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 popolarità grazie alla loro abilità di automatizzare e far rispettare gli accordi tra due soggetti senza la necessità 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 è sicuramente la loro immutabilità, difatti una volta essere stati deployati sulla blockchain questi non possono più essere modificati. Uno dei problemi principali degli SmartContracts è la sicurezza. Infatti, essendo dei contratti che vengono eseguiti in modo automatico, è possibile che ci siano delle vulnerabilità che possono essere sfruttate da malintenzionati. In questo lavoro di tesi verrà presentato un metodo per rilevare le vulnerabilità presenti negli SmartContracts.

Capitolo 2

Lavori Correlati

Il tema della rilevazione delle vulnerabilità all'interno degli Smart Contracts è 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 vulnerabilità all'interno degli Smart Contracts. In questo capitolo verranno presentati alcuni dei lavori più 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 necessità di testare il codice in un ambiente reale. L'esecuzione simbolica è una tecnica particolarmente potente in questo contesto in quanto consente di esplorare tutte le possibili esecuzioni del programma, consentendo di individuare vulnerabilità che potrebbero emergere solo in determinate condizioni. Gli approcci basati su esecuzione simbolica cercano di risolvere queste vulnerabilità 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. Ciò consente di identificare percorsi di esecuzione che potrebbero condurre a condizioni di errore o vulnerabilità. Tuttavia, va notato che l'analisi statica, inclusa l'esecuzione simbolica, può essere complessa e non sempre completa. Alcune vulnerabilità potrebbero sfuggire a questa analisi o richiedere ulteriori tecniche di verifica. Pertanto, è consigliabile combinare l'analisi statica con altre metodologie, come l'analisi dinamica e i test formali, per garantire una copertura completa nella rilevazione di vulnerabilità negli smart contract. Tra i principali strumenti che utilizzano questo tipo di analisi ci sono Maian [12, 11], Oyente [2, 8], Mythril [1], Manticore [10] 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 nvulnerabilità 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 vulnerabilità note e delle best

practice 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 è che i risultati che producono sono limitati al set di regole che è stato implementato, quindi non riescono a riconoscere delle nuove vulnerabilità o vulnerabilità non scoperte precedentemente. Inoltre, un'altra grande limitazione di questi strumenti è il fatto che possano produrre un alto numero di falsi positivi, cioè di situazioni in cui il codice viene segnalato come codice vulnerabile ma in realtà è codice perfettamente sano. Tra i principali strumenti che utilizzano questo tipo di analisi ci sono Slither [4], Securify [17], SmartCheck [16] e altri. Un'altra categoria di strumenti per la rilevazione di vulnerabilità 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 è stato offerto da Wang et al. [18] 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 è risultato il miglior modello. Un altro lavoro che sfrutta tecniche di Machine Learning più tradizionali è 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 vulnerabilità) e poi multiclasse (valutare quale tipo di vulnerabilità il contratto abbia) [9]. 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 è importante citare un'altro lavoro effettuato sullo stesso dataset su cui è basato questo lavoro di tesi. Questo dataset è stato infatti raccolto e pubblicato da un gruppo di ricercatori dell'Università di Bologna che ha effettuato un primo studio utilizzando un approccio basato su reti neurali convoluzionali [15]. L'approccio in questo caso è stato quello di classificare le vulnerabilità in un'impostazione multilabel del problema (in cui la label da predire è un array di elementi, quindi in cui il contratto può appartenere contemporaneamente a più classi) itilizzando delle reti neurali convoluzionali per la rilevazione delle vulnerabilità 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 è possibile rilevare le vulnerabilità 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 è il lavoro proposto da Huang [5] che utilizza anch'egli delle reti neurali convoluzionali per la rilevazione delle vulnerabilità. 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. [3] 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 è 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 Vulnerabilità

