Alma Mater Studiorum · Università di Bologna

SCUOLA DI SCIENZE
Corso di Laurea Magistrale in Informatica

Machine Learning Vulneralibities Detection in SmartContracts .....

Relatore:
Chiar.mo Prof.
Stefano Ferretti

Correlatore:

Presentata da:
Marco Benito Tomasone

Sessione I
Anno Accademico 2023/2024

Alle nonne

Indice

Elenco delle figure

Capitolo 1

Introduzione

Nel mondo delle tecnologie blockchain, gli smart contracts hanno guadagnato popolarit grazie alla loro abilit di automatizzare e far rispettare gli accordi tra due soggetti senza la necessit di un intermediario. Negli ultimi anni una delle tecnologie che sono spopolate e sono arrivate sulla bocca di tutti sono sicuramente la Blockchain e gli SmartContracts. Questi ultimi sono dei contratti digitali che permettono di eseguire delle operazioni in modo automatico e trasparente. Questi contratti sono scritti in un linguaggio di programmazione e vengono eseguiti su una macchina virtuale. Una delle principali caratteristiche peculiari degli SmartContracts sicuramente la loro immutabilit, difatti una volta essere stati deployati sulla blockchain questi non possono pi essere modificati. Uno dei problemi principali degli SmartContracts la sicurezza. Infatti, essendo dei contratti che vengono eseguiti in modo automatico, possibile che ci siano delle vulnerabilit che possono essere sfruttate da malintenzionati. In questo lavoro di tesi verr presentato un metodo per rilevare le vulnerabilit presenti negli SmartContracts.

Capitolo 2

Lavori Correlati

Il tema della rilevazione delle vulnerabilit all'interno degli Smart Contracts un tema che ha guadagnato notevole importanza nel tempo, anche a seguito della grande diffusione della tecnologia blockhain. Proprio per questo motivo, sono stati sviluppati e proposti diversi approcci per la rilevazione automatica di vulnerabilit all'interno degli Smart Contracts. In questo capitolo verranno presentati alcuni dei lavori pi significativi che si sono occupati di questo tema. Tra i principali approcci proposti figurano gli approcci basati su analisi statica basati su tecniche di esecuzione simbolica. L'analisi statica si basa sull'esame del codice sorgente o bytecode di uno smart contract senza effettuarne l'esecuzione effettiva. Questo approccio consente di identificare potenziali problematiche senza la necessit di testare il codice in un ambiente reale. L'esecuzione simbolica una tecnica particolarmente potente in questo contesto in quanto consente di esplorare tutte le possibili esecuzioni del programma, consentendo di individuare vulnerabilit che potrebbero emergere solo in determinate condizioni. Gli approcci basati su esecuzione simbolica cercano di risolvere queste vulnerabilit attraverso la generazione di un grafo di esecuzione simbolico, in cui le variabili sono rappresentate come simboli e le esecuzioni possibili del programma vengono esplorate simbolicamente. Ci consente di identificare percorsi di esecuzione che potrebbero condurre a condizioni di errore o vulnerabilit. Tuttavia, va notato che l'analisi statica, inclusa l'esecuzione simbolica, pu essere complessa e non sempre completa. Alcune vulnerabilit potrebbero sfuggire a questa analisi o richiedere ulteriori tecniche di verifica. Pertanto, consigliabile combinare l'analisi statica con altre metodologie, come l'analisi dinamica e i test formali, per garantire una copertura completa nella rilevazione di vulnerabilit negli smart contract. Tra i principali strumenti che utilizzano questo tipo di analisi ci sono Maian [21, 20], Oyente [2, 17], Mythril [1], Manticore [19] e altri. Un'altro tipo di approcci ad analisi statica sono i tools basati su regole. Questi strumenti usano un set di regole predefinite e pattern per identificare delle potenziali nvulnerabilit nel codice sorgente. Questi tool analizzano il codice sorgente e segnalano tutte le istanze dove il codice viola delle regole predefinite. Le regole sono tipicamente basate su delle vulnerabilit note e delle best prac-

tice di programmazione, come ad esempio evitare dei buffer overflow, usare algoritmi di cifratura sicuri e validare propriamente l'input degli utenti. La limitazione principale di questi strumenti che i risultati che producono sono limitati al set di regole che stato implementato, quindi non riescono a riconoscere delle nuove vulnerabilit o vulnerabilit non scoperte precedentemente. Inoltre, un'altra grande limitazione di questi strumenti il fatto che possano produrre un alto numero di falsi positivi, cio di situazioni in cui il codice viene segnalato come codice vulnerabile ma in realt codice perfettamente sano. Tra i principali strumenti che utilizzano questo tipo di analisi ci sono Slither [7], Securify [29], SmartCheck [28] e altri. Un'altra categoria di strumenti per la rilevazione di vulnerabilit negli smart contract sono gli approcci basati su tecniche di Machine Learning e Deep Learning, tra le quali anche il lavoro di questa tesi va ad inserirsi. Un approccio basato sulla trasformazione degli opcode dei contratti e la sua relativa analisi con dei modelli di Machine Learning tradizionale stato offerto da Wang et al. [31] i quali hanno raggiunto risultati eccellenti, con risultati in termini di F1 Score superiori al 95% in quasi tutte le classi prese in analisi con il modello XGBoost che risultato il miglior modello. Un altro lavoro che sfrutta tecniche di Machine Learning pi tradizionali il lavoro di Mezina e Ometov che hanno utilizzato classificatori come RandomForest, LogisticRegressio, KNN, SVM in un approccio dapprima binario (valutare se il contratto abbia o meno delle vulnerabilit) e poi multiclasse (valutare quale tipo di vulnerabilit il contratto abbia) [18]. I risultati in questo caso hanno mostrato come il modello SVM sia quello che ottiene i migliori risultati in termini di accuratezza.

Spostandoci su lavori che utilizzano tecniche di Deep Learning importante citare un'altro lavoro effettuato sullo stesso dataset su cui basato questo lavoro di tesi. Questo dataset stato infatti raccolto e pubblicato da un gruppo di ricercatori dell'Universit di Bologna che ha effettuato un primo studio utilizzando un approccio basato su reti neurali convoluzionali [25]. L'approccio in questo caso stato quello di classificare le vulnerabilit in un'impostazione multilabel del problema (in cui la label da predire un array di elementi, quindi in cui il contratto pu appartenere contemporaneamente a pi classi) itilizzando delle reti neurali convoluzionali per la rilevazione delle vulnerabilit trasformando in codice Bytecode espresso in esadecimale dei contratti in delle immagini RGB. Questo lavoro ha come risultato principale la dimostrazione che utilizzando delle reti neurali convoluzionali possibile rilevare le vulnerabilit presenti negli SmartContracts con delle buone performance, i migliori risultati si attestano con un MicroF1 score del 0.83% e mostrano come i migliori risultati siano dati da delle resi Resnet con delle convoluzioni unidimensionali. Successivamente, gli stessi autori hanno pubblicato una seconda analisi effettuata sul dataset utilizzando nuovi classificatori come CodeNet, SvinV2-T e Inception, mostrando come i migliori risultati continuino ad essere quelli forniti da reti convoluzionali unidimensionali [**?**]. Altri lavori che utilizzano un approccio basato su tecniche di Deep Learning il lavoro proposto da Huang [11] che utilizza anch'egli delle reti neurali convoluzionali per la rilevazione delle vulnerabilit. I modelli utilizzati sono modelli molto noti come Alexnet, GoogleNet e Inception v3, i risultati migliori in questo

caso si attestano sul 75%. Un importante lavoro offerto da Deng et Al. [4] ha proposto un approccio basato sulla fusione di feature multimodali, analizzando contemporaneamente codice sorgente, bytecod e grafi di controllo del flusso. Per ognuna di queste tre feature stato trainato un semplice classificatore con una rete neurale feedforward e sono poi state combinate le predizioni di questi tre classificatori in un classificatore finale utilizzando un approccio di ensamble learning detto stacking. I risultati ottenuti mostrano come l'approccio proposto abbia ottenuto dei risultati migliori rispetto ad un approccio in cui si utilizzava solo una delle tre feature.

Capitolo 3

Dataset

3.0.1 Vulnerabilit

Access-Control

La vulnerabilit di access control una vulnerabilit presente non solo in Solidity ma in numerosi altri linguaggi di programmazione. Questa vulnerabilit si verifica quando un contratto non controlla correttamente l'accesso alle sue funzioni e ai suoi dati, quindi una vulnerabilit legata al governanre chi pu interagire con le varie funzionalit all'interno del contratto. Un esempio di questo dipo di vulnerabilit legato alla mancata restrizione dell'accesso a funzioni di inizializzazione, ad esempio:

```
function initContract() public {
    owner = msg.sender;
}
```

Questa funzione serve a inizializzare l'owner del contratto, ma non controlla chi pu chiamarla, permettendo a chiunque di chiamarla e diventare l'owner del contratto e non ha nemmeno controlli per prevenire la reinizializzazione. Questo un esempio molto semplice di come una vulnerabilit di access control possa portare a comportamenti inaspettati. Un famoso attacco che ha subito una vulnerabilit di tipo access-control il caso di Parity Multisig Wallet, un contratto che permetteva di creare wallet multi firma. Questo contratto ha subito un attacco nel Luglio 2017 che ha portato alla perdita di una grande quantit di Ether. L'attacco stato effettuato da un utente che ha sfruttato una vulnerabilit di access control per diventare l'owner del contratto e rubare criptovalute ad altri utenti, si stima che la perdita sia stata di circa 30 milioni di dollari.

Arithmetic

Le vulnerabilit di tipo aritmetico [14] sono vulnerabilit che vengono generate come risultato di operazioni matematiche. Una delle vulnerabilit pi significative all'interno di

questa classe rappresentata dagli underflow/overflow, un problema molto comune nei linguaggi di programmazione. Incrementi di valore di una variabile oltre il valore massimo rappresentabile o decrementi al di sotto del valore minimo rappresentabile (detti *wrap around*) possono generare comportamenti indesiderati e risultati errati. In tutte le versioni di Solidity precedenti alla versione 0.8.0, le operazioni aritmetiche non controllano i limiti di overflow e underflow previsti per quel tipo di dato (es.uint64 o uint256), permettendo a un attaccante di sfruttare questa vulnerabilit per ottenere un vantaggio. Ad esempio nel caso in cui si stia utilizzano un uint256 il massimo numero che si piu memorizzare nella variabile $2^256 - 1$, che un numero molto alto, ma resta comunque possibile superare questo limite, facendo entrare in scena l'overflow. Quando si verifica un overflow, il valore della variabile riparte dal piu piccolo valore rappresentabile. Questo pu portare a comportamenti inaspettati e a perdite di fondi. Vediamo un esempio molto banale di contratto vulnerabile:

```
pragma solidity 0.7.0;

contract ChangeBalance {
    uint8 public balance;
    function decrease() public {
        balance--;
    }
    function increase() public {
        balance++;
    }
}
```

Questo codice rappresenta un contratto che molto semplicemente memorizza un saldo all'interno di una variabile di tipo uint8, cio un intero a 8bit ovvero un intero che pu memorizzare valori da 0 a $2^8 - 1$, quindi da 0 a 255. Se un utente chiamasse la funzione `increase()` in modo tale che faccia salire il valore del saldo a 256 il calcolo risulterebbe in un overflow e il valore della variabile ritornerebbe a 0. Questo un esempio molto semplice di come un overflow possa portare a comportamenti inaspettati. L'underflow si verificherebbe nel caso diametralmenmte opposto, in cui viene chiamata la funzione `decrease()` quando il saldo a 0. In questo caso il valore della variabile ritornerebbe a 255. Un esempio di attacco che sfrutta l'overflow stato l'attacco del 23 Aprile 2018 effettuato su uno smart contract di BeautyChain (BEC) che ha causato un importantissimo crash del prezzo. La funzione che ha causato l'overflow permetteva di trasferire una certa somma di denaro presa in input a pi utenti contemporaneamnete e per farlo controllava dapprima che il saldo del contratto fosse maggiore o uguale alla somma da trasferire:

```
function batchTransfer(address[] _receivers, uint256 _value)
   public whenNotPaused returns (bool) {
    uint cnt = _receivers.length;
```

```
        uint256 amount = uint256(cnt) * _value;
        require(cnt > 0 && cnt <= 20);
        require(_value > 0 && balances[msg.sender] >= amount);

        balances[msg.sender] = balances[msg.sender].sub(amount);
        for (uint i = 0; i < cnt; i++) {
            balances[_receivers[i]] = balances[_receivers[i]].add
                (_value);
            Transfer(msg.sender, _receivers[i], _value);
        }
        return true;
    }
```

Figura 3.1: Andamento del prezzo di BEC prima e dopo l'attacco

Dalla versione di Solidity 8.0 tutti i calcoli che superano i limiti di rappresentazione del tipo di dato vengono interrotti e viene lanciata un'eccezione. Questo permette di evitare che si verifichino overflow e underflow. Un'altra soluzione a questo tipo di errori l'utilizzo della libreria SafeMath, che offre operazioni aritmetiche che controllano i limiti di rappresentazione del tipo di dato prima di effettuare i calcoli. Overflow e underflow sono le principali vulnerabilit di tipo aritmetico, ma non sono le uniche. Un'altra vulnerabilit di tipo aritmetico rappresentata dalla divisione per zero. In Solidity nelle versioni

precedenti alla 0.4 la divisione per zero non lancia un'eccezione, ma ritorna il valore 0, questo pu portare a comportamenti inaspettati e a perdite di fondi.

