



UNIVERSITÀ DEGLI STUDI DI SALERNO

Dipartimento di Informatica

Corso di Laurea Magistrale in Informatica

TESI DI LAUREA

Prestazioni a confronto per Large Language Models applicati al Security Mutation Testing

RELATORE

Prof. Fabio Palomba

Dott.ssa Valeria Pontillo

Università degli Studi di Salerno

CANDIDATO

Luca Contrasto

Matricola: 0522501512

Anno Accademico 2023-2024

Questa tesi è stata realizzata nel

sesa^{lab}
SOFTWARE ENGINEERING
SALERNO

Dedica o citazione

Abstract

La crescente complessità dei sistemi software e l'aumento delle minacce alla sicurezza rendono fondamentale sviluppare tecniche di testing più efficaci e scalabili. In questo contesto, la tesi esplora l'applicazione dei Large Language Models nel campo del Security Mutation Testing, una metodologia avanzata che valuta la qualità delle suite di test introducendo intenzionalmente errori (mutanti) nel codice. L'obiettivo è verificare se i casi di test riescono a identificarli, migliorandone così l'efficacia.

La ricerca si concentra su un confronto sperimentale tra diversi modelli di linguaggio, tra cui Gemini e ChatGPT, utilizzati per generare mutanti su progetti Java vulnerabili presenti nel dataset VUL4J. Vengono analizzate le capacità dei modelli di generare mutanti rilevanti e diversificati, la loro influenza sull'efficacia delle suite di test e il costo computazionale associato. Particolare attenzione è dedicata all'ottimizzazione del processo attraverso tecniche di prompt engineering e preprocessing, che rendono possibile la sperimentazione in modo automatico.

I risultati mostrano che gli LLM, grazie alle loro capacità di comprensione semantica, possono produrre mutanti di qualità superiore rispetto agli approcci tradizionali, riducendo il costo computazionale e ampliando il ventaglio di difetti simulati. Tuttavia, emergono anche limiti significativi, come la difficoltà di elaborare codice particolarmente complesso e il consumo elevato di risorse computazionali. La tesi conclude con una riflessione sui vantaggi, i limiti e le potenzialità future di questa tecnologia nel testing del software.

Indice

Elenco delle Figure	iii
Elenco delle Tabelle	v
1 Introduzione	1
1.1 Contesto Applicativo	1
1.2 Motivazioni e Obiettivi	5
1.3 Risultati Ottenuti	6
1.4 Struttura della Tesi	6
2 Background e Stato dell'Arte	8
2.1 Testing del Software	8
2.1.1 Verifica Statica e Dinamica del Software	9
2.1.2 Testing e Debugging	9
2.1.3 Terminologia e Problemi Indecidibili del Testing	10
2.1.4 Tecniche di Testing	11
2.1.5 Black Box Testing	13
2.1.6 White Box Testing	15
2.2 Mutation Testing	17
2.2.1 Mutation Testing Process	18
2.2.2 Problemi del Mutation Testing	19

2.2.3	Tecniche di riduzione del costo del Mutation Testing	20
2.3	Large Language Models	24
2.3.1	Architettura Transformer e fondamenti teorici	25
2.3.2	Applicazioni, Sfide ed Etica degli LLM	29
2.4	Applicazioni degli LLM al Testing del Software	31
2.4.1	Applicazioni degli LLM nel Processo di Testing	32
2.4.2	LLM e Mutation Testing	34
2.4.3	Vantaggi e Limiti degli LLM	36
2.4.4	Motivazioni e Obiettivi	36
3	Sperimentazione e impostazione del progetto	38
3.1	Research Questions	38
3.2	Struttura dell'implementazione	40
3.2.1	VUL4J	41
3.2.2	Gemini	42
3.2.3	ChatGPT	43
3.2.4	μ BERT	45
3.3	Processo sperimentale	46
3.3.1	Inizializzazione	47
3.3.2	Preparazione dei test	47
3.3.3	Preprocessing dell'input	47
3.3.4	Generazione dei mutanti e postprocessing	48
3.3.5	Riesecuzione del test e analisi	49
4	Risultati ottenuti	51
4.1	RQ1: Corrispondenza tra Vulnerabilità in Vuln4J e Vulnerabilità Generate	51
4.2	RQ2: Prompt Engineering	54
4.3	RQ3: Confronto tra modelli	56
5	Conclusioni	59
	Bibliografia	61

Elenco delle figure

2.1	Tipologie di Software Testing.	11
2.2	Tecniche di Software Testing sistematico.	15
2.3	Processo di Mutation Testing	17
2.4	Processo di generazione del testo dei modelli MLM.	27
2.5	Processo di generazione del testo dei modelli ALM.	28
3.1	Estratto del progetto che mostra il prompt 1 contenente le informazioni necessarie alla produzione dei mutanti.	40
3.2	Estratto del progetto che mostra il prompt 2 contenente le informazioni necessarie alla produzione dei mutanti più uno stimolo emotivo per il modello.	40
3.3	Vulnerabilità CVE-2018-17201 (Fig. 1a) che causa "Infinite Loop" il quale permette ulteriormente Denialof-Service (DoS), l'attacco è fixato con l'eccezione condizionale utilizzando la condizione "if" (Fig. 1b). Il mutante (Fig. 1c) modifica la condizione "if" che annulla la correzione e si lega fortemente alla vulnerabilità [1].	49

3.4	Vulnerabilità CVE-2018-1000850 che causa "Path Traversal", che consente quindi l'accesso a una directory riservata (Fig. 2a), è corretta con l'eccezione condizionale nel caso '.' o '..' appare nella variabile newRelativeUrl" (Fig. 2b). Il mutante fortemente accoppiato (fig. 2c) cambia il "newRelativeUrl" passato come argomento in "name", che annulla la correzione e reintroduce il comportamento vulnerabile[1].	50
4.1	Diagramma di Venn contenente le classi di accoppiamento dei test della sperimentazione condotta con Gemini sul prompt 1.	52
4.2	Diagramma di Venn contenente le classi di accoppiamento dei test dell'esperimento con Gemini e prompt 2.	55
4.3	Diagramma di Venn contenente le classi di accoppiamento dei test della sperimentazione condotta da Garg et. al [1].	56
4.4	Diagramma di Venn contenente le classi di accoppiamento dei test della sperimentazione condotta con ChatGPT sul prompt2.	58

Elenco delle tabelle

4.1	Risultati ottenuti dalla valutazione dei mutanti generati mostrando la distribuzione nelle classi di accoppiamento.	54
-----	--	----

CAPITOLO 1

Introduzione

1.1 Contesto Applicativo

Da diversi decenni ormai il software e l'informatica hanno avuto un'evoluzione esponenziale della complessità e delle funzionalità offerte. La diffusione di tecnologie come il cloud computing, l'Internet of Things e l'intelligenza artificiale ha reso il software un elemento fondamentale in quasi ogni ambito della società, dalla sanità alla finanza, dall'intrattenimento alla sicurezza nazionale. Al contempo è emersa la necessità di garantire che i sistemi operino in modo sicuro e affidabile per non incorrere in gravi problemi causati da vulnerabilità o difetti del software. A tal fine il software testing risulta essenziale e necessario per la produzione di software sicuri e di qualità, infatti sono state sviluppate diverse tecniche di testing che, oltre a rilevare eventuali bug e vulnerabilità, possono effettuare test più specifici come stress testing e delle prestazioni che vanno a coprire quindi i requisiti non funzionali del sistema. Come lo sviluppo software, nel corso della storia, il software testing ha subito una forte evoluzione. Inizialmente infatti il esso è nato con lo sviluppo della programmazione stessa agli albori dell'informatica. Questa fase pionieristica si concentrava principalmente sull'esecuzione di prove per assicurarsi che il codice funzionasse come previsto, ma mancava una formalizzazione e una struttura teorica. A metà

degli anni '70, figure come Glenford Myers iniziarono a formalizzare le pratiche di testing, egli con il suo libro "The Art of Software Testing" [2], identificava principi fondamentali del testing e introduceva concetti di testing sistematico. Questo decennio è segnato dalla nascita di tecniche specifiche come il path testing e il testing delle condizioni. A causa dell'aumento della complessità del software, si avvertì la necessità di automatizzare le pratiche di testing, vennero sviluppati quindi il testing unitario, focalizzato sulla verifica di singole componenti del codice e spesso implementato dai programmatori stessi, e il testing di classi e metodi con il diffondersi della programmazione orientata agli oggetti. Successivamente sono stati poi sviluppati il testing di integrazione, volto a garantire che i moduli funzionassero correttamente insieme, e del testing di sistema che mira a verificare la qualità e l'interazione delle componenti. L'inizio del nuovo millennio segnò un cambiamento radicale con l'avvento di metodologie agili e pratiche come il Test-Driven Development (TDD). Questo approccio poneva il testing come primo passo nello sviluppo, con la scrittura dei test case prima del codice. Inoltre in questo periodo iniziarono a svilupparsi e diffondersi tecniche come il DevOps e il Continuous Integration/Continuous Deployment (CI/CD), oltre all'introduzione del testing non funzionale, che includeva il testing delle prestazioni, per valutare la velocità e la scalabilità, e il testing della sicurezza, fondamentale per proteggere i dati sensibili degli utenti.

Negli anni recenti, tecnologie avanzate come il machine learning hanno iniziato a influenzare il software testing, migliorando la capacità di analizzare i risultati dei test e di individuare anomalie. Da qui nascono numerose sfide e ricerche sull'applicabilità del machine learning al testing che mirano a scoprire fin dove è possibile automatizzare il testing ottenendo risultati buoni in termini di efficienza e qualità del test.

Durante gli anni '70, con lo sviluppo di diverse formalizzazioni del testing e delle prime automatizzazioni, è nato il mutation testing, una tecnica sviluppata per la prima volta nel contesto accademico con l'intento di misurare la qualità dei test case attraverso l'introduzione di modifiche deliberatamente difettose nel codice sorgente. Le prime ricerche esplorarono l'idea che modificando intenzionalmente il codice, come con la sostituzione di operatori o cambiando il valore di variabili, fosse possibile verificare se i test case fossero abbastanza robusti da individuare questi errori

introdotti. Il mutation testing rimase principalmente una pratica teorica e accademica, poiché presentava sfide computazionali significative: ogni mutante richiedeva risorse per essere eseguita e testata, generando un carico elevato in termini di tempo e potenza computazionale. Di conseguenza il mutation testing, quindi, fu considerato per lungo tempo troppo costoso e poco pratico per un utilizzo su larga scala. Tuttavia, questo periodo vide lo sviluppo dei primi strumenti automatizzati e delle tecniche per ridurre il numero di mutanti generati, come il selective mutation testing, che seleziona solo mutanti rilevanti. Con l'avanzamento della potenza computazionale e lo sviluppo di nuovi algoritmi il mutation testing cominciò ad essere considerato più praticabile anche in contesti non accademici, trovando applicazioni pratiche specialmente nei settori dove la qualità del software era cruciale. La ricerca in questo ambito avanzava portando alla luce diverse tecniche di riduzione dei costi come il weak mutation testing e il mutation clustering. Negli ultimi anni, con l'avvento dell'intelligenza artificiale e dell'automazione avanzata, il mutation testing ha subito un'importante evoluzione. L'uso di algoritmi di machine learning ha permesso di migliorare la selezione e la generazione dei mutanti, rendendo il processo più efficiente e riducendo il costo computazionale. Oggi, grazie a strumenti come PITest per Java, il mutation testing può essere integrato in pipeline CI/CD, permettendo alle aziende di valutare l'efficacia dei propri test case in modo continuo.

Una delle innovazioni più recenti è l'uso dei Large Language Models, che si sono dimostrati capaci di generare varianti di codice in modo intelligente, e quindi di fungere da generatori avanzati di mutanti. Questa applicazione rappresenta una frontiera nuova per il mutation testing, in quanto consente di esplorare un ventaglio di mutanti più ampio e realistico, potenzialmente più rappresentativo degli errori che si potrebbero verificare nel codice reale.

I Large Language Models hanno rivoluzionato il campo dell'intelligenza artificiale e dell'elaborazione del linguaggio naturale. Grazie alla capacità di generare e comprendere testi con livelli di accuratezza e coerenza senza precedenti, questi modelli sono divenuti strumenti fondamentali per una vasta gamma di applicazioni, dall'assistenza virtuale alla traduzione automatica, dalla generazione di codice fino al supporto alla scrittura creativa. Gli LLM come GPT-3 e GPT-4 di OpenAI, BERT e T5 di Google, e LLaMA di Meta, si distinguono per le dimensioni e la complessità, avendo miliardi

di parametri che consentono loro di catturare sfumature linguistiche complesse e adattarsi a molteplici contesti applicativi. Gli LLM infatti vengono addestrati su enormi dataset, che includono testi di diverse tipologie come articoli, libri, forum e pagine web. La vastità e la varietà dei dati consentono ai modelli di apprendere una vasta gamma di conoscenze e contesti, che possono poi essere applicati durante l'uso per generare testi o rispondere a domande. Per rendere i modelli utili in contesti specifici, si applicano tecniche di prompt engineering e fine-tuning. Con il prompt engineering, gli utenti o gli sviluppatori possono fornire input mirati per ottenere risposte più accurate. Il fine-tuning, invece, prevede l'addestramento del modello su dataset specializzati per migliorarne le prestazioni su compiti specifici. L'uso di LLM nel mutation testing presenta diversi vantaggi:

- efficienza nella generazione dei mutanti, gli LLM possono generare mutanti a una velocità molto elevata, rendendo possibile la creazione di un ampio numero di varianti del codice per testare più a fondo le suite di test;
- varietà dei mutanti, poiché gli LLM possono essere istruiti a generare modifiche semantiche del codice, possono creare mutanti con difetti realistici e non banali, simulando meglio gli errori umani e aumentando il valore del mutation testing;
- riduzione del costo computazionale, rispetto ai metodi tradizionali di mutation testing che possono richiedere un numero enorme di mutanti e causare un elevato costo computazionale, l'uso di LLM permette di generare mutanti con un livello di intelligenza maggiore e quindi, in molti casi, di selezionare solo i mutanti più significativi.

Tuttavia, l'impiego degli LLM nel mutation testing presenta anche sfide che riguardano principalmente il bias nei modelli, a causa del fatto che essi non risposte propriamente ragionate il che potrebbe portare alla generazione di mutanti non rappresentativi. Un'altra difficoltà da tenere in considerazione è la comprensione del codice complesso poiché gli LLM sono molto potenti nel generare codice, ma possono incontrare difficoltà con il codice particolarmente complesso o specifico di un dominio, portando a mutanti che potrebbero non essere utili per il testing. Infine l'addestramento e il deployment degli LLM richiedono risorse considerevoli e

l'applicazione di questi modelli può risultare onerosa, specialmente in ambienti con risorse limitate.

1.2 Motivazioni e Obiettivi

Questo lavoro di tesi avrà lo scopo di studiare e condurre una ricerca sull'applicazione degli LLM nella generazione di mutanti efficaci utili al security testing. La ricerca sarà condotta utilizzando il dataset VUL4J [3] che contiene una serie di progetti JAVA vulnerabili insieme al codice fixato delle vulnerabilità e ai test relativi al codice in analisi. Dovranno quindi essere selezionati i progetti utilizzabili per far generare una serie di mutanti ai modelli Gemini e ChatGPT che sono stati selezionati in base a determinati requisiti. Il progetto sarà sviluppato utilizzando due prompt per facilitare la produzione dei mutanti, fornendo il contesto adeguato e consentendo l'analisi e classificazione del mutante. Molte scelte progettuali seguiranno e applicheranno le scoperte relative agli studi effettuati in ambito Software Testing con l'obiettivo di confrontare Gemini e CHatGPT tra loro e con altri LLM testati in letteratura.

A tal proposito infatti, è necessario considerare che quest'ambito di applicazione degli LLM al Software Testing è ancora poco studiato. I lavori presenti in letteratura in questo campo riguardano principalmente la generazione di mutanti utilizzando gli LLM per identificare difetti nel codice. Una specializzazione di questa pratica è quella di identificare le vulnerabilità del codice poiché anch'esse sono da considerarsi difetti, ma oltre a generare comportamenti inattesi possono causare problemi più gravi che vanno dal furto di informazioni ad attacchi DoS o attacchi più complessi; quello che tutti gli attacchi hanno in comune nel contesto applicativo è la presenza di difetti nel software e più precisamente di vulnerabilità. Uno dei pochi studi, se non l'unico allo stato dell'arte, che affronta il problema del Security Mutation Testing tramite l'utilizzo di LLM risulta essere quello di Garg et al. [1]. In esso viene utilizzato un tool chiamato μ BERT che tramite il Large Language Model CodeBERT maschera e sostituisce i token avendo in input la sola classe Java da mutare. Il preprocessing necessario affinché il modello possa comprendere bene cosa, dove e come applicare

la mutazione è già implementato dal tool stesso. Per utilizzare quindi altri modelli, quali Gemini e ChatGPT, sarà necessario quindi applicare il preprocessing necessario per far generare i mutanti ai modelli, avendo come input solo i progetti vulnerabili contenuti nel dataset VUL4J. Questo studio permetterà, oltre a vedere come si comportano questi modelli nella produzione di mutanti, di confrontare LLM di tipo Masked Language Modeling (CodeBERT) con modelli di tipo Autoregressive Language Modeling (Gemini e ChatGPT); inoltre sarà condotta una fase di prompt engineering per vedere la differenza di risposte cambiando solo i prompt sulla base dello stesso modello e degli stessi dati in input.

