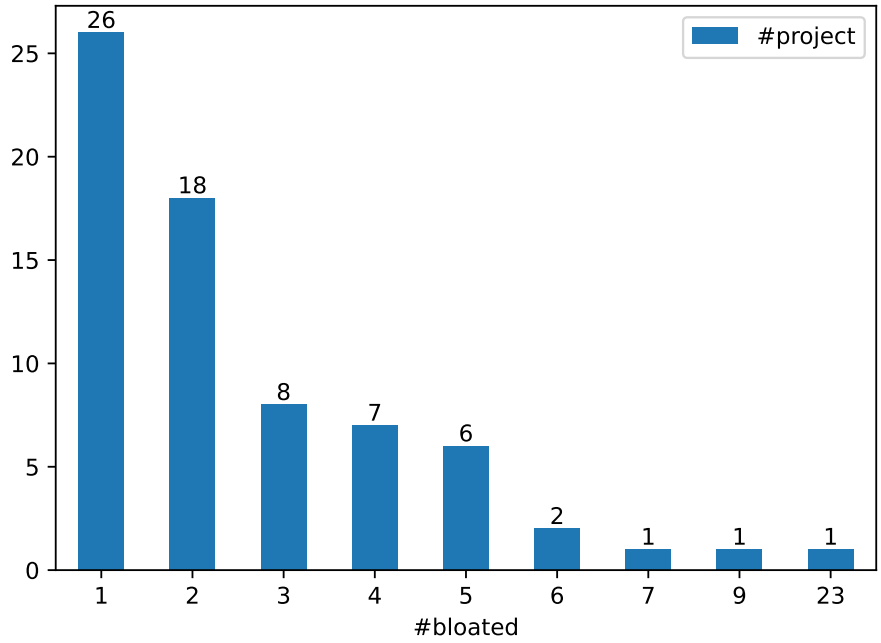

Figura 5.1: Il grafico a barre della diffusione delle bloated dependency.

5.4.2 RQ2: La diffusione delle dipendenze vulnerabili.

RQ$_2$. *Quanto è diffuso il fenomeno delle dipendenze vulnerabili, all'intero di progetti Python che fanno uso di soluzioni ML?*

Dopo aver effettuato il processo di sperimentazione e aver raccolto i risultati, considerando i 365 progetti misurabili, 355 progetti sono affetti dal fenomeno delle dipendenze vulnerabili. Attraverso la diffusione delle dipendenze vulnerabili è possibile osservare che 31 progetti hanno un numero di dipendenze vulnerabili pari a 7 (valore moda). In totale, 309 progetti, 131 più della metà dei 355, hanno un numero di dipendenze vulnerabili minore di 29 (valore mediana). Tra i 46 progetti rimanenti, ci sono in particolare alcuni progetti da dover considerare. Il progetto *Monk_Object_Detection* ha un numero di dipendenze vulnerabili pari a 687 e ha complessivamente un numero di dipendenze pari a 1581. Il progetto *feast* ha un numero di dipendenze vulnerabili pari a 172 e ha complessivamente un numero di dipendenze pari a 1239. Bisognerebbe analizzare approfonditamente i due progetti per capire i motivi di questi numeri. I due progetti sono stati conteggiati anche nella RQ1 ma il loro numero di bloated dependency (rispettivamente pari a 6 e 5) non è stato influen-

zato dal numero complessivo di dipendenze. Risulta essere presente anche un altro progetto con un numero di dipendenze vulnerabili pari a 173, molto simile al numero di *feast*. Si è deciso quindi di considerare il progetto *Monk_Object_Detection* come un outlier per la RQ2 e di non conteggiarlo nei progetti affetti dal fenomeno delle dipendenze vulnerabili. Sarà invece 173 il numero di dipendenze vulnerabili più alto per i progetti analizzati e la media del numero di dipendenze vulnerabili nei progetti è pari a 38. I progetti *CryptTen* e *sigopt-examples* hanno lo stesso numero di dipendenze vulnerabili pari a 44, ma complessivamente il loro numero di vulnerabilità è diverso, rispettivamente pari a 1031 e 103. I motivi dietro questa differenza nel numero di vulnerabilità, sono probabilmente legati alla tipologia di librerie ML utilizzate. Il progetto *CryptTen* fa uso della libreria *Tensorflow* e il progetto *sigopt-examples* fa invece uso della libreria *PyTorch*. Secondo il database di vulnerabilità note *safetydb*, su tutte le sue versioni la liberia *Tensorflow* ha un numero di vulnerabilità pari a 480, a differenza della libreria *PyTorch* che ha un numero di vulnerabilità pari a 1, sintomo di una possibile incompletezza del database di vulnerabilità utilizzato e ulteriore motivazione di questa disparità di numeri. Infine il progetto *TextAttack* ha un numero di dipendenze pari a 32 ma un numero complessivo di vulnerabilità pari a 1627, maggiore del numero di vulnerabilità di *CryptTen* che ha però un numero maggiore di dipendenze vulnerabili. Per questo motivo, non è sempre vero che il numero più alto di dipendenze vulnerabili implica anche il numero complessivo più alto di vulnerabilità. Tutti i valori descritti in questa sezione, sono stati rappresentati attraverso un grafico a barre nella Figura 5.2. Sull'asse x, sono presenti i valori legati ai possibili numeri di dipendenze vulnerabili (#vulnerable), mentre sull'asse y sono presenti i valori legati al numero di progetti (#project) che hanno il corrispondente numero di dipendenze vulnerabili.