Other

In questa classe di vulnerabilit rientrano tutte quelle vulnerabilit che non fanno parte delle altre classi. Uno degli esempi sono le vulnerabilit di *uninitialized-state* che fanno capo a tutte quelle vulnerabilit a seguito di variabili che non vengono inizializzate correttamente. Le variabili in Solidity possono essere memorizzate in *memory*, *storage* o *calldata*. Bisogna assicurarsi che questi diversi storage vengano compresi e inizializzati correttamente, poich ad esempio inzializzare male i puntatori allo storage o lasciarli non inizializzati pu portare a degli errori. Da Solidity 0.5.0 i puntatori allo storage non inizializzati non sono pi un problema poich contratti con puntatori non inizializzati risulteranno in errori di compilazione. Un altra possibile vulnerabilit detta *incorrect-equality*. Questa vulnerabilit si verifica, solitamente, quando si controlla affinche un account ha abbastanza Ether o Tokens utilizzando una uguaglianza stretta, ci un qualcosa che un soggetto malevolo pu facilmente manipolare per attaccare il contratto. Un esempio di questo tipo di vulnerabilit il caso in cui il contratto entra in uno stato di *GridLock*:

```
    /**
    * @dev        Locks up the value sent to contract in a new
       Lock
    * @param      term         The length of the lock up
    * @param      edgewareAddr The bytes representation of the
       target edgeware key
    * @param      isValidator  Indicates if sender wishes to be a
        validator
    */
   function lock(Term term, bytes calldata edgewareAddr, bool
      isValidator)
       external
       payable
       didStart
       didNotEnd
   {
       uint256 eth = msg.value;
       address owner = msg.sender;
       uint256 unlockTime = unlockTimeForTerm(term);
       // Create ETH lock contract
       Lock lockAddr = (new Lock).value(eth)(owner, unlockTime);
       // ensure lock contract has all ETH, or fail
       assert(address(lockAddr).balance == msg.value); // BUG
```

```
        emit Locked(owner, eth, lockAddr, term, edgewareAddr,
            isValidator, now);
    }
```

In questo caso la vulnerabilit rappresentata dall'assert che controlla che il contratto abbia ricevuto la quantit di Ether corretta. Il controllo si basa sull'assunzione che il contratto essendo creato alla riga precedente abbia saldo zero ed essendo precaricato proprio con *msg.value* si suppone che il saldo del contratto sia uguale a *msg.value*. In realt gli Ether possono essere inviati ai contratti prima che vengano istanziati negli indirizzi stessi, poich la generazione degli indirizzi dei contratti in Ethereum un processo basato su dei nonce deterministici. L'attacco DoS che si basa su questa vulnerabilit in questo caso consiste nel pre-calcolare l'indirizzo del contatto *Lock* e mandare dei Wei a quell'indirizzo. Questo forza la funzione *lock* a fallire e a non creare il contratto, bloccando il contratto in uno stato di *GridLock*. Per risolvere questa problematica, si potrebbe adottare l'approccio di sostituire l'uguaglianza stretta con un confronto maggiore o uguale.

Reentrancy

La Reentrancy una classe di vulnerabilit presente negli SmartContracts che permette ad un malintenzionato di rientrare nel contratto in modo inaspettato durante l'esecuzione della funzione originale. Questa vulnerabilit pu essere utilizzata per rubare fondi e rappresenta la vulnerabilit pi impattante dal punto di vista di perdita di fondi a seguito di attacchi. Il caso pi famoso di questo attacco che lo ha anche reso noto il caso di The DAO, un contratto che ha subito un attacco di reentrancy che ha portato alla perdita circa sessanta milioni di dollari in Ether, circa il 14% di tutti gli Ether in circolazione all'epoca. Nonostante dal 2016 ad oggi siano stati fatti numerosi progressi nelle tecnologie e nelle misure di sicurezza questa vulnerabilit rimane comunque una delle minacce pi pericolose per gli SmartContracts, poich negli anni questo tipo di attacchi si ripresentato notevole frequenza [23]. Un attacco di reentrancy pu essere classificato in tre classi differenti:

- **Mono-Function**: la funzione vulnerabile la stesa che viene chiamata pi volte dall'attaccante, prima del completamento delle sue invocazioni precedenti. Questo il caso pi semplice di attacco reentrancy e di conseguenza il pi facile da individuare.
- **Cross-Function**: questo caso molto simile al caso di mono-function Reentrancy, ma in questo caso la funzione che viene chiamata dall'attaccante non la stessa che fa la chiamata esterna. Questo tipo di attacco possibile solo quando una funzione vulnerabile condivide il suo stato con un'altra funzione, risultando in un una situazione fortemente vantaggiosa per l'attaccante.
- **Cross-Contract**: questo tipo di attacco prende piede quando lo stato di un contratto invocato in un altro contratto prima che viene correttamente aggiornato.

Avviene solitamente quando pi contratti condividono una variabile di stato comune e uno di loro la aggiorna in modo non sicuro.

Mostreremo adesso alcuni esempi di contratti vulnerabili a questo tipo di attacco.

```
// UNSECURE
function withdraw() external {
    uint256 amount = balances[msg.sender];
    (bool success,) = msg.sender.call{value: balances[msg.sender
       ]}("");
    require(success);
    balances[msg.sender] = 0;
}
```

In questo caso, il balance dell'utente viene aggiornato solo dopo che la chiamata esterna stata completata. Questo permette all'attaccante di chiamare la funzione withdraw pi volte prima che il balance venga settato a zero, permettendo all'attaccante di rubare fondi allo smart contract. Una versione pi complessa dello stesso processo il caso cross function, di cui mostriamo un esempio:

```
    // UNSECURE
function transfer(address to, uint amount) external {
  if (balances[msg.sender] >= amount) {
    balances[to] += amount;
    balances[msg.sender] -= amount;
  }
}

function withdraw() external {
  uint256 amount = balances[msg.sender];
  (bool success,) = msg.sender.call{value: balances[msg.sender]}(
     "");
  require(success);
  balances[msg.sender] = 0;
}
```

In questo esempio, l'attaccante pu effettuare un attacco di tipo reentrancy avendo una avendo una funzione che chiama `transfer()` per trasferire fondi spesi prima che il bilancio sia settato a zero dalla funzione `withdraw()`. Un nuovo tipo di attacchi sono gli attacchi Read-only Reentrancy, in

Unchecked-Calls

In solidity si possono usare delle chiamate a funzione low level come 'address.call()'

Capitolo 4

Metodologia

Questa sezione descrive in dettaglio l'approccio seguito per affrontare il problema della classificazione degli smart contracts. Questo capitolo fondamentale per comprendere come sono stati raccolti, pre-processati e utilizzati i dati, come sono stati configurati e addestrati i modelli e quali strumenti e tecniche sono stati impiegati per ottenere i risultati presentati.

La metodologia adottata in questa tesi suddivisa nelle seguenti fasi principali:

1. Raccolta e preparazione dei dati: esplorazione del dataset utilizzato, delle tecniche di pre-processing applicate e delle modalit di suddivisione dei dati per l'addestramento e la valutazione.

2. Modellazione: descrizione dell'architettura dei modelli utilizzati, delle scelte di configurazione e delle strategie di addestramento. .

Ogni fase sar trattata in modo dettagliato, evidenziando le scelte metodologiche compiute e le motivazioni alla base di tali scelte. Questo approccio sistematico garantisce trasparenza e replicabilit del lavoro svolto, consentendo ad altri di comprendere e, eventualmente, replicare i risultati ottenuti.

4.1 Esplorazione dei dati

Il dataset [24] utilizzato in questo progetto un dataset disponibile pubblicamente sulla piattaforma HuggingFace una delle pi importanti piattaforme per il Natural Language Processing. HF un'infrastruttura open-source che fornisce accesso a una vasta gamma di modelli di deep learning pre-addestrati, tra cui alcuni dei pi avanzati nel campo del NLP. Questo dataset contiene informazioni su 106.474 SmartContracts pubblicati sulla rete Ethereum. Ogni elemento nel dataset composto da quattro elementi:

- **Address**: l'indirizzo del contratto

- **SourceCode**: il codice sorgente del contratto, scritto in linguaggio Solidity
- **ByteCode**: il codice bytecode del contratto, ottenuto a partire dalla compilazione del codice sorgente utilizzando il compilatore di Solidity. Questo bytecode quello che viene eseguito sulla macchina virtuale di Ethereum (EVM).
- **Slither**: il risultato dell'analisi statica del contratto con Slither, un tool open-source per l'analisi statica di contratti scritti in Solidity. Questo risultato un array di valori che vanno da 1 a 5, dove ogni numero rappresenta la presenza di una vulnerabilit e 4 rappresenta un contratto safe, cio privo di vulnerabilit.