Access-Control

La vulnerabilità di access control è una vulnerabilità presente non solo in Solidity ma in numerosi altri linguaggi di programmazione. Questa vulnerabilità si verifica quando un contratto non controlla correttamente l'accesso alle sue funzioni e ai suoi dati, è quindi una vulnerabilità legata al governanre chi può interagire con le varie funzionalità all'interno del contratto. Un esempio di questo dipo di vulnerabilità è 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 può chiamarla, permettendo a chiunque di chiamarla e diventare l'owner del contratto e non ha nemmeno controlli per prevenire la reinizializzazione. Questo è un esempio molto semplice di come una vulnerabilità di access control possa portare a comportamenti inaspettati. Un famoso attacco che ha subito una vulnerabilità di tipo access-control è 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 quantità di Ether. L'attacco è stato effettuato da un utente che ha sfruttato una vulnerabilità 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 vulnerabilità di tipo aritmetico [6] sono vulnerabilità che vengono generate come risultato di operazioni matematiche. Una delle vulnerabilità più significative all'interno

di questa classe è 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 vulnerabilità per ottenere un vantaggio. Ad esempio nel caso in cui si stia utilizzano un uint256 il massimo numero che si piuò memorizzare nella variabile è $2^256 - 1$, che è 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 piuù piccolo valore rappresentabile. Questo può 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, cioè un intero a 8bit ovvero un intero che può 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 è 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 è a 0. In questo caso il valore della variabile ritornerebbe a 255. Un esempio di attacco che sfrutta l'overflow è 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 più 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 è 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 vulnerabilità di tipo aritmetico, ma non sono le uniche. Un'altra vulnerabilità di tipo aritmetico è 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 può portare a comportamenti inaspettati e a perdite di fondi.

Other

In questa classe di vulnerabilità rientrano tutte quelle vulnerabilità che non fanno parte delle altre classi. Uno degli esempi sono le vulnerabilità di *uninitialized-state* che fanno capo a tutte quelle vulnerabilità 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, poichè ad esempio inzializzare male i puntatori allo storage o lasciarli non inizializzati può portare a degli errori. Da Solidity 0.5.0 i puntatori allo storage non inizializzati non sono più un problema poichè contratti con puntatori non inizializzati risulteranno in errori di compilazione. Un altra possibile vulnerabilità è detta *incorrect-equality*. Questa vulnerabilità si verifica, solitamente, quando si controlla affinche un account ha abbastanza Ether o Tokens utilizzando una uguaglianza stretta, ciò è un qualcosa che un soggetto malevolo può facilmente manipolare per attaccare il contratto. Un esempio di questo tipo di vulnerabilità è 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 vulnerabilità è rappresentata dall'assert che controlla che il contratto abbia ricevuto la quantità 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 realtà gli Ether possono essere inviati ai contratti prima che vengano istanziati negli indirizzi stessi, poichè la generazione degli indirizzi dei contratti in Ethereum è un processo basato su dei nonce deterministici. L'attacco DoS che si basa su questa vulnerabilità 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 è una classe di vulnerabilità presente negli SmartContracts che permette ad un malintenzionato di rientrare nel contratto in modo inaspettato durante l'esecuzione della funzione originale. Questa vulnerabilità può essere utilizzata per rubare fondi e rappresenta la vulnerabilità più impattante dal punto di vista di perdita di fondi a seguito di attacchi. Il caso più famoso di questo attacco che lo ha anche reso noto è 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 vulnerabilità rimane comunque una delle minacce più pericolose per gli SmartContracts, poichè negli anni questo tipo di attacchi si è ripresentato notevole frequenza [13]. Un attacco di reentrancy può essere classificato in tre classi differenti:

- **Mono-Function**: la funzione vulnerabile è la stesa che viene chiamata più volte dall'attaccante, prima del completamento delle sue invocazioni precedenti. Questo è il caso più semplice di attacco reentrancy e di conseguenza il più facile da individuare.

