Alma Mater Studiorum · Università di Bologna

SCUOLA DI SCIENZE
Corso di Laurea Magistrale in Informatica

Utilizzo di modelli basati sui Transformers per la classificazione delle vulnerabilità negli Smart Contracts Ethereum

Relatore:
Chiar.mo Prof.
Stefano Ferretti

Presentata da:
Marco Benito Tomasone

Sessione I
Anno Accademico 2023/2024

A nonna Emanuela e nonna Francesca,
ho finito le "scuole grandi".

Indice

Elenco delle figure

Elenco delle tabelle

Capitolo 1

Introduzione

Negli ultimi anni, gli Smart Contracts hanno acquisito una crescente popolarità, attirando l'attenzione di aziende e sviluppatori grazie alle loro caratteristiche innovative, come l'esecuzione automatica, la trasparenza, l'immutabilità e la decentralizzazione. Gli Smart Contracts sono programmi che eseguono automaticamente azioni al verificarsi di condizioni predefinite, operando su una blockchain, un registro distribuito che mantiene una lista di record, chiamati blocchi, collegati in modo sicuro mediante crittografia.

L'attenzione che gli Smart Contracts stanno generando viene confermata dal loro inserimento all'interno di una proposta di regolamentazione da parte dell'Unione Europea [8], in cui gli Smart Contracts vengono definiti come:

> "programmi informatici su registri elettronici che eseguono e regolano transazioni sulla base di condizioni prestabilite. Tali programmi possono potenzialmente fornire ai titolari e ai destinatari dei dati garanzie del rispetto delle condizioni per la condivisione dei dati."

La più grande piattaforma di Blockchain che permette l'esecuzione degli Smart Contracts è Ethereum. Gli Smart Contracts di Ethereum sono scritti in Solidity, un linguaggio di programmazione Turing-completo ad alto livello. Tuttavia, la scrittura di Smart Contracts sicuri rappresenta una sfida, in quanto anche piccoli errori possono portare a gravi conseguenze. Gli Smart Contracts sono immutabili, di conseguenza una volta pubblicati non possono essere modificati. Questa loro caratteristica, per quanto rappresenti una conferma del fatto che il contratto accettato dalle parti non possa essere modificato, può rilevarsi il loro principale punto debole. Difatti, qualsiasi errore o vulnerabilità presente in uno Smart Contract non può essere corretto, dando la possibilità a soggetti malintenzionati di rubare fondi o causare altri danni.

Diventa, di conseguenza, fondamentale per gli sviluppatori disporre di tool che permettano di rilevare automaticamente le vulnerabilità presenti negli Smart Contracts che

sviluppano, assicurandosi che siano privi di potenziali problemi prima della pubblicazione del contratto. Attualmente, per risolvere questo potenziale problema si fa riferimento a tecniche di analisi statica, che permettono di analizzare il codice sorgente o il 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, di contro però queste tecniche si rivelano essere molto lente e fanno troppo affidamento al riconoscimento di pattern conosciuti o si basano su dei set di regole predefinite, che possono non essere esaustive o non riconoscere nuove vulnerabilità.

Un'alternativa a queste tecniche è l'utilizzo di tecniche di Machine Learning e Deep Learning. Queste tecniche permettono di rilevare le vulnerabilità presenti negli Smart Contracts in maniera più rapida e con una maggiore accuratezza rispetto alle tecniche di analisi statica. In questo campo, negli ultimi anni c'è stato un notevole interesse e sono stati proposti diversi approcci che fanno uso di queste tecniche. Questi approcci si basano sull'addestramento di modelli di Machine Learning e Deep Learning, spaziando da modelli basati su reti neurali ricorrenti a modelli basati su reti neurali convoluzionali.

Questo lavoro di tesi si inserisce in questo contesto e propone un approccio basato su tecniche di Deep Learning utilizzando modelli basati sui Transformers per la rilevazione di vulnerabilità negli Smart Contracts. In particolare, questo lavoro ha l'obiettivo di costruire dei modelli che siano in grado di rilevare cinque classi di vulnerabilità:

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

Il contributo di questo lavoro mira a dimostrare l'efficacia dei modelli basati sui Transformers nella rilevazione di vulnerabilità all'interno degli Smart Contracts. In particolare, sono stati impiegati i modelli BERT, DistilBERT e CodeBERT per la classificazione. Considerando che questi modelli hanno un limite di 512 token per input e che i contratti possono essere significativamente più lunghi, è stato necessario adottare un approccio alternativo. Studi precedenti hanno dimostrato che suddividere i contratti in sottocontratti e classificarli mantenendo la stessa etichetta porta alla non convergenza della loss del modello. Pertanto, è stato sperimentato un metodo che prevede la segmentazione del testo in sottocontratti, seguita dall'aggregazione o concatenazione degli embedding generati per ciascun sottocontratto. Questi modelli sono stati addestrati sia utilizzando il bytecode che il codice sorgente dei contratti. Una volta ottenuti il miglior modello per ognuna delle modalità dei dati in input (codice sorgente e bytecode), è stato utilizzato un modello ensamble per ottenere un modello che effettua le predizioni a partire dalle

predizioni dei migliori modelli addestrati rispettivamente sul codice sorgente e sul bytecode. Infine, data la crescente diffusione di chatbot e assistenti virtuali nelle nostre vite, sia per il supporto a compiti di vario genere sia come ausilio alla programmazione, è stato testato il modello Gemini di Google per valutare la sua performance in un task di classificazione di questo tipo.

I risultati ottenuti dimostrano che i modelli basati sui Transformers sono altamente efficaci nel rilevare le vulnerabilità negli Smart Contracts. Tra questi, CodeBERT, un modello preaddestrato su codice sorgente, ha mostrato una notevole capacità di identificare le vulnerabilità sia nel codice sorgente che nel bytecode esadecimale. I modelli addestrati utilizzando diverse tecniche di aggregazione del testo hanno ottenuto performance superiori, evidenziando l'importanza di considerare l'interezza del contratto. Inoltre, l'uso di meta-classificatori in un modello ensemble ha ulteriormente migliorato le metriche di valutazione, dimostrando l'efficacia delle tecniche di combinazione dei modelli. In contrasto, il modello Gemini di Google, ha ottenuto risultati non soddisfacenti, sottolineando la necessità di utilizzare modelli specializzati per la rilevazione di vulnerabilità negli Smart Contracts.

La suddivisione di questo lavoro è articolata come segue:

- il capitolo due illustra le motivazioni alla base del presente lavoro di tesi, introducendo il concetto di Smart Contract e sottolineando l'importanza della rilevazione delle vulnerabilità in essi presenti. Verranno inoltre descritte alcune delle principali classi di vulnerabilità riscontrabili negli Smart Contract. Successivamente, sarà presentata una revisione della letteratura esistente in questo campo, con una panoramica dei lavori più significativi che hanno trattato questo tema.

- il terzo capitolo espone la metodologia adottata in questo lavoro. Inizialmente, sarà condotta un'analisi esplorativa del dataset, al fine di estrarne informazioni utili per la costruzione dei modelli. Successivamente, saranno introdotti i modelli utilizzati, BERT, DistilBERT e CodeBERT, con una dettagliata descrizione delle loro architetture e delle modalità di addestramento. Saranno illustrate le implementazioni e il setup dei vari esperimenti condotti. Inoltre, in questo capitolo sarà presentato il modello Gemini di Google e verrà spiegato come è stato applicato nel contesto di questo studio.

- il quarto capitolo riporta i risultati ottenuti nel corso del lavoro. Verranno introdotte le metriche di valutazione utilizzate e saranno presentate le performance dei vari modelli addestrati, valutate in termini di accuratezza, precisione, recall e F1-Score. Infine, saranno mostrati i risultati ottenuti dai meta-classificatori e dal modello Gemini di Google.

- il quinto ed ultimo capitolo riporta le conclusioni del lavoro svolto, con una discussione delle possibili direzioni future di ricerca.

Capitolo 2

Motivazioni e Lavori Correlati

In questa sezione della tesi, verranno illustrate in modo approfondito le motivazioni che hanno guidato la scelta del tema di ricerca. Inoltre, saranno presentati e discussi alcuni dei contributi più rilevanti e significativi della letteratura esistente che si sono occupati di effettuare ricerca sul tema.

2.1 Motivazioni

Nel 1994, Nick Szabo, un informatico e crittografo, propose la prima descrizione ufficiale degli smart contracts [43]:

> "Smart contracts combine protocols, users interfaces, and promises expressed via those interfaces, to formalize and secure relationships over public networks. This gives us new ways to formalize the digital relationships which are far more functional than their inanimate paper-based ancestors. Smart contracts reduce mental and computational transaction costs, imposed by either principals, third parties, or their tools."

L'idea di base dietro uno smart contract è che molte clausole contrattuali possono essere incorporate in hardware e software con cui tutti i giorni interagiamo. Un primo esempio di antenato degli smart contracts possono essere considerati i distributori automatici, che erogano un bene (una bevanda) in cambio di un pagamento. Questo processo è automatizzato e non richiede l'intervento di un intermediario. Il distributore automatico rappresenta un contratto con il portatore: chiunque abbia monete può partecipare a uno scambio con il venditore. La cassetta di sicurezza e altri meccanismi di sicurezza proteggono le monete e il contenuto dagli attacchi, sufficientemente da permettere l'installazione redditizia di distributori automatici in una vasta gamma di aree.

2.1.1 Blockchain

Ad oggi, gli Smart Contracts hanno avuto un'ampissima diffusione grazie allo sviluppo e alla diffusione della tecnologia blockchain. La blockhain è una tecnologia che permette di costruire un ledger distribuito, cioè un registro condiviso e sincronizzato tra tutti i nodi della rete. Questo registro è immutabile e contiene tutte le transazioni che sono state effettuate. La blockchain è stata introdotta per la prima volta nel 2008 da un autore (o un gruppo di autori) sotto lo pseudonimo di Satoshi Nakamoto, come parte del progetto Bitcoin [30]. Le principali carratteristiche della blockchain sono [60]:

- **Decentralizzazione**: a differenza dei sistemi di transazione centralizzati, in cui ogni transazione deve essere validata da un'autorità centrale, nella blockchain la validazione delle transazioni avviene in maniera distribuita. In una blockchain, le transazioni sono validate da una rete di nodi distribuiti che collaborano per raggiungere un consenso sulla validità delle transazioni utilizzando specifici algoritmi di consenso.

- **Persistenza:** le transazioni registrate sulla blockchain sono immutabili e non possono essere modificate una volta confermate. I blocchi che contengono transazioni invalide vengono immediatamente rilevati e scartati dalla rete. Questo meccanismo garantisce che il registro sia accurato e affidabile.

- **Anonimità:** ogni utente interagisce con la blockchain con un indirizzo generato, che non rivela la reale indentità dell'utente. Questo garantisce un certo grado di anonimità e privacy. Allo stesso tempo, la blockchain non può garantire un anonimato totale.

A livello architetturale, la blockhain non è altro che una sequenza di blocchi, che tengono una lista completa di record di transazioni, come un ledger pubblico. Ogni blocco contiene un hash crittografico del blocco precedente, un timestamp e un set di transazioni. Questi blocchi sono concatenati insieme per formare una catena, da cui il nome "blockchain". Il primo blocco di una blockchain è chiamato *blocco genesi* ed è l'unico a non avere un blocco precedente. Ogni blocco successivo è collegato al blocco precedente e può avere solo un padre.
Le blockchain possono essere categorizzate in tre tipi principali [6]:

- **Blockchain pubbliche**: sono blockchain che chiunque nel mondo può leggere, utilizzare per inviare transazioni e verificare la validità delle transazioni. Tutti i nodi della rete possono partecipare al processo di consenso e alla validazione delle transazioni. Queste blockchain sono considerate completamente decentralizzate. Un esempio di blockchain pubblica è Bitcoin.

- **Blockchain private**: le blockchain private sono sistemi in cui i permessi di scrittura sono centralizzati e controllati da un'unica organizzazione. I permessi di lettura

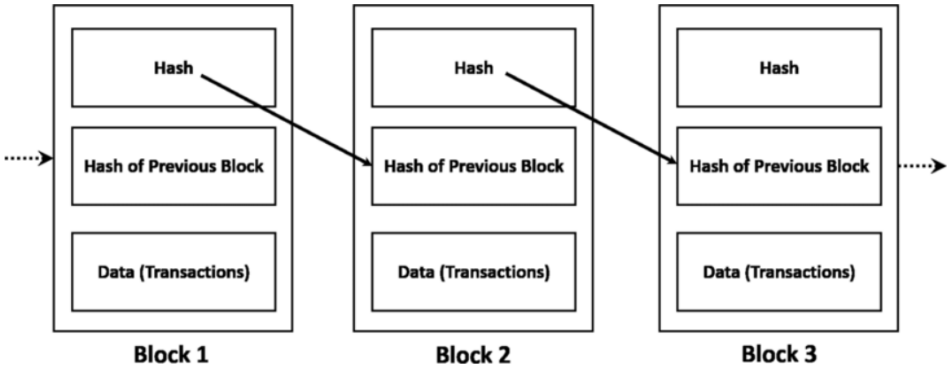


Figura 2.1: Esempio di architettura di una blockchain

possono essere pubblici o limitati a un gruppo selezionato di utenti, a seconda delle esigenze dell'organizzazione. Questo tipo di blockchain viene spesso utilizzato in ambito aziendale per garantire maggiore controllo e sicurezza sulle transazioni e i dati sensibili. Le blockchain private permettono di implementare politiche di accesso più rigorose e di mantenere la privacy delle operazioni interne, rendendole ideali per applicazioni come la gestione della supply chain, la finanza aziendale e la condivisione di dati riservati tra partner commerciali.

- **Blockchain consorziate**: sono blockchain in cui il processo di consenso è controllato da un numero limitato di nodi; ad esempio si può pensare ad un consorzio di istituzioni finanziare in cui ognuna opera come un nodo della rete e per raggiungere un consenso è necessario che la maggioranza dei nodi sia d'accordo. Queste blockchain vengono considerate parzialmente decentralizzate.

Una delle sfide principali della blockchain (e, più in generale, dei sistemi distribuiti) è il problema del consenso. Poichè nella blockchain non ci sono dei nodi centrali che garantiscano che i ledger nei nodi distribuiti siano tutti uguali, è necessaria la presenza di un protocollo che assicuri che i ledger siano consistenti tra i vari nodi. Questo protocollo è chiamato protocollo di consenso. Esistono diversi protocolli di consenso, tra i principali possiamo citare:

- **Proof of Work (PoW)**: è il protocollo di consenso utilizzato da Bitcoin. Nel proof of work, i nodi della rete dimostrano di non essere dei malintenzionati spendendo una grande quantità di lavoro computazionale. Questo lavoro è fatto risolvendo un problema crittografico complesso. Nel PoW, ogni nodo della rete calcola l'hash del blocco, che deve però contenere un nonce ed essere inferiore ad un certo valore target. Quando un nodo trova un hash che soddisfa questi requisiti, il blocco viene inviato a tutti gli altri nodi nella rete che devono confermare la validità del blocco. Questo processo è chiamato "mining" e i nodi che lo fanno sono chiamati

"minatori". Quando un blocco viene confermato, il nodo che ha trovato il blocco riceve una ricompensa in criptovaluta. Tuttavia, PoW richiede una grande quantità di energia elettrica, rendendo il processo costoso e poco sostenibile.

- **Proof of Stake (PoS)**: Il Proof of Stake (PoS) è un protocollo di consenso alternativo a PoW. Poichè PoW richiede un'ampia quantità di lavoro e di energia elettrica, PoS propone un approccio basato sulla quantità di criptovaluta posseduta da un nodo. L'idea di base di PoS è che i nodi che possiedono una grande quantità di criptovaluta sono meno propensi a compiere azioni malevole, poichè ciò potrebbe danneggiare il valore della criptovaluta che possiedono. In PoS, i nodi sono selezionati per validare i blocchi in base alla quantità di criptovaluta che possiedono. Poichè basarsi solo sulla quantità di criptovaluta posseduta potrebbe portare a una centralizzazione del processo di consenso, sono stati proposti diversi algoritmi per selezionare i nodi in modo casuale. Ad esempio, Blackcoin [49] utilizza un algoritmo di selezione casuale basato su una lotteria; Peercoin [59] utilizza un algoritmo di selezione casuale basato sulla quantità di criptovaluta posseduta oltre che sulla quantità di tempo a partire da cui la criptovaluta è stata posseduta. Al contrario di PoW, PoS non richiede una grande quantità di lavoro e di energia elettrica, rendendo il processo di consenso più efficiente e sostenibile, ma al diminuire del costo di mining di un blocco, aumentano le probabilità di attacchi alla blockchain.