Le vulnerabilit che sono state prese in questo lavoro sono le seguenti:

- Access-Control
- Arithmetic
- Other
- Reentrancy
- Safe
- Unchecked-Calls

Prima della costruzione dei modelli stata affrontata una fase di analisi esplorativa dei dati. Questa fase stata svolta per comprendere meglio la struttura del dataset e dei contratti da classificare, per individuare eventuali problemi. A livello pratico, questa fase di analisi esplorativa dei dati stata eseguita utilizzando il linguaggio Python, con l'ausilio di librerie come Pandas, NumPy, Matplotlib e Seaborn per l'analisi e la visualizzazione dei dati. Il dataset diviso in tre sottoinsiemi: training, validation e test set. Il dataset di training composto da 79.641 contratti, il dataset di validazione da 10.861 contratti e il dataset di test da 15.972 contratti. Tutte le informazioni sono presenti per tutti i contratti tranne l'informazione relativa al bytecode, che risulta essere assente per pochissimi contratti come visibile nella Tabella 4.1. Per ottenere una visione d'insieme

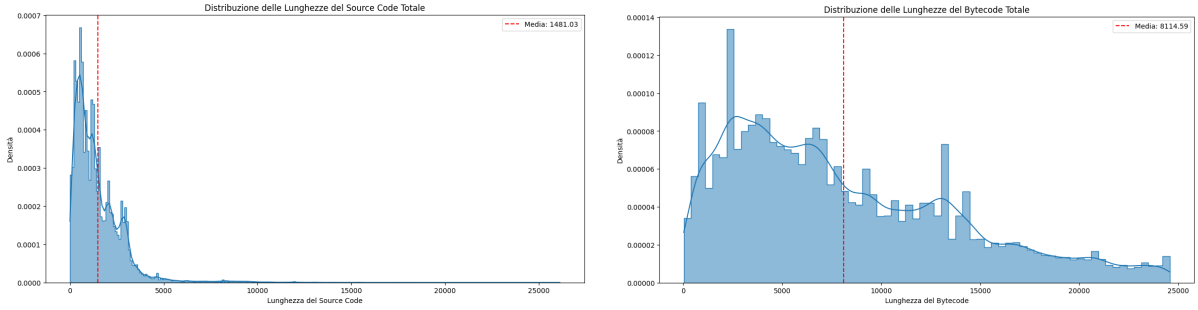
Dataset	**Count**	**%**
Train	227	0.285%
Test	51	0.319%
Validation	30	0.276%

Tabella 4.1: Conteggio e Percentuale di Contratti Senza Bytecode per Dataset

delle lunghezze dei contratti, abbiamo calcolato la lunghezza media del source code e

del bytecode. Prima del preprocessing le lunghezze medie di SourceCode e ByteCode sono rispettivamente di 3155 token e 8114 token. Abbiamo visualizzato la distribuzione delle lunghezze del source code utilizzando un istogramma. Per migliorare la leggibilit del grafico, abbiamo raggruppato i dati per quanto riguarda il source code in intervalli di 500 token. L'istogramma accompagnato da una linea che indica la lunghezza media dei token rappresentata con una linea tratteggiata rossa.

L'inclusione delle lunghezze medie fornisce un punto di riferimento utile per interpretare le distribuzioni e confrontare i singoli esempi di codice rispetto alla media del dataset. Queste analisi sono fondamentali per le successive fasi di preprocessing e modellazione, garantendo che i modelli possano gestire efficacemente la variabilit presente nei dati. Sul bytecode non stato applicato nessun tipo di preprocessing per ridurre la dimensione dei dati. Per quanto riguarda il codice sorgente sono stati eliminati tutti i commenti e le funzioni getter monoistruzione, cio tutte quelle funzioni `getX()` le quali abbiano come unica istruzione una istruzione di return, poich sono state assunte come funzioni corrette, l'eliminazione di queste stringhe avvenuta tramite una ricerca delle stringhe effettuata con una regex. Abbiamo unito i set di dati di addestramento, test e validazione in un unico DataFrame per analizzare le lunghezze del source code e del bytecode. In particolare, sono state calcolate rispettivamente le lunghezze del codice sorgente e del bytecode. Effettuando le rimozioni dei commenti la media del numero di token del sourcecode scende a 1511 token, mostrando come la rimozione dei commenti abbia un grande impatto sulla lunghezza media del codice. Rimuovendo anche le funzioni getter monoistruzione la lunghezza media del source code scende a 1481 token. Poich



(a) Distribuzione delle Lunghezze del Source Code dopo il preprocessing

(b) Distribuzione delle Lunghezze del Bytecode dopo il preprocessing

Figura 4.1: Distribuzioni delle lunghezze del source code e del bytecode.

successivamente andremo a classificare i contratti con dei modelli nella famiglia BERT che prendono in input sequenze di token lunghe al massimo 512 token abbiamo calcolato la percentuale di contratti che non superano questa soglia e in alcuni suoi multipli, per capire quanti contratti riusciamo a classificare per intero e quanti verranno troncati. I risultati sono mostrati nella Tabella 4.2.

Metrica	Sotto 512	Sotto 1024	Sotto 1536	Media
Source Code (%)	21.90	46.04	64.77	62.21
Bytecode (%)	1.56	6.31	8.75	58.69

Tabella 4.2: Percentuale di contratti sotto varie lunghezze in token.

Diventa per importante notare, che per molti casi di contratti che superano i 5000 token questi sono cos lunghi poich riportano in calce al contratto anche il codice sorgente di librerie esterne, che non di interesse per la classificazione delle vulnerabilit.

4.1.1 Distribuzione delle Classi e Matrici di Co-occorrenza

Successivamente, la fase di esplorazione dei dati ha previsto l'analisi delle classi di vulnerabilit dei dati. In questa sezione, presentiamo la distribuzione delle classi e le matrici di co-occorrenza per i dataset di addestramento, test e validazione. Si precisa che i risultati di seguito proposti si riferiscono gi al dataset da cui sono stati sottratti i contratti privi di bytecode.

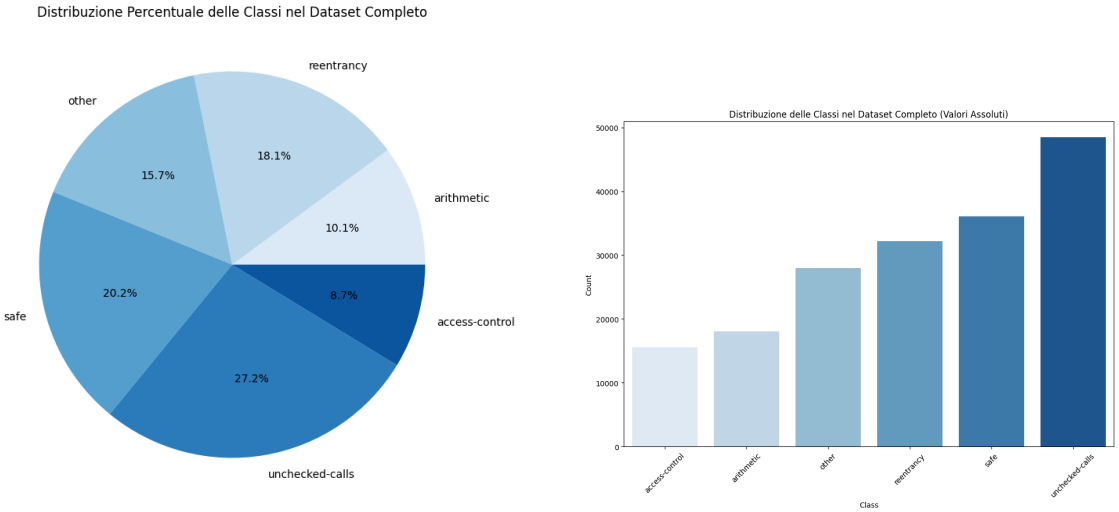
Distribuzione delle Classi

La Tabella 4.3 mostra la distribuzione delle classi per i tre dataset. evidente che la classe 'unchecked-calls' la pi frequente in tutti e tre i dataset, mentre la classe 'access-control' la meno rappresentata.

Class	Train		Test		Validation		Full	
	Count	%	Count	%	Count	%	Count	%
access-control	11619	8.71%	2331	8.71%	1588	8.73%	15538	8.72%
arithmetic	13472	10.10%	2708	10.12%	1835	10.09%	18015	10.10%
other	20893	15.67%	4193	15.67%	2854	15.69%	27940	15.67%
reentrancy	24099	18.07%	4838	18.09%	3289	18.08%	32226	18.08%
safe	26979	20.23%	5405	20.20%	3676	20.21%	36060	20.23%
unchecked-calls	36278	27.21%	7276	27.20%	4951	27.21%	48505	27.21%

Tabella 4.3: Distribuzione delle Classi nei Dataset di Addestramento, Test, Validazione e Completo

Le Figure 4.2a e 4.2b mostrano rispettivamente la distribuzione percentuale e assoluta delle classi nell'intero dataset. Queste visualizzazioni forniscono una panoramica chiara della frequenza delle diverse classi all'interno del dataset, evidenziando le differenze di distribuzione tra le classi. Dalla distribuzione delle classi nei diversi dataset, possiamo osservare che:



(a) Distribuzione Percentuale delle Classi (b) Distribuzione Assoluta delle Classi

Figura 4.2: Distribuzioni delle Classi nell'intero dataset, in termini relativi e assoluti.

- Le classi sono distribuite in modo abbastanza uniforme nei dataset di addestramento, test e validazione, con percentuali simili tra i tre split per classe
- La classe 'unchecked-calls' la pi frequente in tutti e tre i dataset, con una presenza significativa soprattutto nel dataset di addestramento (36278 occorrenze).
- La classe 'access-control' la meno frequente, con il numero pi basso di occorrenze nel dataset di validazione (1588 occorrenze).
- Le classi 'safe' e 'reentrancy' sono anche abbastanza rappresentate

Matrici di Co-occorrenza

Le Tabelle 4.3, 4.4 e 4.5 mostrano le matrici di co-occorrenza per i dataset di addestramento, test e validazione rispettivamente. Le matrici di co-occorrenza indicano la frequenza con cui ogni coppia di classi appare insieme nello stesso elemento.

In questa sezione, vengono presentate le matrici di co-occorrenza per ogni split del dataset, mostrando sia in termini assoluti che relativi il numero di cooccorrenze tra le varie classi.

