POLITECNICO DI TORINO

Cybersecurity

Tesi Di Laurea Magistrale

STRUMENTI PER L'ANALISI VISIVA DI SMART CONTRACTS

Relatrice

Ing. VALENTINA GATTESCHI

Candidato

ID S278024

ALESSANDRO MOZZATO

A.A 2021-2022

Ringraziamenti

Ogni tanto la sera, poco prima di addormentarmi, mi tornano in mente ricordi che sembrano talmente lontani che potrei scambiarli per quelli di una vita passata. Ho paura a guardare indietro e ripensare a questo percorso, ho sempre il timore di non riuscire a gestire bene la malinconia di ogni momento, bello o brutto che sia. Eppure nonostante questo, soprattutto nell'ultimo periodo, continuo a farlo continuamente, quasi fosse una droga di cui sento l'astinenza a fine giornata. Continuo a rimanere vincolato in questa routine già da un po' eppure non trovo una risposta del perché questo continui a succedere ciclicamente nella mia testa, ma forse sta sera, in cui sto scrivendo tutto questo, potrei aver trovato una spiegazione. Il fatto è che probabilmente desidero tornare a quei momenti per poterli rivivere come se fosse la prima volta. Non mi riferisco in particolare solo al periodo didattico in sé ma alle persone con cui ho potuto gioire, piangere e ridere negli ultimi anni. Ed è proprio per questo che scrivo questi ringraziamenti, per tutti coloro grazie ai quali posso tenere in pancia emozioni e ricordi fantastici.

In primo luogo voglio ringraziare la mia famiglia. Le persone che hanno permesso che tutto questo potesse davvero realizzarsi. Grazie per avermi sempre spinto a inseguire i miei sogni senza il peso e la paura di poter fallire. Grazie per avermi insegnato ad essere la persona che sono oggi.

Un ringraziamento speciale a Ilaria. Sei la persona di cui ho bisogno per essere felice, non esiste nessuna copia di me stesso nel multi verso che non venga associata a te. Ti devo tutti i sorrisi che mi hai saputo strappare e tutte le toppe con cui riesci a coprire le ferite che ho nel cuore.

Ringrazio tutti i miei amici. Le persone che ci sono sempre state e che conosco ormai da una vita. I ricordi di questo percorso si intersecano indissolubilmente con le nottate insieme a parlare di tutto e spesso di niente. Grazie per la serenità che mi trasmettete ogni giorno e le nottate che mi permettete di vivere.

Grazie alla mia relatrice. La persona che ha tracciato la strada in cui ora mi trovo. Grazie per la sua disponibilità e per avermi dato la possibilità di scoprire un mondo che ancora mi ero precluso.

I ringraziamenti potrebbero finire qui... eppure vorrei avere la possibilità di scrivere ancora un pensiero. Negli ultimi mesi sono successe molte cose, così

tante che la vita sembra aver preso rapidamente una piega decisamente strana e imprevedibile. In particolare, ci sono persone che ogni tanto vanno da qualche parte e non si fanno più tanto vedere, eppure tornano sempre alla mente senza farlo apposta, in modo molto naturale. Anche oggi sembra quasi di sentirne la voce e la risata risuonare tra il vento e il fruscio delle foglie che cadono e chissà, magari è davvero così. Io di una cosa sono sicuro, queste persone sono più vicine di molte altre. L'unica differenza è che non hanno bisogno di parlare per far sentire la loro presenza perché le sento affianco a me in ogni momento. Grazie per guardare sempre alle mie spalle.

"Con tutto l'affetto che posso"
Alessandro Mozzato

Lista dei Contenuti

Lista delle Tabelle

Lista delle Immagini

Capitolo 1

Introduzione

1.1 Introduzione al tool

La Blockchain, negli ultimi anni, sta diventando di uso comune da parte di molte aziende che vogliono testarne le funzionalità per scopi specifici([1], [2], [3]). Basti pensare che l'Italia è tra i primi dieci Paesi per numero di progetti blockchain implementati, sopra Francia e Germania. Questo soprattutto perché il settore finanziario, che è quello in cui se ne fa maggior uso, è in netta crescita negli ultimi anni.