- **Delegated Proof of Stake (DPoS)**: è una variante di PoS in cui i possessori di criptovaluta possono votare per i nodi che desiderano che validino i blocchi. I nodi più votati vengono selezionati per validare i blocchi. Questo riduce il numero di nodi che devono essere coinvolti nel processo di consenso. Questo sistema mira a migliorare l'efficienza e la velocità delle transazioni, mantenendo un certo grado di decentralizzazione. Tuttavia, può introdurre rischi di centralizzazione del potere di voto e governance all'interno della rete.

Questi protocolli sono solo alcuni dei tanti protocolli di consenso che sono stati proposti. Ogni protocollo ha i suoi vantaggi e svantaggi e può essere adatto a diversi contesti e applicazioni.

2.1.2 Ethereum

Tra le principali blockchain pubbliche, una delle più importanti è Ethereum. Ethereum è una piattaforma open-source basata su blockchain che permette di creare e distribuire applicazioni decentralizzate (dApp). Proposta per la prima volta nel 2014 da Vitalik Buterin [55], Ethereum estende i concetti introdotti da Bitcoin, pur presentando alcune differenze chiave.

Mentre Bitcoin è progettato principalmente come una valuta digitale, Ethereum è pensata per essere una piattaforma per la creazione di applicazioni decentralizzate. La

sua criptovaluta nativa, Ether (ETH), viene utilizzata per pagare le transazioni sulla rete e per incentivare i partecipanti a contribuire alla sicurezza e al funzionamento della rete stessa.

La blockchain di Ethereum è pubblica e permissionless, il che significa che chiunque può partecipare alla rete e inviare transazioni. Inizialmente, la blockchain Ethereum utilizzava il PoW come protocollo di consenso, migrato successivamente al PoS, per migliorare l'efficenza energetica della rete. Ethereum è diventata una delle blockchain più popolari e utilizzate al mondo, con un ecosistema di sviluppatori molto attivo e una vasta gamma di applicazioni decentralizzate che spaziano dai giochi ai servizi finanziari.

2.1.3 Smart Contracts

Una delle principali innovazioni di Ethereum è l'introduzione degli Smart Contracts. Gli smart contracts sono programmi memorizzati ed eseguiti su una blockchain. Possono essere definiti come contenitori di codice che incarnano e replicano i termini di contratti del mondo reale in un dominio digitale [44]. Agiscono come accordi che facilitano lo scambio di denaro, proprietà, azioni o qualsiasi altro valore in modo trasparente, eliminando la necessità di un intermediario. Gli smart contracts vengono eseguiti automaticamente quando vengono soddisfatte determinate condizioni. Ad esempio, un contratto di assicurazione potrebbe pagare automaticamente un risarcimento al verificarsi di un determinato evento. Poichè gli smart contracts sono accessibili ad entrambe le parti, non ci sono dispute una volta che il contratto è stabilito.

Gli smart contracts di Ethereum possono essere visti come agenti autonomi che vivono sulla blockchain Ethereum. Possono inviare e ricevere transazioni, memorizzare dati e interagire con altri contratti. Gli smart contracts sono scritti in un linguaggio di programmazione chiamato Solidity, progettato appositamente per la creazione di smart contracts. Solidity è un linguaggio di programmazione ad alto livello e Turing-completo, progettato per essere eseguito sulla blockchain Ethereum. Gli smart contracts di Ethereum sono eseguiti su una macchina virtuale chiamata Ethereum Virtual Machine (EVM). L'EVM è una macchina virtuale Turing-completa che esegue il codice Solidity e garantisce che il codice sia eseguito in modo deterministico e sicuro.

Poichè gli smart contracts vengono eseguiti sulla blockchain e sono Turing-completi, sono soggetti al problema dell'arresto (Halting Problem), cioè non è possibile sapere a priori se l'esecuzione di un contratto terminerà o meno. Questo potrebbe rendere le blockchain facilmente soggette ad attacchi di tipo Denial of Service. Per evitare questo problema, Ethereum ha introdotto il concetto di gas. Il gas è una misura dell'uso delle risorse computazionali e di memoria di un contratto. Ogni operazione che un contratto esegue costa una certa quantità di gas. Gli utenti che inviano transazioni devono specificare il prezzo del gas che sono disposti a pagare per l'esecuzione della transazione. Se il gas fornito non è sufficiente per completare l'esecuzione del contratto, la transazione viene annullata e il gas speso non viene restituito all'utente. Al contrario, se l'uten-

te consuma meno del gas disponibile, l'eccedenza viene restituita. Questo meccanismo garantisce che i contratti non possano eseguire operazioni infinite e che gli utenti non possano abusare della rete Ethereum inviando transazioni che richiedono una grande quantità di risorse computazionali.
Gli Smart Contracts introducono, numerosi vantaggi [44]:

- **Risparmio**: utilizzando gli smart contracts, si eliminano il tempo e il denaro spesi nell'attesa e nel pagamento di un soggetto terzo per processare una transazione. Le operazioni vengono automatizzate, riducendo così i costi associati ai servizi intermediari e accelerando il processo di transazione.

- **Sicurezza e fiducia:** Una volta che le transazioni sono cifrate e registrate sulla blockchain, sono quasi impossibili da modificare. Questo perchè per alterare una singola transazione sarebbe necessario modificare l'intera catena di blocchi, rendendo l'operazione estremamente complessa e dispendiosa per un eventuale attaccante. Inoltre, gli smart contracts, una volta pubblicati, sono immutabili e non possono essere alterati, garantendo che le parti coinvolte possano fidarsi delle condizioni stabilite nel contratto senza timore di manipolazioni future.

- **Accuratezza, efficenza e rapidità:** Dal momento in cui le condizioni stabilite nel contratto sono soddisfatte, l'esecuzione del contratto avviene immediatamente e automaticamente, eliminando la necessità di un intermediario e la compilazione di documentazione cartacea. Questo automatismo non solo accelera i tempi di esecuzione, ma riduce anche il rischio di errori umani associati alla gestione manuale dei documenti.

2.2 Vulnerabilità negli Smart Contracts

Poichè gli Smart Contracts sono eseguiti su una blockchain pubblica e sono accessibili a tutti, è importante garantire che siano sicuri e privi di vulnerabilità. Come precedentemente già introdotto, una delle caratteristiche principali delli smart contracts è la loro immutabilità. Dopo che un contratto è stato pubblicato sulla blockchain, non può essere modificato o cancellato. Questa proprietà assicura che il comportamento del contratto sia prevedibile e che le parti coinvolte possano fidarsi del contratto. Tuttavia, questa proprietà rende anche gli Smart Contracts vulnerabili a errori e vulnerabilità. Se un contratto contiene una vulnerabilità, essa non può essere corretta e rimarrà vulnerabile per sempre. Questo rende gli Smart Contracts un obiettivo attraente per gli attaccanti che possono sfruttare le vulnerabilità negli Smart Contracts per rubare fondi, bloccare i fondi o causare altri danni [52].
Le vulnerabilità negli Smart Contracts possono essere divise in diverse categorie ed esistono vari criteri di classificazione delle stesse. Una delle classificazioni più comuni è quella proposta da [47] che classifica le vulnerabilità in quattro categorie principali:

- **Security**: vulnerabilità che possono portare a perdite di fondi o a comportamenti inaspettati a seguito dell'interazione con un account o un contratto malevolo.
- **Funzionali**: problemi che causano la violazione delle specifiche funzionalità del contratto, per come esse erano state intese.
- **Operazionali**: vulnerabilità che possono portare a problemi operativi, come performance scarse e tempi di esecuzione lunghi.
- **Sviluppo**: problemi legati al codice sorgente del contratto, come codice di bassa qualità che lo rendono difficile da comprendere e migliorare.

Queste categorie non tengono in considerazione, però, di informazioni e conoscenze di dominio sugli Smart Contracts. Un'altra classificazione delle vulnerabilità è quella proposta da [22] che propone una classificazione basata su conoscenze di dominio sugli Smart Contracts. Ad esempio, poichè gli smart contracts possono interagire con altri contratti e questa interazione può portare a vulnerabilità come l'attacco di reentrancy o denial of service, queste vulnerabilità sono state classificate come vulnerabilità di tipo *inter-contractual*. Questa classificazione conta cinque categorie principali di vulnerabilità:

- **Inter-contractual**
- **Contractual**
- **Arithmetic**
- **Gas-Related**
- **Transactional**

Partendo da questa classificazione è possibile identificare diversi tipi di vulnerabilità, più e meno comuni, che possono essere presenti negli Smart Contracts. In questo lavoro di tesi le vulnerabilità sono state classificate in cinque classi diverse:

- **Reentrancy**
- **Unchecked-Calls**
- **Access-Control**
- **Arithmetic**
- **Others**

Di seguito, verranno presentate alcune delle vulnerabilità più comuni e significative che possono essere presenti negli Smart Contracts, seguendo la classificazione utilizzata nel resto del lavoro ma con un occhio di riguardo alla classificazione proposta da [22].

2.2.1 Reentrancy

Le vulnerabilità di tipo reentrancy sono alcune delle vulnerabilità più comuni e pericolose che possono essere presenti negli Smart Contracts e ricadono nella categoria di vulnerabilità inter-contractual.
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 [9], 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 [34]. Un attacco di reentrancy può essere classificato in tre classi differenti [21, 61]:

- **Mono-Function**: la funzione vulnerabile è la stessa 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 venga 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. Questo esempio mostra un contratto vulnerabile a un attacco di tipo mono-function:

```
// 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 funzione che chiama `transfer()` per trasferire fondi spesi prima che il bilancio sia settato a zero dalla funzione `withdraw()`. Gli attacchi Cross-contract sono anche detti Read-only Reentrancy, in cui l'attaccante invece di rientrare nello stesso contratto in cui i cambiamenti di stato sono stati fatti, rientra in un contratto che legge lo stato dal contratto originale. Un esempio di questo caso è il seguente:

```
    // UNSECURE
contract A {
  // Has a reentrancy guard to prevent reentrancy
  // but makes state change only after external call to sender
  function withdraw() external nonReentrant {
    uint256 amount = balances[msg.sender];
    (bool success,) = msg.sender.call{value: balances[msg.sender
       ]}("");
    require(success);
    balances[msg.sender] = 0;
  }
}

contract B {
  // Allows sender to claim equivalent B tokens for A tokens they
      hold
```

```
  function claim() external nonReentrant {
    require(!claimed[msg.sender]);
    balances[msg.sender] = A.balances[msg.sender];
    claimed[msg.sender] = true;
  }
}
```

Come è possibile vedere, nonostante entrambe le funzioni abbiano il modifier `nonReentrant`, è comunque possibile per un attaccante chiamare `B.claim` durante la callback in `A.withdraw` e poichè l'aggiornamento del bilancio dell'attaccante non è stato ancora completato l'esecuzione ha successo.

2.2.2 Unchecked-Calls

Questa classe di vulnerabilità ricade sempre nella categoria di vulnerabilità inter-contractual ed è nota in letteratura anche come *mishandled exceptions, external-calls* ed *exceptions disorders*.

Questa vulnerabilità si verifica quando un contratto chiama un funzioni di basso livello come `call, send, delegatecall` senza controllare il risultato di queste chiamate. La differenza tra queste funzioni e la funzione `transfer` è che quest'ultima propaga un'eccezione nel caso in cui venga generata un'eccezione nel contratto chiamate, al contrario le altre funzioni in caso di fallimento non generano un'eccezione ma ritornano il booleano false [40], che se non controllato adeguatamento non stoppa l'esecuzione del contratto [36]. Mostriamo ora un esempio di contratto reso protetto rispetto a questo tipo di vulnerabilità, semplicemente controllando il risultato della chiamata:

```
    // Simple transfer of 1 ether
(bool success,) = to.call{value: 1 ether}("");
// Revert if unsuccessful
require(success);
```

2.2.3 Access-Control

La vulnerabilità di access-control è una vulnerabilità di tipo contrattuale. 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 governare chi può interagire con le varie funzionalità all'interno del contratto. Un esempio di questo tipo 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, non avendo allo stesso tempo 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.

2.2.4 Arithmetic

Le vulnerabilità di tipo aritmetico [22] 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 più 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 diametralmente 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, visibile nella figura 2.2. 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;
    }
```

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, portando a comportamenti inaspettati e a perdite di fondi.

Figura 2.2: Andamento del prezzo di BEC prima e dopo l'attacco

2.2.5 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 si riferiscono a tutte quelle vulnerabilità nate 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.

Incorrect-equality

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.

Denial of Service

Un'altra classe di vulnerabilità che cade nella categoria di vulnerabilità inter-contractual è la vulnerabilità di tipo Denial of Service (DoS) [22]. Questa vulnerabilità può avvenire per varie ragioni. Un esempio è il caso in cui si cerca di mandare fondi a un contratto le cui funzionalità dipendono dal successo del trasferimento di questi fondi. Se il trasferimento fallisce, il contratto potrebbe non essere in grado di eseguire le sue funzionalità e rimanere bloccato. Un attaccante potrebbe sfruttare questo comportamento per bloccare il contratto inviando fondi con un contratto che non permette il trasferimento di fondi. Un esempio di questo tipo di vulnerabilità è il seguente:

```
// UNSECURE
contract Auction {
    address currentLeader;
    uint highestBid;

    function bid() payable {
        require(msg.value > highestBid);

        require(currentLeader.send(highestBid)); // Refund
           the old leader, if it fails then revert

        currentLeader = msg.sender;
        highestBid = msg.value;
    }
}
```

In questo esempio qualora un'attaccante faccia la bid da uno smart contract con una funzione che annulla tutti i pagamenti, che quindi non può essere rimborsato, il contratto Auction rimarrebbe bloccato e non potrebbe più ricevere offerte. Questo è un esempio molto semplice di come un attaccante possa sfruttare una vulnerabilità di tipo DoS per bloccare un contratto.

Suicidal

Questa vulnerabulità è presente in letteratura anche come *self-destruct*. Precedenti studi [31] classificano uno smart contract come Suicidal quando utilizza il comando *selfdestruct* senza controllare correttamente il suo utilizzo.

La vulnerabilità suicida degli smart contract rappresenta un difetto critico che consente la terminazione prematura e irreversibile di un contratto sulla blockchain. Questa vulnerabilità si manifesta quando uno smart contract include una funzione che può essere chiamata da utenti malintenzionati per auto-distruggere il contratto stesso.
La funzione `selfdestruct` in Solidity, ad esempio, è stata progettata per consentire agli sviluppatori di rimuovere contratti obsoleti dalla blockchain e restituire i fondi residui

a un indirizzo specificato. Tuttavia, se questa funzione non è adeguatamente protetta o viene esposta accidentalmente, può essere sfruttata per terminare il contratto e compromettere tutte le risorse associate. Per prevenire tale vulnerabilità, è fondamentale implementare rigorosi controlli di accesso e assicurarsi che solo gli utenti autorizzati possano eseguire azioni di questa portata, oltre a condurre audit di sicurezza approfonditi prima del deployment del contratto sulla rete.

Un esempio di smart contract vulnerabile in Solidity è il seguente:

```
pragma solidity ^0.8.0;

contract VulnerableContract {
    address payable owner;

    constructor() {
        owner = payable(msg.sender);
    }

    function destroy() public {
        selfdestruct(owner);
    }
}
```

In questo esempio, la funzione `destroy` può essere chiamata da chiunque, causando la distruzione del contratto e il trasferimento dei fondi rimanenti all'indirizzo del proprietario. Per evitare questa vulnerabilità, la funzione `destroy` dovrebbe essere protetta con un controllo di accesso, come mostrato di seguito:

```
function destroy() public {
    require(msg.sender == owner, "Only the owner can destroy the
       contract");
    selfdestruct(owner);
}
```

2.3 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 blockchain. Proprio per questo motivo, sono stati sviluppati e proposti diversi approcci per la rilevazione automatica di vulnerabilità all'interno degli Smart Contracts. In questa sezione 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 e 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 [32, 29], Oyente [2, 26], Mythril [1], Manticore [28] 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 vulnerabilità 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 [13], Securify [48], SmartCheck [47] 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. [51] 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, LogisticRegression, 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) [27]. 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 [38]. 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) utilizzando 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 Smart-Contracts con delle buone performance, i migliori risultati si attestano con un MicroF1 score del 83% e mostrano come i migliori risultati siano dati da delle reti 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 [18] 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. [10] ha proposto un approccio basato sulla fusione di feature multimodali, analizzando contemporaneamente codice sorgente, bytecode e grafi di controllo del flusso. Per ognuna di queste tre feature è stato addestrato 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.

Questo lavoro di tesi, non è il primo lavoro che fa uso di modelli di Deep Learning basati su transformers per la classificazione delle vulnerabilità. L'approccio proposto da da Tang et al. [45], ha utilizzato le abilità di encoding del modello CodeBERT per la classificazione del codice sorgente Solidity. Il lavoro ha proposto una classificazione degli encoding di CodeBERT tramite una LSTM, una CNN ed un classico layer totalmente connesso. I risultati hanno mostrato come il layer fully-connected ottenga i migliori risultati con un F1-Score del 93%. La principale limitazione di questo lavoro è legata al fatto che il dataset utilizzato è molto piccolo, con meno di 10.000 campioni, inoltre, l'altra grande limitazione è che questi modelli non classificavano interi smart contracts, ma solo singole funzioni.