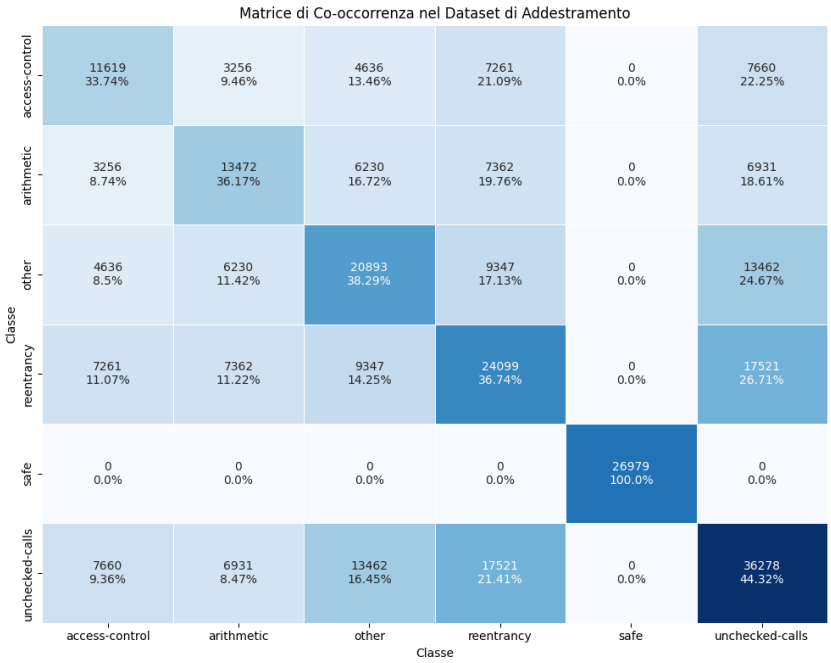


Figura 4.3: Matrice di Co-occorrenza nel Dataset di Addestramento

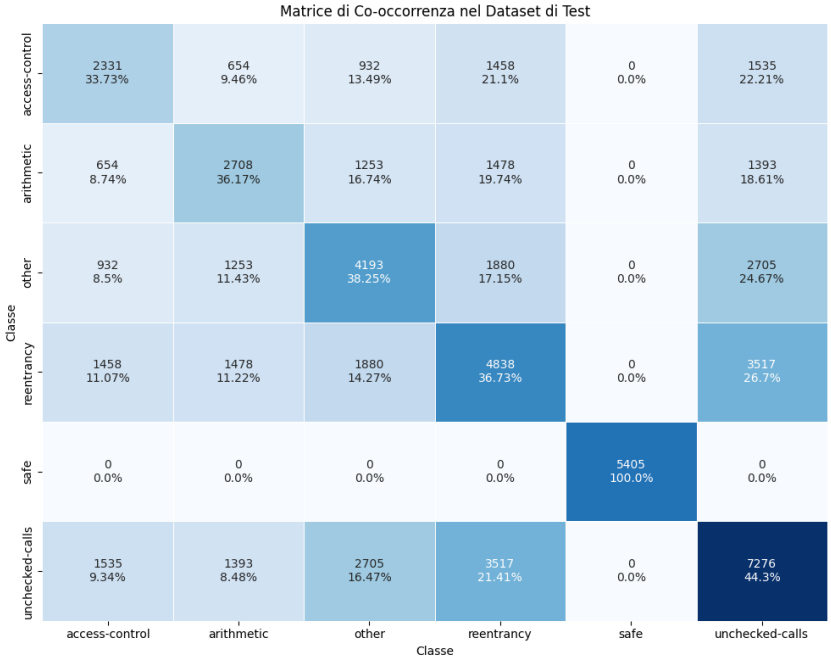


Figura 4.4: Matrice di Co-occorrenza nel Dataset di Test

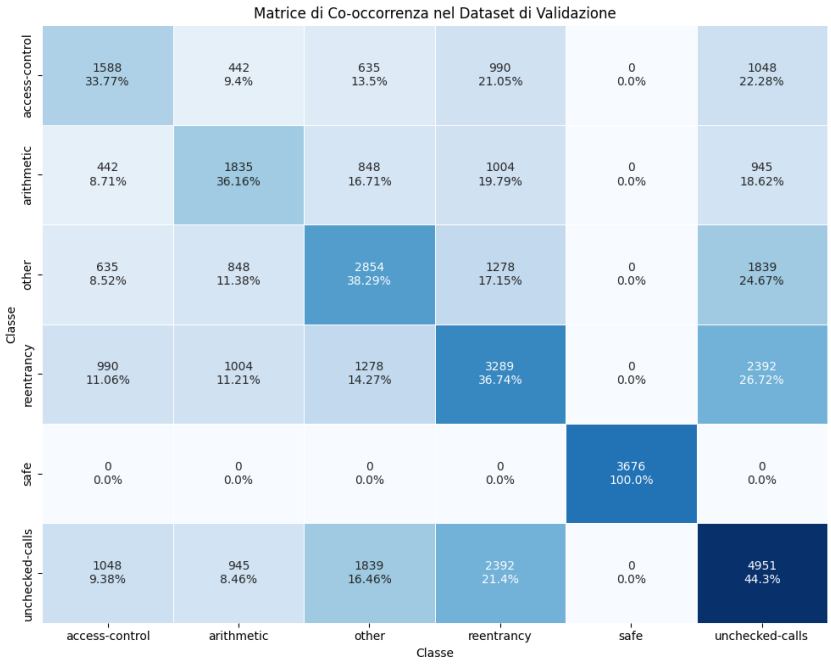


Figura 4.5: Matrice di Co-occorrenza nel Dataset di Validazione

Analizzando le matrici di co-occorrenza, notiamo che:

- La classe safe, che rappresenta i contratti privi di vulnerabilit, correttamente non apparte contemporaneamente a nessuna delle altre classi.
- Le classi 'unchecked-calls' co-occorrono frequentemente con 'reentrancy', 'other', e 'access-control'. Questo suggerisce che i contratti con chiamate non verificate spesso presentano anche altri tipi di vulnerabilit.
- Le classi 'arithmetic' e 'reentrancy' mostrano una co-occorrenza significativa, suggerendo che le vulnerabilit aritmetiche possono spesso essere associate a problemi di rientro.

Questi risultati evidenziano l'importanza di considerare la co-occorrenza delle classi quando si analizzano le vulnerabilit nei contratti intelligenti, poich molte vulnerabilit non si verificano in isolamento ma tendono a manifestarsi insieme ad altre.

4.2 Modellazione

In questa sezione, descriviamo l'architettura dei modelli utilizzati per la classificazione dei contratti intelligenti. In particolare, presentiamo i dettagli relativi ai modelli BERT

utilizzati, alle scelte di configurazione e alle strategie di addestramento. Come si evince dalla sezione precente le feature su cui i modelli dovranno basare le loro predizioni sono il codice sorgente e il bytecode dei contratti, cio dati di natura testuale. La natura dei dati fa s che problema possa essere affrontato efficacemente utilizzando tecniche di elaborazione del linguaggio naturale (NLP, Natural Language Processing).

4.2.1 Natural Language Processing, NLP

L'Elaborazione del Linguaggio Naturale (NLP, da *Natural Language Processing*) un campo di studi interdisciplinare che combina linguistica, informatica e intelligenza artificiale. Si occupa dell'interazione tra computer e linguaggio umano (naturale), in particolare del processamento, analisi e costruzione di modelli riguardanti grandi quantit di dati linguistici naturali [13]. Le due grandi sfide dell'NLP si possono riassumere in due grandi aree di ricerca: la comprensione del linguaggio e la generazione del linguaggio. La comprensione del linguaggio comprende compiti come l'analisi sintattica, l'analisi semantica, il riconoscimento delle entit nominate e la risoluzione delle coreferenze. Questi compiti sono cruciali per la conversione del linguaggio naturale in una rappresentazione formale che le macchine possano elaborare. L'analisi sintattica, ad esempio, mira a determinare la struttura grammaticale di una frase, mentre l'analisi semantica si concentra sulla comprensione del significato del testo. In secondo luogo la generazione del linguaggio riguarda la produzione automatica di testo, che pu includere la sintesi vocale, la traduzione automatica e la generazione di risposte automatiche in chatbot. Questo aspetto dell'NLP fondamentale per creare sistemi che non solo comprendano il linguaggio umano, ma che possano anche comunicare in modo naturale e coerente con gli utenti.

Negli ultimi anni, il campo dell'NLP ha fatto enormi progressi passando dall'epoca delle schede perforate e dell'elaborazione batch (in cui l'analisi di una frase poteva richiedere fino a 7 minuti) all'era di Google e simili (in cui milioni di pagine web possono essere elaborate in meno di un secondo) [3], sino ad arrivare ai giorni d'oggi con l'avvento di modelli di deep learning. Per decenni, l'approccio alla ricerca nel campo dell'NLP prevedeva l'utilizzo di modelli shallow come SVM [6] e regressione logistica allenati su feature sparse e fortemente multidimensionali. Negli ultimi anni, d'altro canto, le reti neurali basati su rappresentazioni di vettori densi hanno prodotto risultati superiori su una grande vastit di task diversi nel mondo dell'NLP [35]. Lo stato dell'arte attuale nell'NLP in molti task rappresentato dall'introduzione di una nuova architettura, che andata a sostituire i modelli RNN e LSTM [9] tradizionali, ovvero i modelli basati su Trasformer, introdotti per la prima volta nel paper "Attention is All You Need" da Vaswani et al. nel 2017 [30]. I Transformer hanno rivoluzionato il campo grazie al meccanismo di self-attention, che consente al modello di valutare e ponderare l'importanza di ogni parola in una frase rispetto alle altre parole della stessa frase, indipendentemente dalla loro distanza posizionale. Questo approccio permette un'elaborazione parallela dei dati, in netto contrasto con la natura sequenziale delle RNN e degli LSTM, migliorando

notevolmente l'efficienza computazionale. La struttura dei Transformer organizzata in blocchi ripetuti di encoder e decoder, dove l'encoder elabora l'input costruendo una rappresentazione interna, e il decoder utilizza questa rappresentazione per generare l'output. Questa architettura ha dimostrato prestazioni eccezionali in molte applicazioni di NLP, tra cui la traduzione automatica, la comprensione e la generazione del linguaggio, la sintesi del testo e il riassunto automatico. Dall'architettura dei Transformer sono derivati molti modelli di successo, tra cui i modelli BERT, la famiglia di modelli GPT e altri.

4.2.2 BERT, Bidirectional Encoder Representations from Transformers

Il modello BERT (Bidirectional Encoder Representations from Transformers) stato presentato da Devlin et al. nel 2018 [5]. BERT un modello di deep learning preaddestrato per l'elaborazione del linguaggio naturale. BERT stato allenato su un corpus di testo molto ampio, comprendente 3.3 miliardi di parole, utilizzando due task di apprendimento supervisionato: il *Masked Language Model* (MLM) e il *Next Sentence Prediction* (NSP). Il Masked Language Model maschera randomicamente alcuni dei token in pinput e l'obiettivo del modello quello di predire l'id nel vocabolario della parola mascherata basandosi solo sul contesto che la circonda, considerando sia il contesto a sinistra che a destra della parola mascherata, in modo da catturare il contesto bidirezionale. Il Next Sentence Prediction, invece, prevede se una frase la successiva rispetto a un'altra frase. Questo task stato introdotto per insegnare al modello a comprendere il contesto e la coerenza tra le frasi. Al momento della sua pubblicazione BERT rappresentava lo stato dell'arte in ben undici diversi task nel campo dell'NLP ed stato il primo modello a raggiungere state-of-the-art performance in molti task sentence-level e token-level, superando anche molte architetture specifiche per task.

Architettura

L'architettura del modello BERT un encoder bidirezionale multi-strato basato sui Transformer, come descritto nell'implementazione originale di Vaswani et al. (2017) [30]. I parametri principali di un'architettura di BERT sono il numero di strati L, la dimensione nascosta H e il numero di self-attention heads A. All'interno dell'architettura di BERT, due concetti fondamentali sono la *hidden size* e le *attention heads*.

La **hidden size** (H) si riferisce alla dimensione dei vettori di rappresentazione nelle varie fasi di elaborazione del modello. In termini pratici, rappresenta la dimensionalit dello spazio in cui le rappresentazioni intermedie dei token vengono proiettate durante l'elaborazione nel modello Transformer. Questa dimensione influisce direttamente sulla capacit del modello di catturare le informazioni a partire dai dati in input; una hidden size maggiore consente al modello di rappresentare e processare informazioni pi dettagliate, a costo per di un incremento dei requisiti computazionali.

Le **attention heads** (A) sono un componente cruciale del meccanismo di self-attention nei Transformer. Ogni attention head esegue una funzione di attenzione, ovvero calcolare un insieme di pesi che determinano l'importanza relativa di ogni token nella sequenza di input rispetto agli altri token, permettendo cos al modello di concentrarsi su diverse parti della sequenza di input simultaneamente.

Modello	Layers L	Hidden Size H	Self-Attention Heads A
$\text{BERT}_{\text{BASE}}$	12	768	12
$\text{BERT}_{\text{LARGE}}$	24	1024	16

Tabella 4.4: Parametri principali dei modelli $\text{BERT}_{\text{BASE}}$ e $\text{BERT}_{\text{LARGE}}$

BERT stato preaddestrato con un embedding WordPiece [34] con un vocabolario di 30.000 token. Il primo token di ogni sequenza sempre un token di classificazione speciale ([CLS]). L'hidden state finale corrispondente a questo token utilizzato come rappresentazione aggregata della sequenza per i task di classificazione, che proprio il modo in cui BERT verr utilizzato in questo lavoro. BERT pu gestire pi sequenze di token in input, ciascuna delle quali seguita da un token speciale ([SEP]), che permette di disambiguare l'appartenenza di un token ad una sequenza piuttosto che ad un'altra.