1.3 Risultati Ottenuti

Dai risultati ottenuti è possibile notare principalmente che i modelli di tipo ALM testati sono leggermente meno performanti rispetto al modello di riferimento di tipo MLM. Infatti, μ BERT produce in totale 39 mutanti utili su 45 progetti testati. Di questi 32 mutanti sono fortemente accoppiati alla vulnerabilità associata, mentre i restanti 7 in modo parziale. Riguardo Gemini esso produce in totale, con il prompt che ha ottenuto performance migliori, 30 mutanti utili di cui 29 sono fortemente accoppiati. ChatGPT infine produce 26 mutanti utili, di questi 21 sono fortemente accoppiati mentre i restanti 5 sono parzialmente accoppiati alle vulnerabilità associate. Nonostante i modelli di tipo ALM siano leggermente meno performanti rispetto a CodeBERT è importante notare che i risultati sono molto simili ma anche che essi dipendono molto anche dalle fasi di preprocessing necessarie per la produzione dei mutanti, per cui questa può essere una delle cause della differenza di performance. Inoltre si potrebbe migliorare ulteriormente il numero di mutanti utili facendoli generare per singolo progetto con un prompt ad hoc per ognuno, ma questo può essere sicuramente applicato in un contesto reale e non di ricerca.

1.4 Struttura della Tesi

Segue la struttura dei capitoli della tesi con una breve descrizione per ognuno. Nel capitolo 2 saranno fornite le conoscenze teoriche necessarie per comprendere

il problema in analisi, gli strumenti utilizzati, oltre ad una descrizione specifica per questi ultimi. Inoltre saranno analizzati gli studi presenti in letteratura riguardo il problema in analisi focalizzandosi sugli eventuali limiti che portano allo studio presente in questa tesi.

Il capitolo 3 riguarda inizialmente la descrizione della sperimentazione con l'obiettivo della tesi e la definizione delle domande di ricerca utili allo studio. Successivamente vengono descritti gli strumenti utilizzati, quali i modelli, il dataset VUL4J ed infine il processo necessario di manipolazione dei progetti affinché possano essere estratti i dati corretti utili alla produzione dei mutanti.

Il capitolo 4 contiene i risultati delle singole research questions mostrati tramite tabelle e grafici oltre alla descrizione associata, vengono quindi illustrate le differenze tra i modelli, i prompt e i risultati ottenuti nei singoli esperimenti.

Infine nel capitolo 5 sono presenti le conclusioni della tesi.

Background e Stato dell'Arte

2.1 Testing del Software

Il software testing è il processo di valutazione di un sistema software o di un'applicazione per verificare che soddisfi i requisiti specificati e che funzioni correttamente nelle condizioni previste. Dare una definizione esaustiva riguardo il software testing è un compito difficile in quanto bisognerebbe considerare diversi aspetti quali: l'ampiezza e la complessità del dominio applicativo, poichè comprende requisiti funzionali e non funzionali per i quali ognuno necessita di valutazioni specifiche, i diversi obiettivi del testing che vanno a definire anche l'approccio da utilizzare, e l'evoluzione continua del software che obbliga la corrispondente evoluzione del testing. Una definizione di Software Testing comunemente accettata in letteratura risulta essere: "Il testing è il processo di esecuzione di un programma con l'intento di trovare errori"[2]. Di conseguenza l'obiettivo del tester sarà quello di stimolare il sistema o l'applicazione tramite gli opportuni input o condizioni ambientali per trovare errori, e far in modo che il sistema o l'applicazione fallisca. Da questa definizione nascono diverse implicazioni come la definizione di test "di successo" e "fallito". In particolare i project manager tendono a definire una "successful test run" quando un test case non trova errori, invece un test case che trova errori è

un test "unsuccessfull". Questa diffusa nomenclatura è infatti concettualmente al contrario in quanto, per la definizione di testing più accettata in letteratura, un test è di successo se trova errori e quindi porta miglioramenti al sistema o all'applicazione. In riferimento sempre alla definizione di testing anche i test "successfull test run" che quindi non trova errori all'interno del software è un test di successo in quanto stimola il test con input che mettono alla prova la solidità del sistema. Gli unici test da considerarsi "unsuccessfull" sono quelli che non vanno a stimolare e verificare il funzionamento del sistema [2].

2.1.1 Verifica Statica e Dinamica del Software

È necessario dare delle definizioni teoriche per comprendere in modo esaustivo l'importanza e l'ampiezza del software testing. In primo luogo il Software testing è una tipologia di testing che si concentra principalmente sul trovare difetti software nel codice, a differenza di altre tipologie di testing, come il testing di sistema, che può considerare anche componenti hardware e quindi valutare altri aspetti del sistema identificabili principalmente nei requisiti non funzionali.

Il software testing si identifica in un tipo di verifica dinamica del software in quanto viene prodotto del codice ad hoc per verificare la correttezza del codice di produzione, con lo scopo di trovare eventuali bug che conducono ad una failure del sistema. Oltre ad una verifica di tipo dinamico quale il software testing che verifica il codice in esecuzione è possibile fare una verifica del codice anche di tipo statico. Essa per definizione è un processo di valutazione di un sistema o di un suo componente basato sulla sua forma, struttura, contenuto e documentazione, la forma di verifica statica più diffusa è l'ispezione del codice [2].

2.1.2 Testing e Debugging

Un'altra differenziazione necessaria nel contesto del Testing del software è quella tra il Software Testing e il Debugging. Mentre lo scopo del software testing è quello di identificare difetti per verificare la qualità del software e che soddisfi i requisiti degli utenti, l'obiettivo del debugging invece è di risolvere problemi specifici che sono

stati identificati durante il testing o in fase di utilizzo dell'applicazione. Le differenze si vedono anche considerando gli strumenti utilizzati in queste fasi poichè per il software testing si utilizzano diverse tecniche quali unit testing, integration testing, system testing e acceptance testing mentre il debugging coinvolge l'uso di debugger, log e analisi del codice per identificare la causa del bug [4].

2.1.3 Terminologia e Problemi Indecidibili del Testing

Concetti fondamentali nell'ambito del software Testing sono le nozioni di fault, error, defect e failure:

- fault è la causa del malfunzionamento presente nel codice;
- error è la differenza tra il comportamento atteso e quello effettivo, risultante da un fault;
- failure è lo stato del sistema in presenza di un error;
- defect è un termine utilizzato per riferirsi sia alla causa (fault) che all'effetto (failure).

Altro concetto fondamentale è l'oracolo, esso rappresenta il comportamento atteso per ogni caso di test e può essere di due tipi: l'oracolo umano che è basato sulle specifiche e sul giudizio di una persona, spesso un tester, e l'oracolo automatico che viene generato dalle specifiche formali definite nelle fasi precedenti di sviluppo.

Nel contesto del software testing ci sono diversi problemi indecidibili, cioè problemi per i quali non esiste un algoritmo generale che possa fornire una soluzione in tutti i casi. Essi derivano principalmente dalla natura computazionale e dalla complessità dei sistemi software, il principale problema indecidibile riguarda la copertura completa del test in quanto la questione di determinare se un insieme di casi di test copre completamente tutti gli stati e le transizioni di un sistema è un problema indecidibile. Ciò vuol dire, in altre parole, che non è possibile garantire che un insieme finito di test possa coprire tutte le possibili interazioni e stati di un programma, specialmente in sistemi complessi o non deterministici [4].

2.1.4 Tecniche di Testing

Esistono diverse tecniche di testing in letteratura, ognuna con un approccio specifico per valutare la qualità del codice e la sua affidabilità. Esse vengono rappresentate in diversi modi a seconda dello scope di riferimento e anche a secondo dall'approccio da utilizzare per il testing. Nella Figura 2.1 viene illustrata la suddivisione più comune riguardo il testing del software in ambito Software Engineering, oltre a queste sono presenti altre forme più specifiche a diversi gradi di utilità.

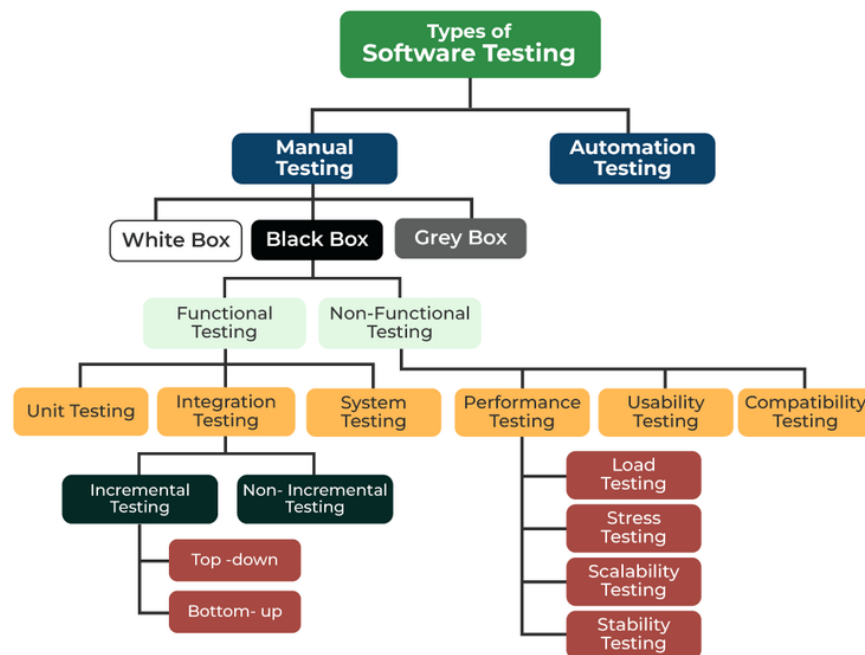


Figura 2.1: Tipologie di Software Testing.

Nel contesto di questa tesi saranno descritte principalmente 3 macroaree del Software Testing, più generali di quelle presenti in figura 2.1, quali testing sistematico, testing statistico e analisi mutazionale.

Testing sistematico

Il testing sistematico è una metodologia di testing del software che segue un approccio strutturato e completo per coprire tutte le funzionalità e i casi d'uso del sistema in modo rigoroso. Questo tipo di testing mira a garantire che tutte le parti del software siano verificate e convalidate, includendo sia gli scenari positivi, cioè quelli in cui il sistema si comporta come previsto, che quelli negativi, ovvero quando si

verificano errori o casi limite. Alcuni degli approcci più comuni nel testing sistematico sono:

- testing basato su specifiche in cui i test vengono progettati sulla base delle specifiche funzionali e non funzionali del sistema;
- testing basato sulla copertura in cui viene verificata la copertura di codice (statement coverage, branch coverage, path coverage, ecc.) per assicurare che tutte le parti del codice siano state testate almeno una volta;
- testing basato sui requisiti in cui si verifica che il software soddisfi tutti i requisiti definiti all'inizio dello sviluppo.

L'obiettivo è ridurre al minimo la possibilità di errori non individuati e garantire che tutte le funzionalità siano correttamente implementate e testate. I vantaggi del testing sistematico sono sicuramente l'elevata accuratezza e copertura dei test, una maggiore sicurezza nel rilevare difetti e malfunzionamenti e la natura della tecnica che garantisce un processo ben strutturato e documentato. Gli svantaggi invece sono che può richiedere molto tempo e risorse e spesso necessita di un grande numero di casi di test.

Testing statistico

Il testing statistico si basa sull'utilizzo di modelli probabilistici o metodi statistici per progettare test o valutare le prestazioni del sistema. L'obiettivo principale è utilizzare la statistica per prevedere l'affidabilità del software e identificare problemi che potrebbero non essere facilmente rilevabili con metodi di testing convenzionali. In questo contesto, i test vengono progettati e selezionati in modo tale da riflettere l'uso realistico del software da parte degli utenti finali o da seguire distribuzioni probabilistiche di input, per questa tecnica esistono due principali approcci descritti di seguito.

- Testing basato su modelli statistici: costruzione di modelli che rappresentano l'uso del sistema e la distribuzione statistica degli input per generare test in modo casuale, ma guidato da probabilità reali.

- Testing basato sull'affidabilità: prevede l'esecuzione del software in diverse condizioni con lo scopo di stimare la sua affidabilità complessiva attraverso metodi di inferenza statistica.

Questa tecnica di testing è particolarmente utile quando è difficile generare un set esaustivo di test o quando si vuole simulare un utilizzo reale del sistema. Ad esempio, potrebbe essere usato in applicazioni critiche per valutare la probabilità di fallimento durante l'esecuzione reale. Questo infatti è uno dei principali vantaggi di questa tecnica, inoltre essa è anche utile per stimare l'affidabilità e identificare problemi che potrebbero verificarsi in ambienti di produzione. Altro fattore da considerare è la potenziale riduzione del numero di test necessari ottimizzando le risorse. Tra gli svantaggi invece si ha una copertura non sempre completa delle funzionalità e la complessità nella costruzione di modelli probabilistici accurati [4, 2].

Riassumendo le tecniche descritte il testing sistematico mira a coprire il software in maniera esaustiva, seguendo un piano preciso, mentre testing statistico si concentra su input probabilistici per riflettere un uso reale e predire l'affidabilità.

2.1.5 Black Box Testing

Una delle tipologie di testing sistematico si identifica nel Black Box Testing. Esso è una tecnica di testing del software in cui il tester esamina il comportamento esterno di un sistema o applicazione senza avere alcuna conoscenza della struttura interna, del codice sorgente o dei dettagli di implementazione. Si concentra quindi su ciò che il software fa, piuttosto che su come lo fa. Questo approccio è particolarmente utile per verificare che il software soddisfi i requisiti funzionali e per trovare eventuali discrepanze tra il comportamento atteso e quello effettivo. Le caratteristiche di questa tipologia di testing sistematico sono principalmente l'indipendenza dall'implementazione in quanto il tester non ha accesso al codice sorgente, quindi non è influenzato dalle scelte di progettazione o di implementazione. Altra caratteristica principale è il focus sui requisiti funzionali infatti il test viene progettato per verificare che il sistema si comporti in modo conforme alle specifiche funzionali: vengono testati input, output, interfacce utente, e comportamenti di sistema, seguendo scenari d'uso tipici o casi limite. Infine un'altra caratteristica si identifica nel fatto che i test sono condotti

dal punto di vista dell'utente, viene infatti simulata l'interazione con il sistema come farebbe un utente finale, testando la facilità d'uso e il rispetto dei requisiti.

Ci sono diverse tecniche usate per il black box testing, che aiutano a definire i casi di test basati su input ed output [4].

- **Equivalence Partitioning (Partizionamento in Classi di Equivalenza):** questa tecnica divide l'input di un sistema in insiemi o classi, con l'idea che tutti gli input di una determinata classe siano trattati in modo simile dal sistema. Si eseguono i test su un singolo input per ciascuna classe, riducendo così il numero di casi di test senza perdere copertura [5].
- **Boundary Value Analysis (Analisi dei Valori ai Limiti):** si concentra sui valori di input che si trovano ai confini di una classe di equivalenza, poiché i bug spesso si manifestano in prossimità di questi limiti.
- **Decision Table Testing (Test basato su tabelle di decisione):** si utilizzano tabelle per rappresentare combinazioni di input e le relative azioni che il sistema deve eseguire. Ogni riga della tabella rappresenta una regola che associa un insieme di condizioni con l'azione attesa.
- **State Transition Testing (Test delle Transizioni di Stato):** questa tecnica è utilizzata per sistemi che cambiano stato in base a input specifici. Il test verifica che il sistema transiti correttamente da uno stato all'altro in base agli input.

I vantaggi del Black Box Testing rispetto ad altre tecniche come il White Box Testing risiedono essenzialmente nelle 3 caratteristiche principali poichè consente di non avere conoscenza sul codice sorgente, testa il codice dal punto di vista dell'utente e identifica quindi problemi di interfaccia. In Figura 2.2 si può infatti notare il diverso grado di conoscenza in base alla tipologia di testing sistematica di riferimento. Gli svantaggi di questa tecnica di testing invece riguardano principalmente la difficoltà di identificare le cause del bug proprio a causa del fatto che il tester non vede il codice sorgente. Inoltre questo è anche il motivo per cui si ha una limitata copertura del codice poiché non è garantito che vengano coperte tutte parti del codice o stabilire una percentuale di copertura.