Capitolo 3

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, quali e 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 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 garantisce trasparenza e replicabilità del lavoro svolto, consentendo ad altri di comprendere e, eventualmente, replicare i risultati ottenuti.

3.1 Esplorazione dei dati

La prima fase, quando ci si approccia ad un problema che riguarda la costruzione di modelli e l'analisi di dati è sicuramente l'analisi esplorativa dei dati. Questa fase è fondamentale per comprendere meglio la struttura del dataset e dei contratti da classificare, per individuare eventuali problemi e per ottenere una visione d'insieme dei dati. 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 [37] utilizzato in questo progetto è un dataset disponibile pubblicamente sulla piattaforma HuggingFace. Hugging Face è una piattaforma e una comunità dedicata all'intelligenza artificiale, nota soprattutto per la sua vasta raccolta di modelli pre-addestrati per il linguaggio naturale, computer vision e altri compiti legati all'IA. Questa piattaforma è diventata una risorsa fondamentale per gli sviluppatori ed i ricercatori che lavorano nell'ambito del machine learning grazie alla sua libreria nominata "Transformers" (che verrà utilizzata in questo lavoro), che offre un'implementazione semplice e potente di alcuni dei modelli più avanzati nel campo del NLP (Natural Language Processing). Hugging Face non solo fornisce modelli pre-addestrati, ma incoraggia anche la collaborazione e la condivisione all'interno della comunità attraverso il suo hub online, dove gli utenti possono accedere, condividere e collaborare su modelli, dataset e altri componenti relativi all'IA. Inoltre, Hugging Face supporta un ecosistema di strumenti e risorse open-source che facilitano lo sviluppo e l'implementazione di soluzioni basate su IA.

Questo dataset contiene informazioni su ben 106.474 Smart Contracts pubblicati sulla rete Ethereum. Il dataset è stato costruito a partire dalla lista Smart Contracts verificati forniti da Smart Contract Sanctuary, una risorsa online che fornisce una lista verificata di Smart Contracts. Successivamente, il codice sorgente dei contratti intelligenti è stato scaricato sia dal suddetto repository che tramite Etherscan e poi appiattito utilizzando lo strumento di appiattimento dei contratti Slither. Il bytecode è stato scaricato utilizzando la libreria Web3.py, in particolare la funzione web3.eth.getCode(). Infine, ogni Smart Contract è stato analizzato utilizzando il framework di analisi statica Slither. Lo strumento ha individuato 38 diverse classi di vulnerabilità nei contratti raccolti e sono stati quindi mappati su 6 etichette. Questi mapping sono stati derivati seguendo le linee guida del Decentralized Application Security Project (DASP) e del Smart Contract Weakness Classification Registry. Ogni entry del dataset è composta da quattro feature:

- **Address**: l'indirizzo del contratto
- **SourceCode**: il codice sorgente del contratto, scritto in linguaggio Solidity
- **ByteCode**: il codice bytecode del contratto in esadecimale, 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
- Unchecked-Calls

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. La tabella 3.1 riassume in termini percentuali la suddivisione dei dati nei tre set.

Split	**Numero di elementi**	**Percentuale**
Training	79,641	74.80%
Validazione	10,861	10.20%
Test	15,972	15.00%

Tabella 3.1: Percentuali di ogni split rispetto al totale

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 3.2. Non ci sono valori nulli nelle altre colonne del dataset.

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

Tabella 3.2: Conteggio e Percentuale di Contratti Senza Bytecode per Dataset

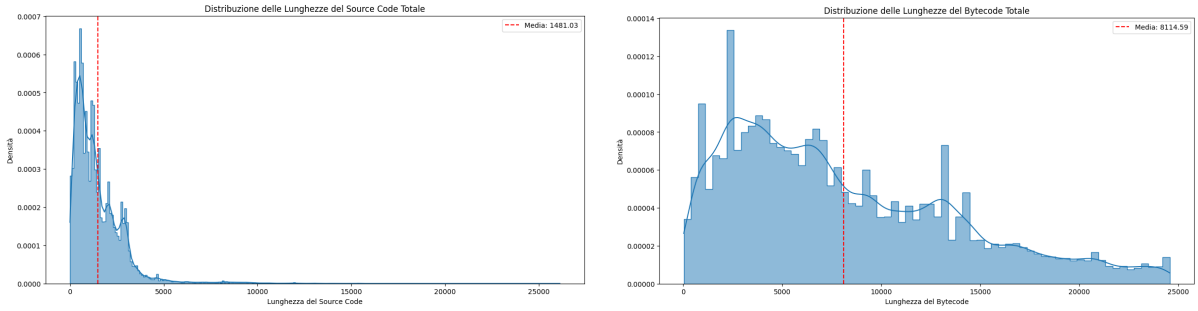
3.1.1 Analisi delle Lunghezze dei contratti

Per ottenere una visione d'insieme 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.

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, poichè essendo il codice esadecimale prodotto direttamente dal compilatore Solc di Solidity, esso è già nella forma più compatta possibile e non richiede ulteriori operazioni di riduzione delle dimensioni.
D'altro canto, 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. 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, mostrata rispettivamente nelle Figure 3.1a e 3.1b.

La media è un indicatore molto utile per guadagnare informazioni sulla lunghezza dei



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

(b) Distribuzione delle Lunghezze del Bytecode dopo il preprocessing

Figura 3.1: Distribuzioni delle lunghezze del source code e del bytecode.

nostri contratti, ma potrebbe essere facilmente influenzato da valori estremi. Per questo motivo, è importante considerare anche la mediana, che rappresenta il valore centrale di un insieme di dati ordinati. La mediana del source code è di 1147 token, mentre la mediana del bytecode è di 6733 token. Entrambi i valori dimostrano come i contratti siano in generale, formati da sequenze molto lunghe. Poichè successivamente andremo a classi-

ficare 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 invece troncati. I risultati sono mostrati nella Tabella 3.3.

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 3.3: 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 sono di interesse per la classificazione delle vulnerabilità.

3.1.2 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.

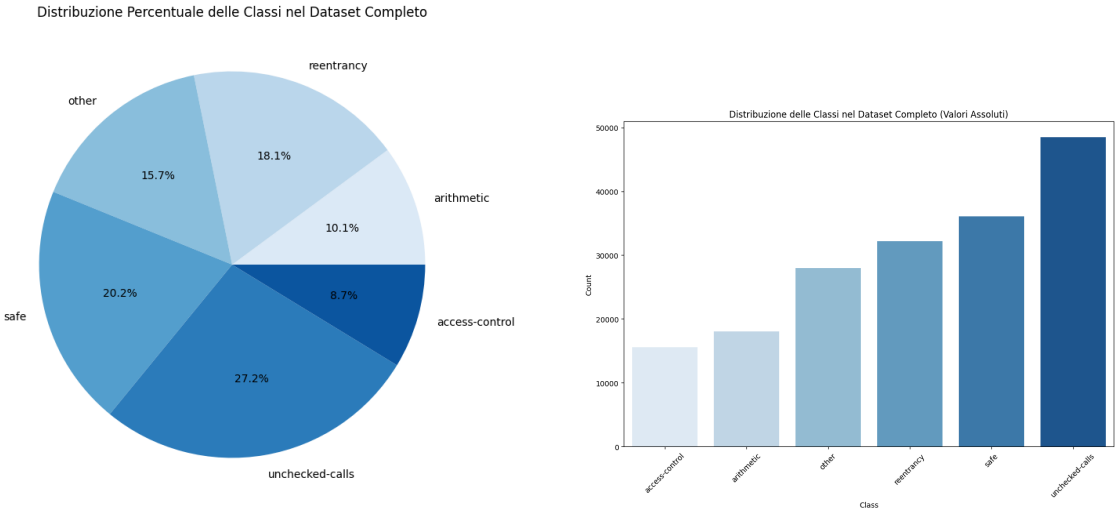
Distribuzione delle Classi

La Tabella 3.4 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 3.4: Distribuzione delle Classi nei Dataset di Addestramento, Test, Validazione e Completo

Le Figure 3.2a e 3.2b 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



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

Figura 3.2: Distribuzioni delle Classi nell'intero dataset, in termini relativi e assoluti.

osservare che:

- 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.
- La classe 'access-control' è la meno frequente.
- Le classi 'safe' e 'reentrancy' sono anche abbastanza rappresentate

Matrici di Co-occorrenza

Le Tabelle 3.3, 3.4 e 3.5 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.

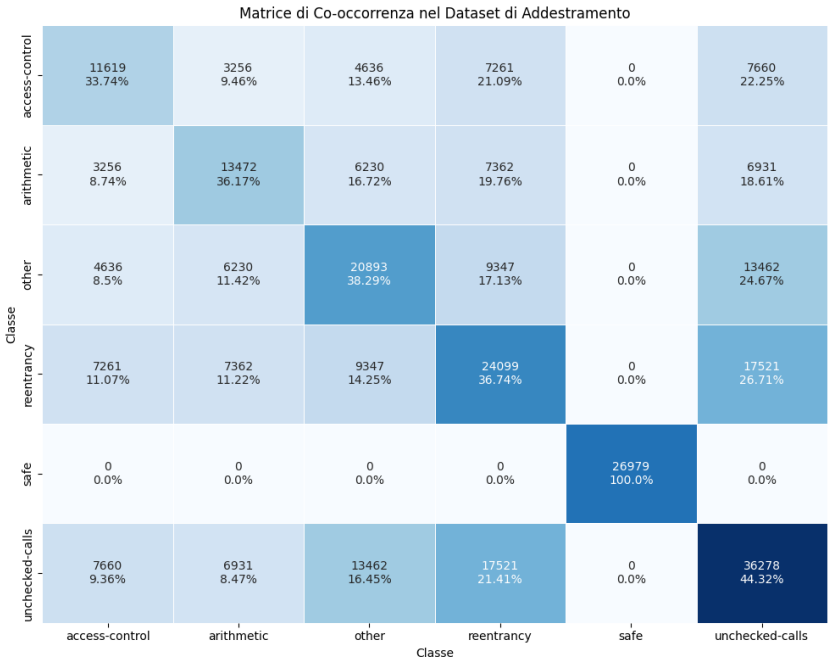


Figura 3.3: Matrice di Co-occorrenza nel Dataset di Addestramento

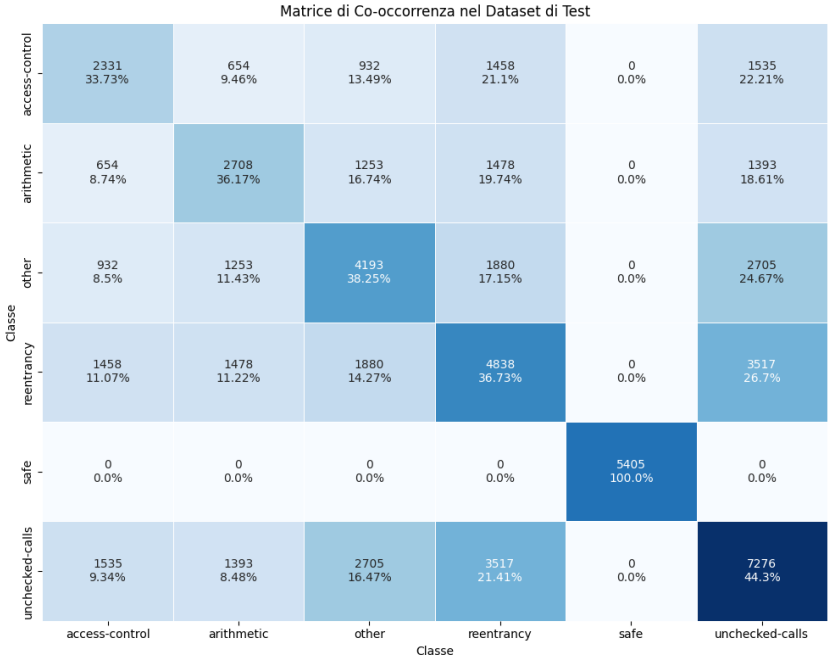


Figura 3.4: Matrice di Co-occorrenza nel Dataset di Test

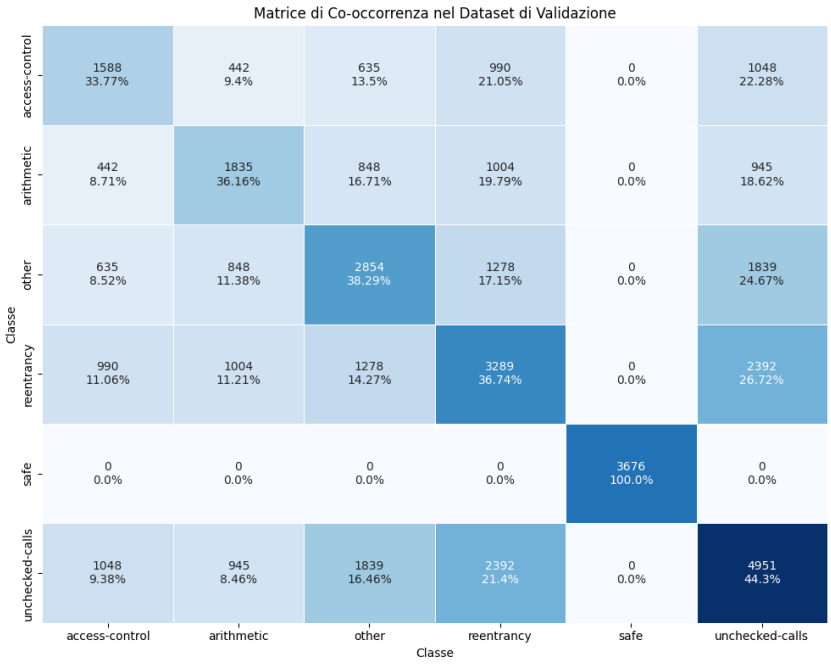


Figura 3.5: Matrice di Co-occorrenza nel Dataset di Validazione

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.

Si è analizzata a questo punto, la percentuale di volte, per ogni classe, per cui la classe stessa appariva non come unica vulnerabilità presente, ma il contratto avesse almeno un'altra vulnerabilità. I risultati sono mostrati nella Tabella 3.5.

Interessante è il risultato relativo alla classe Reentrancy, che pur non essendo la classe più presente all'interno del dataset, è quella che più frequentemente appare in concomitanza con altre classi di vulnerabilità, allo stesso modo della classe Access-Control, che pur essendo la classe minoritaria, apparte in concomitanza con altre classi di vulnerabilità in una percentuale molto alta. Questi risultati evidenziano l'importanza di considerare

Vulnerabilità	%
unchecked-calls	72.32%
reentrancy	91.51%
other	84.17%
arithmetic	77.05%
access-control	87.27%

Tabella 3.5: Percentuale dnelle classi in cui la classe stessa non è l'unica vulnerabilità presente.

la co-occorrenza delle classi quando si analizzano le vulnerabilità negli smart contracts, poichè molte vulnerabilità non si verificano in isolamento ma tendono a manifestarsi insieme ad altre.

3.2 Modellazione

In questa sezione, descriviamo l'architettura dei modelli utilizzati per la classificazione degli Smart Contracts. In particolare, presentiamo i dettagli relativi ai modelli BERT utilizzati, alle scelte di configurazione e alle strategie di addestramento. Come si evince dalla sezione precedente le feature su cui i modelli dovranno basare le loro predizioni sono il codice sorgente e il bytecode dei contratti, cioè dati di natura testuale. La natura dei dati fa sì che problema possa essere affrontato efficacemente utilizzando tecniche di elaborazione del linguaggio naturale (NLP, Natural Language Processing).

3.2.1 Natural Language Processing, NLP

L'Elaborazione del Linguaggio Naturale (NLP, da *Natural Language Processing*) è 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 quantità di dati linguistici naturali [20]. 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 entità 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 può includere la sintesi vocale, la traduzione automatica e la generazione di

risposte automatiche in chatbot. Questo aspetto dell'NLP è 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) [7], 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 [12] e regressione logistica [35] 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 vastità di task diversi nel mondo dell'NLP [57]. Lo stato dell'arte attuale nell'NLP è in molti task rappresentato dall'introduzione di una nuova architettura, che è andata a sostituire i precedenti modelli RNN e LSTM [16] tradizionali, ovvero i modelli basati su Transformers, introdotti per la prima volta nel paper "Attention is All You Need" da Vaswani et al. nel 2017 [50]. 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 è 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 tanti altri modelli diventati oggi lo stato dell'arte nei vari task del mondo dell'NLP.

Di seguito illustreremo i tre modelli che sono stati utilizzati in questo lavoro, ovvero BERT, CodeBERT e DistilBERT, tutti basati sull'architettura dei Transformer.