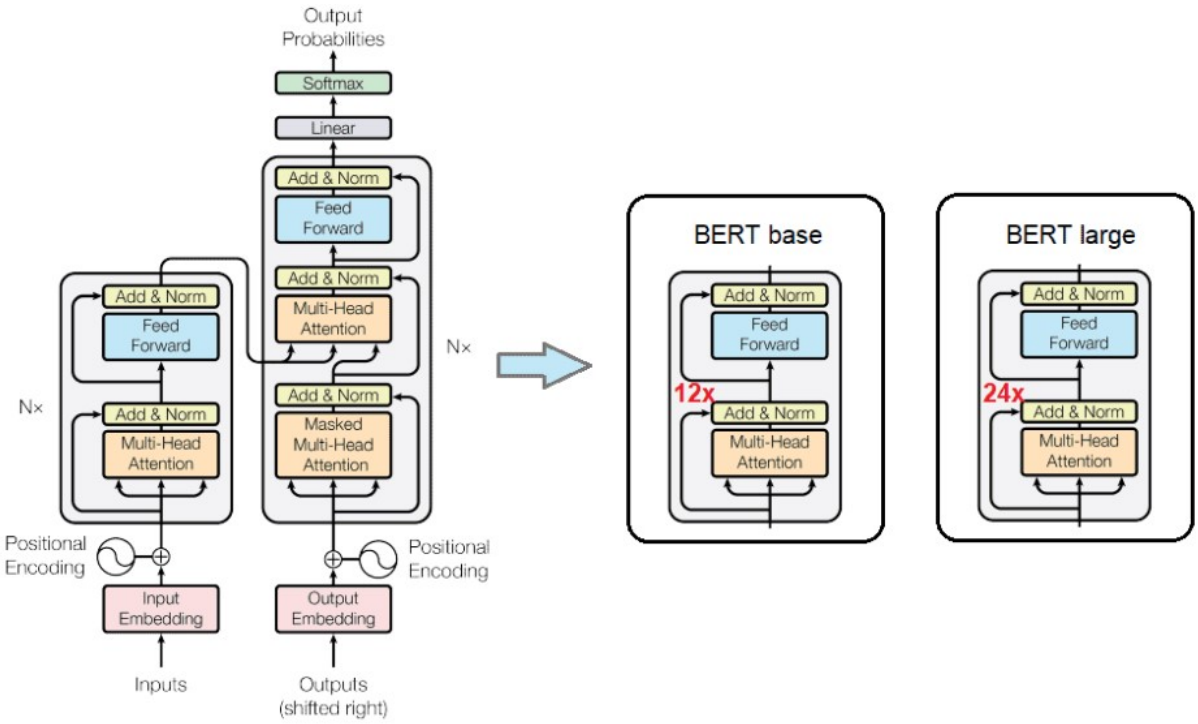


Figura 4.6: Architettura di Transformers, $\text{BERT}_{\text{BASE}}$ e $\text{BERT}_{\text{LARGE}}$

Per ogni token in input, BERT calcola un embedding che dato dalla somma di tre componenti come possibile vedere in Figura 4.7:

- **Token Embeddings**: sono i vettori di embedding per ciascun token nel vocabolario. Questi embedding sono allenati durante il pre-addestramento e sono aggiornati durante il fine-tuning. BERT utilizza una tecnica chiamata Wordpiece tokenization, in cui le parole vengono suddivise in sottostringhe pi piccole chiamate wordpieces. Questa tecnica permette di creare un vocabolario flessibile contenente sia parole che sotto-parole, per esempio prefissi, suffissi o singoli caratteri. Il vocabolario cos creato in grado di gestire tutte le possibili sequenze di caratteri e di evitare lutilizzo di token OOV (Out Of Vocabulary) [34].
- **Segment Embeddings**: sono i vettori di embedding che indicano a quale sequenza appartiene ciascun token. Questi embedding sono utilizzati per distinguere tra le due sequenze di input in un task di classificazione di sequenza.
- **Position Embeddings**: sono una componente critica per aiutare il modello a comprendere la posizione di ciascun token allinterno di una sequenza di testo. Questi embedding consentono a BERT di distinguere tra parole con lo stesso contenuto ma posizionate in posizioni diverse allinterno della frase. Ci contribuisce a catturare le relazioni tra le parole in modo pi completo e consente a BERT di eccellere in una vasta gamma di compiti di elaborazione del linguaggio naturale.

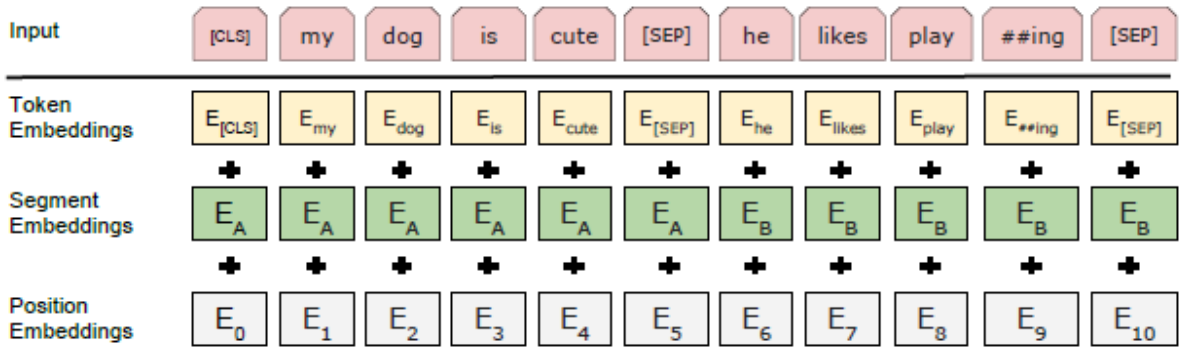


Figura 4.7: Rappresentazione degli input di BERT

Pre-Training

Il pre-training di BERT stato effettuato usando due task di apprendimento non supervisionato:

- **Masked Language Model (MLM)**: in questo task venivano mascherate il 15% dei token in input e si voleva far si che il modello predicesse i token mascherati.

Questo processo in letteratura viene anche spesso chiamato *Cloze* [**?**]. In questo caso i vettori dell'hidden layer finale che si riferisce al token mascherato venivano dati in input ad una funzione softmax sul vocabolario per predire il token mascherato. Per non creare troppo divario tra il pre-addestramento e il fine-tuning, durante questa fase di pre-training con MLM il token speciale [MASK] veniva utilizzato solo l'80% delle volte, il 10% delle volte veniva sostituito con un token casuale e il 10% delle volte veniva lasciato il token originario.

- **Next Sentence Prediction (NSP)**: in molti task di NLP, come Question-Answering necessario che i modelli siano in grado di comprendere relazioni tra due frasi. Per far si che il modello imparasse a riconoscere le relazioni tra frasi BERT stato pre-addestrato su un task di predizione della frase successiva. Nello specifico, prese due frasi A e B il modello doveva predire se la frase B fosse la successiva rispetto alla frase A, questo nel dataset di training era vero nel 50% dei casi.

Fine-tuning di BERT

Il fine-tuning di BERT consiste nell'adattare il modello pre-allenato a compiti specifici, come la classificazione del testo, l'analisi del sentimento, il *question answering* e altri. BERT utilizza l'architettura *Transformer*, che permette di modellare relazioni complesse tra le parole di un testo grazie al meccanismo di *self-attention*. Questo consente a BERT di processare sia singoli testi che coppie di testi.

Durante il fine-tuning, il modello pre-allenato viene ulteriormente addestrato su un dataset specifico del compito da risolvere, modificando tutti i parametri del modello. Questo include i pesi dei livelli di *self-attention*, le rappresentazioni degli *hidden layers* e i parametri degli strati di output. Ad esempio, per un compito di classificazione del testo, il token [CLS], che rappresenta l'intera sequenza, viene utilizzato per determinare la classe del testo. Per compiti a livello di token, come il *named entity recognition* (NER), ogni token del testo viene etichettato individualmente.

Il processo di fine-tuning richiede meno risorse computazionali rispetto al pre-allenamento. Con l'uso di GPU o TPU, il fine-tuning pu essere completato in poche ore, rendendo BERT un'opzione potente e versatile per una variet di applicazioni di elaborazione del linguaggio naturale.

4.2.3 DistilBERT

Nel 2020 stato presentato DistilBERT, un modello pi piccolo e pi veloce rispetto a BERT, sviluppato da Sanh et al. [**?**]. Il modello presentato dichiara che DistilBERT in grado di ridurre la complessit di BERT del 40% pur mantenendo il 97% delle prestazioni di BERT ed essere 60% pi veloce.

I risultati di DistilBERT sono stati ottenuti grazie ad una tecnica chiamata *knowledge distillation*, che una tecnica di compressione in cui modello pi piccolo, detto *modello studente*, allenato per riprodurre i comportamenti di un modello pi grande (o un insieme di modelli) detto *modello insegnante*. Questo processo di distillazione permette di ridurre la complessit del modello studente, riducendo il numero di parametri e la complessit computazionale, mantenendo allo stesso tempo le prestazioni del modello pi grande. Nell'apprendimento supervisionato, un modello di classificazione generalmente allenato per predire l'istanza di una classi massimizzando la stima di probabilit di quella label. Un modello che funziona in maniera ottima predirr una probabilit alta sulla classe corretta e probabilit vicine allo zero per le classi errate.

Il training del modello student si basa su una combinazione di tecniche di distillazione del modello e di apprendimento supervisionato. Viene calcolata una *distillation loss* utilizzando le *soft target probabilities* del modello insegnante. Questa perdita definita come:

$$L_{ce} = \sum_i t_i \log(s_i)$$

dove t_i (rispettivamente s_i) una probabilit stimata dall'insegnante (rispettivamente dallo studente). Questa funzione obiettivo fornisce un segnale di training ricco sfruttando l'intera distribuzione dell'insegnante. Seguendo [**?**], viene utilizzata una *softmax-temperature*, definita come:

$$p_i = \frac{\exp(z_i/T)}{\sum_j \exp(z_j/T)}$$

dove T controlla la morbidezza della distribuzione di output e z_i il punteggio del modello per la classe i. La stessa temperatura T viene applicata sia allo studente che all'insegnante durante il training, mentre in fase di inferenza, T impostata a 1 per tornare ad una funzione *softmax* standard.

L'obiettivo finale del training una combinazione lineare della distillation loss L_{ce} con la loss di training supervisionato, cio la loss del *masked language modeling* L_{mlm}. Per allineare le direzioni dei vettori hidden state del modello student e teacher stata aggiunta una *cosine embedding loss*, L_{cos}. La Loss finale quindi definita come:

$$L = \alpha L_{ce} + \beta L_{mlm} + \gamma L_{cos}$$

Dove α, β, e γ sono pesi che bilanciano i diversi termini di loss. Questa combinazione permette di mantenere la qualit del modello distillato avvicinandolo il pi possibile alla performance del modello insegnante.

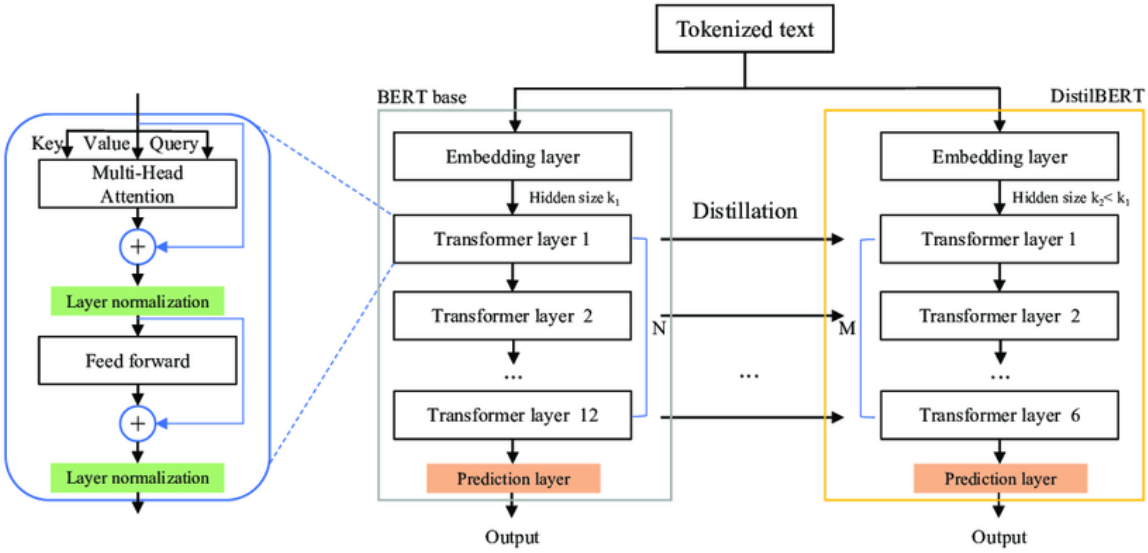


Figura 4.8: Architettura di DistilBERT

Architettura

L'architettura del DistilBERT simile a quella di BERT, ma con alcune differenze chiave. Vengono eliminati i *token-type embedding* e la dimensione in termini di layer viene dimezzata. Sono state ottimizzate la maggior parte delle operazioni usate nell'architettura dei Transformer, come i *linear layer* e *layer normalisation* ed stato dimostrato che ridurre la dimensione dell'hidden state non ha un impatto significativo sulle prestazioni del modello, quindi rimasta invariata. Per l'inizializzazione del modello student stato utilizzato un layer del modello BERT poich hanno la stessa dimensione.