- **Cross-Function**: questo caso è molto simile al caso di mono-function Reentrancy, ma in questo caso la funzione che viene chiamata dall'attaccante non è la stessa che fa la chiamata esterna. Questo tipo di attacco è 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 è invocato in un altro contratto prima che viene correttamente aggiornato. Avviene solitamente quando più 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 è stata completata. Questo permette all'attaccante di chiamare la funzione withdraw più volte prima che il balance venga settato a zero, permettendo all'attaccante di rubare fondi allo smart contract. Una versione più complessa dello stesso processo è 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 può 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 è fondamentale per comprendere come sono stati raccolti, pre-processati e utilizzati i dati, come è 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 è suddivisa nelle seguenti fasi principali:

1. Raccolta e preparazione dei dati: esplorazione del dataset utilizzato, delle tecniche di pre-processing applicate e delle modalità 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 sarà trattata in modo dettagliato, evidenziando le scelte metodologiche compiute e le motivazioni alla base di tali scelte. Questo approccio sistematico garantisce trasparenza e replicabilità del lavoro svolto, consentendo ad altri di comprendere e, eventualmente, replicare i risultati ottenuti.

4.1 Esplorazione dei dati

Il dataset [14] utilizzato in questo progetto è un dataset disponibile pubblicamente sulla piattaforma HuggingFace una delle più importanti piattaforme per il Natural Language Processing. HF è un'infrastruttura open-source che fornisce accesso a una vasta gamma di modelli di deep learning pre-addestrati, tra cui alcuni dei più avanzati nel campo del NLP. Questo dataset contiene informazioni su 106.474 SmartContracts pubblicati sulla rete Ethereum. Ogni elemento nel dataset è 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 è 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 è un array di valori che vanno da 1 a 5, dove ogni numero rappresenta la presenza di una vulnerabilità e 4 rappresenta un contratto safe, cioè privo di vulnerabilità.

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

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

Prima della costruzione dei modelli è stata affrontata una fase di analisi esplorativa dei dati. Questa fase è 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 è 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 è diviso in tre sottoinsiemi: training, validation e test set. Il dataset di training è 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

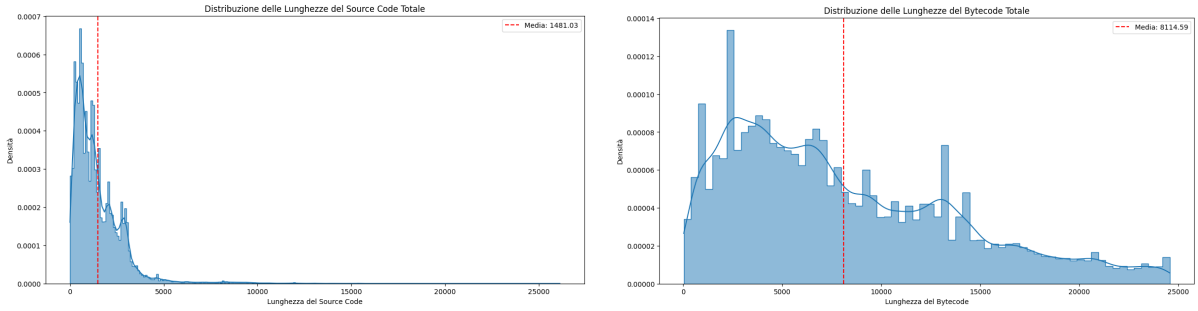
Dataset	**Count**	**%**
Train	227	0.285%
Test	51	0.319%
Validation	30	0.276%

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 leggibilità del grafico, abbiamo raggruppato i dati per quanto riguarda il source code in intervalli di 500 token. L'istogramma è 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 variabilità presente nei dati. Sul bytecode non è 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, cioè tutte quelle funzioni `getX()` le quali abbiano come unica istruzione una istruzione di return, poichè sono state assunte come funzioni corrette, l'eliminazione di queste stringhe è 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. Poichè



(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 però importante notare, che per molti casi di contratti che superano i 5000 token questi sono così lunghi poichè riportano in calce al contratto anche il codice sorgente di librerie esterne, che non è di interesse per la classificazione delle vulnerabilità.