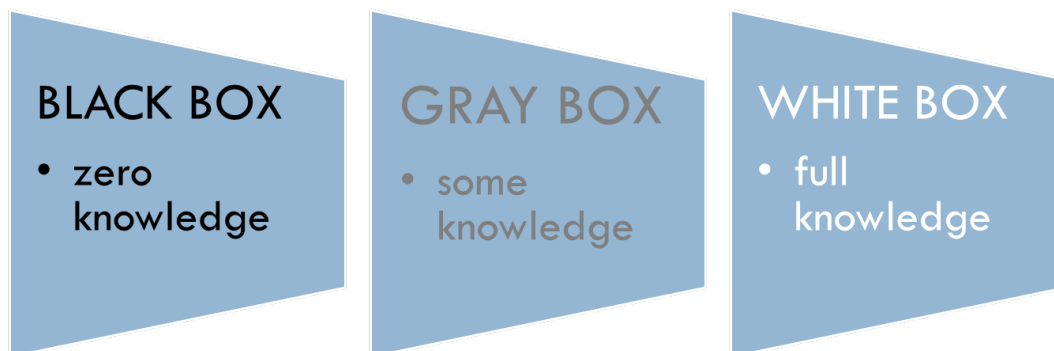


Figura 2.2: Tecniche di Software Testing sistematico.

2.1.6 White Box Testing

Il White Box Testing è una tecnica di testing del software in cui il tester ha accesso alla struttura interna del codice e lo analizza per verificarne il corretto funzionamento. A differenza del Black Box Testing, che si focalizza sul comportamento esterno del software, il white box testing è orientato all'analisi dei dettagli interni come logica, flussi di controllo, cicli, condizioni e percorsi di esecuzione. Esso è caratterizzato dalla visibilità del codice che consente al tester di provare e garantire potenzialmente tutte le logiche del programma. Il White Box Testing infatti, oltre a testare gli output si concentra sull'analisi dei percorsi interni o di altre strutture per garantire determinati vincoli sul criterio di copertura scelto. Esistono infatti diversi criteri di copertura del codice [6]:

- Statement Coverage misura la percentuale di istruzioni eseguite almeno una volta durante il test, l'obiettivo è verificare che tutte le istruzioni nel codice siano state eseguite almeno una volta;
- Branch Coverage si concentra su istruzioni condizionali come if o switch, assicurando che ogni ramo venga eseguito almeno una volta, l'obiettivo infatti è di garantire che ogni possibile risultato di una condizione sia stato testato;
- Condition Coverage verifica che ogni condizione booleana all'interno di un'espressione condizionale (come un if con più condizioni) sia stata verificata con esito sia vero che falso, l'obiettivo è infatti quello di garantire che ogni condizione all'interno di un'espressione sia stata testata sia per il valore vero che per il valore falso;

- Decision Coverage è molto simile alla branch coverage, ma estende il concetto includendo qualsiasi tipo di decisione logica che può influenzare il flusso del programma. Questa include decisioni a livello di istruzioni condizionali (if, switch), ma anche cicli (for, while) e vuole garantire che tutte le decisioni che influenzano il flusso del programma siano testate;
- Multiple Condition Coverage in cui la copertura delle condizioni multiple verifica tutte le possibili combinazioni di condizioni booleane in una singola istruzione condizionale;
- Path Coverage si focalizza sulla verifica di ogni possibile percorso logico attraverso il programma. È una delle metriche più complete, ma anche una delle più complesse da implementare;
- Loop Coverage è una tecnica specifica per verificare il comportamento dei cicli nel codice, come for, while, e do-while. Questo tipo di copertura si concentra sulle esecuzioni dei cicli con zero iterazioni, una singola iterazione, e più iterazioni;
- altri tipi di coverage più specifici e utilizzati in ambiti particolari.

Il White Box Testing risulta essere il metodo più utilizzato ed accettato per la verifica del codice in quanto permette potenzialmente di verificare ogni ramo, condizione e percorso nel codice, riducendo la probabilità di bug nascosti. Esso inoltre ha il vantaggio di poter rilevare bug logici profondi o problemi relativi a condizioni specifiche che non si manifestano facilmente con test a livello più alto, ed è possibile inoltre identificare sezioni di codice inefficiente o inutilizzato, migliorando le prestazioni complessive del software. Di contro gli svantaggi nell'uso del White Box testing sono il costo in termini di tempo poichè richiede un'analisi dettagliata del codice sorgente e la progettazione di casi di test specifici, che per sistemi complessi può diventare un compito complesso.

2.2 Mutation Testing

L'analisi mutazionale (o mutation testing) è una tecnica avanzata di testing del software che valuta la qualità di una suite di test attraverso la creazione di mutanti del programma originale. I mutanti sono versioni leggermente modificate del programma, che introducono piccoli errori o cambiamenti nel codice. L'obiettivo è verificare se la suite di test esistente riesce a identificare questi errori artificiali, ovvero se "uccide" i mutanti. La motivazione per cui vengono introdotti i mutanti è principalmente quella di simulare dei guasti nel software rispetto alla versione corretta del programma, in questo modo si riesce a simulare e testare una buona percentuale di difetti software. Dalla Figura 2.3 possiamo osservare un tipico processo di mutation testing, che sarà descritto più approfonditamente in seguito, nella quale è visibile la fase in cui, grazie ai mutanti, vengono apportate modifiche o aggiunti test, e quindi migliorati.

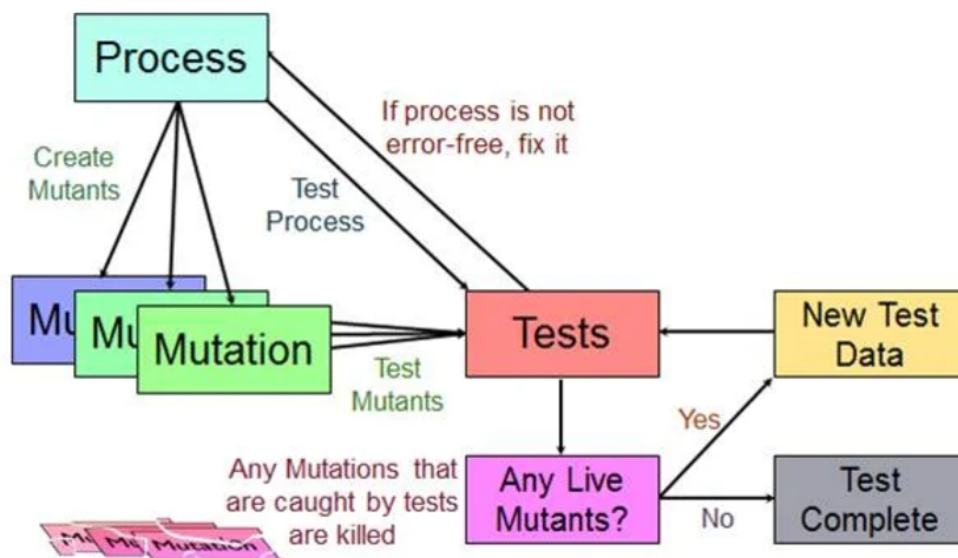


Figura 2.3: Processo di Mutation Testing

Dalla

Gli aspetti teorici del mutation testing sono basati principalmente su due concetti: la Competent Programmer Hypothesis e il Coupling Effect. La prima teoria afferma che i programmatori sono competenti, il che implica che essi tendono a sviluppare programmi vicini alla versione corretta. Di conseguenza, anche se ci possono essere

errori supponiamo che questi difetti siano semplici e che possono essere corretti con qualche piccola modifica di sintassi. Pertanto, nel test di mutazione, vengono applicate diverse semplici modifiche sintattiche per rappresentare i difetti introdotti da "programmatori competenti". [7]

L'effetto di accoppiamento, a differenza della prima ipotesi, riguarda il tipo di guasti utilizzato nell'analisi delle mutazioni. La definizione alla base di questa teoria è che "mutanti complessi sono accoppiati a semplici mutanti in modo tale che un set di dati di test che rileva tutti i mutanti semplici in un programma rileverà anche una grande percentuale dei mutanti complessi". In questa definizione un difetto semplice è rappresentato da un mutante semplice che viene creato effettuando un singolo cambiamento sintattico, mentre un difetto complesso è rappresentato come mutante complesso che è creato effettuando più di un cambiamento. La conseguenza di questa ipotesi è quindi l'utilizzo solo di mutanti semplici nel processo di mutation testing [8].

2.2.1 Mutation Testing Process

A tal proposito, si osserva come un tipico processo di mutation testing viene eseguito: è necessario avere una suite di test corretta per il programma in analisi, che prima di iniziare l'analisi della mutazione, deve essere eseguita con successo contro il programma originale per verificare la correttezza per il caso di prova. Se il programma è errato, deve essere corretto prima di eseguire altri mutanti; altrimenti ogni mutante sarà poi eseguito contro questo set di test errato. Un mutante viene "ucciso" se almeno un test fallisce in presenza della mutazione. Se tutti i test passano anche con la mutazione, significa che la mutazione non è stata rilevata, suggerendo una debolezza nei casi di test (mutante sopravvissuto) [8]. Tuttavia, ci sono alcuni mutanti che non possono mai essere uccisi perché producono sempre lo stesso output del programma originale, questi vengono chiamati Mutanti Equivalenti. Essi sono sintatticamente diversi ma funzionalmente equivalenti al programma originale e la rilevazione automatica di tutti i mutanti equivalenti è impossibile poiché l'equivalenza del programma è un problema indecidibile, questo infatti risulta essere uno dei problemi per cui il mutation testing non è ampiamente utilizzato.

Il processo di mutation testing si conclude con un punteggio di adeguatezza, noto come Mutation Score, che indica la qualità del set di test in ingresso. Il punteggio di mutazione (MS) è il rapporto tra il numero di mutanti uccisi e il numero totale di mutanti non equivalenti. L'obiettivo dell'analisi delle mutazioni è di aumentare il punteggio della mutazione a 1, indicando che la suite di test è sufficiente per rilevare tutti i difetti indicati dai mutanti.

2.2.2 Problemi del Mutation Testing

Sebbene il test delle mutazioni sia in grado di valutare efficacemente la qualità di un set di prove, esso presenta ancora una serie di problemi. Un problema che impedisce al test di mutazione di diventare una tecnica pratica di prova è l'alto costo computazionale di eseguire l'enorme numero di mutanti contro un set di test. Gli altri problemi sono legati alla quantità di sforzo umano necessario per utilizzare il test della mutazione, ad esempio il problema dell'oracolo umano e dei mutanti equivalenti.

Il problema dell'oracolo umano si riferisce al processo di controllo del risultato del programma originale con ogni caso di prova. In senso stretto, questo non è un problema unico per il test di mutazione poiché in tutte le forme di prova, una volta raggiunto un insieme di input, resta il problema del controllo dei risultati. Tuttavia, il test di mutazione è efficace proprio perché è impegnativo in termini di risorse e questo può portare ad un aumento del numero di casi di test, aumentando così il costo degli oracoli. Questo costo di oracolo è spesso la parte più costosa dell'attività globale di prova. Inoltre, a causa dell'indecidibilità dei mutanti equivalenti, la loro rilevazione richiede in genere uno sforzo umano supplementare.

Sebbene sia impossibile risolvere completamente questi problemi, con gli attuali progressi nel campo del mutation testing, il processo può essere automatizzato e l'esecuzione può consentire una scalabilità ragionevole. Saranno analizzate anche le tecniche di riduzione dei costi del mutation testing usate per garantire test efficaci al costo computazionale minore possibile [7].

2.2.3 Tecniche di riduzione del costo del Mutation Testing

Il Mutation Testing è ampiamente considerato una tecnica di test computazionalmente costosa. Tuttavia, questa convinzione si basa in parte sull'assunto ormai superato che tutti i mutanti devono essere presi in considerazione. Per trasformare il test di mutazione in una tecnica di test pratica, sono state proposte numerose tecniche di riduzione dei costi. Queste tecniche sono classificate principalmente in due tipi: riduzione dei mutanti generati e riduzione del costo di esecuzione.

Tecniche di riduzione dei mutanti generati

Una delle principali cause dell'aumento del costo computazionale nel Mutation Testing è rappresentata dal costo di esecuzione del gran numero di mutanti generati contro la suite di test. Di conseguenza, ridurre il numero dei mutanti generati senza perdere l'efficacia del test è diventato un problema di ricerca popolare. Il problema, in termini più formali, diventa quello di trovare un sottoinsieme dei mutanti generati contro una suite di test il cui punteggio risulta essere invariato o di poco inferiore. Le principali tecniche di riduzione sono: Mutant Sampling, Mutant Clustering, Selective Mutation, and Higher Order Mutation.

- il Mutant Sampling è un approccio concettualmente semplice in quanto viene selezionato casualmente un sottoinsieme di mutanti a partire dall'intero insieme di mutanti generati. Uno studio effettuato da Wong et al. ha dimostrato inoltre selezionando una percentuale di mutanti dal 10 al 40 per cento è solo il 16% meno efficace di un set completo di mutanti in termini di punteggio di mutazione [9].
- Invece di selezionare i mutanti in modo casuale il Mutant Clustering sceglie un sottoinsieme di mutanti utilizzando appunto algoritmi di clustering. Il processo inizia dalla generazione di tutti i mutanti del primo ordine, successivamente un algoritmo di clustering viene poi applicato per classificare i mutanti del primo ordine in cluster diversi sulla base dei casi di test eliminabili. Da ciascun gruppo viene selezionato un numero ridotto di mutanti per essere utilizzato nel test delle mutazioni; i mutanti restanti vengono scartati. Nell'esperimento

di Hussain [10] sono stati applicati due algoritmi di clustering, K-means e Agglomerative clustering, il risultato è stato poi confrontato con strategie di selezione casuale e greedy. Questi risultati empirici suggeriscono che il Mutant Clustering è in grado di selezionare meno mutanti ma mantenere comunque il punteggio di mutazione quindi risulta essere una tecnica valida come il mutant sampling ma che non rischia di escludere mutanti significativi poichè considera mutanti per ogni classe identificata.

- l'idea alla base del Selective Mutation è che una riduzione del numero di mutanti può essere ottenuta anche riducendo il numero degli operatori di mutazione applicati, quindi si cerca di trovare un insieme di operatori di mutazione che generino un sottoinsieme di tutti i possibili mutanti senza una significativa perdita di efficacia del test, infatti alcuni operatori di mutazione generano molti più mutanti di altri, molti dei quali possono risultare ridondanti. Sin dal lavoro di Offutt et al. [11] dove veniva utilizzata la 2-selective Mutation, ovvero l'eliminazione degli operatori di mutazione ASR (Assignment Operator Replacement) e SVR (Scalar Variable Replacement), sono stati provati diversi approcci per la riduzione dei mutanti sempre in base alla selezione degli operatori di mutazione. Recenti studi hanno invece formulato il problema della mutazione selettiva come un problema statistico: il problema della selezione o riduzione delle variabili. Sono stati applicati approcci statistici lineari per identificare un sottoinsieme di 28 operatori di mutazione da un totale di 108 operatori di mutazione. I risultati hanno suggerito che questi 28 operatori sono sufficienti per prevedere l'efficacia di una suite di test e ha ridotto il 92% di tutti i mutanti generati. Secondo i risultati ottenuti, questo approccio ha raggiunto il più alto tasso di riduzione rispetto ad altri approcci.
- La Higher Order Mutation è una forma relativamente nuova di test delle mutazioni che ha come obiettivo quello di trovare quei rari ma preziosi mutanti di ordine superiore che denotano difetti sottili. Nei tradizionali test di mutazione, i mutanti possono essere classificati in mutanti del primo ordine e mutanti di ordine superiore. I primi vengono creati applicando un operatore di mutazione una sola volta, mentre i secondi sono generati applicando gli operatori di mu-

tazione più di una volta. Sono stati proposti diversi algoritmi per combinare mutanti del primo ordine per generare quelli del secondo ordine, e dai risultati empirici ottenuti si evidenzia che l'applicazione di mutanti del secondo ordine ha ridotto il costo di test di circa il 50% senza grande perdita di efficacia della prova.

Tecniche di riduzione del costo di esecuzione

Oltre a ridurre il numero di mutanti generati, il costo computazionale può anche essere ridotto ottimizzando il processo di esecuzione per singolo mutante. Le tecniche di riduzione del costo sono classificate in: Strong, Weak, and Firm Mutation, Runtime Optimization Techniques e Advanced Platforms Support for Mutation Testing.

- In base al modo in cui viene analizzato e deciso se un mutante viene ucciso durante il processo di esecuzione, le tecniche di test delle mutazioni possono essere classificate in tre tipi: Strong, Weak, and Firm Mutation. La Strong Mutation è spesso associata al tradizionale Test di mutazione cioè che per un dato programma p , si dice che un m mutante del programma p sia ucciso solo se il m mutante dà un risultato diverso dal programma originale p . Per ottimizzare l'esecuzione della mutazione forte è stata proposta la mutazione debole. Nella Weak Mutation si presume che un programma p sia costruito da un insieme M di componenti, supponendo che il mutante m sia fatto cambiando una componente cm ; esso si dice essere ucciso se qualsiasi esecuzione di componente cm è diverso da mutante m . Di conseguenza, nella Weak Mutation, invece di controllare mutanti dopo l'esecuzione del programma intero, i mutanti devono essere controllati solo immediatamente dopo il punto di esecuzione del componente mutante o mutato, questo risulta essere proprio il vantaggio di questa tecnica in quanto si può verificare se il mutante è ucciso e terminare quindi l'esecuzione subito dopo l'esecuzione della componente associata, tralasciando il codice successivo. L'idea della Firm Mutation invece è quella di superare gli svantaggi delle mutazioni sia deboli che forti fornendo un continuum di possibilità intermedie. Cioè, lo "stato di confronto" della mutazione Firm si trova tra gli stati intermedi dopo l'esecuzione (mutazione debole) e l'output

finale (mutazione forte). Purtroppo attualmente non esiste uno strumento di Firm Mutation disponibile al pubblico e quindi analizzabile.