3.2.2 BERT, Bidirectional Encoder Representations from Transformers

Il modello BERT (Bidirectional Encoder Representations from Transformers) è stato presentato da Devlin et al. nel 2018 [11]. BERT è un modello di deep learning preaddestrato per l'elaborazione del linguaggio naturale. BERT è 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 input con l'obiettivo 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 è la successiva rispetto a un'altra frase. Questo task è 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 è 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 è un encoder bidirezionale multi-strato basato sui Transformer, come descritto nell'implementazione originale di Vaswani et al. (2017) [50]. 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 dimensionalità dello spazio in cui le rappresentazioni intermedie dei token vengono proiettate durante l'elaborazione nel modello Transformer. Questa dimensione influisce direttamente sulla capacità del modello di catturare le informazioni a partire dai dati in input; una hidden size maggiore consente al modello di rappresentare e processare informazioni più dettagliate, a costo però 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 così al modello di concentrarsi su diverse parti della sequenza di input simultaneamente.

La tabella 3.6 riassume i parametri principali dei modelli $\text{BERT}_{\text{BASE}}$ e $\text{BERT}_{\text{LARGE}}$, mentre un'immagine rappresentativa dell'architettura dei Transformer, di $\text{BERT}_{\text{BASE}}$ e $\text{BERT}_{\text{LARGE}}$ è mostrata in Figura 3.6.

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

BERT è stato preaddestrato con un embedding WordPiece [56] con un vocabolario di 30.000 token. Il primo token di ogni sequenza è sempre un token di classificazione

speciale ([CLS]). L'hidden state finale corrispondente a questo token è utilizzato come rappresentazione aggregata della sequenza per i task di classificazione, che è proprio il modo in cui BERT verrà utilizzato in questo lavoro. BERT può gestire più sequenze di token in input, ciascuna delle quali è seguita da un token speciale ([SEP]), che permette di disambiguare l'appartenenza di un token ad una sequenza piuttosto che ad un'altra.

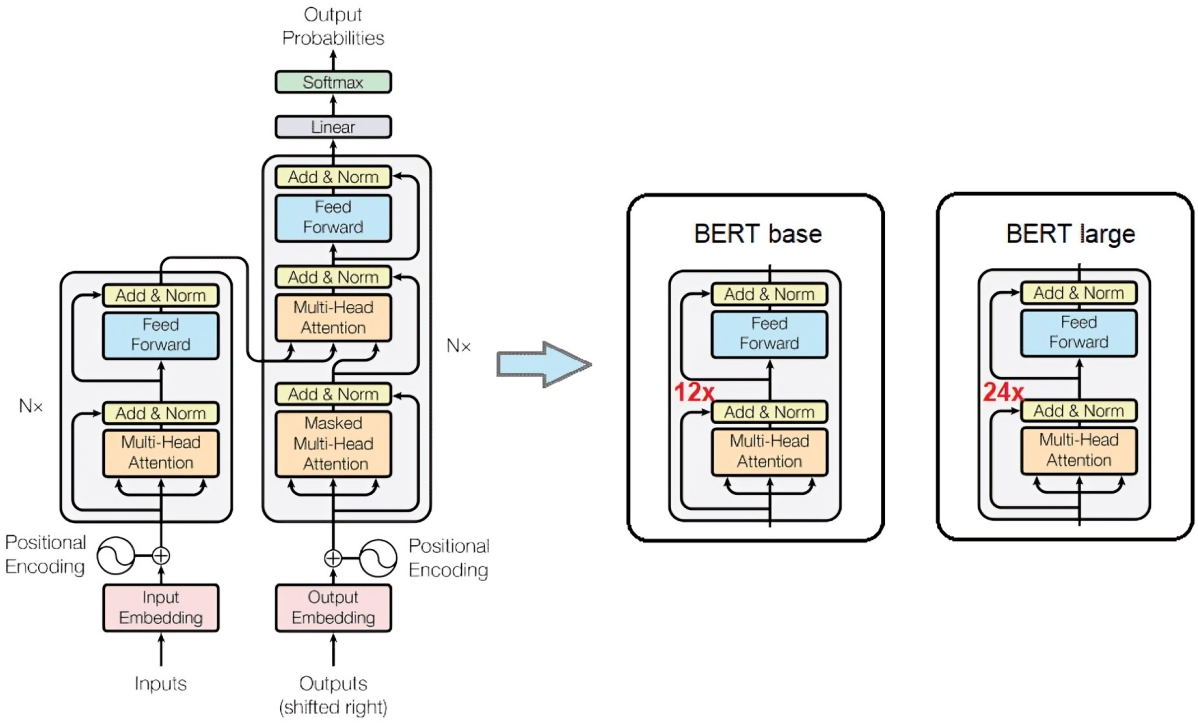


Figura 3.6: Architettura di Transformers, $\text{BERT}_{\text{BASE}}$ e $\text{BERT}_{\text{LARGE}}$. Immagine di [23]

Per ogni token in input, BERT calcola un embedding che è dato dalla somma di tre componenti come è possibile vedere in Figura 3.7:

- **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 più 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 così creato è in grado di gestire tutte le possibili sequenze di caratteri e di evitare l'utilizzo di token OOV (Out Of Vocabulary) [56].
- **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 all'interno di una sequenza di testo. Questi embedding consentono a BERT di distinguere tra parole con lo stesso contenuto ma posizionate in posizioni diverse all'interno della frase. Ciò contribuisce a catturare le relazioni tra le parole in modo più completo e consente a BERT di eccellere in una vasta gamma di compiti di elaborazione del linguaggio naturale.

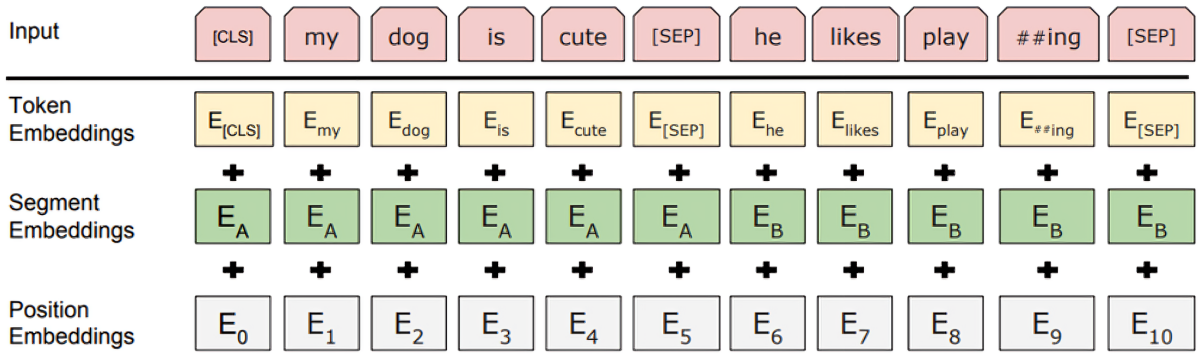


Figura 3.7: Rappresentazione degli input di BERT. Immagine di [11]

Pre-Training

Il pre-training di BERT è 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* [46]. 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 è 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 è 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 può essere completato in poche ore, rendendo BERT un'opzione potente e versatile per una varietà di applicazioni di elaborazione del linguaggio naturale.

3.2.3 DistilBERT

Nel 2020 è stato presentato DistilBERT, un modello più piccolo e più veloce rispetto a BERT, sviluppato da Sanh et al. [41]. Il modello presentato dichiara che DistilBERT è in grado di ridurre la complessità di BERT del 40% pur mantenendo il 97% delle prestazioni di BERT ed essere 60% più veloce.

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

Il training del modello studente 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 è definita come:

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

dove t_i (rispettivamente s_i) è una probabilità stimata dall'insegnante (rispettivamente dallo studente). Questa funzione obiettivo fornisce un segnale di training ricco sfruttando l'intera distribuzione dell'insegnante. Seguendo [15], 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 è 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 è impostata a 1 per tornare ad una funzione *softmax* standard.

L'obiettivo finale del training è una combinazione lineare della distillation loss L_{ce} con la loss di training supervisionato, cioè la loss del *masked language modeling* L_{mlm}. Per allineare le direzioni dei vettori hidden state del modello student e teacher è stata aggiunta una *cosine embedding loss*, L_{cos}. La Loss finale è 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 qualità del modello distillato avvicinandolo il più possibile alla performance del modello insegnante.

Architettura

L'architettura del DistilBERT è 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 normalization* ed è stato dimostrato che ridurre la dimensione dell'hidden state non ha un impatto significativo sulle prestazioni del modello, quindi è rimasta invariata. Per l'inizializzazione del modello student è stato utilizzato un layer del modello BERT poichè hanno la stessa dimensione.

3.2.4 RoBERTa e CodeBERT

A partire da BERT, nel 2019 è stato presentato RoBERTa (Robustly optimized BERT approach) da Liu et al. [24]. RoBERTa è 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

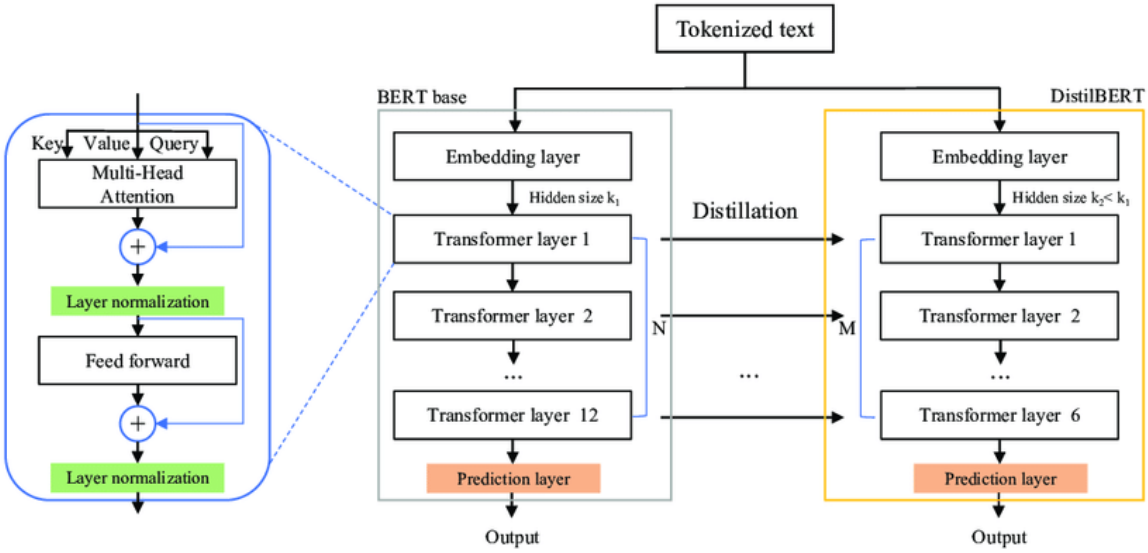


Figura 3.8: Architettura di DistilBERT. Immagine di [3]

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

Il modello CodeBERT è un modello presentato per la prima volta da Microsoft nel 2020 [14]. Il modello è stato costruito con la stessa architettura del modello RoBERTa-base, avendo quindi un numero totale di parametri pari a 125M.

Fase di Pre-Training di CodeBERT

Nella fase di pre-training, l'input è 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 è testo in linguaggio naturale e l'altro è codice di un determinato linguaggio di programmazione. [CLS] è 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*, è stato considerato un testo in linguaggio naturale come una sequenza di parole, splittandolo utilizzando *WordPiece*. Allo stesso modo, il codice sorgente è 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:* è rappresentato da una singola funzione a cui è associata della documentazione in linguaggio naturale.
- *unimodale:* è rappresentato da una singola funzione senza documentazione in linguaggio naturale.

Nello specifico, è stato utilizzato un dataset offerto da [19] 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 CodeBERT è stato effettuato utilizzando due diverse funzioni obiettivo. Il primo metodo è stato il MLM, che è stato utilizzato per preaddestrare il modello a predire i token mascherati, questo è stato applicato sui dati bimodali. Anche in questo caso, seguendo l'approccio usato dal modello BERT originario [11], sono stati mascherati il 15% dei token tra token appartenenti al linguaggio naturale e token facenti parte

del codice sorgente. Il secondo metodo applicato è stato il *replaced token detection* che utilizza sia i dati bimodati che quelli unimodali. Durante il pre-training, alcuni token nell'input originale, che può essere testo in linguaggio naturale o codice, vengono sostituiti con token casuali. Il compito del modello è 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 probabilità che indichi se il token è stato sostituito o meno. L'obiettivo di training per RTD è 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 più ricche e accurate sia per il linguaggio naturale che per il codice.

Per quanto riguarda il fine-tuning, allo stesso modo di BERT, CodeBERT può 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.

3.3 Implementazione

In questa sezione, descriveremo a livello pratico quali sono state le scelte implementative effettuate per la costruzione dei modelli di classificazione degli smart contracts, descrivendo sia il pre-processing dei dati che la costruzione e l'addestramento dei modelli, nonchè le librerie utilizzate.

I tre modelli precedenti, BERT, DistilBERT e CodeBERT sono quelli scelti ed utilizzati in questo lavoro per la classificazione di vulnerabilità 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 vulnerabilità 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 vulnerabilità presente in uno smart contract posto che questo possa avere un solo tipo di vulnerabilità.

- **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 più classi contemporaneamente.

Questo lavoro è stato affrontato come un task di classificazione multilabel, in quanto uno smart contract può avere più di una vulnerabilità contemporaneamente.

Il codice per il pre-processing dei dati, la costruzione, il training e la valutazione dei modelli è stato scritto in Python. Python è un linguaggio di programmazione Turing-completo ad alto livello, ampiamente riconosciuto per la sua semplicità e leggibilità. La vasta disponibilità 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 è supportato da una vasta comunità di sviluppatori e ricercatori, che lo rendono uno dei linguaggi più utilizzati a livello mondiale per la manipolazione di dati e per lo sviluppo di modelli di Machine e Deep Learning. In aiuto a Python è stata utilizzata la libreria PyTorch, una libreria di deep learning altamente performante e flessibile, sviluppata da Facebook AI Research (FAIR) [33]. 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 è supportato da una comunità 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 è 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, è 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 [37] utilizzato in questo lavoro, semplificando il processo di raccolta dei dati.

Di seguito, verranno presentate le tecniche utilizzate per il proceprocessing dei dati e tutti gli esperimenti effettuati e i vari modelli costruiti, i cui risultati verranno poi discussi nel capitolo 4 successivo.

3.3.1 Pre-Processing dei Dati

Come detto in precedenza, nella sezione dell'analisi esplorativa dei dati (vedi sezione 3.1), il dataset utilizzato per questo lavoro è stato fornito da Rossini et al. [37]. Il dataset è stato ottenuto dalla piattaforma Hugginface, le quali librerie sono state utilizzate per dividere il dataset in training, validation e test set.

Per quanto riguarda il pre-processing del codice sorgente sono state rimosse tramite espressioni regolari le parti di codice riguardanti i commenti e riguardanti le funzioni

getter monoistruzione. Mostriamo ora un esempio di codice per il pre-processing del codice sorgente:

```
    def remove_comments(string):
    pattern = r"(\".*?\"|\'.*?\')|(/\*.*?\*/|//[^\r\n]*\$)"
    # first group captures quoted strings (double or single)
    # second group captures comments (//single-line or /* multi-line
                                        */)
    regex = re.compile(pattern, re.MULTILINE|re.DOTALL)
    def _replacer(match):
        # if the 2nd group is not None, then we have captured a real
                                            comment string.
        if match.group(2) is not None:
            return ""
        else: # otherwise, we will return the 1st group
            return match.group(1)
    return regex.sub(_replacer, string)
```

Dopo aver rimosso anche le funzioni getter monoistruzione, il codice sorgente viene passato ad un tokenizer, il cui compito è trasformare il testo in una serie di token, ossia unità fondamentali come parole, sottoparole o caratteri. Il tokenizer di BERT utilizza la tecnica WordPiece per segmentare il testo, frammentando le parole rare in sottoparole più comuni e preservando il contesto originale. Il tokenizer di CodeBERT, specializzato per il codice sorgente, riconosce le sintassi dei linguaggi di programmazione e segmenta il codice in token significativi.

Il processo di tokenizzazione include i seguenti passaggi:

- **Padding**: Quando le sequenze di token hanno lunghezze diverse, si utilizza il padding per uniformare la lunghezza delle sequenze al valore massimo prestabilito. Vengono aggiunti token speciali (di solito uno zero) alle sequenze più corte fino a raggiungere la lunghezza desiderata. Questo è essenziale per creare batch di dati di dimensioni uniformi per l'elaborazione parallela.

- **Token IDs**: Ogni token viene convertito in un ID numerico univoco in base al vocabolario del tokenizer. Questi ID sono utilizzati dal modello per eseguire l'elaborazione e l'analisi del testo. Per esempio, la parola "hello" potrebbe essere convertita nell'ID 123, mentre "world" potrebbe corrispondere all'ID 456.