4.2.4 RoBERTa e CodeBERT

A partire da BERT, nel 2019 stato presentato RoBERTa (Robustly optimized BERT approach) da Liu et al. [15]. RoBERTa un modello di deep learning pre-addestrato per l'elaborazione del linguaggio naturale, che migliora le prestazioni di BERT attraverso una serie di modifiche e ottimizzazioni. A livello architetturale, BERT e RoBERTa sono quasi identici, entrambi basati sull'architettura *Transformer* con encoder bidirezionali. Tuttavia, le differenze principali tra i due risiedono nel processo di pre-training e nelle scelte di ottimizzazione. BERT utilizza due obiettivi di pre-training: il *Masked Language Modeling* (MLM) e il *Next Sentence Prediction* (NSP), mentre RoBERTa si concentra esclusivamente su MLM, eliminando l'obiettivo NSP. RoBERTa introduce anche il *dynamic masking*, dove le maschere applicate ai token cambiano ad ogni epoca di training, rispetto al *static masking* utilizzato da BERT. Inoltre, RoBERTa utilizza una quantit di

dati di training molto pi grande e adotta configurazioni di iperparametri pi aggressive, come dimensioni del batch maggiori e tassi di apprendimento pi elevati. Queste modifiche rendono RoBERTa pi efficace e robusto, migliorando le sue performance su una variet di compiti di elaborazione del linguaggio naturale rispetto a BERT.

Il modello CodeBERT un modello presentato per la prima volta da Microsoft nel 2020 [8]. Il modello stato costruito con la stessa architettura del modello RoBERTa-base, avendo quindi un numero totale di parametri pari a 125M.

Nella fase di pretraining per questo modello l'input stata una concatenazione di due segmenti:

4.3 Fase di Pre-Training di CodeBERT

Nella fase di pre-training, l'input costituito dalla concatenazione di due segmenti con un token separatore speciale, ovvero:

$$[CLS], w_1, w_2, \ldots, w_n, [SEP], c_1, c_2, \ldots, c_m, [EOS]$$

. Un segmento testo in linguaggio naturale e l'altro codice di un determinato linguaggio di programmazione. [CLS] un token speciale posizionato all'inizio dei due segmenti, la cui rappresentazione nascosta finale viene considerata come la rappresentazione aggregata della sequenza per task di classificazione o ranking. Seguendo il metodo standard di elaborazione del testo nei *Transformer*, stato considerato un testo in linguaggio naturale come una sequenza di parole, splittandolo utilizzando in *WordPiece*. Allo stesso modo, il codice sorgente stato considerato come una sequenza di token. L'output di CodeBERT offre:

- la rappresentazione vettoriale contestuale di ciascun token, sia per il linguaggio naturale che per il codice
- la rappresentazione di [CLS], che funziona come rappresentazione aggregata della sequenza, allo stesso modo che in BERT.

I dati di training che sono stati utilizzati nel pretraining sono sia dati *bimodali*, ovvero dati che contengono sia testo scritto in linguaggio naturale che codice di un linguaggio di programmazione, che dati *unimodali*, ovvero dati che contengono solo codice senza linguaggio naturale.

I dati sono stati raccolti da repository Github, dove un datapoint:

- *bimodale:* rappresentato da una singola funzione a cui associata della documentazione in linguaggio naturale.
- *unimodale:* rappresentato da una singola funzione senza documentazione in linguaggio naturale.

Nello specifico, stato utilizzato un dataset offerto da [12] che contiene 2.1M di dati bimodali e 6.4M di dati unimodali suddivisi in 6 linguaggi di programmazione: Python, Java, JavaScript, Ruby, Go e PHP. Tutti i dati erano provenienti da repository pubblici e open-source su Github e sono stati filtrati sulla base cinque criteri:

1. ogni progetto deve essere usato da almeno un altro progetto
2. ogni documentazione viene troncata al primo paragrafo
3. le documentazioni inferiori a tre token sono state rimosse
4. funzioni piu corte di tre linee di codice sono state rimosse
5. le funzioni che abbiamo nel nome la sottostringa "test" sono state rimosse

Il pretraining di stato effettuato utilizzando due diverse funzioni obiettivo. Il primo metodo stato il MLM, che stato utilizzato per preaddestrare il modello a predire i token mascherati, questo stato applicato sui dati bimodali. Anche in questo caso, seguendo l'approccio usato dal modello BERT originario [5], sono stati mascherati il 15% dei token tra token appartenenti al linguaggio naturale e token facenti parte del codice sorgente. Il secondo metodo applicato stato il *replaced token detection* che utilizza sia i dati bimodati che quelli unimodali. Durante il pre-training, alcuni token nell'input originale, che pu essere testo in linguaggio naturale o codice, vengono sostituiti con token casuali. Il compito del modello quindi rilevare quali token sono stati sostituiti. Il processo funziona nel seguente modo: il modello riceve una sequenza di token contenente sia token originali che token sostituiti. Per ogni token, il modello deve prevedere una probabilit che indichi se il token stato sostituito o meno. L'obiettivo di training per RTD minimizzare la perdita di classificazione binaria tra i token originali e quelli sostituiti. Questo metodo sfrutta l'intero contesto della sequenza, permettendo a CodeBERT di apprendere rappresentazioni pi ricche e accurate sia per il linguaggio naturale che per il codice.

Per quanto riguarda il fine-tuning, allo stesso modo di BERT, CodeBERT pu essere specializzato su task specifici tunando tutti i suoi parametri in modo molto efficiente. Il modello ha ottenuto performance allo stato dell'arte per task che includono ricerca di codice tramite linguaggio naturale, generazione di documentazione a partire dal codice.

4.4 Implementazione

I tre modelli precedenti, BERT, DistilBERT e CodeBERT sono quelli scelti ed utilizzati in questo lavoro per la classificazione di vulnerabilit degli smart contracts. I task di classificazione possono dividersi in tre categorie principali:

- **Classificazione Binaria:** in cui il modello deve predire se un dato elemento appartiene o meno ad una classe specifica. Facendo un esempio nel contesto degli smart contracts, un task di classificazione binaria comporterebbe la predizione della presenza o meno di vulnerabilit in uno smart contract.

- **Classificazione Multiclasse:** in cui il modello deve predire, all'interno di un insieme di classi, a quale classe appartiene un dato elemento, posto che possa appartenere ad una sola classe. Ad esempio, un task di classificazione multiclasse potrebbe essere quello di prevedere la vulnerabilit presente in uno smart contract posto che questo possa avere un solo tipo di vulnerabilit.

- **Classificazione Multilabel:** in cui il modello deve predire, all'interno di un insieme di classi, a quali classi appartiene un dato elemento, posto che possa appartenere a pi classi contemporaneamente.

Questo lavoro stato affrontato come un task di classificazione multilabel, in quanto uno smart contract pu avere pi di una vulnerabilit contemporaneamente.

Il codice per il pre-processing dei dati, la costruzione, il training e la valutazione dei modelli stato scritto in Python. Python un linguaggio di programmazione Touring-completo ad alto livello, ampiamente riconosciuto per la sua semplicit e leggibilit. La vasta disponibilit di librerie e strumenti specifici per l'elaborazione del linguaggio naturale e l'apprendimento automatico, come NumPy, Pandas e Scikit-learn, rende Python una scelta ideale per questo tipo di ricerca. Inoltre, Python supportato da una vasta comunit di sviluppatori e ricercatori, che lo rendono uno dei linguaggi pi utilizzati a livello mondiale per la manipolazione di dati e per lo sviluppo di modelli di Machine e Deep Learning. In aiuto a Python stata utilizzata la libreria PyTorch, una libreria di deep learning altamente performante e flessibile, sviluppata da Facebook AI Research (FAIR) [22]. PyTorch offre un'interfaccia intuitiva e dinamica per la costruzione e l'addestramento di modelli di apprendimento profondo, facilitando la sperimentazione e l'ottimizzazione dei modelli. Inoltre, PyTorch supportato da una comunit di ricerca attiva e in crescita, che contribuisce con numerosi modelli pre-addestrati e risorse che accelerano lo sviluppo e la valutazione dei modelli. Queste caratteristiche rendono Python e PyTorch l'ovvia scelta per questo lavoro.

Fondamentali per lo sviluppo sono state altre due librerie offerte da HuggingFace, rispettivamente Transformers e Datasets. Transformers una libreria open-source che offre un'implementazione di modelli di apprendimento profondo pre-addestrati, tra cui BERT, RoBERTa, DistilBERT e molti altri. Questa libreria offre un'interfaccia semplice e intuitiva per caricare, costruire e addestrare modelli di apprendimento profondo, facilitando la sperimentazione e l'ottimizzazione dei modelli. Datasets, invece, una libreria open-source che offre un'interfaccia semplice e intuitiva per caricare e manipolare dataset che sono stati caricati sulla piattaforma HuggingFace. Questa libreria offre una

vasta gamma di dataset pre-caricati, tra cui il dataset *rossini2022slitherauditedcontracts* utilizzato in questo lavoro, semplificando il processo di raccolta dei dati.

Di seguito, verranno presentati tutti gli esperimenti effettuati e i vari modelli costruiti, i cui risultati verranno poi discussi nel capitolo 5 successivo.

4.4.1 BERT-base

Il primo modello ricostruito stato un modello BERT base. Il modello pretrainato a cui si fatto riferimento stato *bert-base-uncased*, un modello BERT base pre-addestrato su un corpus di testo in lingua inglese. Dopo essere stato caricato il modello ne viene estratto il pooling output per il primo token, il token [CLS] che come descritto nella sezione 4.2.2 utilizzato come rappresentazione aggregata della sequenza per task di classificazione. Questo output viene poi passato ad un layer di classificazione lineare, che prende in input un vettore di dimensione pari all'hidden state di BERT, quindi 768, e restitusce un vettore di output con dimensione pari al numero di classi (nel nostro caso 5). Questo vettore di output viene poi passato ad una funzione di attivazione softmax, che restituisce una distribuzione di probabilit su tutte le classi. Questo modello viene utilizzato per la classificazione sia del codice sorgente che del bytecode dei contratti. Mostriamo ora un esempio di codice per la costruzione del modello:

```
class BERTClassifier(torch.nn.Module):
def __init__(self):
    super(BERTClassifier, self).__init__()
    self.bert_model = transformers.BertModel.from_pretrained('bert-
                                        base-uncased')
    self.dropout = torch.nn.Dropout(0.3)
    self.linear = torch.nn.Linear(768, NUM_CLASSES)

def forward(self, input_ids, attention_mask, token_type_ids):
    _, pooled_output = self.bert_model(input_ids, attention_mask=
                                        attention_mask,
                                        token_type_ids=
                                        token_type_ids, return_dict
                                        =False)
    dropout_output = self.dropout(pooled_output)
    linear_output = self.linear(dropout_output)
    return linear_output
```

Il *pooled_output* la rappresentazione vettoriale estratta dal modello BERT che rappresenta un'aggregazione delle informazioni apprese dal testo di input. Questa rappresentazione viene ottenuta tramite un processo di "pooling" delle rappresentazioni dei token di input prodotte dal modello BERT. Il layer di Dropout stato aggiunto per evitare l'overfitting del modello. Il modello stato addestrato utilizzando l'ottimizzatore Adam con un learning rate di 1×10^{-5} e una dimensione del batch di 8.