Oltretutto questo ampio utilizzo è motivato dal modo con cui vengono effettuati gli scambi economici e in cui si tiene traccia delle varie transazioni [4].

In tutto questo trovano un ruolo fondamentale gli Smart contract. Si tratta di programmi salvati sulla Blockchain (solo di Ethereum) che possono essere richiamati dai nodi che partecipano alla rete per offrire dei servizi specifici.

Fatte queste iniziali considerazioni, l'obiettivo di questa ricerca si sviluppa principalmente sui seguenti punti:

- Trovare un modo che possa permettere un primo approccio agli Smart contract da parte delle persone meno esperte (e non solo) dell'ambito Blockchain. In particolare per quella classe di persone che ha bisogno di effettuare un'analisi superficiale di alcuni specifici contratti senza dover ispezionare appositamente il codice, ma anche per coloro che sono più esperti e hanno bisogno di valutare un contratto in maniera differente;

- Permettere agli utenti di osservare uno Smart contract da un nuovo punto di vista, in modo che possano risaltare delle specifiche caratteristiche intrinseche ed estrinseche al contratto, ovvero influenzate anche dal comportamento in tempo reale della Blockchain Ethereum. Si vuole inoltre permettere un confronto ad alto livello su più smart contract contemporaneamente;

- Valutare quanto la visualizzazione grafica e l'estrazione delle informazioni mediante una piattaforma web basata su Javascript, HTML, CSS, d3js, Web3, Nodejs possa essere appropriata in questo specifico contesto.

Osservando gli strumenti attualmente disponibili sul mercato, molti tentano di concentrarsi, in particolare, sull'aspetto riguardante la sicurezza [5], [6]. Questo perché è molto comune imbattersi in smart contract con codice altamente insicuro che potrebbe permettere ad hacker della rete di sfruttare queste vulnerabilità per creare danni e perdite ingenti di denaro [7], [8].

D'altro canto però, non sono presenti molti tool o software che permettano di visualizzare e reperire le informazioni presenti nel codice Solidity di uno o più contratti. Da un punto di vista di analisi e programmazione questo potrebbe risultare interessante nel momento in cui dovesse essere necessario fare dei confronti. L'approccio e il modo in cui il tool cerca di visualizzare i risultati delle analisi di alcuni smart contract è particolarmente innovativo per il fatto che permette di osservare da un nuovo punto di vista il contratto.

1.2 Introduzione alla rete Blockchain

Una blockchain è un registro distribuito, decentralizzato e immutabile d' informazioni la cui consistenza è garantita attraverso dei meccanismi di consenso.

- Distribuito: vuol dire che ogni nodo ha una connessione verso gli altri nodi, e la logica di esecuzione è distribuita tra tutti i nodi;
- Decentralizzato: vuol dire che la logica di esecuzione e i dati sono replicati su ogni client, o nodo, collegato alle rete;
- Immutabile: vuol dire che i dati, una volta inseriti, non possono essere modificati o rimossi.

All'interno di questo registro vengono salvati dei messaggi, più correttamente delle transazioni, che danno informazioni su indirizzo che effettua un pagamento, indirizzo verso cui viene effettuato il pagamento e quantità di denaro utilizzata. Per ogni indirizzo presente nella blockchain si è in grado di risalire a tutta la storia delle sue transazioni in qualunque momento da qualsiasi nodo della rete.

Alcuni nodi della rete svolgono il ruolo di 'miners', ovvero nodi che generano blocchi di transazioni da aggiungere in coda alla blockchain.

Esistono diversi algoritmi di mining che utilizzano approcci differenti per generare blocchi, ma ogni algoritmo di mining ha come risultato un nuovo blocco all'interno del quale sono presenti un certo numero di transazioni fino a quel momento ancora non confermate.

Ogni blocco creato da un miner contiene un certo numero di transazioni ed un hash crittografico del blocco precedente. Il numero massimo di transazioni presenti all'interno di un blocco è dettato dalla dimensione massima, in bytes, del blocco stesso: ogni transazione ha una dimensione variabile, e un blocco non può avere una dimensione maggiore ad un valore predefinito. Quando un blocco viene creato da un miner, viene propagato a tutti i nodi a cui esso è connesso, i quali lo verificano e, se valido, lo propagano ulteriormente: il nodo viene dunque aggiunto alla blockchain.