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 vulnerabilità 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 già al dataset da cui sono stati sottratti i contratti privi di bytecode.

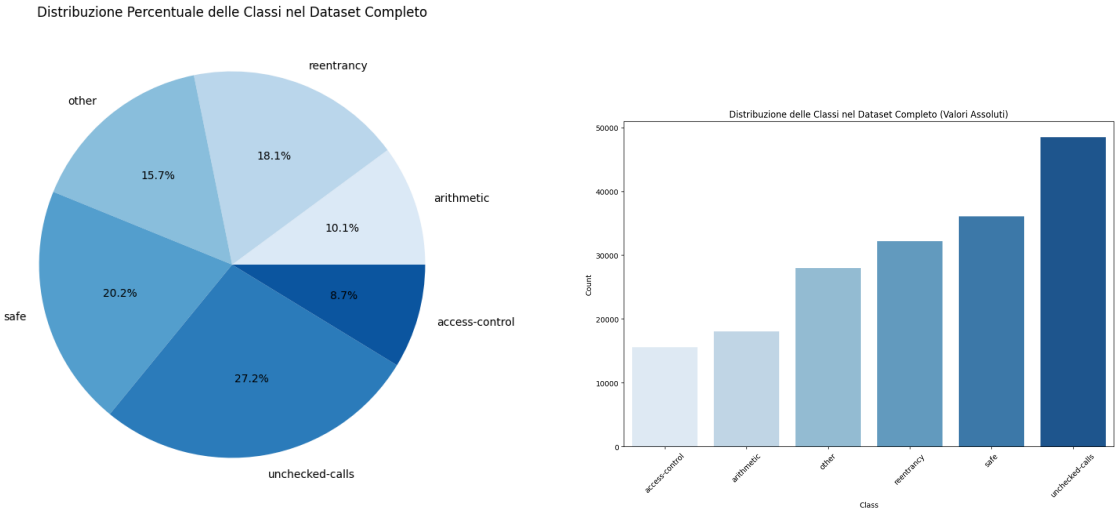
Distribuzione delle Classi

La Tabella 4.3 mostra la distribuzione delle classi per i tre dataset. È evidente che la classe 'unchecked-calls' è la più frequente in tutti e tre i dataset, mentre la classe 'access-control' è 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' è la più frequente in tutti e tre i dataset, con una presenza significativa soprattutto nel dataset di addestramento (36278 occorrenze).
- La classe 'access-control' è la meno frequente, con il numero più 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.