👉 **Per rispondere alla RQ_2.** 355 progetti su 365 sono affetti dal fenomeno delle dipendenze vulnerabili. A differenza delle bloated dependency, le ragioni per cui una dipendenza è identificata come dipendenza vulnerabile sono condizioni intriseche della dipendenza stessa e solo il lavoro degli sviluppatori nell'aggiornare e rilasciare nuove versioni della libreria associata alla dipendenza, può far diminuire il numero di vulnerabilità di una dipendenza vulnerabile.

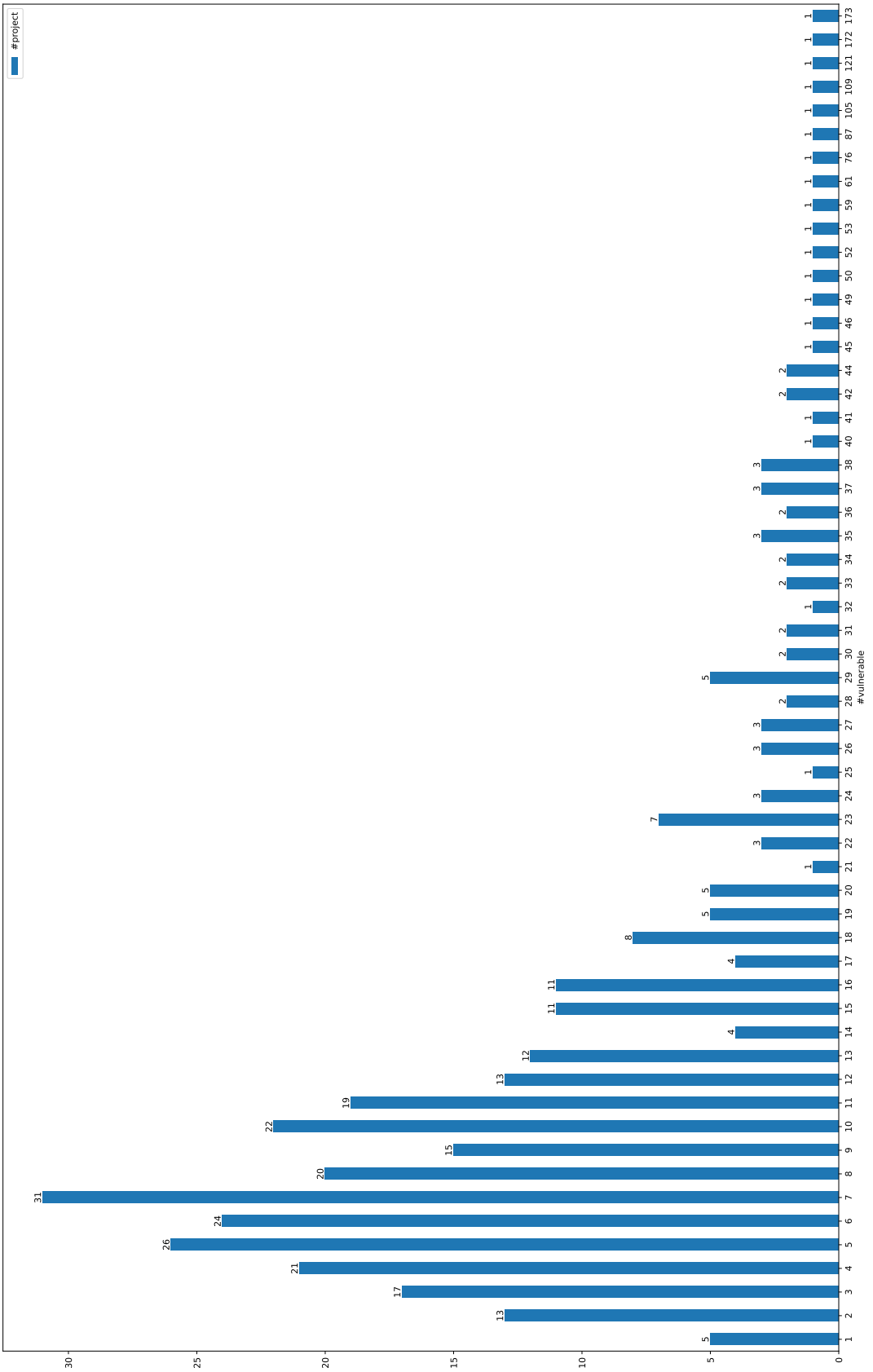

Figura 5.2: Il grafico a barre della diffusione delle dipendenze vulnerabili.

5.4.3 RQ3: Il legame tra bloated dependency e dipendenze vulnerabili

RQ$_3$. *Come sono legate tra loro le diffusioni dei fenomeni delle bloated dependency e delle dipendenze vulnerabili, all'interno di progetti Python che fanno uso di soluzioni ML?*

Dopo aver effettuato il processo di sperimentazione e aver raccolto i risultati, considerando i 365 progetti misurabili e in particolare i 70 affetti dal fenomeno delle bloated dependency, 38 progetti sono affetti da entrambi i fenomeni delle bloated dependency e delle dipendenze vulnerabili. Come per le dipendenze vulnerabili, anche in questo caso si è deciso di non conteggiare il progetto *Monk_Object_Detection* considerabile come un outlier. In media il numero di dipendenze bloated e vulnerabili nei progetti è pari a 3. Attraverso la diffusione in contemporanea dei due fenomeni, è possibile osservare che 29 progetti hanno un numero di dipendenze bloated e vulnerabili pari a 1 (valore moda) e 4 progetti hanno un numero di dipendenze bloated e vulnerabili pari a 2. In totale, 33 progetti, 14 più della metà dei 38, hanno un