- La tecnica basata sul compilatore è l'approccio più comune per ottenere la mutazione di un programma dove ogni mutante è prima compilato in un programma eseguibile che viene poi eseguito da un numero di casi di prova. Per ottimizzare le prestazioni delle tecniche tradizionali basate sul compilatore è stata proposta la tecnica di integrazione del compilatore. Poiché c'è solo una piccola differenza sintattica tra ogni mutante e il programma originale, compilando ogni mutante separatamente nella tecnica basata sul compilatore si otterrà un costo di compilazione ridondante. Nella tecnica di compilazione integrata, un compilatore modificato è destinato per generare e compilare i mutanti. Quest'ultimo genera due output dal programma originale: un codice oggetto eseguibile per il programma originale e una serie di patch per i mutanti. Ogni patch contiene istruzioni che possono essere applicate per convertire l'immagine di codice oggetto eseguibile originale direttamente in codice eseguibile per un mutante. Di conseguenza, questa tecnica può ridurre efficacemente il costo ridondante dalla compilazione individuale. Il lavoro più recente sulla riduzione del costo di compilazione è la tecnica di traduzione del byte code. Questa tecnica prevede che i mutanti sono generati dal codice oggetto compilato del programma originale, invece che dal codice sorgente. Di conseguenza, i "mutanti bytecode" generati possono essere eseguiti direttamente senza compilazione. Oltre a risparmiare sui costi di compilazione, Bytecode Translation può anche gestire programmi off-the-shelf che non hanno codice sorgente disponibile. Tuttavia, non tutti i linguaggi di programmazione forniscono un modo facile per manipolare il codice oggetto intermedi e non tutti gli operatori di mutazione possono essere rappresentati a livello di byte.
- Il Mutation testing è stato applicato anche a molte architetture avanzate per distribuire il costo complessivo del calcolo tra molti processori. Le prove effettuate sono state: eseguire test di mutazione su un sistema di processori vettoriali, tramite l'esecuzione simultanea di mutanti in macchine SIMD (Single Instruction, Multiple Data), e distribuendo il costo di esecuzione del test della

mutazione attraverso macchine MIMD (Multiple Instruction, Multiple Data) [7].

In conclusione quindi si possono analizzare i diversi vantaggi offerti dell'analisi mutazionale, il primo risulta essere sicuramente il fatto che fornisce un indicatore molto preciso della qualità dei test, inoltre identifica punti deboli della suite di test e promuove la creazione di test più robusti. Lo svantaggio principale invece, che rende questa tecnica non diffusa, risulta essere l'elevato costo computazionale in quanto la generazione di ogni mutante richiede una nuova esecuzione dei test e anche che la generazione di un gran numero di mutanti può essere complessa. L'applicazione di diverse tecniche di riduzione del costo del Mutation Testing, come descritto in precedenza, rende questa tecnica di testing utilizzabile ed efficace in determinati contesti.

2.3 Large Language Models

Storicamente è sempre stato argomento di ricerca consentire alle macchine leggere, scrivere e comunicare come gli esseri umani. In generale la modellazione del linguaggio ha come obiettivo quello di modellare la probabilità generativa di sequenze di parole, in modo da prevedere le probabilità di token futuri o mancanti. La ricerca è condotta su molteplici argomenti che possono essere suddivisi in quattro principali fasi di sviluppo.

- Modelli statistici di linguaggio (SLM) che sono stati sviluppati a partire dagli anni '90 sulla base di metodi statistici di apprendimento. L'idea alla base è quella di predire la prossima parola in base al contesto più recente che è stato fornito, questi sono stati ampiamente applicati per migliorare le prestazioni dei compiti nel recupero delle informazioni ma soffrono di un problema di dimensionalità in quanto è difficile stimare i modelli linguistici di alto ordine in quanto deve essere stimato un numero esponenziale di probabilità di transizione.
- Modelli di linguaggio neurale (NLM) caratterizzati dalla probabilità di sequenze di parole prodotte da reti neurali come multi-layer perceptron (MLP) e reti

neurali ricorrenti (RNNs). Tramite questi modelli viene introdotto il concetto di rappresentazione distribuita delle parole e viene definita la funzione di previsione della parola, condizionata dalle caratteristiche del contesto aggregato. Queste reti tuttavia presentano limiti nel gestire sequenze lunghe, a causa di problemi come il vanishing gradient.

- Modelli linguistici preaddestrati (PLM) caratterizzati principalmente da BERT, un Transformers con un'architettura parallelizzabile e con meccanismi di autoattenzione, che quindi valuta le parole in base alla loro rilevanza nel contesto della frase. BERT è stato proposto per creare modelli di linguaggio bidirezionale preaddestrati appositamente progettati su larga scala senza dati etichettati. Queste rappresentazioni di parole preaddestrate e sensibili al contesto sono molto efficaci come caratteristiche semantiche generiche e hanno alzato notevolmente la barra delle prestazioni del NLP. Seguendo questo paradigma, sono stati sviluppati un gran numero di studi sui PLM, introducendo diverse architetture (ad esempio, GPT-2 e BART) oppure strategie di pre-training migliorate. È importante notare che questo paradigma spesso richiede un fine tuning del PLM per adattarsi a diversi compiti a valle.
- Large Language Models (LLM): i ricercatori scoprono che scalare i PLM (ad esempio, aumentando la dimensione del modello o la dimensione dei dati) spesso porta a una migliore capacità del modello per le attività a valle. Anche se la scalabilità è principalmente condotta in formato modello questi grandi PLM mostrano comportamenti diversi da PLM più piccoli (ad esempio, BERT con parametro 330M e GPT-2 con parametro 1,5B) avendo capacità sorprendenti nel risolvere una serie di compiti complessi[12].

2.3.1 Architettura Transformer e fondamenti teorici

L'architettura Transformer, introdotta da Vaswani et al. [13], ha rivoluzionato il campo dell'elaborazione del linguaggio naturale (NLP). Questa architettura è costruita su una sequenza di blocchi di self-attention che, come introdotto in precedenza, permettono al modello di analizzare e ponderare le relazioni tra le parole di una frase

indipendentemente dalla loro posizione. Il meccanismo di self-attention consente infatti di calcolare pesi (detti "attenzioni") che indicano l'importanza relativa delle parole tra loro, eliminando i limiti dei modelli ricorrenti che soffrivano nel cogliere relazioni di lungo raggio e che non erano facilmente parallelizzabili.

UnTransformer si compone principalmente di due parti: encoder e decoder, organizzati in blocchi ripetuti. Mentre l'encoder cattura rappresentazioni contestuali delle parole di input, il decoder sfrutta tali rappresentazioni per generare previsioni su sequenze di output rendendo questo modello versatile in compiti come la traduzione o la sintesi del testo. Inoltre, gli strati di multi-head attention ampliano la capacità del modello di apprendere diverse sfumature del contesto, suddividendo l'attenzione in più sotto-spazi. Questo processo è ulteriormente migliorato dall'uso di embeddings posizionali, che permettono al modello di mantenere informazioni sull'ordine sequenziale delle parole, nonostante il calcolo in parallelo. Questi aspetti consentono ai Transformer di eccellere in compiti che richiedono una comprensione approfondita del linguaggio, come la traduzione automatica, il riassunto testuale e la generazione di testo. L'efficienza di calcolo resa possibile dalla parallelizzazione ha anche favorito la scalabilità, contribuendo alla diffusione dei Transformer come base dei principali LLM attuali, tra cui BERT, GPT e T5.

Fasi dell'Addestramento

Per raggiungere un livello di comprensione e generalizzazione elevato, gli LLM passano attraverso un processo di addestramento in due fasi, il pre-allenamento e il fine-tuning. Il pre-allenamento avviene su grandi quantità di testo, che possono includere libri, articoli, pagine web e dialoghi. Durante questa fase, il modello apprende le strutture linguistiche e costruisce una rappresentazione delle parole e delle frasi, utilizzando tecniche di apprendimento non supervisionato come il masked language modeling (MLM) e l'autoregressive language modeling (ALM).

Nel masked language modeling, utilizzato da modelli come BERT, alcune parole di una frase vengono mascherate e il modello è addestrato a predirle in base al contesto. Questo approccio permette di migliorare la capacità del modello di comprendere le dipendenze tra le parole, poiché ogni parola viene analizzata in relazione alle altre.

Nella Figura 2.4 è possibile osservare il processo di generazione di testo che viene eseguito da questa tipologia di modelli.

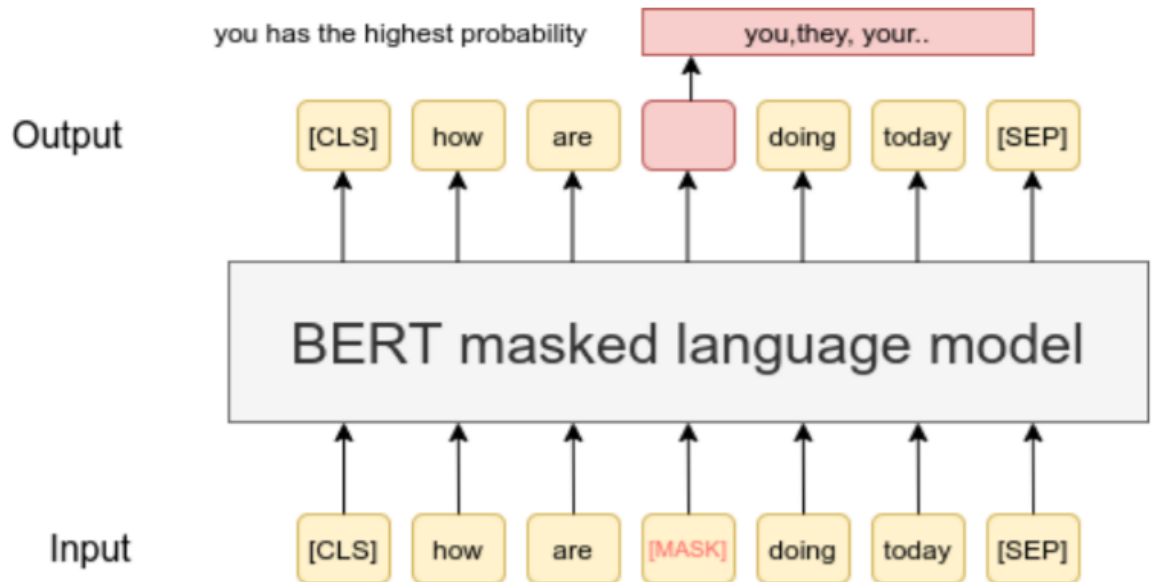


Figura 2.4: Processo di generazione del testo dei modelli MLM.

Nell'autoregressive language modeling, come nel caso di GPT, il modello impara a prevedere la parola successiva in una sequenza, apprendendo progressivamente il contesto e le relazioni sintattiche. La combinazione di queste tecniche permette al modello di sviluppare una comprensione approfondita e sfumata della lingua [14].

Il fine-tuning, invece, consiste nell'adattare il modello pre-allenato a compiti specifici mediante un dataset annotato e più ristretto. Questa fase consente al modello di specializzarsi in compiti mirati, come la classificazione, il riconoscimento delle entità o la risposta a domande. Con l'evoluzione delle tecniche di transfer learning, si è assistito alla nascita di metodi avanzati di few-shot e zero-shot learning, in cui il modello, grazie alla struttura generalizzata appresa nel pre-allenamento, può essere adattato a nuovi compiti con pochi esempi (few-shot) o addirittura senza esempi (zero-shot), sfruttando tecniche di prompt engineering per ottimizzare l'interazione con il modello [15]. La Figura 2.5 mostra il processo di generazione del testo seguito dai modelli ALM, che è un processo più strutturato e complesso rispetto a quelli di tipo MLM.

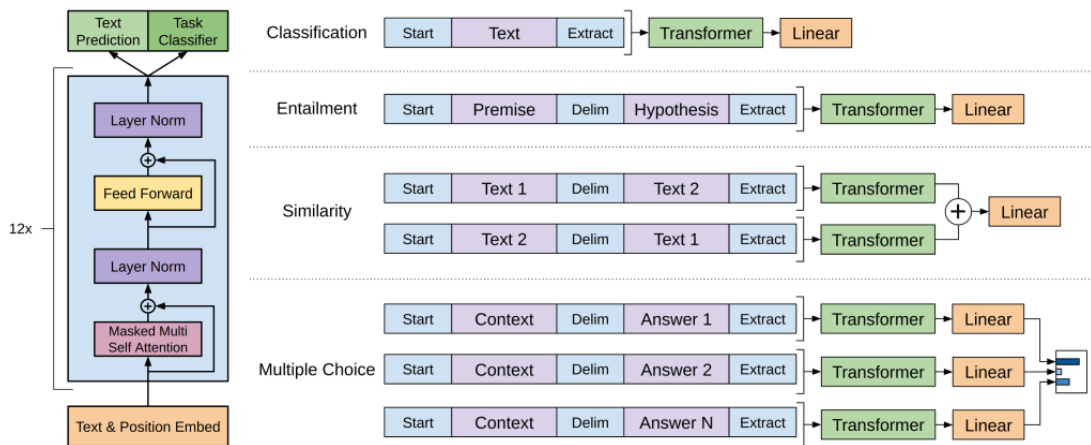


Figura 2.5: Processo di generazione del testo dei modelli ALM.

Scalabilità e gestione delle risorse computazionali

La scalabilità rappresenta una delle sfide maggiori nello sviluppo degli LLM, poiché richiede infrastrutture computazionali avanzate. Gli LLM di ultima generazione possono avere decine o centinaia di miliardi di parametri, richiedendo cluster di GPU o TPU per il loro addestramento. Tecniche come il data parallelism e il model parallelism vengono applicate per dividere i dati e i parametri del modello su più dispositivi, ottimizzando così il processo di addestramento. Ad esempio, nel data parallelism, i dati vengono suddivisi in batch elaborati parallelamente su più GPU, mentre nel model parallelism, il modello stesso viene partizionato per ridurre la domanda di memoria su ogni singolo dispositivo.

Per affrontare le sfide computazionali, si utilizzano anche tecniche di ottimizzazione del modello:

- la distillation, che consiste nel trasferire le conoscenze da un modello di grandi dimensioni a uno più piccolo mantenendo prestazioni comparabili;
- la quantization, che riduce la precisione numerica dei parametri comprimendo il modello e rendendolo più leggero;
- la sparsity, che introduce parametri nulli per semplificare i calcoli, riducendo la complessità del modello e migliorando l'efficienza computazionale.

Tuttavia, l'addestramento e l'esecuzione di questi modelli richiedono ancora risorse significative, portando a costi elevati e a un impatto ambientale notevole[16].

2.3.2 Applicazioni, Sfide ed Etica degli LLM

L'impiego dei Large Language Models (LLM) sta rapidamente crescendo in numerosi settori, grazie alla loro abilità nel comprendere e generare testo in modo fluido e contestuale. Con l'espansione delle capacità dei modelli, sono emerse questioni fondamentali legate alla loro efficacia, alle limitazioni e alle implicazioni etiche del loro utilizzo. L'adozione di LLM su larga scala ha dimostrato di poter supportare e trasformare le modalità di interazione tra le aziende e gli utenti finali, di automatizzare compiti linguistici complessi e di migliorare le operazioni quotidiane in diversi ambiti. Tuttavia, nonostante le loro potenzialità, l'implementazione e l'utilizzo degli LLM comportano anche una serie di sfide tecniche ed etiche.

Applicazioni principali

Gli LLM (Large Language Models) si sono rivelati strumenti straordinariamente versatili, trovando applicazione in una varietà di settori. Nel servizio clienti, per esempio, vengono impiegati per alimentare chatbot e sistemi di risposta automatica, offrendo alle aziende la possibilità di gestire rapidamente e con precisione grandi volumi di richieste. Nel campo della creazione di contenuti, gli LLM supportano la scrittura automatica di articoli, descrizioni di prodotti, riassunti e persino opere creative, come poesie o racconti.

Nel settore sanitario, gli LLM vengono utilizzati per analizzare documenti clinici, come cartelle sanitarie e risultati di laboratorio, contribuendo a migliorare l'efficienza nella diagnosi e nella pianificazione dei trattamenti. In ambito legale, questi modelli possono semplificare l'analisi di contratti complessi, identificare clausole rilevanti e accelerare la ricerca giuridica, riducendo drasticamente i tempi di revisione dei documenti. Infine, nel contesto aziendale, oltre a monitorare la reputazione online e analizzare il sentiment, gli LLM sono impiegati nella generazione automatica di report e nella valutazione dei mercati, rendendo più efficienti i processi decisionali.