- **Attention Mask**: La maschera di attenzione è un array di valori binari che indica quali token devono essere considerati dal modello durante l'addestramento o l'inferenza. I token reali ricevono un valore di 1, mentre i token di padding ricevono un valore di 0. Questo permette al modello di ignorare i token di padding durante il calcolo dell'attenzione.

Di seguito viene riportato un esempio di implementazione di una classe `CustomDataset` in Python, che utilizza un tokenizer per preparare il codice sorgente per l'input del modello:

```
class CustomDataset(Dataset):

    def __init__(self, dataframe, tokenizer, max_len):
        self.tokenizer = tokenizer
        self.data = dataframe
        self.sourceCode = dataframe["source_code"]
        self.targets = dataframe["label"]
        self.max_len = max_len

    def __len__(self):
        return len(self.sourceCode)

    def __getitem__(self, index):
        sourceCode = str(self.sourceCode[index])
        sourceCode = " ".join(sourceCode.split())

        inputs = self.tokenizer.encode_plus(
            sourceCode,
            None,
            add_special_tokens=True,
            max_length=self.max_len,
            pad_to_max_length=True,
            return_token_type_ids=True
        )
        ids = inputs['input_ids']
        mask = inputs['attention_mask']
        token_type_ids = inputs["token_type_ids"]

        return {
            'ids': torch.tensor(ids, dtype=torch.long),
            'mask': torch.tensor(mask, dtype=torch.long),
            'token_type_ids': torch.tensor(token_type_ids, dtype=torch.
                                                    long),
            'targets': torch.tensor(self.targets[index], dtype=torch.
                                                    float)
        }
```

Come si può vedere dal codice, la funzione di tokenizzazione si incarica di aggiungere i token speciali [CLS] e [SEP] all'inizio e alla fine di ogni sequenza, rispettivamente, oltre che ricevere come parametri configurazioni su come produrre l'output, come la lunghezza massima della sequenza e se aggiungere il padding. Tutti i dati che vengono passati al modello vengono convertiti in tensori di PyTorch, che sono strutture dati multidimensionali che contengono i dati e le informazioni di forma necessarie per l'elaborazione dei dati e poi caricati in memoria GPU per l'elaborazione parallela.

Per quanto riguarda il preprocessing del bytecode, il processo è molto simile a quello del codice sorgente. Anche in questo caso, il bytecode viene passato ad un tokenizer, che si occupa di trasformare il testo in una serie di token. La differenza sta nel preprocessing iniziale subito dalle stringhe. Innanzitutto, viene rimossa la stringa di prefisso "0x", comune nei dati esadecimali, per standardizzare il formato del bytecode. Questo avviene tramite una verifica iniziale della presenza del prefisso e la sua successiva eliminazione, se presente. Successivamente, il bytecode viene suddiviso in token di 2 caratteri ciascuno. Questa operazione segmenta il bytecode in coppie di cifre esadecimali che rappresentano le istruzioni di bytecode.

3.3.2 BERT-base

Il primo modello ricostruito è stato un modello BERT base. Il modello pretrainato a cui si è fatto riferimento è 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 3.2.2 è 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 probabilità 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* è la rappresentazione vettoriale estratta dal modello BERT che rappresenta un'aggregazione delle informazioni apprese dal testo di input. Questa rappresen-

tazione viene ottenuta tramite un processo di "pooling" delle rappresentazioni dei token di input prodotte dal modello BERT. Il layer di Dropout è stato aggiunto per evitare l'overfitting del modello. Il modello è 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 è la `BCEWithLogitsLoss`, una funzione di perdita disponibile nella libreria PyTorch, particolarmente utile per problemi di classificazione multilabel. In questi problemi, ogni esempio può appartenere a più di una classe simultaneamente, richiedendo una previsione indipendente per ciascuna etichetta. `BCEWithLogitsLoss` combina una `Sigmoid` layer e la funzione di perdita `Binary Cross-Entropy (BCE)` in un'unica classe, rendendo il calcolo più numericamente stabile ed efficiente. La `Sigmoid` layer viene applicata all'output grezzo (logits) del modello per mappare i valori in un intervallo tra 0 e 1, rappresentando così le probabilità predette per ogni etichetta. Successivamente, la funzione di perdita BCE calcola la differenza tra le probabilità predette e le etichette target reali per ciascuna etichetta, trattando ogni etichetta come un problema di classificazione binaria indipendente. Utilizzando `BCEWithLogitsLoss`, si garantisce una gestione numerica più stabile, evitando problemi di underflow o overflow che possono sorgere quando si combinano separatamente la `Sigmoid` layer e la `BCE` loss, assicurando risultati più accurati e affidabili nel processo di addestramento del modello per la classificazione multilabel.

$$\text{BCEWithLogitsLoss}(y, \hat{y}) = -\frac{1}{N} \sum_{i=1}^{N} [y_i \cdot \log(\sigma(\hat{y}_i)) + (1 - y_i) \cdot \log(1 - \sigma(\hat{y}_i))]$$

Dove:

- N è il numero di etichette.
- y_i è il valore target per l'etichetta i-esima.
- $\hat{y}_i$ è il logit predetto per l'etichetta i-esima.
- $\sigma(\hat{y}_i)$ è la funzione sigmoide applicata al logit $\hat{y}_i$, definita come $\sigma(x) = \frac{1}{1+e^{-x}}$.

3.3.3 DistilBERT

La costruzione di un modello DistilBERT avviene in modo pressochè identico a quello del modello BERT. Anche in questo caso si è 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 è 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 è stato aggiunto per evitare l'overfitting del modello.

Come si può 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.

3.3.4 CodeBERT

Anche il modello CodeBERT è stato costruito in modo simile ai modelli BERT e DistilBERT. Il modello pre-addestrato utilizzato è 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 è 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 è stato aggiunto per evitare l'overfitting del modello.

3.3.5 CodeBERT con aggregazione

Una delle principali limitazioni riscontrate nel training dei precedenti modelli, è che i modelli della famiglia BERT hanno una capacità massima in input di 512 token. I nostri smart contract, come visto nella sezione 3.1 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, è 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 è 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.

3.3.6 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 è 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 più 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 è stato testato sia con due che con tre blocchi di codice, quindi si classificavano rispettivamente 1024 e 1536 token.

La scelta di utilizzare aggregazioni e concatenazioni per ovviare alla limitazione dei token in input per il modello BERT è stata sostenuta dai risultati ottenuti da [45], che hanno mostrato come la classificazione delle varie sottoparti e la successiva aggregazione delle predizioni porti a risultati peggiori. Questo è probabilmente dovuto al fatto che suddividendo il contratto in sottoparti mantenendo la label di vulnerabilità può portare a confusione nel modello, in quanto si andrebbero a indicare parti come soggette ad una certa vulnerabilità in quanto non lo sono. Per questo motivo, sono stati evitati approcci come Sliding Window sul testo (che sarebbero risultate anche più computazionalmente onerose) e si è preferito utilizzare aggregazioni e concatenazioni, classificando sempre l'intero contratto e non le sue singole parti.

3.3.7 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, è stato utilizzato un layer di Dropout con un rate del 30%. La funzione di loss impiegata è stata la BCEWithLogitsLoss, scel-

ta per la classificazione multilabel. Come ottimizzatore è stato utilizzato Adam. Gli addestramenti sono stati eseguiti su una GPU NVIDIA 2080Ti.

A causa delle limitazioni della potenza di calcolo e della grandezza del dataset, in alcuni casi è stato necessario ridurre la batch size. In particolare, per i modelli di aggregazione e concatenazione che utilizzavano due blocchi di codice (1024 token), la batch size è stata ridotta a 4, mentre per i modelli che utilizzavano tre blocchi di codice (1536 token), la batch size è stata ridotta a 2. Questa limitazione potrebbe influire negativamente sulle prestazioni del modello, poichè una batch size ridotta può portare a un addestramento più lento e a risultati meno accurati.

Questa limitazione suggerisce che ulteriori ricerche potrebbero beneficiare di potenze di calcolo maggiori, consentendo un tuning dei parametri più accurato e potenzialmente migliorando le prestazioni dei modelli.

3.4 Stacking

Seguendo l'approccio proposto da [10] è stato effettuato un esperimento di stacking per combinare il miglior modello costruito sul codice sorgente e il miglior modello costruito sul bytecode.

Per multimodalità si intende la descrizione dello stesso oggetto da punti di vista differenti. I nostri smart contracts, nel nostro dataset, ad esempio, vengono rappresentati sia dal codice sorgente che dal bytecode. Questi due punti di vista sono due punti di vista diversi ma che rappresentano la stessa cosa in modo diverso. L'idea alla base è quella che modalità diverse della stessa rappresentazione possano fornire informazioni complementari che possono essere utilizzate per migliorare le prestazioni del modello. Esistono tre modi per combinare le informazioni provenienti da diverse modalità, visibili in figura 3.9:

- **Fusione basata sulle feature**: le feature estratte da diverse modalità vengono combinate direttamente e l'unione delle stesse viene utilizzata per l'addestramento del modello. Questa tecnica permette di combinare le informazioni estratte da diverse modalità in un'unica rappresentazione, ma può rendere il training molto più complesso poichè spesso si lavora con feature altamente dimensionali e dati eterogenei.

- **Fusione basata sulle decisioni**: si addestrano modelli separati per ciascuna modalità e si combinano le decisioni ottenute da ciascun modello per allenare un meta-classificatore che fornisce il risultato finale. Questa tecnica ha il grande vantaggio di riuscire a unire dati eterogenei ma può soffrire di una perdita di informazioni.

- **Fusione Ibrida**: combina le caratteristiche della fusione basata su caratteristiche e la fusione basata su decisioni. Questa tecnica combina i vantaggi delle due tecniche di fusione precedenti, ma rende il modello e l'apprendimento più complesso.

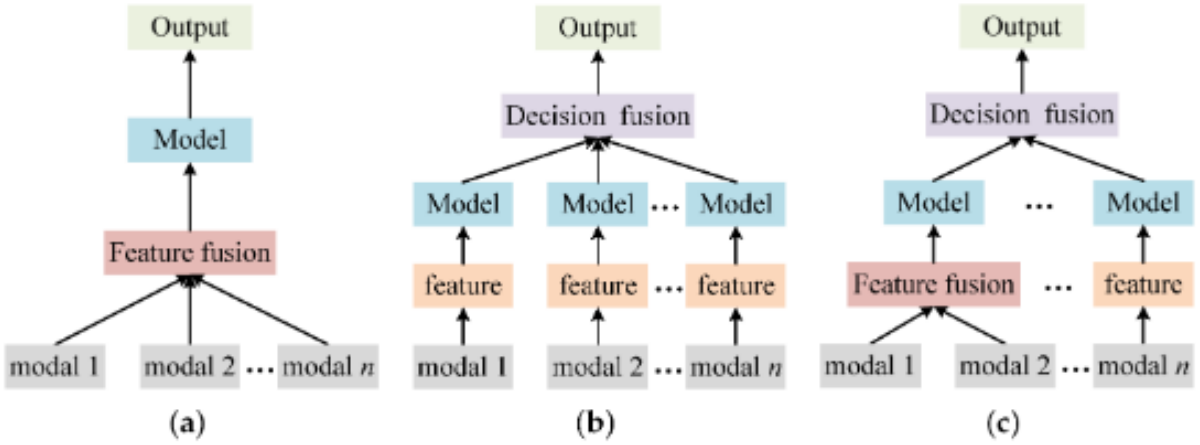


Figura 3.9: Confronto tra i tre metodi di fusione. (a) Fusione basata su caratteristiche. (b) Fusione basata su decisioni. (c) Fusione Ibrida.

In questo lavoro, è stato utilizzato un approccio di fusione basato su decisioni, in cui i modelli costruiti su codice sorgente e bytecode sono stati addestrati separatamente e le decisioni di entrambi i modelli sono state combinate per addestrare un meta-classificatore che fornisce il risultato finale.

Questo approccio è stato testato con differenti modelli utilizzati come meta-classificatori, tra cui un modello di regressione logistica, un modello di support vector machine, un modello di random forest, un decision tree, un modello bayesiano ed un modello Gradient Boost. Tutti i modelli sono stati addestrati utilizzando la libreria Scikit-learn e l'API MultiOutputClassifier, che permette di addestrare un classificatore multilabel e sono stati tunati gli iperparametri specifici di ogni modello utilizzando la funzione GridSearchCV con una cross-validation pari a 5.

I risultati ottenuti sono stati confrontati con i modelli costruiti su codice sorgente e bytecode per valutare l'efficacia dell'approccio di stacking.

3.5 Gemini

Al giorno d'oggi, chatbot basati su Large Language Models come Gemini e ChatGPT vengono sempre più spesso utilizzati come strumenti di supporto per le attività di tutti i giorni, come la ricerca di informazioni, la creazione di contenuti e la gestione delle attività quotidiane. Lo sviluppo software non si esime da questa tendenza, e sempre più spesso si fa ricorso a chatbot per la gestione di task di programmazione e sviluppo software sempre più complessi. Data l'importanza che questi strumenti stanno assumendo nelle

nostre vite è importante valutarne le performance e la qualità. Diventa quindi importante confrontare i modelli costruiti in questo lavoro di tesi con modelli già esistenti e ampiamente utilizzati come Gemini e ChatGPT.

Gemini è 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 [54]. Questa famiglia di modelli, è 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 [53].

La prima generazione di Gemini ("Gemini 1") ha tre modelli, con la stessa architettura software. Si tratta di transformer 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 più grandi, progettati per l'uso da parte di dispositivi con potenza di calcolo limitata come gli smartphone. Poichè Gemini è multimodale, ogni finestra di contesto può contenere più forme di input. Le diverse modalità 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 è 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.

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 è stata utilizzata la versione gratuita delle API di Gemini 1.5 Flash per testarlo in un task di classificazione multilabel delle vulnerabilità degli smart contracts. Non è 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, è composto da 1169 elementi a partire dal dataset di test. Questo è stato utilizzato come indicazione delle performance del modello rispetto ai modelli costruiti in questo lavoro. Non è stato possibile raccogliere un campione di dati più 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 è 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 già 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 può notare nel codice sopra riportato, l'utilizzo del modello Gemini 1.5 Flash è 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 è un oggetto di tipo GenerativeModelResponse che contiene il testo generato dal modello. Il testo generato è stato controllato per assicurarsi che la generazione abbia prodotto solo un array di 5 elementi, ognuno dei quali è 0 o 1, come richiesto dal prompt. Questo è risultato vero per tutti i 1169 elementi del campione

di dati. Successivamente, è stato convertito il testo generato in un array di 5 elementi, ognuno dei quali è 0 o 1, per poter calcolare le metriche di valutazione.

Capitolo 4

Risultati

Questo capitolo finale del lavoro di tesi è dedicato alla valutazione e all'analisi dei risultati ottenuti da tutti gli esperimenti condotti. Nei prossimi paragrafi verrà introdotta la struttura del capitolo.

Inizieremo introducendo le metriche di valutazione utilizzate per misurare le prestazioni dei modelli, cercando di fornire una panoramica delle metriche utilizzate e delle loro applicazioni.

Successivamente, presenteremo i risultati ottenuti dai modelli di classificazione applicati alla classificazione del bytecode. A seguire, verranno illustrati i risultati dei modelli utilizzati per la classificazione del codice sorgente Solidity. In entrambi i casi, verranno presentati i risultati di tutti gli esperimenti nelle varie configurazioni, seguite da un'analisi dei risultati ottenuti.

Dopo aver esaminato i risultati dei modelli sulle diverse modalità in input, presenteremo i risultati ottenuti tramite lo stacking dei modelli. Questa tecnica avanzata, che combina più modelli per migliorare la precisione e la robustezza delle previsioni, sarà discussa in termini di efficacia rispetto ai singoli modelli utilizzati.

Infine, concluderemo con la presentazione dei risultati ottenuti dal modello Gemini di Google. Questa sezione fornirà un'analisi critica delle performance del modello Gemini, confrontandole con i risultati ottenuti dai modelli precedenti e valutando il suo contributo complessivo all'interno del contesto della ricerca.

4.1 Metriche di Valutazione

In questo capitolo verranno introdotte le metriche di valutazione utilizzate per analizzare le performance dei modelli implementati. Le metriche di valutazione sono strumenti fondamentali per misurare l'efficacia di un modello di classificazione, confrontando le sue previsioni con le etichette reali del dataset. Le metriche sono calcolate a partire dalle seguenti misurazioni:

- **True Positives (TP)**: Il numero di istanze correttamente classificate come appartenenti a una specifica classe.
- **True Negatives (TN)**: Il numero di istanze correttamente classificate come non appartenenti a una specifica classe.
- **False Positives (FP)**: Il numero di istanze erroneamente classificate come appartenenti a una specifica classe.
- **False Negatives (FN)**: Il numero di istanze erroneamente classificate come non appartenenti a una specifica classe.

Queste misurazioni costituiscono la base per il calcolo delle diverse metriche di valutazione, che forniscono una visione dettagliata delle prestazioni del modello in vari aspetti chiave.