Loss Function

La funzione di loss scelta stata la BCEWithLogitsLoss, che stata utilizzata per la classificazione multilabel. La funzione di perdita *BCEWithLogitsLoss* una combinazione efficiente della *Binary Cross Entropy* (BCE) e di una funzione logistica di attivazione, spesso utilizzata per compiti di classificazione binaria in reti neurali. Questa funzione di loss applica un'attivazione sigmoide ad una Binary Cross Entropy.

$$\ell(x, y) = L = \left[l_1, \dots, l_N\right]^\top, \quad l_n = -w_n \left[y_n \cdot \log \sigma(x_n) + (1 - y_n) \cdot \log(1 - \sigma(x_n))\right],$$

4.4.2 DistilBERT

La costruzione di un modello DistilBERT avviene in modo pressoch identico a quello del modello BERT. Anche in questo caso si utilizzato il modello pre-addestrato *distilbert-base-uncased*, proveniente dalla libreria Transformers. Mostriamo ora il codice per la costruzione del modello:

```
class DistilBERTClass(torch.nn.Module):
def __init__(self, NUM_CLASSES):
    super(DistilBERTClass, self).__init__()
    self.num_classes = NUM_CLASSES
    self.distilbert = DistilBertModel.from_pretrained('distilbert-
                                        base-uncased')
    self.dropout = torch.nn.Dropout(0.3)
    self.fc = torch.nn.Linear(768, NUM_CLASSES)

def forward(self, input_ids, attention_mask):
    outputs = self.distilbert(input_ids=input_ids, attention_mask=
                                        attention_mask)
    pooled_output = outputs.last_hidden_state[:, 0]
    pooled_output = self.dropout(pooled_output)
    output = self.fc(pooled_output)
    return output
```

Anche in questo caso, il modello stato addestrato utilizzando l'ottimizzatore Adam con un learning rate di 1×10^{-5} e una dimensione del batch di 8 e la funzione di loss BCEWithLogitsLoss. Come per il modello BERT, il layer di Dropout stato aggiunto per evitare l'overfitting del modello.

Come si pu notare dal codice sopra riportato, a differenza del modello BERT, il modello DistilBERT non prende in input i token type ids, in quanto non sono presenti nella struttura del modello DistilBERT.

4.4.3 CodeBERT

Anche il modello CodeBERT stato costruito in modo simile ai modelli BERT e DistilBERT. Il modello pre-addestrato utilizzato stato *microsoft/codebert-base*, proveniente dalla libreria Transformers. Mostriamo ora il codice per la costruzione del modello:

```
class CodeBERTClass(torch.nn.Module):
    def __init__(self, NUM_CLASSES):
        super(CodeBERTClass, self).__init__()
        self.num_classes = NUM_CLASSES
        self.codebert = AutoModel.from_pretrained('microsoft/
                                                  codebert-base')
        self.dropout = torch.nn.Dropout(0.3)
        self.fc = torch.nn.Linear(768, NUM_CLASSES)

    def forward(self, ids, mask):
        outputs = self.codebert(input_ids=ids, attention_mask=mask)
        pooled_output = outputs.pooler_output
        pooled_output = self.dropout(pooled_output)
        output = self.fc(pooled_output)
        return output
```

Anche in questo caso, il modello stato addestrato utilizzando l'ottimizzatore Adam con un learning rate di 1×10^{-5} e una dimensione del batch di 8 e la funzione di loss BCEWithLogitsLoss. Come per i modelli BERT e DistilBERT, il layer di Dropout stato aggiunto per evitare l'overfitting del modello.

4.4.4 CodeBERT con aggregazione

Uno dei problemi principali riscontrati nel training dei precedenti modelli i modelli della famiglia BERT hanno una capacit massima in input di 512 token. I nostri smart contract, come visto precedentemente hanno una lunghezza media di circa 1500 token, dare quindi in input al modello solo 512 token significa far si che il modello analizzi, in molti casi, solo una piccola porzione iniziale dei nostri contratti. Per ovviare a questo problema, stato deciso di dividere i nostri contratti in sotto-contratti di lunghezza massima 512 token e poi aggregare gli embedding prodotti da CodeBERT per ogni sotto-contratto tramite delle funzioni di aggregazione come media e massimo. Questo approccio permette di far si che il modello possa analizzare l'intero contratto, anche se in sotto-parti. Mostriamo ora il codice per la costruzione del modello:

```
class CodeBERTAggregatedClass(torch.nn.Module):
    def __init__(self, num_classes, aggregation='mean', dropout=0.3
                                                    ):
        super(CodeBERTAggregatedClass, self).__init__()
        self.codebert = AutoModel.from_pretrained('microsoft/
                                                  codebert-base',
                                                  cache_dir="./cache")
```

```
        self.dropout = torch.nn.Dropout(dropout)
        self.fc = torch.nn.Linear(self.codebert.config.hidden_size,
                                  num_classes)
        self.aggregation = aggregation

    def forward(self, input_ids, attention_masks):
        batch_size, seq_len = input_ids.size()

        # Divide input_ids e attention_masks in blocchi di 512
                                  token
        num_chunks = (seq_len + 511) // 512
        input_ids = input_ids[:, :512*num_chunks].view(batch_size *
                                  num_chunks, 512)
        attention_masks = attention_masks[:, :512*num_chunks].view(
                                  batch_size * num_chunks
                                  , 512)

        outputs = self.codebert(input_ids=input_ids, attention_mask
                                  =attention_masks)

        last_hidden_states = outputs.last_hidden_state
        cls_tokens = last_hidden_states[:, 0, :]

        cls_tokens = cls_tokens.view(batch_size, num_chunks, -1)

        if self.aggregation == 'mean':
            aggregated_output = torch.mean(cls_tokens, dim=1)
        elif self.aggregation == 'max':
            aggregated_output, _ = torch.max(cls_tokens, dim=1)
        else:
            raise ValueError("Aggregation must be 'mean' or 'max'")

        aggregated_output = self.dropout(aggregated_output)
        output = self.fc(aggregated_output)
        return output
```

Questo approccio stato testato sia con la funzione di aggregazione *mean* che con la funzione di aggregazione *max*, che restituiscono rispettivamente la media e il massimo degli embedding prodotti da CodeBERT per ogni sotto-contratto. Inoltre, sono stati testati aprrocci per entrambi in cui venivano utilizzati due o tre blocchi di codice, quindi si classificavano rispettivamente 1024 e 1536 token.

4.4.5 CodeBert con concatenazione

Un altro approccio complementare alla risoluzione del problema legato al massimo numero di token che i modelli della famiglia BERT possono accettare come input stato quello di concatenare gli embedding prodotti da CodeBERT per ogni sotto-contratto.

Questo approccio permette di mantenere l'informazione relativa a tutti i token del contratto e di fornire al modello un'informazione pi completa. Mostriamo ora il codice per la costruzione del modello:

```
    class CodeBERTConcatenatedClass(torch.nn.Module):
    def __init__(self, num_classes, dropout=0.3):
        super(CodeBERTConcatenatedClass, self).__init__()
        self.codebert = AutoModel.from_pretrained('microsoft/codebert-
                                                  base', cache_dir="./cache")
        self.dropout = torch.nn.Dropout(dropout)
        # Multiply hidden_size by the number of chunks you're
                                                  concatenating
        self.fc = torch.nn.Linear(self.codebert.config.hidden_size *
                                                  CODE_BLOCKS, num_classes)

    def forward(self, input_ids, attention_masks):
        batch_size, seq_len = input_ids.size()

        # Divide input_ids and attention_masks into chunks of 512
                                                  tokens
        num_chunks = (seq_len + 511) // 512
        input_ids = input_ids[:, :512*num_chunks].reshape(batch_size *
                                                  num_chunks, 512)
        attention_masks = attention_masks[:, :512*num_chunks].reshape(
                                                  batch_size * num_chunks,
                                                  512)

        outputs = self.codebert(input_ids=input_ids, attention_mask=
                                                  attention_masks)

        last_hidden_states = outputs.last_hidden_state
        cls_tokens = last_hidden_states[:, 0, :]

        # Concatenate the CLS tokens of the various chunks
        cls_tokens = cls_tokens.reshape(batch_size, num_chunks, -1)
        concatenated_output = cls_tokens.reshape(batch_size, -1)

        concatenated_output = self.dropout(concatenated_output)
        output = self.fc(concatenated_output)
        return output
```

In questo caso, si divide la stringa in input in chunk da 512 token e si ottengono gli embedding prodotti da codeBERT per ognuno di questi chunk. Questi embedding vengono poi concatenati e passati ad un layer di classificazione lineare. Anche in questo caso l'approccio stato testato sia con due che con tre blocchi di codice, quindi si classificavano rispettivamente 1024 e 1536 token.

4.4.6 Train e Validation

Tutti i modelli sono stati addestrati (o fine-tunati) per 20 epoche, con un learning rate di 1×10^{-5}. Per evitare l'overfitting, stato utilizzato un layer di Dropout con un rate del 30%. La funzione di loss utilizzata stata la BCEWithLogitsLoss, che stata utilizzata per la classificazione multilabel. L'ottimizzatore utilizzato stato Adam. I modelli sono stati addestrati su una GPU NVIDIA 2080Ti. A seguito di limitazioni dovute alla potenza di calcolo e alla grandezza del dataset in alcuni casi stato necessario ridurre la batch size. Nello specifico, per i modelli di aggregazione e concatenazione che utilizzavano due blocchi di codice (1024 token) la batch size stata ridotta a 4, mentre per i modelli che utilizzavano tre blocchi di codice (1536 token) la batch size stata ridotta a 2. Questa potrebbe essere sicuramente una limitazione al lavoro, in quanto una batch size ridotta potrebbe portare ad un addestramento pi lento e a risultati meno accurati. Si rimanda a futura ricerca la possibilit di addestrare i modelli con potenze di calcolo maggiori per effettuare dei tuning dei parametri pi accurati.

4.5 Gemini

Gemini una famiglia di large language model multimodali sviluppato da Deep Mind Google, l'azienda fondata nel 2010 con sede a Londra che ha il compito di fare ricerca e sviluppo nel campo dell'intelligenza artificiale [33]. Questa famiglia di modelli, nata come successore di LaMDA e PaLM2, e comprende Gemini Ultra, Gemini Pro e Gemini Nano. Questi modelli sono stati poi resi famosi per dar vita al chatbot Gemini di Google, che si pone come principale competitor di GPT-4 di OpenAI [32].

La prima generazione di Gemini ("Gemini 1") ha tre modelli, con la stessa architettura software. Si tratta di tranformer solo-decoder. Hanno una lunghezza del contesto di 32768 token. Sono poi state distillare due versioni di Gemini Nano: Nano-1 con 1,8 miliardi di parametri e Nano-2 avente 3,25 miliardi di parametri. Questi modelli sono distillati da modelli Gemini pi grandi, progettati per l'uso da parte di dispositivi con potenza di calcolo limitata come gli smartphone. Poich Gemini multimodale, ogni finestra di contesto pu contenere pi forme di input. Le diverse modalit possono essere interlacciate e non devono essere presentate in un ordine fisso, consentendo una conversazione multimodale. Ad esempio, l'utente potrebbe aprire la conversazione con una miscela di testo, immagini, video e audio, presentata in qualsiasi ordine e Gemini potrebbe rispondere con lo stesso ordine libero.