Limiti e sfide

Nonostante le straordinarie potenzialità, gli LLM presentano anche una serie di limiti e sfide significative. Uno dei problemi principali riguarda le "allucinazio-

ni", quando il modello genera contenuti che sembrano verosimili ma sono in realtà inaccurati o completamente inventati. Questo accade perché gli LLM si basano su correlazioni statistiche apprese durante l'addestramento, senza una vera comprensione dei fatti. Un altro limite importante è la capacità di ragionamento logico, che risulta ancora insufficiente: gli LLM tendono a riprodurre schemi linguistici piuttosto che a sviluppare un ragionamento autentico e coerente. Inoltre, esiste il rischio di bias nei dati di addestramento. Se il dataset include pregiudizi o stereotipi, il modello può finire per riprodurli, amplificando discriminazioni già presenti. Questo è un problema particolarmente grave in ambiti come il reclutamento o la giustizia, dove pregiudizi non controllati possono influenzare negativamente le decisioni. Infine, c'è la questione dell'enorme consumo di risorse computazionali: l'addestramento di LLM a grande scala richiede una notevole quantità di energia, sollevando preoccupazioni ambientali e facendo lievitare i costi per le aziende che vogliono implementare questi modelli.

Implicazioni etiche

L'adozione diffusa degli LLM solleva una serie di complesse questioni etiche e legali. Una delle preoccupazioni principali riguarda la privacy dei dati: gli LLM vengono addestrati su enormi quantità di dati, spesso prelevati da fonti pubbliche o semi-private, senza ottenere un consenso esplicito da parte degli utenti. Questo solleva interrogativi etici sul rispetto della privacy e sulla trasparenza delle aziende nella gestione di questi dati. Dal punto di vista legale, l'utilizzo di dati personali senza un adeguato consenso potrebbe violare normative come il GDPR (Regolamento Generale sulla Protezione dei Dati) in Europa, che impone che i dati siano trattati in modo lecito, trasparente e sicuro. Le aziende che utilizzano gli LLM devono poter dimostrare che i dati utilizzati per l'addestramento sono conformi a queste leggi, altrimenti rischiano sanzioni legali. Un altro problema riguarda la generazione di contenuti fuorvianti, come notizie false o disinformazione. Poiché gli LLM sono in grado di creare testi molto realistici e dettagliati, potrebbero essere sfruttati per ingannare gli utenti, manipolare l'opinione pubblica o diffondere disinformazione. Questo rende urgente l'introduzione di regolamentazioni legali specifiche per gestire l'uso

di queste tecnologie. Le leggi contro la diffamazione e la manipolazione elettorale potrebbero essere applicate a chi utilizza gli LLM per creare contenuti ingannevoli. Tuttavia, la natura dinamica e autonoma degli LLM complica l'applicazione delle leggi attuali, rendendo necessaria una revisione o riforma delle normative esistenti. Inoltre, non è ancora chiaro chi sia legalmente responsabile per i danni causati dalla disinformazione generata dagli LLM, soprattutto quando i contenuti falsi vengono distribuiti tramite piattaforme online. Anche l'accesso agli LLM solleva preoccupazioni relative all'equità: attualmente, è dominato da poche grandi aziende tecnologiche, che hanno le risorse necessarie per sviluppare e mantenere modelli su questa scala. Questo potrebbe accentuare le disuguaglianze digitali, con le piccole imprese o i paesi con risorse limitate che potrebbero restare indietro. Da una prospettiva legale, ciò potrebbe generare problemi di concorrenza e monopolio. Se alcune aziende controllano il mercato degli LLM, potrebbero esercitare un potere eccessivo sulla tecnologia, limitando l'innovazione o manipolando il suo utilizzo a loro favore. Per garantire un uso etico e socialmente responsabile degli LLM, è fondamentale che le aziende e i ricercatori adottino approcci trasparenti e regolamentati. Le linee guida etiche dovrebbero essere accompagnate da regolamentazioni legali che tutelino la privacy, prevenendo l'abuso di disinformazione, garantiscano un accesso equo alla tecnologia e promuovano politiche per mitigare gli effetti negativi sul lavoro. Promuovere lo sviluppo di leggi antidiscriminatorie e altre normative specializzate sarà cruciale per limitare i rischi e massimizzare i benefici di queste tecnologie.

2.4 Applicazioni degli LLM al Testing del Software

Come è stato descritto all'inizio del capitolo il testing del software è un'attività critica nel processo ingegneristico per la produzione di software, ed ha come obiettivo garantire che i sistemi siano affidabili, sicuri e performanti. Il software testing è costituito da diverse attività e fasi come la progettazione di casi di test, l'esecuzione di test su diverse piattaforme e ambienti, il testing di regressione e la validazione dei risultati. Tutte queste attività richiedono un significativo intervento umano, spesso comportando inefficienze dovute alla ripetitività e alla difficoltà di coprire tutti gli scenari possibili in un sistema complesso. Con l'avvento dei Large Language Models

che, come è stato illustrato in precedenza, sono degli strumenti promettenti per la comprensione e generazione di testo, è possibile automatizzare diverse attività del testing del software.

2.4.1 Applicazioni degli LLM nel Processo di Testing

Le applicazioni principali degli LLM al software testing riguardano diverse categorie: la generazione automatica di casi di test, la verifica della copertura del codice, il rilevamento di vulnerabilità e difetti del codice ed infine nell'automatizzazione del debug e analisi dei log.

Generazione automatica dei casi di test

La generazione dei casi di test è un processo che serve a garantire che il software funzioni correttamente in una vasta gamma di scenari, a seconda dei diversi input e delle condizioni specificate da testare. Solitamente la creazione dei test case viene affidata agli ingegneri del software che scrivono il codice e che analizzano le specifiche e i requisiti del sistema. Questa è una pratica lunga e, poiché i test case sono scritti manualmente, può essere incline ad errori umani. Gli LLM diventano quindi uno strumento utile alla generazione automatica di test case in quanto, analizzando la documentazione di progetto, requisiti funzionali ed anche il codice stesso, riescono a generare test case autonomamente suggerendo combinazioni di input relative a scenari complessi che potrebbero non essere immediatamente evidenti. Questo approccio non solo aumenta l'efficienza e riduce il tempo necessario per scrivere i test, ma migliora anche la copertura dei test, poiché l'LLM può esplorare una varietà più ampia di possibilità rispetto a quanto farebbe un tester umano. Inoltre, gli LLM possono essere integrati con tecniche di test basate su metodi automatici, come il fuzzing che testa il codice con input casuali o la generazione di mutanti per il mutation testing, dove le modifiche al codice vengono utilizzate per valutare la resistenza del software a errori introdotti.

Verifica della copertura del codice

La copertura del codice è una misura fondamentale calcolata sulla base del codice eseguito durante il test e seguendo un criterio di copertura specifico. L'ideale è avere una copertura alta in quanto questo indica che le parti critiche del software siano state eseguite e quindi testate sotto vari input, questo però può risultare complesso per software molto ampi di conseguenza sono necessari strumenti automatici che ispezionano le linee di codice non coperte e testate. Gli LLM possono essere utilizzati per migliorare questo processo per analizzare i risultati dei test e suggerire test aggiuntivi aumentando così la qualità del testing. Questo processo non si limita a semplici test di unità, ma può essere esteso a test di integrazione, test di sistema e test di accettazione, migliorando complessivamente l'efficacia del testing. Gli LLM possono essere inoltre utilizzati anche per analizzare il flusso di esecuzione e identificare sezioni di codice che potrebbero non essere testate a causa di condizioni di ramo o cicli complessi.

Rilevamento di vulnerabilità e difetti del codice

Una delle applicazioni più critiche degli LLM nel campo del testing riguarda il rilevamento di vulnerabilità e difetti del codice in quanto questi ultimi possono compromettere la qualità, sicurezza e stabilità del sistema oltre alle perdite di dati e compromissioni relative allo sfruttamento delle vulnerabilità. Tradizionalmente il rilevamento di vulnerabilità e difetti viene fatto tramite strumenti automatici come scanner oppure tramite una verifica manuale, che però diventa impraticabile e poco efficiente per software ampi e complessi. Di conseguenza gli LLM possono essere utilizzati per riconoscere pattern di vulnerabilità e difetti e possono essere impiegati anche per fornire eventuali strategie di mitigazione, essi infatti sono ideali per integrarsi in ambienti di testing continuo dove il codice è frequentemente aggiornato e testato.

Automatizzazione del debug e analisi dei log

Come anticipato in precedenza gli LLM possono essere utilizzati anche per fornire delle risoluzioni ai problemi identificati sempre in modo automatico nel codice,

questa attività è strettamente collegata al debugging e all'analisi dei log. Così come questi ultimi consentono di identificare e correggere errori nel software, anche gli LLM sono in grado di farlo automatizzando e ottimizzando entrambi questi processi. Gli LLM possono infatti analizzare i messaggi di errore, i traceback e le eccezioni nel codice per determinare la causa di un bug suggerendo soluzioni basate su pattern di vulnerabilità riconosciute in precedenza.

2.4.2 LLM e Mutation Testing

Il test di mutazione è una delle tecniche di test del software più ampiamente studiate a causa della sua efficienza ma anche difficoltà di implementazione. Esso è eseguito producendo varianti di programma facendo cambiamenti sintattici semplici, cioè mutazioni nel codice sorgente del programma di origine, generando quindi dei mutanti. Questi ultimi costituiscono gli obiettivi del test, nel senso che è necessario testarli per innescare comportamenti diversi sul programma originale e sui mutanti. Un mutante viene definito ucciso o vivo in base al fatto che l'esecuzione del test porti ad un output di programma diverso da quello del programma originale oppure no. La proporzione dei mutanti uccisi rispetto all'intero gruppo di mutanti costituisce una metrica di adeguatezza del test denominata punteggio di mutazione. Le tecniche tradizionali di generazione dei mutanti utilizzano semplici trasformazioni sintattiche chiamate operatori di mutazione. Questi approcci basati su regole purtroppo però producono un gran numero di mutanti ridondanti che non sono utili ai fini del test causando un overhead computazionale non indifferente [17].

In letteratura sono già presenti diversi progetti e ricerche che studiano il comportamento degli LLM nella generazione di mutanti applicati principalmente a codice Java.

Nello specifico nel progetto di Wang et al. [17] sono stati testati gli LLM più recenti per la generazione del codice nella produzione di mutanti sulla base del dataset Defects4J. Analizzando i risultati ottenuti da questo progetto è di notevole importanza notare che innanzitutto l'approccio con LLM risulta computazionalmente più costoso in termini di tempo dove gli LLM sono battuti da strumenti che utilizzano i tradizionali approcci basati su regole. Successivamente è stata fatta un'analisi che

studia la bontà dei prompt dati in input al modello e l'utilizzo di differenti modelli. Riguardo i prompt analizzati, quello che ha avuto le prestazioni è stato quello di default che fornisce un contesto al modello, ossia il codice del metodo, un target cioè la linea di codice da mutare ed un esempio di mutazione, oltre ad altre informazioni aggiuntive tralasciabili. Riguardo l'analisi tra i diversi LLM impiegati i modelli GPT-4 risultano avere le prestazioni migliori in termini di tempo e risultati ottenuti rispetto a GPT-3.5 e DeepSeek.

Un altro progetto recente riguarda LLMorpheus [18] che studia una tecnica per generare mutanti di codice Javascript utilizzando LLM basati sul Masked Language Modeling. I risultati della valutazione empirica su 13 applicazioni dimostrano che LLMorpheus è in grado di produrre mutanti simili a veri bug, che non possono essere prodotti con operatori standard di mutazione. Il progetto ha rilevato che il 63,2% dei mutanti sopravvissuti prodotti da LLMorpheus sono cambiamenti comportamentali e che il numero di mutanti (quasi) equivalenti è inferiore al 20%. Gli esperimenti che studiano le variazioni dei prompt rivelano che il modello "completo" che include tutte le informazioni funziona meglio e che omettere parti delle informazioni può influenzare i risultati in modi diversi. Infine, utilizzando tre LLM, codellama-34b-instruct ha prodotto generalmente il maggior numero di mutanti e di mutanti vivi. Lo studio che più si avvicina al progetto proposto in questa ricerca risulta essere quello di Garg et al. [1] poiché esso utilizza una variante di CodeBERT chiamata μ BERT per generare mutanti sulla base del dataset Vul4J per vedere in che misura i mutanti sono associati a vulnerabilità del software. Utilizzando quindi un LLM basato sul Masked Language Modeling il progetto ha dimostrato che μ BERT può generare mutanti che fanno fallire gli stessi test e per le stesse ragioni del 32% dei 45 progetti vulnerabili studiati. Inoltre, μ BERT può generare mutanti che, anche se non per le stesse ragioni, fanno fallire gli stessi test delle altre 7 vulnerabilità. Nel complesso, la mutazione basata sul Large Language Model è riuscita a dimostrare che può produrre mutazioni che deviano i comportamenti del programma nello stesso modo delle vulnerabilità.

Tenendo come riferimento quest'ultimo progetto, la ricerca e il progetto di questa tesi si propongono di studiare il comportamento di un LLM di tipo Autoregressive Language Modeling utilizzando Gemini e basandosi sui progetti utilizzabili presenti

nel dataset Vul4J.

2.4.3 Vantaggi e Limiti degli LLM

L'adozione degli LLM nel processo di testing del software porta con sé numerosi benefici principalmente riguardo l'incremento significativo dell'efficienza. L'automazione dei compiti ripetitivi, come la generazione di test, la verifica della copertura del codice e l'esecuzione di test su differenti configurazioni, riduce notevolmente il carico di lavoro manuale e accelera l'intero ciclo di sviluppo. L'utilizzo degli LLM permette anche di esplorare una gamma di test più ampia e diversificata rispetto ai metodi tradizionali, poiché i modelli possono produrre casi di test innovativi basati su un'analisi approfondita dei requisiti e delle specifiche del sistema. Nonostante i numerosi vantaggi, l'adozione degli LLM nel testing del software non è priva di sfide e limitazioni. Una delle principali difficoltà riguarda l'affidabilità dei modelli, infatti, sebbene gli LLM siano in grado di generare output di alta qualità, non sono infallibili e possono produrre errori o risultati non pertinenti, soprattutto quando il contesto è complesso o il codice non segue convenzioni standard. La comprensione del codice da parte degli LLM è basata su pattern statistici e non su una vera "comprensione" del comportamento del software, il che può comportare errori di interpretazione, soprattutto in scenari non convenzionali o quando il codice contiene anomalie.

2.4.4 Motivazioni e Obiettivi

I lavori presenti in letteratura basati sullo studio degli LLM applicati al Mutation Testing in generale sono pochi, e ancor di meno sono le ricerche in ambito Security Mutation Testing dove si passa dall'imitare difetti software a vere e proprie vulnerabilità di sicurezza. Di conseguenza, la ricerca e il progetto di questa tesi si propongono di ampliare questo ramo di applicazione degli LLM studiando la loro applicazione al Security Mutation Testing. Più in particolare, tenendo in considerazione il lavoro svolto da Garg et al. [1] dove viene utilizzato lo strumento μ BERT per produrre mutanti sulla base delle vulnerabilità, lo si vuole ampliare per iniziare ad effettuare confronti tra diversi LLM in questo campo di applicazione. Sarà quindi studiato e sviluppato il comportamento di un LLM di tipo Autoregressive Language Modeling,

Gemini di Google, basandosi sui progetti presenti nel dataset Vul4J. L'utilizzo di questo tipo di modello solleva diverse sfide in quanto lo strumento μ BERT incorpora le fasi di preprocessing necessarie che consentono, a partire da una classe vulnerabile fixata, di mascherare i token da mutare nelle specifiche posizioni indicate; l'utilizzo di un modello diverso, di tipo ALM, rende necessario lo sviluppo di un approccio differente. Come sarà descritto successivamente nella sezione relativa alla progettazione, a differenza di μ BERT, non dovranno essere mascherati i token da mutare, proprio a causa della natura del modello, ma sarà fornito il contesto necessario e le indicazioni su dove applicare la mutazione al modello in modo da riuscire a produrre dei mutanti specifici per il problema in analisi. I mutanti prodotti dal modello saranno poi valutati nello stesso modo in cui è stato fatto per il progetto di Garg et al. [1] in modo da poter effettuare un confronto concreto sul risultato dei test e vedere come i modelli di tipo ALM producono mutanti nel dominio di interesse.

Sperimentazione e impostazione del progetto

Come descritto nei precedenti capitoli la ricerca nell'ambito dei Large Language Models ha dimostrato un potenziale notevole in diversi campi applicativi, tra cui il security mutation testing. Poiché i metodi classici richiedono spesso un intervento manuale significativo o strumenti specifici che non sempre si adattano a scenari complessi, l'impiego degli LLM promette un'automazione avanzata nella generazione di mutazioni e una maggiore precisione nel rilevamento di vulnerabilità, per cui l'obiettivo della tesi diventa proprio quello di studiare come gli LLM si comportano in questo ambito:

© **Our Goal.** Studiare e testare come si comportano gli LLM nella produzione di mutanti applicati al security mutation testing, in particolare sul dataset VUL4J.