4.1.1 Accuracy

L'accuracy misura la proporzione degli esempi che sono stati correttamente classificati. Più precisamente, è la somma dei true positives e dei true negatives diviso per il numero totale di istanze. La formula per il calcolo dell'accuracy è la seguente:

$$\text{Accuracy} = \frac{TP + TN}{TP + TN + FP + FN}$$

L'accuracy è semplice da interpretare e calcolare, specialmente in contesti di classificazione binaria. Tuttavia, in contesti di classificazione multilabel con classi sbilanciate, può risultare fuorviante poichè non considera la distribuzione delle classi.

Ad esempio, in un problema di classificazione multilabel dove ciascuna istanza può appartenere a più classi contemporaneamente, consideriamo un dataset con le seguenti distribuzioni: il 70% delle istanze appartiene alla classe A, il 20% alla classe B, e il 10% alla classe C. Un modello che classifica tutte le istanze come appartenenti solo alla classe A potrebbe ottenere un'accuracy elevata, ignorando completamente le classi B e C. Questo risultato non rifletterebbe adeguatamente la capacità del modello di gestire tutte le classi presenti.

Pertanto, sebbene l'accuracy possa fornire una misura rapida delle performance di un modello, è importante considerarla insieme ad altre metriche di valutazione, come precision, recall e F1-score, specialmente in contesti di classificazione multilabel, per ottenere una visione più completa e accurata delle capacità del modello.

4.1.2 Precision

La Precision misura la proporzione di predizioni corrette tra tutte le predizioni di una certa classe. In altre parole è la proporzione dei true positive rispetto a tutte le predizioni positive.

$$Precision = \frac{TP}{TP + FP}$$

In un contesto di classificazione multilabel, la precision può essere calcolata in tre modi differenti:

- **Micro Precision**: Aggrega i contributi di tutte le classi per calcolare la precision complessiva.

$$\text{Micro Precision} = \frac{\sum TP}{\sum TP + \sum FP}$$

- **Macro Precision**: Calcola la precision per ogni classe e poi ne fa la media.

$$\text{Macro Precision} = \frac{1}{N} \sum_{i=1}^{N} \text{Precision}_i$$

- **Weighted Precision**: Calcola la precision per ogni classe ponderata per il numero di veri positivi.

$$\text{Weighted Precision} = \frac{\sum_{i=1}^{N} \text{Precision}_i \times w_i}{\sum_{i=1}^{N} w_i}$$

La precision ha il vantaggio che indica quanto rilevanti siano le previsioni fatte, cioè la proporzione di istanze classificate correttamente tra quelle classificate come positive. In contesti in cui è importante evitare falsi positivi la precision è una metrica molto utile. Al contrario, se il modello ha pochi falsi positivi ma molti falsi negativi, la precisio può essere ingannevole. La precision non è molto utile se considerata isolatamente, poichè non rifletta la capacità del modello di identificare tutte le istanze rilevanti.

4.1.3 Recall

La Recall misura la proporzione di istanze rilevanti che sono state recuperate. Di conseguenza, è la proporzione dei veri positivi su tutti i positivi.

$$Recall = \frac{TP}{TP + FN}$$

Anche la Recall, come la precision, in un contesto multilabel può essere calcolata in tre modi differenti:

- **Micro Recall**: Aggrega i contributi di tutte le classi per calcolare la recall complessiva.

$$\text{Micro Recall} = \frac{\sum TP}{\sum TP + \sum FN}$$

- **Macro Recall**: Calcola la recall per ogni classe e poi ne fa la media.

$$\text{Macro Recall} = \frac{1}{N} \sum_{i=1}^{N} \text{Recall}_i$$

- **Weighted Recall**: Calcola la recall per ogni classe ponderata per il numero di veri positivi.

$$\text{Weighted Recall} = \frac{\sum_{i=1}^{N} \text{Recall}_i \times w_i}{\sum_{i=1}^{N} w_i}$$

A differenza delle Precision, la recall indica quanto bene il modello riesce a identificare tutte le istanze rilevanti di una classe, cioè la proporzione di veri positivi tra tutte le istanze che avrebbero dovuto essere classificate come positive. La recall è particolarmente utile in contesti in cui è importante evitare falsi negativi, come il rilevamento di malattie o, come nel nostro caso, la rilevazione delle vulnerabilità all'interno degli Smart Contracts. Tuttavia, la recall non è molto utile se considerata isolatamente, poichè non riflette la capacità del modello di evitare falsi positivi.

4.1.4 F1 Score

L'F1-score è la media armonica della precision e della recall, ed è utilizzato per trovare un equilibrio tra queste due metriche, soprattutto in contesti con classi sbilanciate.

$$F1 = 2 \times \frac{Precision \times Recall}{Precision + Recall}$$

Nella classificazione multilabel, l'F1-score può essere calcolato in tre modi distinti: Micro, Macro e Weighted.

- **Micro F1**: Combina micro precision e micro recall.

$$\text{Micro F1} = 2 \times \frac{\text{Micro Precision} \times \text{Micro Recall}}{\text{Micro Precision} + \text{Micro Recall}}$$

- **Macro F1**: Calcola l'F1 score per ogni classe e poi ne fa la media.

$$\text{Macro F1} = \frac{1}{N} \sum_{i=1}^{N} \text{F1}_i$$

- **Weighted F1**: Calcola l'F1 score per ogni classe ponderata per il numero di veri positivi.

$$\text{Weighted F1} = \frac{\sum_{i=1}^{N} \text{F1}_i \times w_i}{\sum_{i=1}^{N} w_i}$$

Il grande vantaggio dell'F1 score è quello di integrare sia la precision che la recall, offrendo una misura bilanciata delle perfonace del modello, che è particolamente utile quando si ha un compromesso tra queste due metriche. L'F1 score è particolarmente utile in contesti in cui le classi sono sbilanciate, poichè considera sia i falsi positivi che i falsi negativi. Di contro, l'F1 score può essere meno interpretabile rispetto alla precision e alla recall, poichè offrendo un compromesso tra le due non rende chiaro su quale delle due il modello eccelle e su quale fallisce. Inoltre, in contesti con necessità specifiche come minimizzare i falsi negativi (dando priorità alla recall) o minimizzare i falsi positivi (dando priorità alla precision), l'F1 score potrebbe non fornire l'informazione più rilevante.

In conclusione, in questo lavoro di tesi, verranno prese in considerazione tutte le metriche sopra descritte, ma, data la natura del problema, poichè avere falsi negativi è molto più grave che avere falsi positivi, verrà data maggiore importanza alla recall rispetto alla precision.

4.2 Risultati modelli sul Bytecode

In questa sezione verranno presentati i risultati ottenuti dai modelli di classificazione utilizzati per la classificazione del bytecode. I modelli sono stati allenati sul dataset di training, validati sul dataset di validazione e testati sul dataset di test. Le metriche di valutazione, descritte nelle sezioni precedenti, sono state utilizzate per calcolare le prestazioni dei modelli.

Per la classificazione del bytecode, inizialmente sono stati utilizzati due modelli: BERT e CodeBERT, con in input ls sequenza massima di 512 token. Dopo aver confrontato i risultati, il modello CodeBERT ha dimostrato prestazioni superiori rispetto al modello BERT. Di conseguenza, CodeBERT è stato scelto per gli esperimenti successivi.

4.2.1 BERT

L'accuratezza del modello è del 70,48%. Di seguito il classification report del modello:

Class	Precision	Recall	F1-Score	Support
access-control	0.87	0.67	0.76	2331
arithmetic	0.88	0.59	0.71	2708
other	0.81	0.73	0.76	4193
reentrancy	0.88	0.78	0.83	4838
unchecked-calls	0.90	0.87	0.88	7276
Micro avg	0.8726	0.7654	0.8155	21346
Macro avg	0.8694	0.7287	0.7895	21346
Weighted avg	0.8724	0.7654	0.8126	21346

Tabella 4.1: Classification Report del modello BERT sul bytecode

4.2.2 CodeBert

L'accuratezza del modello è del 72,54%. Di seguito il classification report del modello:

Class	Precision	Recall	F1-Score	Support
access-control	0.87	0.72	0.79	2331
arithmetic	0.81	0.69	0.75	2708
other	0.85	0.73	0.78	4193
reentrancy	0.88	0.81	0.84	4838
unchecked-calls	0.93	0.86	0.89	7276
Micro avg	0.8800	0.7869	0.8309	21346
Macro avg	0.8661	0.7622	0.8104	21346
Weighted avg	0.8787	0.7869	0.8298	21346

Tabella 4.2: Classification Report del modello CodeBert sul bytecode

4.2.3 CodeBert Aggregazione di due chunk

Aggregazione con funzione Mean

L'accuratezza del modello è del 76.13%. Di seguito il classification report del modello:

Class	Precision	Recall	F1-Score	Support
access-control	0.88	0.77	0.82	2331
arithmetic	0.82	0.76	0.79	2708
other	0.86	0.78	0.82	4193
reentrancy	0.89	0.84	0.86	4838
unchecked-calls	0.91	0.91	0.91	7276
Micro avg	0.8805	0.8349	0.8571	21346
Macro avg	0.8708	0.8121	0.8400	21346
Weighted avg	0.8795	0.8349	0.8561	21346

Tabella 4.3: Classification Report per il modello CodeBERT sul bytecode con aggregazione a due chunk usando la media

Aggregazione con funzione Max

L'accuratezza del modello è del 76.10%. Di seguito il classification report del modello:

Class	Precision	Recall	F1-Score	Support
access-control	0.85	0.78	0.81	2331
arithmetic	0.86	0.73	0.79	2708
other	0.83	0.82	0.82	4193
reentrancy	0.87	0.86	0.86	4838
unchecked-calls	0.90	0.93	0.92	7276
Micro avg	0.8691	0.8513	0.8601	21346
Macro avg	0.8612	0.8245	0.8415	21346
Weighted avg	0.8682	0.8513	0.8588	21346

Tabella 4.4: Classification Report per il modello codeBERT sul bytecode con aggregazione a due chunk usando il massimo

4.2.4 CodeBert Aggregazione di tre chunk

Aggregazione con funzione Mean

L'accuratezza del modello è del 76.75%. Di seguito il classification report del modello:

Class	Precision	Recall	F1-Score	Support
access-control	0.86	0.78	0.82	2331
arithmetic	0.81	0.77	0.79	2708
other	0.82	0.83	0.82	4193
reentrancy	0.88	0.86	0.87	4838
unchecked-calls	0.92	0.92	0.92	7276
Micro avg	0.8725	0.8530	0.8626	21346
Macro avg	0.8590	0.8302	0.8440	21346
Weighted avg	0.8722	0.8530	0.8622	21346

Tabella 4.5: Classification Report per CodeBERT sul bytecode con aggregazione a tre chunk usando la media

Aggregazione con funzione Max

L'accuratezza del modello è del 76.60%. Di seguito il classification report del modello:

Class	Precision	Recall	F1-Score	Support
access-control	0.83	0.80	0.82	2331
arithmetic	0.85	0.74	0.79	2708
other	0.84	0.82	0.83	4193
reentrancy	0.89	0.84	0.86	4838
unchecked-calls	0.92	0.92	0.92	7276
Micro avg	0.8796	0.8462	0.8626	21346
Macro avg	0.8659	0.8242	0.8441	21346
Weighted avg	0.8789	0.8462	0.8619	21346

Tabella 4.6: Classification Report per CodeBERT con aggregazione a tre chunk usando il massimo

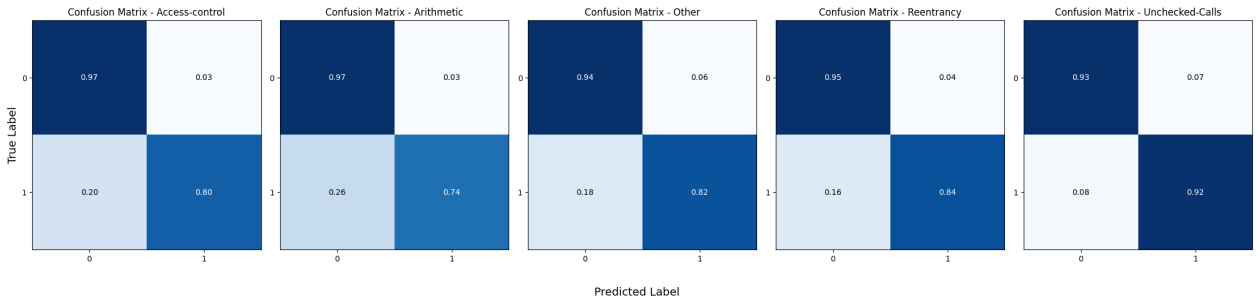


Figura 4.1: Confusion Matrices per le diverse classi del modello CodeBERT con aggregazione di tre chunk usando la funzione max sul bytecode

4.2.5 CodeBERT con Concatenazione

CodeBert Concatenazione di due chunk

L'accuratezza del modello è del 76,26%. Di seguito il classification report del modello:

Class	Precision	Recall	F1-Score	Support
access-control	0.86	0.78	0.82	2331
arithmetic	0.82	0.76	0.79	2708
other	0.84	0.81	0.83	4193
reentrancy	0.89	0.85	0.87	4838
unchecked-calls	0.90	0.93	0.91	7276
Micro avg	0.8725	0.8487	0.8604	21346
Macro avg	0.8619	0.8250	0.8427	21346
Weighted avg	0.8715	0.8487	0.8596	21346

Tabella 4.7: Classification Report per CodeBERT con concatenazione di due chunk

CodeBert Concatenazione di tre chunk

L'accuratezza del modello è del 76.80%. Di seguito il classification report del modello:

Class	Precision	Recall	F1-Score	Support
access-control	0.84	0.79	0.82	2331
arithmetic	0.87	0.73	0.80	2708
other	0.84	0.81	0.83	4193
reentrancy	0.90	0.84	0.87	4838
unchecked-calls	0.92	0.92	0.92	7276
Micro avg	0.8862	0.8435	0.8643	21346
Macro avg	0.8749	0.8197	0.8457	21346
Weighted avg	0.8856	0.8435	0.8634	21346

Tabella 4.8: Classification Report per il modello CodeBERT concatenando tre chunk

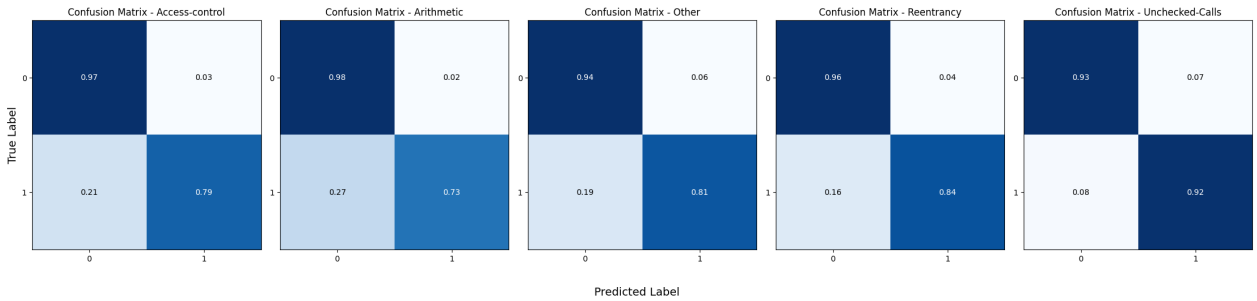


Figura 4.2: Confusion Matrices per le diverse classi del modello CodeBERT con concatenazione di tre chunk sul bytecode

4.2.6 Analisi

Per i modelli allenati sul bytecode, i risultati migliori, sugli esperimenti iniziali utilizzando un blocco di codice da 512 token per la classificazione, sono stati ottenuti utilizzando il modello CodeBERT, che ha superato il modello BERT in tutte le metriche di valutazione, eccetto per la recall della classe "unchecked-calls". Di conseguenza, è stato utilizzato il modello CodeBERT per le ulteriori analisi, poichè è stato quello con le migliori performance.

Successivamente, si è esaminato l'effetto dell'aumento del numero di chunk da due a tre. Tuttavia, non si è osservato un miglioramento significativo delle prestazioni del modello con il passaggio da due a tre chunk di codice. Il miglior modello, seppur con un vantaggio di pochi decimi, è stato quello che ha utilizzato la concatenazione di tre chunk, raggiungendo un'accuratezza del 76,80% e un F1 Micro dell'86,43% sul test set.

Analizzando, però, le performance per le singole classi, si è osservato che il miglior modello in termini di accuratezza non ha ottenuto le migliori performance in termini di recall per nessuna classe specifica. In dettaglio:

- Per la classe "arithmetic", la miglior performance in termini di recall è stata offerta dal modello con aggregazione di tre chunk utilizzando la funzione di media (mean).
- Per le restanti classi, il miglior modello in termini di recall è stato quello con aggregazione di tre chunk utilizzando la funzione massima (max).