Nelle blockchain pubbliche sono presenti dei meccanismi che incentivano la creazione di blocchi corretti da parte di miner. Un miner malevolo potrebbe infatti introdurre delle transazioni costruite ad arte per effettuare dei cambi di stato che non dovrebbero poter avvenire (ad esempio operazioni di double-spend [9], [10]). Il meccanismo più utilizzato per incentivare i miner a comportarsi correttamente è pagando il miner stesso per il lavoro svolto e, in taluni casi, penalizzarlo nel momento in cui cerca di aggirare le regole della blockchain. Sebbene il termine mining focalizzi l'attenzione sul 'premio' ricevuto dai miner per il lavoro svolto, l'obiettivo principale del mining non è quello di premiare i miner, ma quello di permettere l'emergere di un consenso distribuito e garantire la sicurezza della blockchain. L'emissione di un premio per i miner è un mezzo per allineare l'interesse dei miner con la sicurezza della blockchain.

I blocchi creati dai miners devono però essere verificati dagli altri nodi della rete prima di essere effettivamente caricate sulla blockchain. Questo avviene attraverso specifici algoritmi di consenso [11] [12], per evitare che attori malevoli possano inserire sulla blockchain false transazioni.

1.3 Bitcoin

Il primo esempio d' implementazione di una tecnologia basata su blockchain è stata Bitcoin, una delle più famose e conosciute cripto valute in circolazione. Bitcoin rappresenta l'unione di diverse tecnologie che vanno dal dominio della crittografia al dominio della finanza, aggiungendo un algoritmo di consenso, il PoW, che fino a quel momento non era mai stato implementato e che risolve problemi fino ad allora non ancora risolti in maniera decentralizzata, come il problema del double spend.

Bitcoin è stato rilasciato con la pubblicazione del white paper [14], pubblicato sotto lo pseudonimo di Satoshi Nakamoto.

La funzione principale di Bitcoin è quella di essere una moneta di scambio digitale completamente decentralizzata e resistente alla censura, con una emissione mediamente costante e una quantità di coin limitata superiormente con l'obiettivo di diventare, a tendere, deflazionaria. La blockchain di Bitcoin contiene quindi nel suo stato interno l'associazione tra indirizzi e bitcoin posseduti, e le transazioni

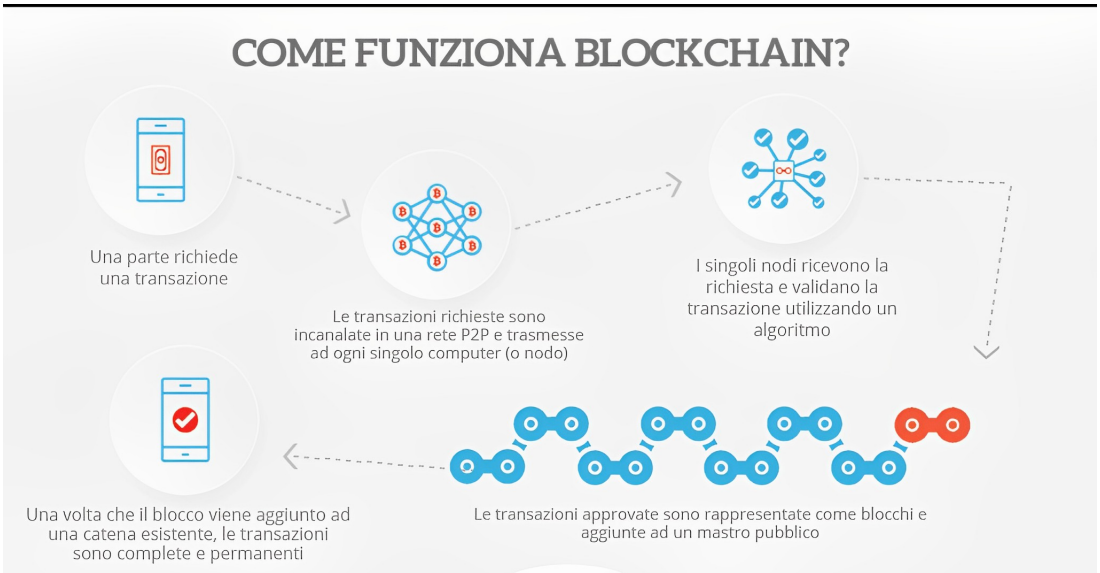

Figura 1.1: Schema del funzionamento della Blockchain [13]