3.1 Research Questions

Per poter dare analizzare bene il caso di studio è necessario suddividere la ricerca in più parti analizzando nello specifico il comportamento degli LLM applicati al problema. Per poter fare ciò si risponde alle seguenti domande di ricerca:

Q RQ₁. *Che corrispondenza c'è tra le vulnerabilità contenute nei progetti del dataset VUL4J e mutanti prodotti da un LLM?*

Per poter rispondere a questa domanda saranno analizzati 45 progetti presenti nel dataset VUL4J e i mutanti generati dagli LLM sulla base della versione fixata del codice del progetto. I mutanti risultanti saranno poi testati e i risultati dei test confrontati con i test vulnerabili per poter stabilire il grado di accoppiamento tra le versioni di codice vulnerabile e mutato.

Q RQ₂. *In che modo il prompt dato in input al modello influenza i risultati ottenuti?*

Nella sperimentazione saranno prodotti principalmente due prompt. Il primo di base che fornisce indicazioni sulla produzione dei mutanti come ad esempio vincoli riguardo i mutanti generati che non devono essere uguali tra loro, non devono essere uguali alla versione fixata o vulnerabile data in input e il numero di mutanti da produrre, inoltre saranno date informazioni di contesto quali il metodo fixato e vulnerabile e un commento nelle righe dove deve essere effettuata la mutazione. Il secondo prompt invece aggiunge una frase in cui si danno stimoli emotivi ai modelli in quanto è stato dimostrato che attraverso questo tipo di prompt è possibile ottenere delle prestazioni migliori nella produzione delle risposte [19].

Nella Figura 3.1 viene mostrato il codice Python che definisce il prompt da utilizzare, come è stato descritto in precedenza esso contiene le informazioni di base e necessarie affinché possano essere generati i mutanti in un formato utile alla successiva elaborazione da parte del progetto.

Il prompt 2 visibile nella Figura 3.2 invece è composto dallo stesso testo definito nel prompt 1 più una frase che dà un contesto emozionale e personale al modello, creando una dipendenza sulla risposta che esso dovrà generare.

Q RQ₃. *Che differenza c'è tra LLM di tipo Masked Language Modeling e Autoregressive Language Modeling nella produzione di mutanti per il security testing?*

La risposta a questa domanda sarà basata sul confronto tra i risultati ottenuti da questo lavoro di ricerca ed uno studio già presente in letteratura condotto da Garg et al. [1]. Quest'ultimo infatti è uno dei pochi studi allo stato dell'arte riguardanti


```

prompt1 = (f"Create a mutant for the following method by applying one of the mutation operators used for mutation "
    f"testing only in the code line marked with //MUTATE. "
    f"The task is to improve code testing by checking whether changes made to fixed code introduce or "
    f"detect vulnerabilities. it is very important that you do not produce mutants "
    f"equal to fixed code or identical to each other .\n Fixed code: {fixed_code}\n"
    f"Vulnerable code: {vulnerable_code}\n Please consider changes such as using different operators, "
    f"changing constants, referring to different variables, object properties, functions, methods or any "
    f"other mutation operator you consider useful for the purpose. "
    f"Provide ten answers as fenced code blocks containing only"
    f" the mutated lines, using the following template: "
    f"Option 1: The lines can be replaced with: ```java <code fragment> ``` "
    f"Option 2: The lines can be replaced with: ```java <code fragment> ``` "
    f"Option 3: The lines can be replaced with: ```java <code fragment> ``` "
    f"... "
    f"Option 10: The lines can be replaced with: ```java <code fragment> ``` ")

```

Figura 3.1: Estratto del progetto che mostra il prompt 1 contenente le informazioni necessarie alla produzione dei mutanti.

```

prompt2 = (f"Create a mutant for the following method by applying one of the mutation operators used for mutation "
    f"testing only in the code line marked with //MUTATE. "
    f"The task is to improve code testing by checking whether changes made to fixed code introduce or "
    f"detect vulnerabilities. it is very important that you do not produce mutants "
    f"equal to fixed code or identical to each other .\n Fixed code: {fixed_code}\n"
    f"Vulnerable code: {vulnerable_code}\n Please consider changes such as using different operators, "
    f"changing constants, referring to different variables, object properties, functions, methods or any "
    f"other mutation operator you consider useful for the purpose . "
    f"Provide ten answers as fenced code blocks containing only"
    f" the mutated lines, using the following template: "
    f"Option 1: The lines can be replaced with: ```java <code fragment> ``` "
    f"Option 2: The lines can be replaced with: ```java <code fragment> ``` "
    f"Option 3: The lines can be replaced with: ```java <code fragment> ``` "
    f"... "
    f"Option 10: The lines can be replaced with: ```java <code fragment> ``` "
    f"Please take your time with the implementation, as this is very important to my career")

```

Figura 3.2: Estratto del progetto che mostra il prompt 2 contenente le informazioni necessarie alla produzione dei mutanti più uno stimolo emotivo per il modello.

l'applicazione di LLM al security mutation testing poiché, tramite uno strumento basato sull'LLM CodeBERT produce mutanti delle versioni di codice fixate di VUL4J. Allo stesso modo questo lavoro di tesi produrrà mutanti sulle versioni di codice fixato di VUL4J utilizzando però due LLM di tipo ALM, in particolare Gemini e Chat GPT, per poter confrontare i risultati ottenuti dai diversi LLM.

3.2 Struttura dell'implementazione

La descrizione dell'implementazione del progetto relativo al caso di studio sarà strutturata in modo da delineare ogni entità facente parte della ricerca e successiva-

mente sarà illustrato come queste vengono usate e il rationale che segue l'approccio utilizzato.

Il codice dell'intero progetto, da cui sono stati estratti anche i prompt presentati in precedenza è disponibile al link `LLM_Mutation_Testing`.

3.2.1 VUL4J

Lo studio sarà basato sull'analisi dei dati provenienti dal dataset VUL4J [3] che è un dataset specifico per il dominio della sicurezza del software, progettato per fornire una raccolta di vulnerabilità documentate nel codice Java.

Scopo e Struttura del dataset

L'obiettivo di VUL4J è quello di supportare la ricerca in ambito cybersecurity in particolare riguardo alla sicurezza del software. Esso infatti è una raccolta di 79 progetti JAVA vulnerabili che offre una base di riferimento per confrontare strumenti e approcci di analisi delle vulnerabilità. VUL4J viene utilizzato principalmente negli ambiti: testing di strumenti automatici per la rilevazione di vulnerabilità, ricerca sull'efficacia di strumenti per l'analisi statica e dinamica ed infine il campo in cui si posiziona questa ricerca che è la creazione di mutanti per il mutation testing.

La struttura del dataset è infatti organizzata in modo da essere facilmente navigabile per scopi di analisi, esso contiene per ogni progetto analizzato informazioni relative alla vulnerabilità e informazioni relative al patch delle stesse indicando anche il commit della patch applicata. Le vulnerabilità sono di diverso tipo, le maggiori identificate sono: SQL Injection, Cross-Site Scripting, Insecure Deserialization, Command Injection e Path Traversal. Per ognuna di esse sono disponibili metadati che forniscono ulteriori informazioni e contesto quali il tipo della vulnerabilità, la gravità, dipendenze esterne e librerie utilizzate. Una caratteristica fondamentale di questo dataset è che, una volta scaricato ed installato seguendo la guida presente sul repository del progetto, si possono facilmente eseguire operazioni sui progetti contenuti nel dataset da linea di comando. Le operazioni principali che saranno anche utilizzate in questa ricerca sono:

- Checkout che permette di effettuare il checkout di uno specifico progetto vulnerabile in una cartella a piacere;
- Compile che compila il progetto scaricato in precedenza;
- Test che esegue la suite di test per lo specifico progetto.

Sono poi disponibili altri comandi non utilizzati come `reproduce` che verifica la riproducibilità della vulnerabilità nello specifico progetto oppure `sast` che esegue una SpotBugs analysis.

Infine, per dare una descrizione esaustiva del progetto VUL4J, è necessario illustrare la facilità di installazione dello stesso in quanto viene fornito dagli sviluppatori un docker container pronto all'uso con tutte le librerie e dipendenze necessarie già installate, per cui è possibile direttamente eseguire i comandi illustrati precedentemente per utilizzare il dataset. Viene poi fornita anche una guida per l'installazione con tutti i requisiti e dipendenze necessarie.

Vantaggi e Limiti

Oltre alla semplicità di installazione ed utilizzo è necessario descrivere ulteriori vantaggi che si hanno nell'uso di VUL4J, esso infatti risulta essere un dataset completo perché per ogni progetto contenuto in esso fornisce vulnerabilità documentate e le patch corrispondenti, è un dataset flessibile perché può essere usato in diversi contesti e in diversi ambiti di applicazione come ad esempio il machine learning, analisi statica del codice, analisi manuale del codice e automazione del testing. Infine un altro vantaggio da tenere in considerazione è la presenza di riferimenti per ogni vulnerabilità, ognuna di esse è infatti corredata di informazioni dettagliate semplificando l'analisi e la replicabilità.

3.2.2 Gemini

Gemini, chiamato in precedenza anche Bard, è un Large Language Model sviluppato dal Gemini Team di Google nel 2023 [20]. Esso viene descritto come un Large Language Model che avanza di gran lunga le capacità del modello in tutti i campi

dell'elaborazione quali testo, codice, immagini, video e audio. Gemini è disponibile in più varianti, ma quella più potente risulta essere Gemini Ultra. In particolare esso supera le prestazioni di esperti umani sul benchmark MMLU, con un punteggio del 90,0%, che è stato una misura de facto dei progressi per i LLM sin da quando è stato rilasciato nel 2020. Nel dominio multimodale, Gemini Ultra stabilisce nuovi standard di riferimento sulla maggior parte della comprensione dell'immagine, della comprensione video e della comprensione audio senza modifiche o ottimizzazioni specifiche per il compito. Le capacità di ragionamento multimodale di Gemini Ultra sono infatti evidenti dalla sua performance allo stato dell'arte sul benchmark MMMU [21] che comprende domande sulle immagini che richiedono conoscenza del soggetto a livello universitario e ragionamento logico.

Oltre ai risultati numerici ottenuti dal benchmark, nel concreto le nuove capacità dei modelli Gemini consentono di analizzare immagini complesse, come grafici o infografiche, ragionando su sequenze interfogliate di immagini, audio e testo, e generano testi e immagini interfogliati a loro volta come risposte aprendo una vasta gamma di nuove applicazioni. I modelli Gemini infatti possono consentire nuovi approcci in settori quali l'istruzione, la risoluzione quotidiana dei problemi, la comunicazione multilingue, il riassunto delle informazioni, l'estrazione e la creatività. Nonostante le loro notevoli capacità, è da notare che ci sono limitazioni all'uso degli LLM in generale e quindi anche di questo caso particolare. Vi è un continuo bisogno di ricerca e sviluppo sulle "hallucinations" generate dai LLM per garantire che i risultati del modello siano più affidabili e verificabili. Queste ultime si verificano in compiti che richiedono capacità di ragionamento ad alto livello come la comprensione causale, deduzione logica e ragionamento controfattuale, anche se i LLM raggiungono prestazioni impressionanti sui benchmark di riferimento. Ciò sottolinea la necessità di valutazioni più impegnative e robuste per misurare la loro vera comprensione, poiché gli attuali LLM all'avanguardia saturano molti parametri di riferimento [20].

3.2.3 ChatGPT

ChatGPT è stato sviluppato da OpenAI, azienda leader nel settore dell'Intelligenza Artificiale e Machine Learning, utilizzando un'architettura basata sui Transformer.

Il modello si basa su una rete neurale profonda progettata per gestire sequenze di dati, ottimizzando l'uso dell'attenzione auto-regressiva per modellare la dipendenza tra i token. Concretamente, il modello lavora prevedendo il prossimo token (parola, simbolo, o carattere) in una sequenza, basandosi sul contesto precedente.

L'architettura segue la struttura del Generative Pre-trained Transformer (GPT) ed è composta da:

- Encoder-decoder transformer: ottimizzato per comprensione e generazione del linguaggio;
- Struttura gerarchica: include decine di strati con miliardi di parametri ottimizzati tramite addestramento supervisionato e fine-tuning;
- Self-attention mechanism: consente al modello di dare peso differente alle parole di una frase, cogliendo sfumature semantiche e contesti complessi.

ChatGPT utilizza tecniche avanzate di addestramento su larga scala basando il suo apprendimento su dataset eterogenei, comprendenti libri, articoli scientifici, pagine web e altri tipi di documenti, includendo anche codice sorgente. L'addestramento avviene con tecnologie di calcolo distribuito su GPU e TPU, richiedendo enormi risorse computazionali, necessarie anche per l'elaborazione delle risposte.

ChatGPT offre numerosi vantaggi distintivi rispetto ad altri modelli di grandi dimensioni, che lo rendono una soluzione particolarmente versatile ed efficace in una vasta gamma di applicazioni. Uno dei principali punti di forza risiede nella sua capacità di adattarsi al contesto di una conversazione, fornendo risposte coerenti e dettagliate anche in situazioni complesse. Questa capacità deriva dall'architettura basata sui Transformer, che consente al modello di cogliere sfumature linguistiche e relazioni semantiche tra i termini, garantendo un'interpretazione accurata delle richieste. Un altro vantaggio fondamentale di ChatGPT è la sua integrazione con strumenti esterni. A differenza di molti altri modelli, può utilizzare moduli specifici per compiti aggiuntivi, come eseguire calcoli complessi, generare immagini o cercare informazioni in tempo reale. Questo lo trasforma in una piattaforma completa, capace di ampliare le sue funzionalità a seconda delle esigenze dell'utente, superando i limiti tipici di un modello puramente testuale.

ChatGPT si distingue anche per il metodo di addestramento adottato, che combina fine-tuning supervisionato con tecniche avanzate di reinforcement learning basato sul feedback umano. Questo approccio ibrido permette di affinare continuamente le risposte, riducendo errori e migliorando la qualità delle interazioni. La componente umana nell'addestramento gioca un ruolo cruciale, poiché consente di identificare e correggere potenziali bias o incoerenze presenti nel modello. A tal proposito è necessario specificare che questo modello, come anche gli LLM in generale, presenta dei limiti nell'applicazione. Una delle problematiche principali è legata alla presenza di bias nei dati di addestramento poiché, essendo stato formato su un vasto insieme di dati raccolti da fonti pubblicamente disponibili, il modello può riflettere pregiudizi culturali, linguistici o sociali presenti in tali dati. Questo può portare a risposte che, pur tecnicamente corrette, risultano inadeguate o non neutrali in determinati contesti. Un altro limite significativo riguarda la dipendenza dai dati utilizzati durante il pre-training. ChatGPT non ha accesso diretto a informazioni aggiornate in tempo reale, a meno che non utilizzi strumenti specifici come browser integrati. Di conseguenza, la sua conoscenza è limitata al momento in cui è stato addestrato e può non essere accurata per eventi recenti o in rapida evoluzione. Questo può rappresentare un ostacolo nelle applicazioni che richiedono informazioni costantemente aggiornate.

3.2.4 μ BERT

μ BERT [22] è uno strumento utilizzato nel campo del mutation testing che utilizza CodeBERT per generare mutanti mediante mascheramento e sostituzione dei token. μ BERT è utile poiché integra il preprocessing necessario all'input da dare al modello per fargli produrre dei mutanti utilizzabili. Esso infatti prende una classe Java ed estrae le espressioni per mutare, successivamente maschera il token di interesse, ad es. un nome variabile, e invoca CodeBERT per completare la sequenza mascherata (cioè per prevedere il token mancante). Questo approccio si è dimostrato efficace nel l'aumentare il rilevamento dei fault nelle test suite e nel migliorare la precisione dei bug-detector basati sul machine learning. Data una sequenza di codice CodeBERT predice i 5 token più probabili per sostituire quello mascherato, successivamente μ BERT prende queste previsioni e genera mutanti sostituendo il token mascherato

con quelli previsti (quindi per ogni token mascherato crea cinque mutanti). Nella fase di postprocessing μ BERT scarta i mutanti non compilabili e quelli sintatticamente uguali al programma originale (casi in cui CodeBERT predice il token mascherato originale) [1].

3.3 Processo sperimentale

Per poter confrontare al meglio i due LLM e le due strategie di analisi delle domande e generazione delle risposte, quali MLM e ALM, ci serviamo della classificazione dei mutanti generati seguendo il lavoro svolto da Garg et al. [1] per stabilire il grado di accoppiamento tra un mutante con una vulnerabilità analizzando i test falliti e le ragioni dei fallimenti. Questo metodo di classificazione è preso a sua volta dal lavoro svolto da Gay et al. [23] e consiste nella definizione di diverse classi, avendo come input un mutante, una vulnerabilità e la suite di test scritta dallo sviluppatore:

- Strong Coupling: se il mutante e la vulnerabilità falliscono esattamente sugli stessi test PoV per le stesse ragioni (i.e. stessa eccezione/errore viene lanciato);
- Test Coupling: se il mutante e la vulnerabilità falliscono esattamente sugli stessi test PoV ma uno o più falliscono per motivi diversi;
- Partial Coupling: se il mutante e la vulnerabilità falliscono su alcuni, ma non tutti, i test PoV per gli stessi motivi;
- Partial Test Coupling: se il mutante e la vulnerabilità falliscono su alcuni, ma non tutti i test PoV ma uno o più falliscono per motivi diversi;
- No Coupling: se il mutante viene ucciso solo da test che non attivano la vulnerabilità (i POV non uccidono il mutante).