numero di dipendenze bloated e vulnerabili minore di 3 (valore mediana). Tra i 6 progetti rimanenti, 4 progetti hanno un numero di dipendenze bloated e vulnerabili pari a 3, 1 progetto ha un numero di dipendenze bloated e vulnerabili pari a 4 e l'ultimo progetto ha il numero più alto di dipendenze bloated e vulnerabili pari a 5. Inoltre in 9 progetti, il numero di dipendenze bloated coincide con quelle vulnerabili, in 8 casi c'è un'unica dipendenza bloated e vulnerabile, nell'altro caso ce ne sono 2. Oltre questi 9 progetti, altri 20 progetti hanno un numero di dipendenze bloated e vulnerabili pari ad almeno la metà del loro numero di dipendenze bloated. Il progetto che ha il numero più alto di dipendenze bloated (23) non presenta alcuna dipendenza sia bloated che vulnerabile. Infine è importante sottolineare che i due fenomeni possono non combinarsi tra loro ma risultare comunque più pericolosi in termini di numeri. Ad esempio consideriamo il progetto *botbuilder-pytorch* che ha 2 dipendenze bloated, 38 dipendenze vulnerabili, 1 dipendenza sia bloated che vulnerabile e nel complesso ha 161 vulnerabilità. Invece il progetto *Adlik* ha 3 dipendenze bloated, 38 dipendenze vulnerabili ma nessuna dipendenza sia bloated che vulnerabile e nel complesso ha 1980 vulnerabilità. Tutti i valori descritti in questa sezione, sono stati rappresentati attraverso un grafico a barre nella Figura 5.3. Sull'asse x, sono presenti i valori legati

ai possibili numeri di dipendenze bloated e vulnerabili (#bloatedvulnerable), mentre sull'asse y sono presenti i valori legati al numero di progetti (#project) che hanno il corrispondente numero di dipendenze bloated e vulnerabili.