rappresentano un passaggio di proprietà di bitcoin da un indirizzo all'altro. La condizione necessaria e sufficiente per poter scambiare bitcoin è quella di possedere la chiave privata associata a un indirizzo. La conoscenza della chiave privata determina quindi, univocamente, il possesso dei bitcoin, che non possono essere altrimenti, in nessun modo, trasferiti.

1.4 Ethereum

Lanciata nel 2015, Ethereum è una blockchain che implementa una infrastruttura di calcolo decentralizzata.

La creazione di Ethereum, il cui whitepaper [15] è stato pubblicato nel 2013 da Vitalik Buterin, è motivata dalla volontà di voler costruire una blockchain le cui applicazioni andassero oltre quelle delle cripto valute. Sebbene Bitcoin permetta lo sviluppo di scripts, ci sono molte limitazioni che impediscono lo sviluppo di diverse applicazioni a meno di non utilizzare ulteriori layer off-chain. A differenza di Bitcoin, la cui blockchain traccia il passaggio di proprietà di una unità di scambio monetaria, la blockchain di Ethereum traccia l'evoluzione del contenuto di una memoria di archiviazione dati. Lo scopo di Ethereum infatti è quello di essere una infrastruttura di calcolo decentralizzata general-purpose, come indicato nel suo yellow paper [16].

Come ogni computer general-purpose, Ethereum può caricare dai dati salvati all'interno della sua memoria dei programmi all'interno della macchina a stati ed

eseguirli, salvando il risultato dell'esecuzione. La differenza tra un computer general-purpose ed Ethereum risiede nel fatto che, in Ethereum, i cambi di stato sono governati da delle regole di consenso e che lo stato è distribuito su più nodi. I cambi di stato della blockchain di Ethereum sono gestiti dalla EVM, o Ethereum Virtual Machine, una virtual machine che esegue del bytecode risultato dalla compilazione di programmi chiamati Smart Contracts, scritti in linguaggi di alto livelo. La EVM è in esecuzione su ciascun nodo che partecipa alla rete, dal momento che ogni nodo ha il compito di verificare le transazioni che riceve ed eseguire ogni smart contract che viene chiamato.

Capitolo 2

Stato dell'arte

2.1 Smart-Graph: Rappresentazione grafica degli smart contract sulla blockchain di Ethereum

Si tratta di un web tool che permette la rappresentazione grafica di uno smart contract [17]. La rappresentazione parte con una considerazione che viene fatta riguardo alla metodologia Agile, ovvero un insieme di metodi di sviluppo del software emersi a partire dai primi anni 2000 che si basano su un insieme di principi comuni, derivati dai principi del 'Manifesto per lo sviluppo agile del software'.

Partendo appunto dall'analisi del manifesto risulta evidente che la metodologia Agile si basa sul 'linguaggio di modellizzazione unificato' (**UML** [18]), si tratta di un linguaggio che permette di creare un modo standard per visualizzare il design di un sistema [19]. UML offre un sistema per visualizzare gli schemi architettonici del sistema; come il sistema lavora in fase di 'run'; come le entità interagiscono le une con le altre (componenti e interfacce) [20].

L'obiettivo è quello di creare un tool, 'Smart-Graph', per visualizzare le entità degli smart contract e come interagiscono. Riassumendo, il web tool, permette di generare un rappresentazione grafica in 2D del codice dello smart contract attraverso dei diagrammi di tipo UML. In particolare, si tratta di una versione migliorata rispetto all'UML normale in quanto possono essere estratte informazioni specifiche del linguaggio Solidity su, per esempio, 'Function Modifier' e 'Function Fallback' [21] [22].

Il software ha come vantaggio il fatto di non dover installare sulla propria macchina nessun programma permettendo ai programmatori di avere una vista dì generale dello smart contract in analisi e allo stesso tempo la possibilità di entrare nel dettaglio.