Lo studio condotto sui modelli Gemini e ChatGPT ha seguito diversi step, oltre alle fasi iniziali e di postprocessing è stato necessario implementare una fase di preprocessing necessaria per formattare l'input da dare al modello che nello studio di riferimento per il confronto era incorporata in μ BERT. Seguono le fasi necessarie alla sperimentazione del progetto.

3.3.1 Inizializzazione

La prima operazione effettuata è stata innanzitutto l'installazione di VUL4J. Il metodo di installazione seguito è stato quello tramite Docker container poiché sono stati evitati eventuali problemi di installazione e compatibilità delle librerie necessarie. Una volta installato Docker, scaricato ed importato il container è stato solo necessario testare l'approccio più semplice per poter interagire con il container tramite script Python.

La macchina utilizzata per la sperimentazione è un notebook con cpu Intel Core i7 11th Gen, 36GB di RAM DDR4 a 3200Mhz e una scheda grafica NVIDIA GeForce MX350, la generazione dei mutanti sarà però effettuata tramite richieste inoltrate direttamente a Gemini e ChatGPT, quindi da remoto.

3.3.2 Preparazione dei test

Le operazioni da effettuare per ogni progetto da testare sono innanzitutto il checkout e successivamente il compile e l'esecuzione dei test sul progetto che risulta essere nella versione fixata dopo il checkout. Durante la fase di test è stato rilevato che per buona parte dei progetti in analisi, nonostante non fossero presenti vulnerabilità, erano presenti test falliti. Tutti questi progetti con una test suite unsuccessful sono stati scartati poiché è necessario avere una suite di test di partenza pulita, ovvero con nessun test fallito. La motivazione risiede nel fatto che se fossero presenti test falliti, successivamente alla produzione dei mutanti e il loro test, nella fase di analisi non è possibile stabilire quali test sono falliti a causa dei mutanti e per quale ragione. I progetti risultanti che hanno una test suite pulita sono 45, per cui per ognuno di essi vengono effettuate le operazioni di checkout, compile e test.

3.3.3 Preprocessing dell'input

Per ognuno dei progetti in analisi, dopo che è stato eseguito il comando vul4j per il test, vengono selezionati e preprocessati i metodi da mutare tramite Gemini. Vul4j mette a disposizione già le classi del progetto che contengono la vulnerabilità e la classe risultante dal fix della vulnerabilità, sarà necessario però introdurre un

commento `//<MUTATE>` nelle righe che devono essere modificate per generare il mutante. Le due classi, vulnerabile e fixata, vengono quindi comparate e viene aggiunto il commento `//<MUTATE>` nelle righe della classe fixata che differenziano dalla classe vulnerabile, per cui è quello il punto in cui viene rimossa la vulnerabilità ed introdotto il fix per la stessa, e queste quindi sono anche le righe in cui è necessario introdurre modifiche per la generazione dei mutanti.

Successivamente a questa fase iniziale di preprocessing è necessario dare in input il codice preprocessato ai due modelli, tramite il prompt selezionato nel quale viene inserito sempre sia il codice fixato, al quale sono stati aggiunti i commenti `//<MUTATE>`, che il codice del metodo vulnerabile, per fornire del contesto aggiuntivo al modello. Vengono generati 10 mutanti per ogni metodo della classe che contiene commenti `//<MUTATE>` quindi, in alcuni casi, anche per una singola classe segnalata come vulnerabile, vengono eseguite più iterazioni e quindi generazioni di mutanti a seconda di quanti sono i metodi nei quali era presente la vulnerabilità. I mutanti vengono infatti identificati in base al nome del metodo ed un numero che va da 1 a 10.

3.3.4 Generazione dei mutanti e postprocessing

I mutanti vengono generati dai modelli in base all'input descritto in precedenza, nella risposta prodotta è stato indicato dal prompt di inserire solo le linee di codice mutate per una questione di lunghezza infatti, poiché è necessario generare 10 mutanti per risposta, se così non fosse, e quindi nella risposta sarebbe contenuto il codice di tutto il metodo, il modello non riuscirebbe a generare tutto il testo necessario. Per questo motivo le righe mutate presenti nella risposta devono essere sostituite a quelle originali del codice fixato in modo da avere delle classi che contengono i metodi mutati rispetto alla classe fixata d'origine. Nella Figura 3.3 e Figura 3.4 vengono presentati due esempi di mutazione paragonata ad una vulnerabilità del software per meglio comprendere l'utilità di questo tipo di testing concretamente e cosa ci si aspetta che venga prodotto dai modelli in analisi.

<pre>private static void decompress (final InputStream in, final byte[] out) throws IOException { int position = 0; final int total = out.length; while (position < total) { final int n = in.read(); if (n > 128) { final int value = in.read(); for (int i = 0; i < (n & 0x7f); i++) { out[position++] = (byte) value; } else { for (int i = 0; i < n; i++) { out[position++] = (byte) in.read(); } } } } }</pre>	<pre>private static void decompress (final InputStream in, final byte[] out) throws IOException { int position = 0; final int total = out.length; while (position < total) { final int n = in.read(); if (n < 0) { throw new ImageReadException("Error decompressing RGBE file"); } if (n > 128) { final int value = in.read(); for (int i = 0; i < (n & 0x7f); i++) { out[position++] = (byte) value; } else { for (int i = 0; i < n; i++) { out[position++] = (byte) in.read(); } } } } }</pre>	<pre>private static void decompress (final InputStream in, final byte[] out) throws IOException { int position = 0; final int total = out.length; while (position < total) { final int n = in.read(); if (n == 0) { // '<' modified to '=' throw new ImageReadException("Error decompressing RGBE file"); } if (n > 128) { final int value = in.read(); for (int i = 0; i < (n & 0x7f); i++) { out[position++] = (byte) value; } else { for (int i = 0; i < n; i++) { out[position++] = (byte) in.read(); } } } } }</pre>
(a) Vulnerable Code (CVE-2018-17201)	(b) Fixed Code	(c) Vulnerability-coupled Mutant

Figura 3.3: Vulnerabilità CVE-2018-17201 (Fig. 1a) che causa "Infinite Loop" il quale permette ulteriormente Denialof-Service (DoS), l'attacco è fixato con l'eccezione condizionale utilizzando la condizione "if" (Fig. 1b). Il mutante (Fig. 1c) modifica la condizione "if" che annulla la correzione e si lega fortemente alla vulnerabilità [1].

3.3.5 Riesecuzione del test e analisi

In questa fase si parte dalle classi risultanti dalla fase precedente, che quindi contengono i metodi mutati, per rieseguire la compilazione e il test dell'intero progetto per vedere innanzitutto se i mutanti sono sintatticamente corretti quindi la compilazione ha successo, successivamente vengono valutate le somiglianze comportamentali tra i mutanti e vulnerabilità per determinare la loro categoria di accoppiamento secondo la definizione descritta all'inizio della sezione.

Le fasi che partono dalla generazione dei mutanti a seguire vengono ripetute anche per la valutazione di prompt differenti per poter rispondere alla research question che ne valuta l'efficacia.

```

void addPathParam(String name, String
value, boolean encoded) {
    if (relativeUrl == null) {
        throw new AssertionError(); }

    relativeUrl = relativeUrl.replace("{ " +
        name + "}",
        canonicalizeForPath(value,
        encoded));
}

```

(a) Vulnerable Code (CVE-2018-1000850)

```

void addPathParam(String name, String
value, boolean encoded) {
    if (relativeUrl == null) {
        throw new AssertionError(); }
    String replacement =
        canonicalizeForPath(value, encoded);
    String newRelativeUrl =
        relativeUrl.replace("{ " + name +
        "}", replacement);
    if (PATH_TRAVERSAL
        .matcher(newRelativeUrl)
        .matches()) {
        throw new IllegalArgumentException(
            "@Path parameters shouldn't perform
            path traversal ('.' or '..'): " +
            value); }
    relativeUrl = newRelativeUrl;
}

```

(b) Fixed Code

```

void addPathParam(String name, String
value, boolean encoded) {
    if (relativeUrl == null) {
        throw new AssertionError(); }
    String replacement =
        canonicalizeForPath(value, encoded);
    String newRelativeUrl =
        relativeUrl.replace("{ " + name +
        "}", replacement);
    if (PATH_TRAVERSAL
        .matcher(name)//passed argument changed
        .matches()) {
        throw new IllegalArgumentException(
            "@Path parameters shouldn't perform
            path traversal ('.' or '..'): " +
            value); }
    relativeUrl = newRelativeUrl;
}

```

(c) Vulnerability-coupled Mutant

Figura 3.4: Vulnerabilità CVE-2018-1000850 che causa "Path Traversal", che consente quindi l'accesso a una directory riservata (Fig. 2a), è corretta con l'eccezione condizionale nel caso '.' o '..' appare nella variabile newRelativeUrl" (Fig. 2b). Il mutante fortemente accoppiato (fig. 2c) cambia il "newRelativeUrl" passato come argomento in "name", che annulla la correzione e reintroduce il comportamento vulnerabile[1].

CAPITOLO 4

Risultati ottenuti

In questo capitolo saranno mostrati i risultati ottenuti per ogni research question cercando di dare una risposta esaustiva ad ognuna.

4.1 RQ1: Corrispondenza tra Vulnerabilità in Vuln4J e Vulnerabilità Generate

Per valutare i risultati ottenuti dalla produzione dei mutanti per i progetti considerati, presenti nel dataset VUL4J, saranno analizzate tutte le classi di accoppiamento dei mutanti definite nel capitolo 3 alla sezione 3, supportando la comprensione tramite diagrammi di Venn.

Dai risultati ottenuti, mettendoli a confronto con quelli osservati dal paper di riferimento [1], che utilizza μ BERT come modello, si possono fare delle valutazioni sulla bontà dei mutanti prodotti in base all'andamento dei test. Per l'esperimento condotto sul modello Gemini dando in input il prompt 1 visibile in Figura 3.1 le osservazioni in base all'accoppiamento dei test vengono riportate in Figura 4.1 e si può constatare principalmente che il modello ha prodotto almeno un mutante per 25 progetti in cui quest'ultimo è accoppiato in modo forte o parziale con la vulnerabilità.

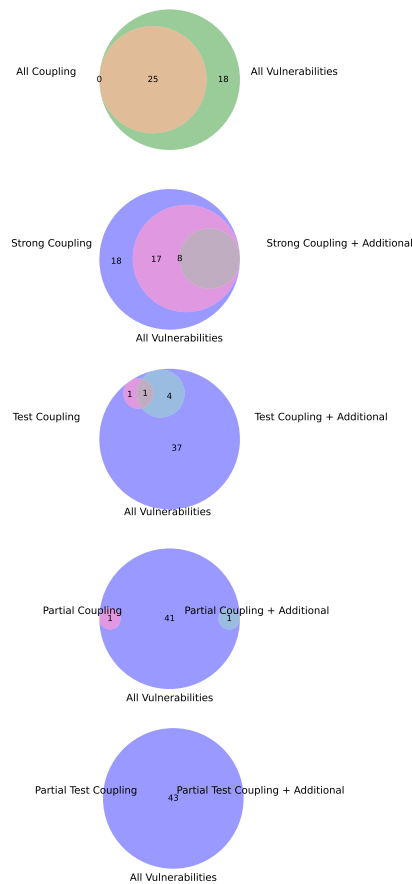


Figura 4.1: Diagramma di Venn contenente le classi di accoppiamento dei test della sperimentazione condotta con Gemini sul prompt 1.


Per le restanti 18 vulnerabilità testate nessun mutante generato condivide una somiglianza comportamentale con le vulnerabilità esposte. Saranno analizzate di seguito le categorie delle classi di accoppiamento in modo più approfondito.

- **Strong Coupling:** per 17 dei progetti analizzati il modello è stato in grado di produrre mutanti fortemente accoppiati, ossia che rompono gli stessi test per le stesse ragioni della vulnerabilità in analisi. Per altre 8 vulnerabilità, oltre ai test della vulnerabilità, falliscono altri test per cui siamo nel caso Strong Coupling + Additional. Considerando questi due gruppi di mutanti 25 progetti di quelli considerati possono portare alla scoperta di ulteriori vulnerabilità nel codice di partenza.
- **Test Coupling:** per 2 progetti Gemini può generare almeno un mutante che non supera gli stessi test che non supera la vulnerabilità, ma solo alcuni dei

fallimenti sono per una ragione diversa. Questo numero arriva a 6 se consideriamo anche i mutanti che non passano alcuni test aggiuntivi rispetto alle vulnerabilità. Nel complesso, i mutanti di tipo test coupling possono aiutare a rilevare deviazioni di programma molto simili, ma non uguali alla vulnerabilità, il che li rende un insieme molto promettente per valutare la sicurezza.

- **Partial Coupling:** per 2 progetti testati, i mutanti generati da Gemini si accoppiano parzialmente con la rispettiva vulnerabilità. Questo inferisce che questi mutanti falliscono alcuni dei test (ma non tutti) che vengono attivati dalle vulnerabilità e anche per le stesse ragioni. Uno di questi progetti oltre a fallire alcuni dei test, fallisce anche su altri test aggiuntivi rispetto a quelli della vulnerabilità. Questi 2 mutanti possono servire come requisiti di test parziali per la progettazione di test suite specifiche per la sicurezza.
- **Partial Test Coupling:** questa categoria riguarda i mutanti che falliscono sugli stessi test della vulnerabilità indipendentemente dalla motivazione per cui il test fallisce. Gemini non ha prodotto mutanti che rompono alcuni dei test innescati anche dalla vulnerabilità, indipendentemente dal motivo del fallimento.

Da questi risultati si può definire il principale finding di questo esperimento che pone le basi per rispondere alla research question:

 **Finding 1.** La sperimentazione di base condotta con Gemini, avendo in input il prompt 1 mostra risultati promettenti nel campo del Security Mutation Testing, in particolare nella generazione di mutanti fortemente accoppiati alla vulnerabilità in analisi.

Nella Tabella 4.1 è possibile visionare i risultati ottenuti per ogni vulnerabilità ed ogni classe di accoppiamento analizzata, considerando anche il totale e i mutanti creati da Gemini per cui il progetto ha fallito nella compilazione.

Tabella 4.1: Risultati ottenuti dalla valutazione dei mutanti generati mostrando la distribuzione nelle classi di accoppiamento.

vul_id	CE	MNK	NS	PS	PS+ATF	PTS	SS	SS+ATF	TS	TS+ATF	Totale
VUL4J-1	3	3	0	1	0	0	2	0	0	1	10
VUL4J-10	0	1	0	9	0	0	0	0	0	0	10
VUL4J-11	17	0	0	0	0	0	3	0	0	0	20
VUL4J-13	10	0	0	0	0	0	0	0	0	0	10
VUL4J-17	10	0	0	0	0	0	0	0	0	0	10
VUL4J-20	8	0	0	0	0	1	0	0	1	0	10
VUL4J-22	6	0	0	0	0	0	1	1	1	1	10
VUL4J-24	10	0	0	0	0	0	0	0	0	0	10
VUL4J-25	10	0	0	0	0	0	0	0	0	0	10
VUL4J-26	10	0	0	0	0	0	0	0	0	0	10
VUL4J-29	10	0	0	0	0	0	0	0	0	0	10
VUL4J-30	7	0	0	0	0	0	2	1	0	0	10
VUL4J-31	9	0	0	0	0	0	0	0	1	0	10
VUL4J-34	2	1	1	6	0	0	0	0	0	0	10
VUL4J-36	0	4	0	0	0	0	6	0	0	0	10
VUL4J-38	12	0	0	0	0	0	8	0	0	0	20
VUL4J-40	0	0	0	9	0	0	0	0	1	0	10
VUL4J-41	10	0	0	0	0	0	0	0	0	0	10
VUL4J-42	20	0	0	0	0	0	9	1	0	0	30
VUL4J-43	10	0	0	0	0	0	0	0	0	0	10
VUL4J-44	0	0	0	3	2	0	2	3	0	0	10
VUL4J-45	7	0	0	0	0	0	3	0	0	0	10
VUL4J-46	5	0	0	0	0	0	2	0	0	3	10
VUL4J-47	7	0	0	0	0	2	1	0	0	0	10
VUL4J-48	9	0	0	1	0	0	0	0	0	0	10
VUL4J-49	10	0	0	0	0	0	0	0	0	0	10
VUL4J-50	0	3	0	0	0	2	5	0	0	0	10
VUL4J-57	4	0	0	0	0	0	6	0	0	0	10
VUL4J-68	14	0	0	2	0	0	3	1	0	0	20
VUL4J-76	8	0	0	1	0	0	1	0	0	0	10
VUL4J-77	10	0	0	0	0	0	0	0	0	0	10
VUL4J-9	10	0	0	0	0	0	0	0	0	0	10
Totale	150	10	5	45	5	3	50	10	15	5	298
%	50.34%	3.36%	1.68%	15.10%	1.68%	1.01%	16.78%	3.36%	5.03%	1.68%	100%

Mostrata questa panoramica sui risultati ottenuti dal modello Gemini, di tipo ALM, avendo in input il prompt che fornisce informazioni di base per la generazione dei mutanti è possibile dare una risposta alla prima research question:

🔗 **Answer to RQ₁.** Gemini si dimostra efficace nella produzione di mutanti per il Security Mutation Testing, in particolare per quelli utili al testing circa il 60% mostrano un accoppiamento forte con la rispettiva vulnerabilità. Questo consente al tester di trovare potenzialmente ulteriori vulnerabilità nel codice.