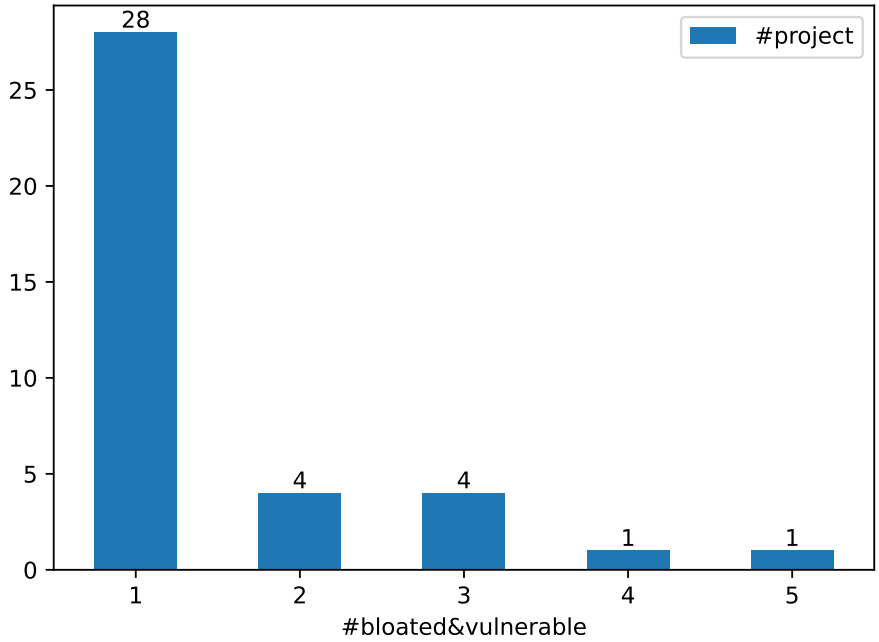

Figura 5.3: Il grafico a barre della diffusione delle dipendenze bloated e vulnerabili.

👉 **Per rispondere alla RQ_3.** 38 progetti su 70 progetti bloated presentano dipendenze contemporaneamente affette sia dal fenomeno delle bloated dependency che delle dipendenze vulnerabili. I due fenomeni potrebbero comunque presentarsi in maniera separata su dipendenze distinte del progetto e portare a risultati più pericolosi, di quanti se ne hanno nel caso in cui i due fenomeni si combinano sulle stesse dipendenze.

CAPITOLO 6

Conclusioni

In questo capitolo sono descritte le attuali limitazioni e lo strumento realizzato. Sono anche considerati degli sviluppi futuri legati ad aspetti collaterali che possono comunque diminuire la diffusione dei fenomeni delle bloated dependency, delle dipendende vulnerabili e della loro combinazione.

6.1 Limitazioni attuali

Per la realizzazione di BloatWeak, sono stati utilizzati dei tool già esistenti. Questi tool sono funzionanti anche in maniera separata e l'approccio che è stato utilizzato per la loro integrazione e combinazione, è di tipo black box. Durante questo studio, è stato assunto che i tool interni fossero già stati ampiamenti verificati e validati come strumenti in grado di fornire risultati precisi. Come già discusso nella Sezione 4.2 - Metodo di validazione e risultati, i limiti legati al valore semantico di alcune dichiarazioni dipendono dal tool interno fawltydeps. Il tool fawltydeps è in grado di identificare le dipendenze che categorizzano un fenomeno più generale rispetto alle bloated dependency detto unused dependency. Sono quindi considerate anche le dipendenze inutilizzate per altre fasi oltre alla build/deploy e installazione dell'artefatto software ottenuto, come ad esempio le fasi di documentazione, im-

plementazione e testing. Queste fasi potrebbero prevedere l'utilizzo di particolari dipendenze che nella realtà non sono bloated poichè non saranno davvero necessarie alla corretta esecuzione del software. Considerando un qualunque progetto da analizzare, nonostante il filtraggio effettuato da BloatWeak relativamente a queste dipendenze e gli ulteriori problemi relativi al valore semantico di alcune dichiarzioni, potrebbero esserci delle dipendenze che vengono erroneamente identificate come bloated. Inoltre bisogna considerare anche le limitazioni dovute al database di vulnerabilità utilizzato. Questo database fornito in modalità gratutita è una versione ridotta di un archivio più grande ma che è disponibile esclusivamente a pagamento. Utilizzando un database più completo, sarebbe possibile individuare al meglio il fenomeno delle dipendenze vulnerabili e ottenere informazioni più precise su come si combina con il fenomeno delle bloated dependency. Infine bisogna considerare il dataset utilizzato che rappresenta un campione molto ristretto di progetti analizzabili. In alcuni casi, BloatWeak ha interrotto la sua esecuzione, restituendo diversi errori rispetto all'analisi di vari progetti. Per migliorare questo studio, bisognerebbe analizzare approfonditamente i motivi di questi errori nell'esecuzione e utilizzare un campione ben più grande del dataset attuale, ampliando la verifica manuale dei progetti sull'intero campione.