Il dataset di Gemini multimodale e multilingue, composto da "documenti web, libri e codice, e include dati di immagini, audio e video".

La seconda generazione di Gemini ("Gemini 1.5") ha due modelli pubblicati finora:

- *Gemini 1.5 Pro*: si tratta di un mixture-of-expertise sparso multimodale.

- *Gemini 1.5 Flash*: un modello distillato da Gemini 1.5 Pro, con una lunghezza del contesto superiore a 2 milioni.

Al giorno d'oggi, chatbot come Gemini e ChatGPT vengono sempre pi spesso utilizzati come strumenti di supporto per le attivit di tutti i giorni, come la ricerca di informazioni, la creazione di contenuti e la gestione delle attivit quotidiane. Lo sviluppo software non si esime da questa tendenza, e sempre pi spesso si fa ricorso a chatbot per la gestione di task di programmazione e sviluppo software sempre pi complessi. Data l'importanza che questi strumenti stanno assumendo nelle nostre vite importante valutarne le performance e la qualit. Diventa quindi importante confrontare i modelli costruiti in questo lavoro di tesi con modelli gi esistenti e ampiamente utilizzati come Gemini e ChatGPT.

Per valutare quanto i modelli costruiti in questo lavoro siano performanti rispetto a chatbot disponibili online gratuitamente (o in versioni Pro a pagamento) come ChatGPT e Gemini stata utilizzata la versione gratuita delle API di Gemini 1.5 Flash per testarlo in un task di classificazione multilabel delle vulnerabilit degli smart contracts. Non stato possibile effettuare gli stessi test utilizzando le API di ChatGPT in quanto esse non sono disponibili in forma di prova gratuita.

Il campione di dati raccolto per la valutazione del modello Gemini 1.5 Flash, composto da 1169 elementi a partire dal dataset di test. Questo stato utilizzato come indicazione delle performance del modello rispetto ai modelli costruiti in questo lavoro. Non stato possibile raccogliere un campione di dati pi ampio (come ad esempio l'intero dataset di test) a causa delle limitazioni della versione gratuita delle API.

Il prompt utilizzato per la valutazione di Gemini 1.5 Flash stato scritto in lingua inglese e recita:

> Analyze the following smart contract for the presence of the following vulnerabilities:
>
> access-control
>
> arithmetic
>
> other
>
> reentrancy
>
> unchecked-calls
>
> Reply ONLY with an array of 5 elements where each element is either 0 or 1. A 1 indicates the presence of the corresponding vulnerability, and a 0 indicates its absence.
>
> For example, if the contract has arithmetic and reentrancy vulnerabilities, the output should be [0,1,0,1,0].

A questo prompt veniva concatenato la stringa del codice sorgente del contratto da analizzare, opportunamente gi preprocessato con le stesse tecniche utilizzate per i modelli

descritti in precedenza, quindi la rimozione dei commenti e delle funzioni getter monoistruzione. Mostriamo quindi un esempio di codice per l'utilizzo delle API di Gemini 1.5 Flash:

```
import pathlib
import textwrap

import google.generativeai as genai

from IPython.display import display
from IPython.display import Markdown
from google.colab import userdata
GOOGLE_API_KEY=userdata.get('GOOGLE_API_KEY')

genai.configure(api_key=GOOGLE_API_KEY)
model = genai.GenerativeModel('gemini-1.5-flash')
response = model.generate_content(basePrompt + sourceCode)
```

Come si pu notare nel codice sopra riportato, l'utilizzo del modello Gemini 1.5 Flash estremamente semplice e intuitivo. Dopo aver configurato l'API key, si crea un oggetto di tipo GenerativeModel passando come parametro il nome del modello, in questo caso "gemini-1.5-flash". Successivamente, si chiama il metodo generate_content passando come parametro il prompt da utilizzare per la generazione del testo. Il risultato della chiamata al metodo generate_content un oggetto di tipo GenerativeModelResponse che contiene il testo generato dal modello. Questo testo stato poi processato per estrarre l'array di 5 elementi che rappresenta la presenza o meno delle vulnerabilit all'interno del contratto.

Capitolo 5

Results

Capitolo 6

Conclusioni

Bibliografia

[1] Mythril github repository. https://github.com/ConsenSys/mythril.

[2] Opyente. https://pypi.org/project/oyente/.

[3] Erik Cambria and Bebo White. Jumping nlp curves: A review of natural language processing research [review article]. *IEEE Computational Intelligence Magazine*, 9(2):48–57, 2014.

[4] Weichu Deng, Huanchun Wei, Teng Huang, Cong Cao, Yun Peng, and Xuan Hu. Smart contract vulnerability detection based on deep learning and multimodal decision fusion. *Sensors*, 23(16), 2023.

[5] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. BERT: pre-training of deep bidirectional transformers for language understanding. *CoRR*, abs/1810.04805, 2018.

[6] Theodoros Evgeniou and Massimiliano Pontil. Support vector machines: Theory and applications. volume 2049, pages 249–257, 09 2001.

[7] Josselin Feist, Gustavo Grieco, and Alex Groce. Slither: A static analysis framework for smart contracts. In *2019 IEEE/ACM 2nd International Workshop on Emerging Trends in Software Engineering for Blockchain (WETSEB)*, pages 8–15, 2019.

[8] Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, and Ming Zhou. Codebert: A pre-trained model for programming and natural languages. *CoRR*, abs/2002.08155, 2020.

[9] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural Comput.*, 9(8):17351780, nov 1997.

[10] Austin Huang, Suraj Subramanian, Jonathan Sum, Khalid Almubarak, and Stella Biderman. The annotated transformer. https://nlp.seas.harvard.edu/annotated-transformer/.

[11] TonTon Hsien-De Huang. Hunting the ethereum smart contract: Color-inspired inspection of potential attacks. *CoRR*, abs/1807.01868, 2018.

[12] Hamel Husain, Ho-Hsiang Wu, Tiferet Gazit, Miltiadis Allamanis, and Marc Brockschmidt. Codesearchnet challenge: Evaluating the state of semantic code search. *CoRR*, abs/1909.09436, 2019.

[13] Daniel Jurafsky and James H Martin. *Speech and Language Processing*. Pearson Prentice Hall, 2nd edition, 2009.

[14] Zulfiqar Ali Khan and Akbar Siami Namin. Ethereum smart contracts: Vulnerabilities and their classifications. In *2020 IEEE International Conference on Big Data (Big Data)*, pages 1–10, 2020.

[15] Yinhan Liu, Myle Ott, Naman Goyal, Jingfei Du, Mandar Joshi, Danqi Chen, Omer Levy, Mike Lewis, Luke Zettlemoyer, and Veselin Stoyanov. Roberta: A robustly optimized BERT pretraining approach. *CoRR*, abs/1907.11692, 2019.

[16] Oliver Lutz, Huili Chen, Hossein Fereidooni, Christoph Sendner, Alexandra Dmitrienko, Ahmad-Reza Sadeghi, and Farinaz Koushanfar. ESCORT: ethereum smart contracts vulnerability detection using deep neural network and transfer learning. *CoRR*, abs/2103.12607, 2021.

[17] Loi Luu, Duc-Hiep Chu, Hrishi Olickel, Prateek Saxena, and Aquinas Hobor. Making smart contracts smarter. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, CCS '16, page 254269, New York, NY, USA, 2016. Association for Computing Machinery.

[18] Anzhelika Mezina and Aleksandr Ometov. Detecting smart contract vulnerabilities with combined binary and multiclass classification. *Cryptography*, 7(3), 2023.

[19] Mark Mossberg, Felipe Manzano, Eric Hennenfent, Alex Groce, Gustavo Grieco, Josselin Feist, Trent Brunson, and Artem Dinaburg. Manticore: A user-friendly symbolic execution framework for binaries and smart contracts. *CoRR*, abs/1907.03890, 2019.

[20] Bernhard Mueller. Smashing ethereum smart contracts for fun and real profit. *9th HITB Security Conference*, 2018.

[21] Ivica Nikolic, Aashish Kolluri, Ilya Sergey, Prateek Saxena, and Aquinas Hobor. Finding the greedy, prodigal, and suicidal contracts at scale. *CoRR*, abs/1802.06038, 2018.

[22] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Köpf, Edward Z. Yang, Zach DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. Pytorch: An imperative style, high-performance deep learning library. *CoRR*, abs/1912.01703, 2019.

[23] pcaversaccio. A historical collection of reentrancy attacks. https://github.com/pcaversaccio/reentrancy-attacks, 2023.

[24] Martina Rossini. Slither audited smart contracts dataset, 2022.

[25] Martina Rossini, Mirco Zichichi, and Stefano Ferretti. On the use of deep neural networks for security vulnerabilities detection in smart contracts. In *2023 IEEE International Conference on Pervasive Computing and Communications Workshops and other Affiliated Events (PerCom Workshops)*, pages 74–79, 2023.

[26] Alexander Rush. The annotated transformer. In Eunjeong L. Park, Masato Hagiwara, Dmitrijs Milajevs, and Liling Tan, editors, *Proceedings of Workshop for NLP Open Source Software (NLP-OSS)*, pages 52–60, Melbourne, Australia, July 2018. Association for Computational Linguistics.

[27] Victor Sanh, Lysandre Debut, Julien Chaumond, and Thomas Wolf. Distilbert, a distilled version of BERT: smaller, faster, cheaper and lighter. *CoRR*, abs/1910.01108, 2019.

[28] Sergei Tikhomirov, Ekaterina Voskresenskaya, Ivan Ivanitskiy, Ramil Takhaviev, Evgeny Marchenko, and Yaroslav Alexandrov. Smartcheck: static analysis of ethereum smart contracts. In *Proceedings of the 1st International Workshop on Emerging Trends in Software Engineering for Blockchain*, WETSEB '18, page 916, New York, NY, USA, 2018. Association for Computing Machinery.

[29] Petar Tsankov, Andrei Dan, Dana Drachsler-Cohen, Arthur Gervais, Florian Bnzli, and Martin Vechev. Securify: Practical security analysis of smart contracts. pages 67–82, 10 2018.

[30] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention is all you need. *CoRR*, abs/1706.03762, 2017.

[31] Wei Wang, Jingjing Song, Guangquan Xu, Yidong Li, Hao Wang, and Chunhua Su. Contractward: Automated vulnerability detection models for ethereum smart contracts. *IEEE Transactions on Network Science and Engineering*, 8(2):1133–1144, 2021.

[32] Wikipedia. Gemini (language model) — Wikipedia, the free encyclopedia. http://en.wikipedia.org/w/index.php?title=Gemini%20(language%20model)&oldid=1227938870, 2024.

[33] Wikipedia. Google DeepMind — Wikipedia, the free encyclopedia. http://en.wikipedia.org/w/index.php?title=Google%20DeepMind&oldid=1228075682, 2024.

[34] Yonghui Wu, Mike Schuster, Zhifeng Chen, Quoc V. Le, Mohammad Norouzi, Wolfgang Macherey, Maxim Krikun, Yuan Cao, Qin Gao, Klaus Macherey, Jeff Klingner, Apurva Shah, Melvin Johnson, Xiaobing Liu, Lukasz Kaiser, Stephan Gouws, Yoshikiyo Kato, Taku Kudo, Hideto Kazawa, Keith Stevens, George Kurian, Nishant Patil, Wei Wang, Cliff Young, Jason Smith, Jason Riesa, Alex Rudnick, Oriol Vinyals, Greg Corrado, Macduff Hughes, and Jeffrey Dean. Google's neural machine translation system: Bridging the gap between human and machine translation. *CoRR*, abs/1609.08144, 2016.

[35] Tom Young, Devamanyu Hazarika, Soujanya Poria, and Erik Cambria. Recent trends in deep learning based natural language processing. *CoRR*, abs/1708.02709, 2017.