4.2 RQ2: Prompt Engineering

Questa domanda di ricerca si pone come obiettivo quello di provare diversi prompt per lo stesso modello cercando di migliorare le prestazioni ottenute. In particolare, come descritto in precedenza, il modello potrebbe comportarsi in modo migliore sulla base di stimoli emotivi. L'aggiunta di questo stimolo emotivo è stata effettuata sul prompt 2 sulla base del prompt 1. I risultati ottenuti dalla sperimentazione con questo tipo di prompt sono effettivamente quelli attesi. Il modello si

comporta molto meglio nella generazione dei mutanti, in particolare nella generazione di mutanti fortemente accoppiati alla vulnerabilità in quanto si passa da 25 mutanti di tipo Strong Coupling e Strong Coupling + Additional alla cifra di 30, un miglioramento di circa il 10%. Nella Figura 4.2 è possibile visionare la distribuzione dei mutanti, essa infatti è molto vicina alla distribuzione avuta per μ BERT.

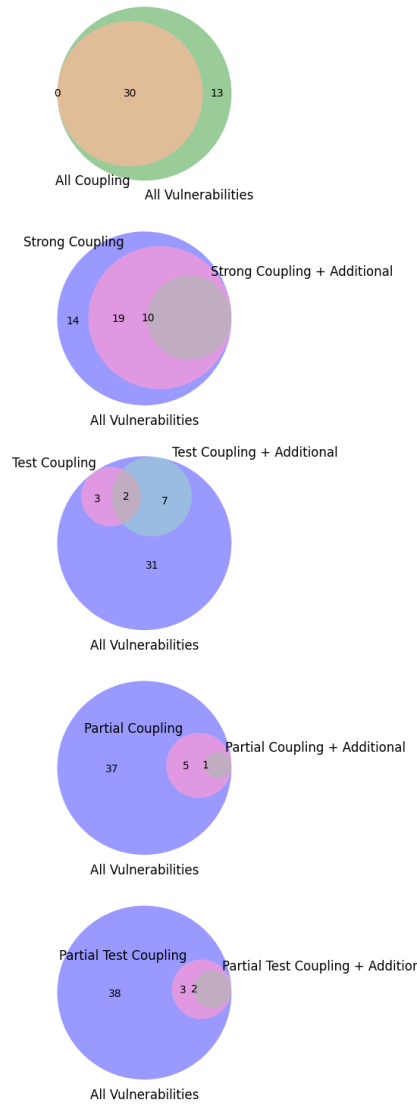


Figura 4.2: Diagramma di Venn contenente le classi di accoppiamento dei test dell'esperimento con Gemini e prompt 2.

In conclusione quindi, osservando i risultati ottenuti dai due diversi prompt per il modello Gemini è possibile dare una risposta alla research question:

🔗 **Answer to RQ₂.** L'utilizzo di due prompt differenti ha mostrato una differenza sostanziale nella produzione dei mutanti: l'introduzione di uno stimolo emotivo migliora di molto la bontà dei mutanti generati soprattutto nella generazione di mutanti fortemente accoppiati alla vulnerabilità in analisi.

4.3 RQ3: Confronto tra modelli

La risposta a questa domanda può essere facilmente ricavata mettendo a confronto i risultati ottenuti dall'esperimento con Gemini e ChatGPT con i risultati ottenuti dall'esperimento con μ BERT nel paper di riferimento [1]. La distribuzione in classi di accoppiamento dei mutanti generati con μ BERT è rappresentata nella Figura 4.3

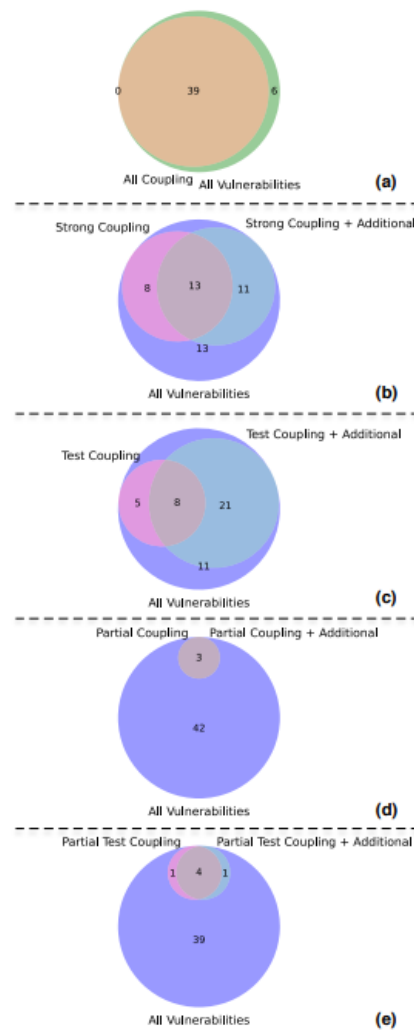


Figura 4.3: Diagramma di Venn contenente le classi di accoppiamento dei test della sperimentazione condotta da Garg et. al [1].

Il secondo finding dell'esperimento risulta essere:

🔍 Finding 2. Nonostante le buone prestazioni ottenute tramite l'esperimento condotto su Gemini e il prompt 1, μ BERT risulta essere leggermente migliore.

Da questa scoperta ci si chiede quindi con un prompt diverso ma soprattutto un modello diverso come cambia il confronto rispetto al modello di tipo MLM. Saranno di seguito riportati i risultati ottenuti con ChatGPT. Nella Figura 4.4 è possibile visionare i risultati ottenuti da questo esperimento. Principalmente, ChatGPT si colloca, considerando l'utilità dei mutanti generati, tra i due esperimenti condotti con Gemini. A parità di prompt infatti, Gemini si comporta meglio rispetto a ChatGPT, che a sua volta è migliore rispetto a Gemini nell'esperimento condotto con il prompt 1.

Dall'analisi effettuata si ricava un altro finding:

🔍 Finding 3. Confrontando diversi modelli sullo stesso prompt i risultati sono simili ma Gemini è più efficace di ChatGPT.

In definitiva, sulla base degli esperimenti condotti tra diversi modelli, confrontandoli con i risultati ottenuti da Garg et al. [1] la risposta alla seconda research question risulta:

👉 Answer to RQ₃. I modelli di tipo MLM risultano leggermente migliori, anche se molto simili nella produzione di mutanti fortemente accoppiati, rispetto a quelli di tipo ALM testati, ovvero Gemini e ChatGPT. Sulla base dei risultati ottenuti però risulta fondamentale il prompt dato in input e il preprocessing effettuato sui singoli progetti. In un contesto reale, non di ricerca, si può adattare il prompt e il preprocessing necessario in modo specifico per avere risultati migliori.

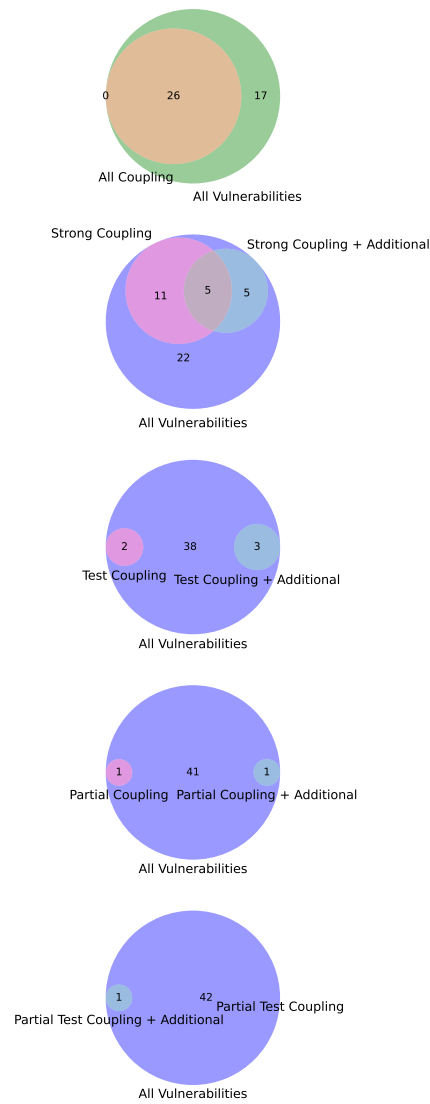


Figura 4.4: Diagramma di Venn contenente le classi di accoppiamento dei test della sperimentazione condotta con ChatGPT sul prompt2.

CAPITOLO 5

Conclusioni

Questa tesi ha dimostrato il potenziale innovativo dei Large Language Models nel Security Mutation Testing, un ambito in cui l'innovazione è fondamentale per affrontare la crescente complessità del software e la necessità di garantire sicurezza e qualità. Gli esperimenti condotti hanno evidenziato che i LLM, come Gemini e ChatGPT, possono generare mutanti in modo più rapido e diversificato rispetto ai metodi tradizionali. Ciò si traduce in una maggiore capacità delle suite di test di rilevare vulnerabilità critiche, contribuendo significativamente alla robustezza dei sistemi software.

Le tecniche di prompt engineering e preprocessing hanno giocato un ruolo chiave nell'ottimizzazione dei risultati, confermando che un uso strategico dei modelli può massimizzare il loro impatto. Tuttavia, la ricerca ha anche messo in luce alcune limitazioni, ovvero la difficoltà degli LLM nel comprendere codice particolarmente complesso o specifico di dominio e il consumo elevato di risorse computazionali rappresentano ostacoli rilevanti. Questi aspetti suggeriscono la necessità di ulteriori miglioramenti tecnici per rendere l'uso degli LLM più accessibile ed efficiente. Tuttavia, in una casistica reale e non di ricerca, è possibile adattare i prompt dati al modello e la progettazione in base al caso specifico e non a quello generale utilizzato per ottenere risultati tangibili sui progetti valutati all'interno del dataset VUL4J.

In un'ottica futura, si apre la strada a numerose opportunità di ricerca e applicazione. Un primo passo potrebbe essere l'integrazione tra LLM e strumenti tradizionali di mutation testing, sfruttando i punti di forza di entrambi per superare le rispettive debolezze. Inoltre, lo sviluppo di modelli più leggeri e specializzati attraverso tecniche come il fine-tuning su dataset mirati potrebbe ridurre significativamente il costo computazionale, rendendo la tecnologia accessibile anche in contesti industriali con risorse limitate.

Un'altra direzione interessante riguarda l'applicazione degli LLM in scenari reali, come le pipeline di Continuous Integration/Continuous Deployment (CI/CD), dove potrebbero automatizzare e potenziare il testing in ambienti di produzione.

Questa ricerca rappresenta una continuazione al processo che promette di trasformare e migliorare il modo in cui il testing del software viene realizzato, fornendo le basi per ampliare la varietà dei test implementati per un determinato progetto, fornendo una diversità tale da scovare possibili vulnerabilità software.

Bibliografia

- [1] A. Garg, R. Degiovanni, M. Papadakis, and Y. Le Traon, “On the coupling between vulnerabilities and llm-generated mutants: A study on vul4j dataset,” in *2024 IEEE Conference on Software Testing, Verification and Validation (ICST)*. IEEE, 2024, pp. 305–316. (Citato alle pagine iii, iv, 5, 35, 36, 37, 39, 46, 49, 50, 51, 56 e 57)
- [2] G. J. Myers, C. Sandler, and T. Badgett, *The art of software testing*. John Wiley & Sons, 2011. (Citato alle pagine 2, 8, 9 e 13)
- [3] Q.-C. Bui, R. Scandariato, and N. E. D. Ferreyra, “Vul4j: A dataset of reproducible java vulnerabilities geared towards the study of program repair techniques,” in *2022 IEEE/ACM 19th International Conference on Mining Software Repositories (MSR)*, 2022, pp. 464–468. (Citato alle pagine 5 e 41)
- [4] M. Pezzè and M. Young, *Software Testing and Analysis: Process, Principles, and Techniques*. Wiley India Pvt. Limited, 2008. [Online]. Available: <https://books.google.it/books?id=7x10CgAAQBAJ> (Citato alle pagine 10, 13 e 14)
- [5] T. J. Ostrand and M. J. Balcer, “The category-partition method for specifying and generating functional tests,” *Commun. ACM*, vol. 31, no. 6, p. 676–686, Jun. 1988. [Online]. Available: <https://doi.org/10.1145/62959.62964> (Citato a pagina 14)

-
- [6] P. Ammann and J. Offutt, *Introduction to software testing*. Cambridge University Press, 2017. (Citato a pagina 15)
- [7] Y. Jia and M. Harman, “An analysis and survey of the development of mutation testing,” *IEEE Transactions on Software Engineering*, vol. 37, no. 5, pp. 649–678, 2011. (Citato alle pagine 18, 19 e 24)
- [8] A. Offutt, “The coupling effect: fact or fiction,” *SIGSOFT Softw. Eng. Notes*, vol. 14, no. 8, p. 131–140, Nov. 1989. [Online]. Available: <https://doi.org/10.1145/75309.75324> (Citato a pagina 18)
- [9] W. E. Wong, *On mutation and data flow*. Purdue University, 1993. (Citato a pagina 20)
- [10] R. A. DeMillo, D. S. Guindi, W. McCracken, A. J. Offutt, and K. N. King, “An extended overview of the mothra software testing environment,” in *Workshop on Software Testing, Verification, and Analysis*. IEEE Computer Society, 1988, pp. 142–143. (Citato a pagina 21)
- [11] E. S. Mresa and L. Bottaci, “Efficiency of mutation operators and selective mutation strategies: An empirical study,” *Software Testing, Verification and Reliability*, vol. 9, no. 4, pp. 205–232, 1999. (Citato a pagina 21)
- [12] W. X. Zhao, K. Zhou, J. Li, T. Tang, X. Wang, Y. Hou, Y. Min, B. Zhang, J. Zhang, Z. Dong *et al.*, “A survey of large language models,” *arXiv preprint arXiv:2303.18223*, 2023. (Citato a pagina 25)
- [13] A. Vaswani, “Attention is all you need,” *Advances in Neural Information Processing Systems*, 2017. (Citato a pagina 25)
- [14] J. D. M.-W. C. Kenton and L. K. Toutanova, “Bert: Pre-training of deep bidirectional transformers for language understanding,” in *Proceedings of naacL-HLT*, vol. 1. Minneapolis, Minnesota, 2019, p. 2. (Citato a pagina 27)
- [15] T. B. Brown, “Language models are few-shot learners,” *arXiv preprint arXiv:2005.14165*, 2020. (Citato a pagina 27)

-
- [16] J. Kaplan, S. McCandlish, T. Henighan, T. B. Brown, B. Chess, R. Child, S. Gray, A. Radford, J. Wu, and D. Amodei, “Scaling laws for neural language models,” *arXiv preprint arXiv:2001.08361*, 2020. (Citato a pagina 28)
- [17] B. Wang, M. Chen, Y. Lin, M. Papadakis, and J. M. Zhang, “An exploratory study on using large language models for mutation testing,” *arXiv preprint arXiv:2406.09843*, 2024. (Citato a pagina 34)
- [18] F. Tip, J. Bell, and M. Schäfer, “Llmorpheus: Mutation testing using large language models,” *arXiv preprint arXiv:2404.09952*, 2024. (Citato a pagina 35)
- [19] C. Li, J. Wang, Y. Zhang, K. Zhu, W. Hou, J. Lian, F. Luo, Q. Yang, and X. Xie, “Large language models understand and can be enhanced by emotional stimuli,” 2023. [Online]. Available: <https://arxiv.org/abs/2307.11760> (Citato a pagina 39)
- [20] G. Team, R. Anil, S. Borgeaud, J.-B. Alayrac, J. Yu, R. Soricut, J. Schalkwyk, A. M. Dai, A. Hauth, K. Millican *et al.*, “Gemini: a family of highly capable multimodal models,” *arXiv preprint arXiv:2312.11805*, 2023. (Citato alle pagine 42 e 43)
- [21] X. Yue, Y. Ni, K. Zhang, T. Zheng, R. Liu, G. Zhang, S. Stevens, D. Jiang, W. Ren, Y. Sun *et al.*, “Mmmu: A massive multi-discipline multimodal understanding and reasoning benchmark for expert agi,” in *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, 2024, pp. 9556–9567. (Citato a pagina 43)
- [22] R. Degiovanni and M. Papadakis, “ μ bert: Mutation testing using pre-trained language models,” in *2022 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*. IEEE, 2022, pp. 160–169. (Citato a pagina 45)
- [23] G. Gay and A. Salahirad, “How closely are common mutation operators coupled to real faults?” in *2023 IEEE Conference on Software Testing, Verification and Validation (ICST)*. IEEE, 2023, pp. 129–140. (Citato a pagina 46)

Ringraziamenti

Ringraziamenti qui...