6.2 Sviluppi Futuri

Ci sono anche alcuni fattori esterni che potrebbero influenzare la diffusione di questi fenomeni. Bisognerebbe riflettere e analizzare meglio quello che è il gestore pacchetti di Python. Attraverso una analisi approfondita di pip, potrebbe essere possibile capire se alcune problematiche relative alle dipendenze, possano essere già "risolte a monte", integrando stesso nel gestore pacchetti alcune funzionalità di supporto nella corretta gestione delle dipendenze di un progetto. Inoltre sarebbe interessante analizzare le cosiddette "good and bad practises" degli sviluppatori, per educarli al meglio nell'utilizzo dei file di configurazione. Ad esempio le dipendenze necessarie alla fase di testing sono in alcuni casi inserite nella sezione relative alle dipendenze da installare per l'esecuzione del software, come anche bisognerebbe evitare la dipersione delle dichiarazioni delle dipendenze in troppi file di configura-

zione. Infine andrebbe anche analizzata più nel dettaglio la stessa pratica del riuso, specialmente nella situazione in cui bisogna sviluppare una libreria di terze parti. Quando si pensa allo sviluppo di una libreria, bisognerebbe avere un approccio "di basso livello". Laddove possibile, sarebbe ideale diminuire le dipendenze transitive e nelle situazioni meno complesse dove i costi di tempo lo permettono, implementare da zero alcune soluzioni "ad hoc", per rendere la libreria meno esosa in termini di spazio di archiviazione sulla macchina e introducendo codice nuovo e controllato dagli stessi sviluppatori della liberia principale, evitando ulteriori rischi legati alla sicurezza del software realizzato.

6.3 Considerazioni Finali

Grazie a questo studio è stato possibile realizzare BloatWeak, uno strumento che combina il funzionamento di tool già esistenti, automatizzando il processo di identificazione di bloated dependency e relative vulnerabilità di un progetto analizzato. La fase di validazione ha permesso di verificare il funzionamento dello strumento realizzato, ma le metriche hanno anche mostrato che è possibile migliorarne la precisione. Nonostante ciò, la sperimentazione ha raccolto diversi risultati rispetto al dataset utilizzato. BloatWeak si propone come uno strumento in grado di facilitare la vita di uno sviluppatore nell'individuare le bloated dependency, mostrando inoltre eventuali vulnerabilità a cui il software analizzato si espone, in relazione alle varie dipendenze. Sarà poi compito dello sviluppatore, stabilire se risolvere queste problematiche o trascurarle. Risulta importante ricordare che la manutenzione di un software ha dei costi, i quali possono influenzare queste scelte. In alcuni casi, è anche vero che lo sforzo impiegato nel voler risolvere queste problematiche, non è sempre equilibrato alle gravità dei difetti introdotti nel software dalle bloated dependency, scegliendo quindi di trascurare le problematiche. Questo studio analizza il fenomeno delle bloated dependency, su di un particolare dataset di progetti. Per proseguire l'avanzamento della ricerca in questo campo, lo studio attuale si pone come una base di partenza, utilizzabile da altri ricercatori per ampliare e approfondire l'analisi di questo fenomeno, andando a considerare ulteriori dataset di progetti e cercando di fornire ulteriori metriche e risultati migliori, ai fini di superare le limitazioni attuali.

Bibliografia

[1] S. Olbrich, D. S. Cruzes, V. Basili, and N. Zazworka, "The evolution and impact of code smells: A case study of two open source systems," in *2009 3rd International Symposium on Empirical Software Engineering and Measurement*, 2009, pp. 390–400. (Citato a pagina 5)

[2] S. Romano, C. Vendome, G. Scanniello, and D. Poshyvanyk, "A multi-study investigation into dead code," *IEEE Transactions on Software Engineering*, vol. 46, no. 1, pp. 71–99, 2020. (Citato a pagina 5)

[3] C. Soto-Valero, T. Durieux, and B. Baudry, "A longitudinal analysis of bloated java dependencies," in *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ser. ESEC/FSE 2021. New York, NY, USA: Association for Computing Machinery, 2021, p. 1021–1031. [Online]. Available: https://doi.org/10.1145/3468264.3468589 (Citato a pagina 6)