Poichè la metrica di valutazione più importante per il nostro problema è la recall, il modello con aggregazione di tre chunk utilizzando la funzione max è stato scelto come il miglior modello per la classificazione del bytecode.

4.3 Risultati sul Codice Sorgente Solidity

Questa sezione presenta i risultati ottenuti dai modelli di classificazione applicati al codice sorgente Solidity. Anche in questo caso, i modelli sono stati allenati sul dataset di training, validati sul dataset di validazione e testati sul dataset di test. I risultati sono stati valutati utilizzando le metriche di valutazione precedentemente descritte.

Analogamente al bytecode, sono stati inizialmente utilizzati un modello BERT e un modello CodeBERT per la classificazione del codice sorgente Solidity, impiegando 512 token in input. Successivamente, poichè anche in questo caso i migliori risultati sono stati ottenuti con il modello CodeBERT, questo è stato utilizzato per gli esperimenti successivi, che prevedevano l'uso di porzioni più ampie dei dati disponibili.

4.3.1 BERT

L'accuratezza del modello è del 70.46%. Di seguito il classification report del modello:

Class	**Precision**	**Recall**	**F1-Score**	**Support**
access-control	0.82	0.60	0.69	2346
arithmetic	0.79	0.73	0.76	2715
other	0.79	0.71	0.75	4209
reentrancy	0.87	0.77	0.82	4846
unchecked-calls	0.91	0.88	0.89	7289
Micro avg	0.8568	0.7702	0.8112	21405
Macro avg	0.8384	0.7370	0.7828	21405
Weighted avg	0.8550	0.7702	0.8093	21405

Tabella 4.9: Classification Report per il modello BERT sul codice sorgente Solidity

4.3.2 CodeBert

Class	**Precision**	**Recall**	**F1-Score**	**Support**
access-control	0.81	0.74	0.77	2346
arithmetic	0.81	0.77	0.79	2715
other	0.78	0.76	0.77	4209
reentrancy	0.88	0.81	0.84	4846
unchecked-calls	0.91	0.91	0.91	7289
Micro avg	0.8563	0.8199	0.8377	21405
Macro avg	0.8391	0.7969	0.8172	21405
Weighted avg	0.8558	0.8199	0.8372	21405

Tabella 4.10: Classification Report per il modello di CodeBERT

4.3.3 DistilBert

L'accuratezza del modello è del 70.35%. Di seguito il classification report del modello:

Class	Precision	Recall	F1-Score	Support
access-control	0.82	0.60	0.69	2346
arithmetic	0.79	0.71	0.75	2715
other	0.78	0.72	0.75	4209
reentrancy	0.86	0.78	0.81	4846
unchecked-calls	0.91	0.88	0.89	7289
Micro avg	0.8487	0.7734	0.8093	21405
Macro avg	0.8308	0.7378	0.7799	21405
Weighted avg	0.8466	0.7734	0.8072	21405

Tabella 4.11: Classification Report per il modello DistilBert sul codice sorgente

4.3.4 CodeBERT con concatenazione

CodeBert Concatenazione di due chunk

L'accuratezza del modello è del 76,36%. Di seguito il classification report del modello:

Class	Precision	Recall	F1-Score	Support
access-control	0.86	0.78	0.82	2331
arithmetic	0.81	0.81	0.81	2708
other	0.82	0.78	0.80	4193
reentrancy	0.90	0.80	0.85	4838
unchecked-calls	0.94	0.91	0.93	7276
Micro avg	0.8819	0.8342	0.8574	21346
Macro avg	0.8662	0.8172	0.8406	21346
Weighted avg	0.8821	0.8342	0.8571	21346

Tabella 4.12: Classification Report del modello CodeBERT con concatenazione di due chunk

CodeBert Concatenazione di tre chunk

L'accuratezza del modello è del 79,39%. Di seguito il classification report del modello:

Class	Precision	Recall	F1-Score	Support
access-control	0.87	0.82	0.84	2331
arithmetic	0.90	0.81	0.85	2708
other	0.87	0.79	0.83	4193
reentrancy	0.91	0.84	0.87	4838
unchecked-calls	0.95	0.93	0.94	7276
Micro avg	0.9103	0.8561	0.8824	21346
Macro avg	0.8994	0.8387	0.8677	21346
Weighted avg	0.9093	0.8561	0.8816	21346

Tabella 4.13: Classification Report del modello codeBERT con concatenazione di tre chunk

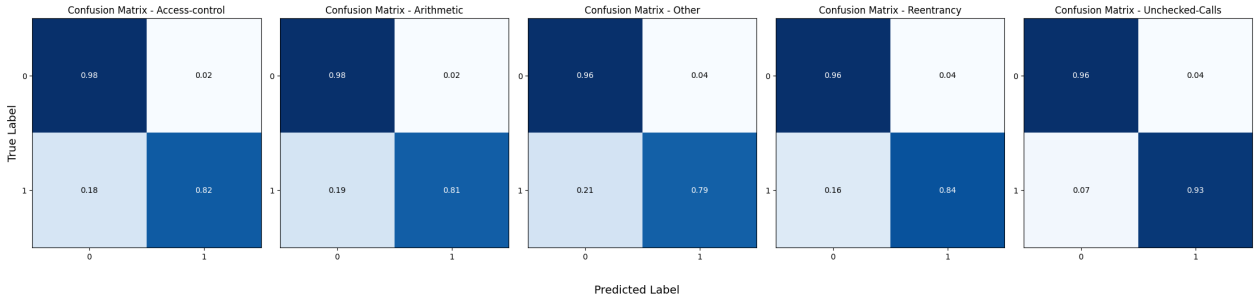


Figura 4.3: Confusion Matrices per le diverse classi del modello CodeBERT con concatenazione di tre chunk sul codice sorgente Solidity

4.3.5 CodeBert Aggregazione di due chunk

Aggregazione con funzione Max

L'accuratezza del modello è del 76.50%. Di seguito il classification report del modello:

Class	Precision	Recall	F1-Score	Support
access-control	0.87	0.77	0.82	2331
arithmetic	0.83	0.80	0.82	2708
other	0.80	0.80	0.80	4193
reentrancy	0.90	0.81	0.85	4838
unchecked-calls	0.94	0.91	0.93	7276
Micro avg	0.8816	0.8369	0.8586	21346
Macro avg	0.8687	0.8181	0.8422	21346
Weighted avg	0.8821	0.8369	0.8585	21346

Tabella 4.14: Classification Report del modello codeBERT con aggregazione di due chunk

Aggregazione con funzione Mean

L'accuratezza del modello è del 75.84%. Di seguito il classification report del modello:

Class	Precision	Recall	F1-Score	Support
access-control	0.84	0.77	0.81	2331
arithmetic	0.82	0.80	0.81	2708
other	0.81	0.77	0.79	4193
reentrancy	0.90	0.80	0.85	4838
unchecked-calls	0.94	0.90	0.92	7276
Micro avg	0.8830	0.8233	0.8521	21346
Macro avg	0.8657	0.8068	0.8350	21346
Weighted avg	0.8833	0.8233	0.8520	21346

Tabella 4.15: Classification Report del modello CodeBERT con aggregazione di due chunk

4.3.6 CodeBert Aggregazione di tre chunk

Aggregazione con funzione Max

L'accuratezza del modello è del 79.08%. Di seguito il classification report del modello:

Class	Precision	Recall	F1-Score	Support
access-control	0.87	0.83	0.85	2331
arithmetic	0.86	0.84	0.85	2708
other	0.84	0.82	0.83	4193
reentrancy	0.91	0.84	0.87	4838
unchecked-calls	0.95	0.94	0.94	7276
Micro avg	0.8983	0.8671	0.8824	21346
Macro avg	0.8855	0.8524	0.8685	21346
Weighted avg	0.8983	0.8671	0.8822	21346

Tabella 4.16: Classification Report del modello con aggregazione di tre chunk usando la funzione Max

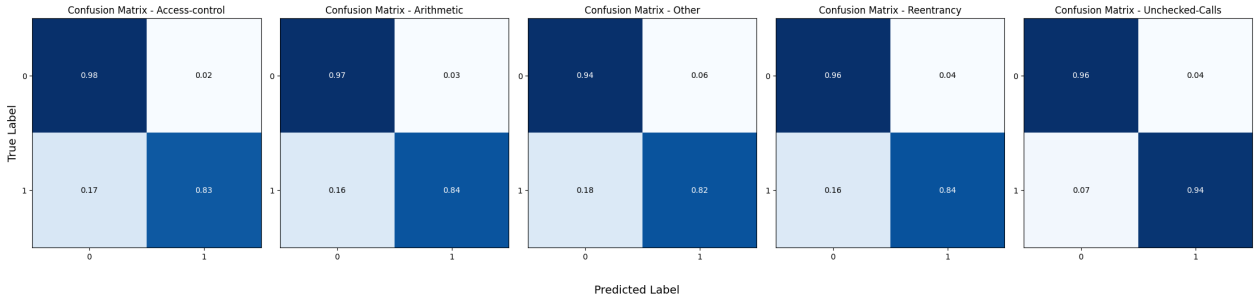


Figura 4.4: Confusion Matrices per le diverse classi del modello CodeBERT con aggregazione di tre chunk usando la funzione Max sul codice sorgente Solidity

Aggregazione con funzione Mean

L'accuratezza del modello è del 78.97%. Di seguito il classification report del modello:

Class	**Precision**	**Recall**	**F1-Score**	**Support**
access-control	0.89	0.82	0.85	2331
arithmetic	0.89	0.81	0.85	2708
other	0.83	0.83	0.83	4193
reentrancy	0.92	0.82	0.86	4838
unchecked-calls	0.95	0.93	0.94	7276
Micro avg	0.9047	0.8566	0.8800	21346
Macro avg	0.8959	0.8411	0.8672	21346
Weighted avg	0.9051	0.8566	0.8797	21346

Tabella 4.17: Classification Report del modello con aggregazione di tre chunk usando la funzione Mean

4.3.7 Analisi

I risultati ottenuti dai modelli sul codice sorgente Solidity riflettono tendenze simili a quelle riscontrate nel bytecode.

Innanzitutto, anche in questo caso si evidenzia un netto miglioramento del modello CodeBERT rispetto a BERT nei test iniziali con 512 token in input. Di conseguenza, è stato scelto di focalizzarsi sul modello CodeBERT per gli esperimenti successivi, che hanno coinvolto porzioni più ampie dei dati a disposizione.

Il modello che ha mostrato le migliori performance in termini di accuratezza e F1-Score è stato quello con concatenazione di tre chunk, come nel caso dei modelli addestrati sul bytecode, registrando un'accuratezza del 79,39% e un F1 Micro dell'88,24% sul test set. Questo risultato in F1-Score è condiviso con il modello che utilizza l'aggregazione di tre chunk con funzione Max, il quale ha però un'accuratezza leggermente inferiore.

Rispetto ai risultati ottenuti sul bytecode, l'aumento da due a tre chunk ha prodotto miglioramenti significativi in tutti gli esperimenti condotti sul codice sorgente Solidity. Questo fenomeno, potrebbe probabilmente essere dovuto alla lunghezza media del bytecode rispetto a quella del codice sorgente (ricordiamo circa 8000 contro circa 1500), quindi l'utilizzo di tre chunk di codice sorgente Solidity permette di catturare più informazioni utili rispetto al bytecode.

Nonostante il modello con concatenazione di tre chunk abbia ottenuto i migliori risultati in termini di accuratezza complessiva, è importante notare che in termini di recall per le singole classi, il modello con aggregazione di tre chunk e funzione Max ha dimostrato di essere il migliore per tutte le classi, tranne che per la classe "Other". In quest'ultimo caso, il modello con aggregazione di tre chunk e funzione Mean ha ottenuto i risultati migliori.

Basandosi sull'analisi dei risultati, il modello con aggregazione di tre chunk e funzione Max è stato selezionato come il miglior modello per la classificazione del codice sorgente Solidity.

4.4 Risultati Stacking

In questa sezione vengono presentati i risultati dei meta-classificatori addestrati sulle predizioni dei modelli di classificazione per il bytecode e il codice sorgente Solidity. Entrambi i modelli base selezionati utilizzano CodeBERT con aggregazione di tre chunk e funzione Max.

4.4.1 Risultati meta-classificatori

Di seguito, mostriamo i risultati dei meta classificatori utilizzati. Per questi modelli verranno presentate delle statistiche aggregate in quanto i risultati ottenuti sono molto simili tra loro.

I risultati delle metriche di valutazione indicano chiaramente che il Random Forest ha ottenuto le performance più elevate in termini di Accuracy, Precision e F1 Score tra tutti i meta-classificatori testati. Tuttavia, per le metriche di Recall, il modello Gaussian Naive Bayes ha mostrato risultati marginalmente superiori.

Nonostante ciò, nessuno dei modelli esaminati è stato selezionato come il miglior candidato. Questa decisione è motivata dal fatto che tutti i modelli, al momento della valutazione sui dati di training, hanno mostrato segni di overfitting, fenomeno osservato in misura differente tra i vari modelli, presentando metriche significativamente superiori rispetto a quelle ottenute sui dati di test. Il Random Forest, è emerso come il modello più affetto da overfitting mentre il modello di Gaussian Naive Bayes è risultato essere il

	Gaussian NB	SVC	GBoost	Decision Tree	Random Forest
Accuracy	0.8111	0.8256	0.8295	0.7959	**0.8346**
Precision Micro	0.9038	0.9238	0.9252	0.8945	**0.9287**
Precision Macro	0.8922	0.9153	0.9163	0.8834	**0.9212**
Precision Weighted	0.9040	0.9235	0.9249	0.8952	**0.9283**
Recall Micro	**0.8944**	0.8900	0.8930	0.8797	0.8939
Recall Macro	**0.8816**	0.8752	0.8789	0.8653	0.8801
Recall Weighted	**0.8944**	0.8900	0.8930	0.8797	0.8939
F1 Score Micro	0.8990	0.9066	0.9088	0.8871	**0.9110**
F1 Score Macro	0.8867	0.8947	0.8971	0.8740	**0.9001**
F1 Score Weighted	0.8990	0.9063	0.9086	0.8872	**0.9107**

Tabella 4.18: Confronto delle performance dei cinque modelli

meno affetto. Questa disparità può essere facilmente attribuibile alla natura stessa dei modelli: il Random Forest essendo un modello complesso che combina un numero molto alto di alberi decisionali, è più incline a sovradattarsi ai dati di training rispetto ad un modello più semplice come il Gaussian Naive Bayes.

Nel tentativo di mitigare questo problema attraverso cross-validation e semplificando gli iperparametri dei modelli (ad esempio costruendo modelli Random Forest più piccoli e meno profondi), i risultati migliorativi non sono stati sufficienti per raggiungere un livello di generalizzazione adeguato per l'implementazione pratica del modello.

4.4.2 Regressione Logistica

Il modello di regressione logistica si è dimostrato il miglior classificatore ottenuto, confermando i risultati emersi nello studio di [10]. L'accuratezza del modello è del 81.30%. Di seguito il classification report del modello:

Class	Precision	Recall	F1-Score	Support
access-control	0.91	0.82	0.86	2331
arithmetic	0.90	0.85	0.88	2708
other	0.85	0.87	0.86	4193
reentrancy	0.91	0.87	0.89	4838
unchecked-calls	0.96	0.94	0.95	7276
Micro avg	0.9144	0.8835	0.8987	21346
Macro avg	0.9074	0.8678	0.8868	21346
Weighted avg	0.9147	0.8835	0.8986	21346

Tabella 4.19: Classification Report per il modello di Regressione Logistica

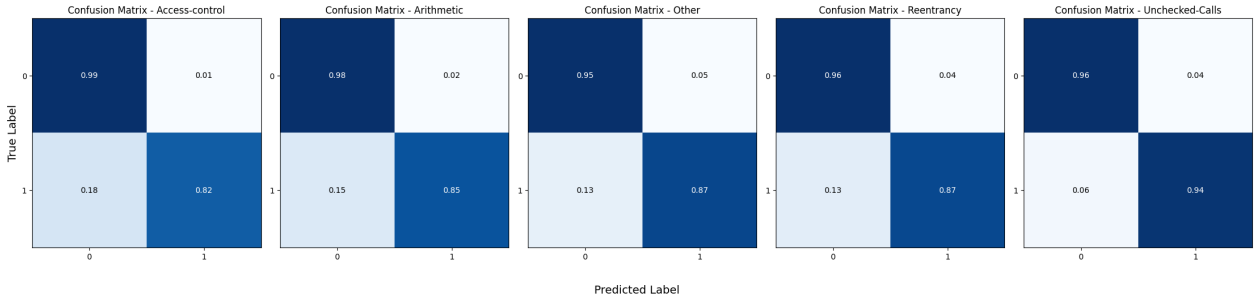


Figura 4.5: Confusion Matrices per le diverse classi del modello Logistic Regression

Rispetto agli altri modelli utilizzati come meta-classificatori, la regressione logistica mostra miglioramenti significativi in tutte le metriche di valutazione, particolarmente evidenti nella precisione e nel recall. L'unica eccezione è rappresentata dal recall per la classe access-control, la quale è la classe minoritaria in questo contesto di classificazione.