2.1.1 Architettura del Sistema

Il sistema è composto da due entità: back-end e front-end, che sono unite da delle API. Entrambi utilizzano delle tecnologie 'open source'. I componenti principali sono:

- Express è una libreria utilizzabile tramite 'Node.js' che permette di gestire specifiche richieste;
- Webpack è un modulo JavaScript 'open-source' che permette di generare il grafo delle dipendenze partendo dal codice presente.

1. Frontend: E' creato mediante l'ambiente web. Questo è un vantaggio in quanto è indipendente dalla piattaforma. Questo significa che è semplicemente necessario un browser web per utilizzare il tool. L'interfaccia grafica è suddivisa in due sezioni:
   - 'Smart Graph Form': presenta un campo di testo come input e un bottone chiamato 'Generate the diagram' per ottenere l'output. Il form di input permette all'utente di inserire l'indirizzo di uno smart contract. Quando si clicca sul bottone il backend gestirà una serie di dati e restiuirà l'UML che verrà appositamente visualizzato nella pagina web.
   - 'Smart Graph Diagram Container': Si tratta dell'area in cui l'UML classico e l'UML versione migliorata vengono visualizzati

   Il grafico che si visualizza è interattivo e permette all'utente di entrare più nello specifico attraverso una sorta di 'zoom in' oppure elevarsi a livelli più alti per una visione d'insieme utilizzando lo 'zoom out'.
2. Backend: E' scritto in javascript per avere la possibilità di scrivere codice asincrono attraverso le Promise. In casi in cui la mole di dati è molto elevata, queste non bloccano l'esecuzione del programma, ma in modo asincrono cercano di prelevare e analizzare i dati. Il tool preleva il codice dello smart contract utilizzando Etherscan. Si tratta di un tool che permette di fare ricerche sulla blockchain di ethereum.

2.1.2 Considerazioni finali

Ci sono due limitazioni nell'utilizzo dell'UML per la visualizzazione degli smart contract. Esso si pone come obiettivo quello di creare un diagramma per un'intera DApp, ma è molto difficile riuscire a visualizzare una mole di dati e informazioni cosi elevata in modo comprensibile per un utente. Ci sono infatti molti smart contract e tante relazioni che li uniscono rendendo molto fitto e poco chiara la

visualizzazione. Come altra nota negativa c'è il fatto che la rappresentazione degli smart contract mediante UML non riesce a tenere in considerazione informazioni di vitale importanza per un programmatore Solidity, ovvero 'Receive Ether Function', 'Function Modifer', 'Function Fallback' e 'Pure Function'.

2.2 Slither: Framework di analisi statico per gli smart contracts

Slither [23], un framework di analisi per gli smart contracts, fornisce informazioni dettagliate riguardo agli smart contract della blockchain di ethereum. Slither utilizza usa un apposito intermediario, SlithIR, che permette di eseguire delle analisi sul codice Solidity.

Slither viene creato partendo da alcune considerazioni a cui vorrebbe arrivare:

- Un buon livello di astrazione: né troppo né troppo poco altrimenti potrebbero esserci una serie di problematiche;
- Robustezza: Deve poter analizzare il codice senza 'crash';
- Performance: L'analisi dev'essere rapida, anche per molti contratti insieme;
- Accuratezza: Si deve cercare di manetenere il più basso possibile il numero di falsi positivi;

Esso utilizza delle tecniche di analisi molto conosciute come il 'data flow' e il 'taint tracking' per estrarre e rifinire le informazioni.

Slither è un framework open source in modo che anche esterni o terze parti possano validare e migliorare il tool.

2.2.1 Descrizione framework

Il tool permette le seguenti funzionalità:

- Rilevamento automatico delle vulnerabilità: una consistente varietà di 'bug' presenti negli smart contract può essere trovata senza l'intervento dell'utente;
- Rilevamento automatico delle ottimizzazioni: Slither permette di trovare ottimizzazioni del codice che il compilatore non fa [24];
- Comprensione del codice: permette di visualizzare le informazioni che vengono estrapolate dall'analisi del codice;
- Revisione assistita del codice: attraverso apposite API, un utente può interagire con Slither.