[4] C. Soto-Valero, N. Harrand, M. Monperrus, and B. Baudry, "A comprehensive study of bloated dependencies in the Maven ecosystem," *Empir. Software Eng.*, vol. 26, no. 3, pp. 45–44, Mar. 2021. (Citato alle pagine 7 e 10)

[5] S. Mukherjee, A. Almanza, and C. Rubio-González, "Fixing dependency errors for python build reproducibility," in *Proceedings of the 30th ACM*

SIGSOFT International Symposium on Software Testing and Analysis, ser. ISSTA 2021. New York, NY, USA: Association for Computing Machinery, 2021, p. 439–451. [Online]. Available: https://doi.org/10.1145/3460319.3464797 (Citato a pagina 7)

[6] Y. Cao, L. Chen, W. Ma, Y. Li, Y. Zhou, and L. Wang, "Towards better dependency management: A first look at dependency smells in python projects," *IEEE Transactions on Software Engineering*, vol. 49, no. 4, pp. 1741–1765, 2023. (Citato alle pagine 8 e 11)

[7] S. E. Ponta, W. Fischer, H. Plate, and A. Sabetta, "The used, the bloated, and the vulnerable: Reducing the attack surface of an industrial application," in *2021 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, 2021, pp. 555–558. (Citato a pagina 10)

[8] J. Hejderup, M. Beller, K. Triantafyllou, and G. Gousios, "Präzi: from package-based to call-based dependency networks," *Empir. Software Eng.*, vol. 27, no. 5, pp. 102–42, May 2022. (Citato a pagina 10)

[9] N. E. M. Maria Knorps, "FawltyDeps: A dependency checker for Python." [Online]. Available: https://github.com/tweag/FawltyDeps/blob/main/docs/DesignDoc.md (Citato a pagina 13)

[10] J. D. Hunter, "Matplotlib: A 2d graphics environment," *Computing in Science & Engineering*, vol. 9, no. 3, pp. 90–95, 2007. (Citato a pagina 29)

[11] R. Widyasari, Z. Yang, F. Thung, S. Q. Sim, F. Wee, C. Lok, J. Phan, H. Qi, C. Tan, Q. Tay, and D. Lo, "NICHE: A Curated Dataset of Engineered Machine Learning Projects in Python," *arXiv*, Mar. 2023. (Citato a pagina 29)

[12] N. Munaiah, S. Kroh, C. Cabrey, and M. Nagappan, "Curating GitHub for engineered software projects," *Empir. Software Eng.*, vol. 22, no. 6, pp. 3219–3253, Dec. 2017. (Citato a pagina 30)

Ringraziamenti

Sono giunto ai titoli di coda di questo percorso. Durante i primi mesi di lezioni, ero molto titubante riguardo la mia scelta di proseguire gli studi, non ero convinto di potercela fare e arrivare fino in fondo. Nel corso di questi tre anni ho invece avuto il piacere di ricredermi. Un ringraziamento va quindi a me stesso per aver continuato a crederci e di non essersi buttato giù alle prime difficoltà, riuscendo a sfruttare con il giusto tempismo tutte le opportunità che mi sono state offerte, aggiungendoci anche un pizzico di fortuna. La stesura di questa tesi non sarebbe stata possibile senza l'aiuto di diverse figure che si sono alternate nel corso dei mesi. Ringrazio il Prof. Gravino e il Prof. Palomba per avermi dato la possibilità di studiare queste tematiche e aver sostenuto la realizzazione di questo lavoro. Laddove non c'erano i Professori, ho avuto la possibilità di confrontarmi e discutere con il Dott. Giordano e il Dott. Iannone che si sono sempre dimostrati super disponibili. Ho potuto vivere l'Università in maniera serena e spensierata, senza dovermi preoccupare di nessun altra problematica esterna. Ringrazio i miei genitori, mia sorella e tutta la mia famiglia per avermi supportato emotivamente ed economicamente durante l'intero periodo della triennale. Infine voglio fare un ringraziamento a tutti i miei compagni di corso. Ho legato con tante persone durante questi tre anni, non mi aspettavo di trovare una comunità così bella, mi sono sentito parte di un gruppo e sono felice di non essere rimasto lì da solo nel mezzo dei banchi di una grande aula universitaria. Spero di rivedervi alla magistrale. Grazie ancora a tutti. Un saluto da Daniele.

Questa tesi ha contribuito a piantare un albero in Camerun tramite il progetto Treedom.

https://www.treedom.net/it/user/sesalab/event/sesa-random-forest