Inoltre, a differenza degli altri modelli utilizzati come meta-classificatori, la regressione logistica non mostra segni di overfitting. Questo suggerisce che un approccio più semplice possa essere più adatto per questo tipo di problemi di classificazione.

4.5 Risultati Gemini

In questa sezione verranno presentati i risultati ottenuti utilizzando le API del modello Gemini, valutati secondo le metriche descritte in precedenza.
L'accuratezza del modello è del 27.54%. Di seguito il classification report del modello:

Class	**Precision**	**Recall**	**F1-Score**	**Support**
access-control	0.24	0.04	0.06	171
arithmetic	0.19	0.04	0.07	198
other	0.28	0.49	0.36	282
reentrancy	0.41	0.08	0.14	364
unchecked-calls	0.45	0.20	0.28	475
Micro avg	0.3271	0.1872	0.2382	1490
Macro avg	0.3137	0.1706	0.1800	1490
Weighted avg	0.3490	0.1872	0.2056	1490

Tabella 4.20: Classification Report del modello Gemini

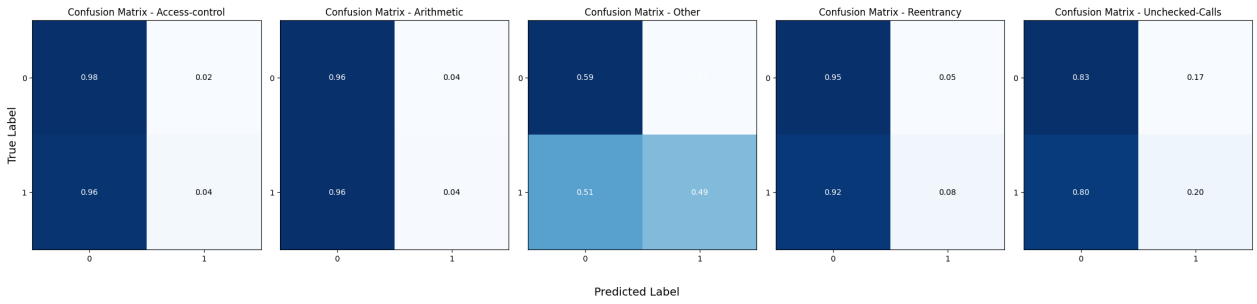


Figura 4.6: Confusion Matrices per le diverse classi del modello Gemini

I risultati del modello Gemini indicano una performance complessivamente scarsa nelle metriche di valutazione. L'Accuracy del 27.54% evidenzia che meno di un terzo delle previsioni del modello sono corrette. La Precision micro del 32.71% indica che solo un terzo delle istanze classificate positivamente sono effettivamente corrette, mentre la Precision macro e weighted, rispettivamente 31.37% e 34.90%, mostrano variazioni nella performance tra le diverse classi. Il Recall micro dell'18.72% suggerisce che il modello riesce a identificare meno di un quinto delle istanze rilevanti, con valori macro e weighted simili, indicando che il modello ha difficoltà a catturare tutte le istanze corrette. L'F1 Score, che bilancia precision e recall, è basso in tutte le versioni, suggerendo un compromesso non soddisfacente tra la capacità del modello di evitare falsi positivi e falsi negativi.
Analizzando le performance per classe, si osserva che il modello ha ottenuto risultati migliori solo per la classe "Other", con un Recall del 49%. Interessante è il risultato per la classe "reentrancy", una delle vulnerabilità più comuni, che ha ottenuto un Recall dell'8%. Questo evidenzia una specifica difficoltà del modello nel riconoscere questa particolare categoria di vulnerabilità.
Complessivamente, il modello Gemini non ha raggiunto risultati soddisfacenti nella classificazione delle vulnerabilità del codice sorgente. Le metriche di valutazione indicano la necessità di miglioramenti significativi per rendere il modello più affidabile e preciso nella sua capacità di identificare correttamente le vulnerabilità del codice sorgente Solidity, mostrando la necessità di avere modelli ad hoc per task specifici e complessi come questo.

Capitolo 5

Conclusioni e sviluppi futuri

In questo lavoro di tesi, sono stati utilizzati dei modelli di Deep Learning basati sui Transformers per la classificazione delle vulnerabilità negli Smart Contracts. Sono stati inizialmente introdotti gli Smart Contracts e le tecnologie correlate, come la blockchain, delineando il loro funzionamento e la loro importanza nel panorama tecnologico attuale. In seguito, è stata effettuata un'analisi delle vulnerabilità più comuni che affliggono gli Smart Contracts, soffermandosi sopratutto sulle vulnerabilità prese in considerazione durante la fase sperimentale di questo lavoro. Successivamente, è stata esaminanta anche la letteratura presente e le metodologie di classificazione attuali.

Il focus si è, poi, spostato sulla presentazione dei modelli utilizzati in questo studio, in particolare sui modelli della famiglia BERT basati sull'architettura dei Transformers. Tra le principali contribuzioni di questo lavoro vi è la dimostrazione della maggior efficacia dei modelli CodeBERT, preaddestrati su codice sorgente, rispetto ai modelli BERT pre-addestrati su testi generici. Inoltre, sono state testate diverse metodologie che hanno permesso di superare il limite di lunghezza di sequenze di token in input ai modelli BERT, consentendo di analizzare interi (o quasi) Smart Contracts. Nello specifico, il codice dei contratti è stato suddiviso in blocchi da 512 token, e sono stati utilizzati approcci in cui gli embedding prodotti dal modello per ciascun blocco sono stati concatenati o aggregati. Questi approcci di concatenazione degli embedding sono stati scelti poichè precedenti esperimenti effettuati in letteratura hanno dimostrato come approcci più classici di Sliding Window sul testo o di segmentazione semplicemente in sottoparti peggiorino le performance dei modelli. Infine, è stato presentato un approccio di stacking, che ha permesso di combinare il miglior modello preaddestrato sul codice sorgente ed il miglior modello preaddestrato sul bytecode degli Smart Contracts. Nello stacking un meta-classificatore ha il compito di predire la classificazione finale a partire dalle predizioni dei classificatori base. Sono stati testati vari algoritmi di machine learning utilizzati come meta-classificatori con l'obiettivo di migliorare le performance dei modelli base. I risultati ottenuti hanno dimostrato come l'approccio di stacking permetta di ottenere risultati migliori rispetto all'utilizzo di un singolo modello, ma sia facilmente

prono all'overfitting dei modelli. Nello specifico, i risultati ottenuti dimostrano come modelli più complessi come SVM e Random Forest siano più soggetti all'overfitting rispetto a modelli più semplici come il Gaussian Naive Bayes. Il miglior modello, però, è risultato essere la regressione logistica, che ha permesso di ottenere i risultati migliori in termini di accuratezza, F1 score e recall, senza mostrare segni di overfitting.

I risultati di questo lavoro hanno dapprima dimostrato come modelli preaddestrati su codice sorgente, come CodeBERT, ottengano risultati migliori rispetto ai modelli preaddestrati su testi generici. Si è dimostrato, inoltre, come la quantità di testo che viene analizzata sia un fattore chiave nel migliorare le performance dei modelli. I risultati hanno mostrato, però, come il passaggio da due a tre blocchi di codice (quindi il passaggio da 1024 a 1536 token in input), migliori significativamente le performance del modello solo nei modelli che utilizzano il codice sorgente. Questo può essere spiegato sicuramente dalle differenze di lunghezza media tra i due tipi di testo, che fanno si che per il bytecode (in media molto più lungo del codice sorgente) il passaggio da due a tre blocchi non porti a miglioramenti significativi. In termini di risultati ottenuti, sia per il bytecode che per il codice sorgente, il modello che ha ottenuto i risultati migliori è stato il modello che utilizza tre blocchi di codice concatenati con un'accuratezza del 79,39% ed un F1-Scode dell'88,24% nel caso del codice sorgente ed un'accuratezza del 76.80% ed un F1-Score dell'86,43% per il modello addestrato sul bytecode. Le migliori metriche di recall sono state ottenute invece sui modelli che utilizzano un'aggregazione con funzione max di tre blocchi di codice, con valori di Micro Recall pari all'84% e all'86% rispettivamente per il bytecode e per il codice sorgente. Il modello di stacking con Regressione Logistica ha raggiunto un'accuratezza dell'81,30% ed un F1-Score del quasi 90%. Inoltre, le metriche di recall tutte superiori all'85% a meno della classe access-control (la classe minoritaria del dataset) che si attesta comunque all'82% mostrando come il modello sia in grado di classificare correttamente la maggior parte delle vulnerabilità presenti nei contratti, dimostrando come questo approccio, se addestrato su ottimi modelli base, possa essere molto efficace.

Per valutare la necessità di utilizzare modelli ad hoc per task complessi come questo, rispetto ai modelli LLM (Large Language Models) come GPT di OpenAI o Gemini di Google entrati ormai nelle nostre vite sotto forma di chatbot utilizzatissimi per tantissimi task, sono stati effettuati test comparativi. Utilizzando la versione gratuita delle API di Gemini, è stato raccolto un campione di 1169 classificazioni dal dataset di test. I risultati ottenuti sono stati molto scarsi, con un'accuratezza del 27% e delle recall pari quasi allo zero in tutte le classi a meno della classe Other, che raggiunge livelli di recall poco piu bassa del 50%. Questi risultati dimostrano come per task complessi come questo sia ancora necessario utilizzare modelli ad hoc.

Possibili direzioni future di ricerca di questo lavoro implicano sicuramente un miglior tuning dei modelli, sopratutto per quanto riguarda la Batch Size. A seguito, infatti, della limitata potenza di calcolo non è stato possibile tunare correttamente la batch size, che in molti esperimenti è stata obbligata ad essere impostata a valori molto bassi.

Sarebbe inoltre, molto interessate utilizzare modelli di classificazione che possano gestire sequenze di token di lunghezza maggiore come Big Bird [58] e LongFormer [5]. Allo stesso tempo, avendo a disposizione più risorse computazionali, sarebbe interessante provare a fare finetuning su dei modelli LLM divenuti lo stato dell'arte al giorno d'oggi come GPT, LLaMa o Mistral. Durante lo svolgimento di questo lavoro, è stato già tentato un approccio di fine-tuning di questi modelli, ma la limitata potenza di calcolo non ha permesso di avere tempi di training ragionevoli con una lunghezza delle sequenze di token accettabile. Altre possibili direzioni di ricerca potrebbero vedere l'estensione della classificazione ad altre classi di vulnerabilità, non presenti nel dataset utilizzato in questo studio.

Bibliografia

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

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

[3] Hadeer Adel, Abdelghani Dahou, Alhassan Mabrouk, Mohamed Elsayed Abd Elaziz, Mohammed Kayed, Ibrahim El-henawy, Samah Alshathri, and Abdelmgeid Ali. Improving crisis events detection using distilbert with hunger games search algorithm. *Mathematics*, 10:447, 01 2022.

[4] Rachit Agarwal, Tanmay Thapliyal, and Sandeep Kumar Shukla. Vulnerability and transaction behavior based detection of malicious smart contracts. *CoRR*, abs/2106.13422, 2021.

[5] Iz Beltagy, Matthew E. Peters, and Arman Cohan. Longformer: The long-document transformer. *CoRR*, abs/2004.05150, 2020.

[6] Vitalik Buterin. On public and private blockchains. `https://blog.ethereum.org/2015/08/07/on-public-and-private-blockchains`, 2015.

[7] 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.

[8] European Commission. Regulation of the european parliament and of the council on harmonised rules on fair access to and use of data (data act). `https://eur-lex.europa.eu/legal-content/EN/TXT/?uri=COM%3A2022%3A68%3AFIN`, 2022.

[9] Phil Daian. Analysis of the dao exploit. `https://hackingdistributed.com/2016/06/18/analysis-of-the-dao-exploit/`, June 18, 2016.

[10] 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.

[11] 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.

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

[13] 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.

[14] 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.

[15] Geoffrey Hinton, Oriol Vinyals, and Jeff Dean. Distilling the knowledge in a neural network, 2015.

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

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

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

[19] 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.

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

[21] Sukrit Kalra, Seep Goel, Mohan Dhawan, and Subodh Sharma. Zeus: Analyzing safety of smart contracts. In *Network and Distributed System Security Symposium*, 2018.

[22] 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.

[23] Sushant Kumar. Bert: Bidirectional encoder representations from transformers, 2024.

[24] 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.

[25] 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.

[26] 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.

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

[28] 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.

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

[30] Satoshi Nakamoto. Bitcoin: A peer-to-peer electronic cash system. May 2009.

[31] 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.

[32] 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.

[33] 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.

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

[35] Joanne Peng, Kuk Lee, and Gary Ingersoll. An introduction to logistic regression analysis and reporting. *Journal of Educational Research - J EDUC RES*, 96:3–14, 09 2002.

[36] Daniel Perez and Benjamin Livshits. Smart contract vulnerabilities: Does anyone care?, 02 2019.

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

[38] 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.

[39] 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.

[40] Gernot Salzer and Monika Di Angelo. A survey of tools for analyzing ethereum smart contracts. 04 2019.

[41] 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.

[42] Nick Szabo. Smart contracts: Building blocks for digital markets. *Extropy*, 1996.

[43] Nick Szabo. Formalizing and securing relationships on public networks. *First Monday*, 2(9), Sep. 1997.

[44] Hamed Taherdoost. Smart contracts in blockchain technology: A critical review. *Information*, 14:117, 02 2023.

[45] Xueyan Tang, Yuying Du, Alan Lai, Ze Zhang, and Lingzhi Shi. Deep learning-based solution for smart contract vulnerabilities detection. *Scientific Reports*, 13, 11 2023.

[46] Wilson L. Taylor. cloze procedure: A new tool for measuring readability. *Journalism & Mass Communication Quarterly*, 30:415 – 433, 1953.

[47] 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.

[48] 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.

[49] Pavel Vasin. Blackcoin's proof-of-stake protocol v2. *URL: https://blackcoin. co/blackcoin-pos-protocol-v2-whitepaper. pdf*, 71, 2014.

[50] 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.

[51] 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.

[52] Yajing Wang, Jingsha He, Nafei Zhu, Yuzi Yi, Qingqing Zhang, Hongyu Song, and Ruixin Xue. Security enhancement technologies for smart contracts in the blockchain: A survey. *Transactions on Emerging Telecommunications Technologies*, 32, 12 2021.

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

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

[55] Gavin Wood. Ethereum: A secure decentralised generalised transaction ledger.

[56] 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.

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

[58] Manzil Zaheer, Guru Guruganesh, Avinava Dubey, Joshua Ainslie, Chris Alberti, Santiago Ontañón, Philip Pham, Anirudh Ravula, Qifan Wang, Li Yang, and Amr Ahmed. Big bird: Transformers for longer sequences. *CoRR*, abs/2007.14062, 2020.

[59] Wenbing Zhao, Shunkun Yang, Xiong Luo, and Jiong Zhou. On peercoin proof of stake for blockchain consensus. In *Proceedings of the 2021 3rd International Conference on Blockchain Technology*, ICBCT '21, page 129134, New York, NY, USA, 2021. Association for Computing Machinery.

[60] Zibin Zheng, Shaoan Xie, Hong-Ning Dai, Xiangping Chen, and Huaimin Wang. An overview of blockchain technology: Architecture, consensus, and future trends. 06 2017.

[61] Kaden Zipfel. Smart contract vulnerabilities. `https://github.com/kadenzipfel/smart-contract-vulnerabilities?tab=readme-ov-file`, 2023.

Appendice

Iperparametri del modello Alberi di Decisione

Gli iperparametri scelti per il modello di decision tree sono stati:

- **Criterion**: gini
- **Max Depth**: 15
- **Min Samples Leaf**: 1
- **Min Samples Split**: 2
- **Splitter**: random

Iperparametri del modello Random Forest

Gli iperparametri scelti per il modello di random forest sono stati:

- **Max Depth**: 15
- **Min Samples Leaf**: 4
- **Min Samples Split**: 10
- **N Estimators**: 300
- **Bootstrap**: True

Per l'eliminazione del comportamento di overfitting, gli iperparametri scelti sono stati:

- **Max Depth**: 10
- **Min Samples Leaf**: 2
- **Min Samples Split**: 10
- **N Estimators**: 100
- **Bootstrap**: True

Iperparametri del modello Support Vector Machine

Gli iperparametri scelti per il modello di support vector machine sono stati:

- **C**: 1
- **Kernel**: rbf
- **Gamma**: scale

Iperparametri del modello Gaussian NaiveBayes

L'iperparametro scelto per il modello di Gaussian NaiveBayes stato:

- **Var Smoothing**: 1e-09

Iperparametri del modello Logistic Regression

Gli iperparametri scelti per il modello di logistic regression sono stati:

- **C**: 1
- **Max Iter**: 100
- **Solver**: liblinear