Slither prende come input iniziale l'abstract Syntax tree (**AST**) del codice Solidity generato dal compilatore Solidity. In un primo momento, Slither cerca di recuperare le informazioni riguardo all'ereditarietà del grafo del contratto, il 'control flow graph' (**CFG**), e la lista delle espressioni. Successivamente, Slither trasforam tutto il codice del contratto in SlithIR, il suo linguaggio di programmazione interno. All'ultimo step Slither esegue delle operazioni che generano informazioni migliorate per gli altri moduli.

2.2.2 Analisi del codice

Slither include i 'printers', che permettono all'utente di comprendere velocemente cosa fa un contratto e come è strutturato. I 'printers' permettono:

- Esportazione di differenti rappresentazioni con grafo, incluso il grafo delle dipendenze (ereditarietà), il 'control flow graph', e il grafo delle chiamate di ogni contratto;
- Un sommario leggibile del contratto, che include il numero di problemi relativi al codice che sono stati trovati e informazioni riguardanti la qualità del codice;
- Un sommario con gli accessi autorizzati e le variabili che possono essere cambiati dal possessore (intestatario) dello smart contract.

2.2.3 Conclusioni

Slither è uno dei pochi framework che nello stesso tempo può essere utilizzato per trovare bugs, proporre ottimizzazioni del codice, aumentare la comprensione del codice per un certo contratto. E' stato creato affinché possa essere ulteriormente esteso da terze parti. Confrontando il tool con altri che permettono di eseguire operazioni simili, risulta essere decisamente migliore in termini di efficienza e rapidità. In futuro potrebbe esserci la possibilità di adattare questo framework per essere utilizzato non solo in Solidity ma anche in altri linguaggi come Vyper.

2.3 Mythril

Si tratta di un tool [25] da linea di comando scritto in linguaggio Python per l'analisi interattiva di smart contract. Esegue l'EVM bytecode in maniera simbolica e visualizza il CFG, con i nodi contenenti il codice disassemblato e gli archi che rappresentano formule di percorso. L'SMT solver Z3 è usato per ridurre lo spazio di ricerca e per calcolare valori concreti per sfruttare una potenziale vulnerabilità.

2.4 SmartCheck: Analisi statica degli Smart Contracts

Questo tool [26] viene creato partendo dalla premessa che la sicurezza è l'aspetto più importante nella programmazione su blockchain Ethereum. Questo per una serie di ragioni:

- **Ambiente di esecuzione sconosciuto**. Ethereum differisce dagli ambienti di esecuzione centralizzati.
- **Nuovo stack software**. Lo stack Ethereum è ancora in fase di sviluppo. Le vulnerabilità vengono trovate giorno per giorno.
- **Attacchi anonimi per fini finanziari**. Hackerare gli smart contract risulta essere molto remunerativo sia in termini di usabilità del denaro rubato (che può essere subito speso), sia per il fatto che il rischio di essere scoperti si riduce notevolmente grazie all'anonimato.
- **Rapido ritmo di sviluppo**. Le compagnie blockchain cercano di rilasciare i loro prodotti il più in fretta possibile, ma questo spesso va a spese della sicurezza.
- **Linguaggio ad alto livello non ottimale**.

2.4.1 SmartCheck analisi del tool

Si tratta di un tool di analisi per gli smart contract sviluppato con linguaggio Java. Utilizza ANTLR e una grammatica Solidity personalizzata per generare un albero XML che funge da rappresentazione intermedia (RI). Si fa un'analisi per verificare e trovare vulnerabilità grazie all'utilizzo di query XPath sulla RI. Smart Check permette una copertura totale: Il codice viene totalmente analizzato e tradotto nella RI, e tutti gli elementi possono essere verificati mediante XPath.

Gli attributi RI possono essere arricchiti con informazioni aggiuntive quando nuovi metodi di analisi sono implementati. Il tool può essere esteso per supportare altri linguaggi di programmazione per gli smart contracts aggiungendo la grammatica ANTLR e un apposito database.

2.4.2 Conclusioni

Il tool è stato testato su molti smart contract ed ha permesso di trovare molte vulnerabilità. Ovviamente sono comunque presenti altri validi tool che permettono di identificare le vulnerabilità presenti nel codice Solidity.

Il tool può essere migliorato in varie direzioni: Miglioramento della grammatica, creare dei 'pattern' più precisi, creare nuovi 'pattern', creare dei metodi di analisi più sofisticati, aggiungere un supporto per altre lingue.