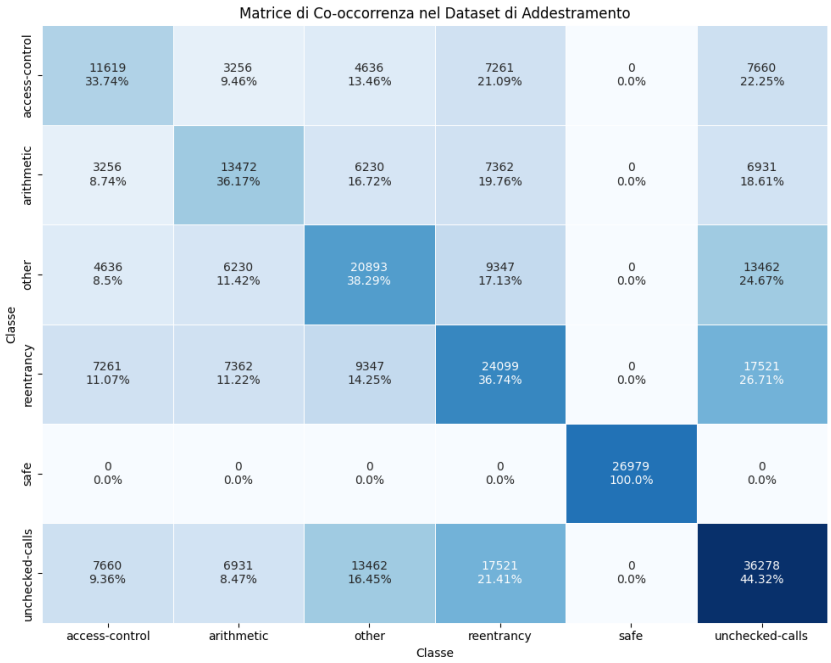


Figura 4.3: Matrice di Co-occorrenza nel Dataset di Addestramento

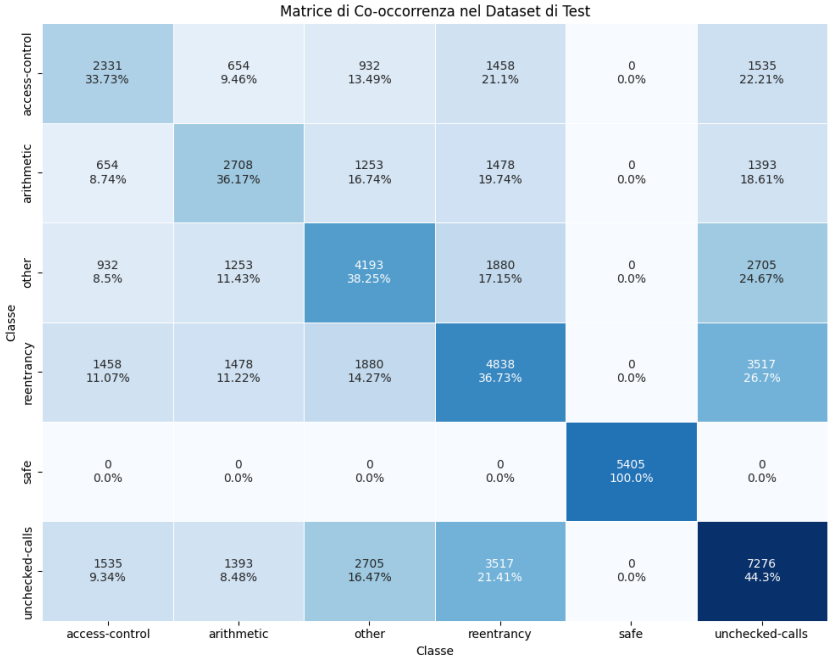


Figura 4.4: Matrice di Co-occorrenza nel Dataset di Test

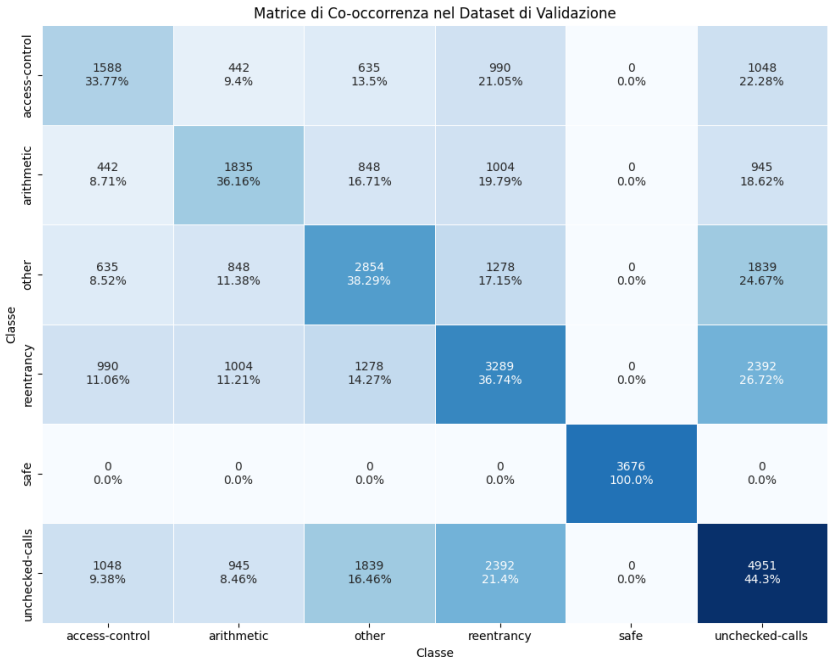


Figura 4.5: Matrice di Co-occorrenza nel Dataset di Validazione

Analizzando le matrici di co-occorrenza, notiamo che:

- La classe safe, che rappresenta i contratti privi di vulnerabilità, 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 vulnerabilità.
- Le classi 'arithmetic' e 'reentrancy' mostrano una co-occorrenza significativa, suggerendo che le vulnerabilità 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 vulnerabilità nei contratti intelligenti, poiché molte vulnerabilità non si verificano in isolamento ma tendono a manifestarsi insieme ad altre.

4.2 BERT

Il modello BERT (Bidirectional Encoder Representations from Transformers) rappresenta un pilastro fondamentale nel campo del Natural Language Processing (NLP),

grazie alla sua capacità di comprensione del contesto delle parole all'interno di una frase o di un testo più ampio. Questo modello, sviluppato da Google, si basa sull'architettura dei transformer, una classe di modelli neurali che ha dimostrato notevole successo nell'analisi del linguaggio naturale.

BERT si distingue per la sua capacità bidirezionale di elaborare il contesto linguistico. A differenza dei modelli NLP precedenti, che processavano il testo in modo sequenziale, interpretando le parole una dopo l'altra, BERT considera sia il contesto precedente sia quello successivo di ciascuna parola all'interno di una frase. Questo approccio bidirezionale consente a BERT di catturare relazioni semantiche più complesse e di fornire una rappresentazione più accurata del significato del testo.

Il funzionamento di BERT può essere compreso attraverso due fasi principali: l'addestramento e l'utilizzo.

Durante la fase di addestramento, BERT viene esposto a enormi quantità di testo, proveniente da varie fonti e domini. Utilizzando un processo noto come "pre-addestramento", il modello apprende i modelli linguistici e il contesto delle parole. Questo pre-addestramento coinvolge due compiti principali: la predizione di parole mascherate e la predizione della successione di frasi. Nel primo compito, BERT impara a prevedere le parole mancanti in una frase fornita, mentre nel secondo compito, il modello apprende a determinare se due frasi sono consecutive in un testo o sono state estratte casualmente da testi diversi.

Una volta completata la fase di addestramento, BERT può essere utilizzato per una vasta gamma di compiti NLP senza la necessità di ulteriori addestramenti specifici. Durante l'utilizzo, il modello riceve in input una sequenza di token, che possono essere parole, frammenti di testo o segmenti di frasi. Ogni token viene rappresentato come un vettore di caratteristiche, derivato dal contesto bidirezionale fornito da BERT durante l'addestramento. Queste rappresentazioni vettoriali possono essere utilizzate per svariati compiti, come classificazione di testo, analisi del sentiment, risposta alle domande, traduzione automatica e molto altro ancora.

In sintesi, il modello BERT ha rivoluzionato il campo del NLP introducendo una comprensione più approfondita e contestualizzata del linguaggio naturale. La sua capacità di catturare il contesto bidirezionale delle parole ha portato a miglioramenti significativi nelle prestazioni dei sistemi NLP su una vasta gamma di compiti e applicazioni. BERT rimane un pilastro fondamentale nell'ambito della comprensione automatica del linguaggio umano, consentendo a macchine e sistemi di interagire e comprendere il linguaggio umano in modo più naturale e preciso.

Capitolo 5

Results

Capitolo 6

Conclusioni

Bibliografia

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

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

[3] 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.

[4] 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.

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

[6] 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.

[7] 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.

[8] 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 254–269, New York, NY, USA, 2016. Association for Computing Machinery.

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

[10] 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.

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

[12] 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.

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

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

[15] 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.

[16] 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 9–16, New York, NY, USA, 2018. Association for Computing Machinery.

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

[18] 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.