2.5 Solgraph

Si tratta di un tool [27] che permette di visualizzare il flusso delle chiamate nei contratti scritti in linguaggio Solidity per supportare gli utenti nelle loro analisi. Questo permette di leggere il codice Solidity e produrre un grafo, con i nodi che rappresentano le funzioni e gli archi che rappresentano le chiamate alle funzioni. Il tool cerca di mettere in mostra delle possibili vulnerabilità nel codice Solidity.

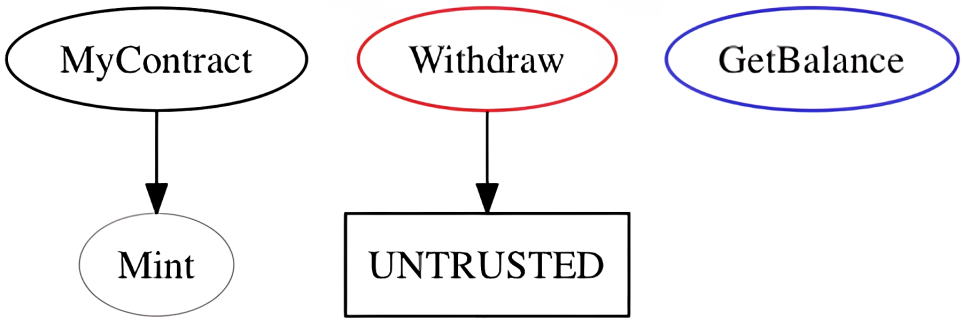

Figura 2.1: Output di Solgraph nell'analisi di uno smart contract

Legenda:

- Rosso: manda a un indirizzo esterno;
- Blu: funzione costante;
- Giallo: funzione di tipo 'view';
- Verde: funzione di tipo 'pure';
- Arancione: chiamata;
- Viola: funzione 'transfer';
- Lilla: funzione di tipo 'payable'.

Si tratta di un tool da linea di comando scritto in JavaScript. Utilizza il compilatore Solidity per la creazione dell'AST e GraphViz per la visualizzazione del grafo.

2.6 SolMet

Si tratta di un tool [28] [29] in grado di scomporre un Solidity smart contracts e fare delle analisi per calcolare delle informazioni intrinseche del codice. Il tool è stato verificato su un elevato numero di smart contract con risultati soddisfacenti. In accordo coi risultati ottenuti si possono notare alcune caratteristiche interessanti negli smart contract. Sembra infatti che gli smart contract siano molto corti e 'piatti'senza una reale complessità in termini di 'McCabe cyclomatic complexity' [30] o di livello di annidamento delle strutture di controllo.

2.6.1 Analisi e studio degli smart contract

Il tool propone di calcolare le seguenti informazioni:

- SLOC - numero di linee di codice del contract, library o interface;
- LLOC - numero di linee di codice logiche (commenti e linee vuote sono ignorate) del contract, library o interface;
- CLOC - numero di linee di commento presenti in un contract, library, interface;
- NF - numero di funzioni in un contract, library o interface;
- WMC - la complessità pesata delle funzioni in un contract, library, interface, che è la somma del 'McCabe cyclomatic complexities' [30] delle funzioni;
- NL - la somma del livello di nidificazione più alto delle strutture di controllo nell'ambito delle funzioni di un contract, library o interface.

Il tool è stato implementato utilizzando il linguaggio Java. Per fare il 'parsing' del codice Solidity viene usato un parser basato su una versione leggermente modificata del ANTLR4. Sono state fatte delle leggere modifiche nella grammatica ANTLR per essere in grado di fare il 'parsing' sia dei contratti più recenti che dei più datati.

Il calcolo delle informazioni sopracitate invece è permesso grazie a visite multiple sull'AST.

2.6.2 Analisi dei risultati

Con questo tool sono stati analizzati all'incirca 10.206 contratti Solidity. Da questi sono state prelevate le seguenti informazioni.

In media, ogni Solidity file contiene 4.4 contracts e in uno smart contract su due troviamo la presenza di una library. Una statistica interessante è che l'80% di queste librerie sono connesse alle stesse funzionalità, in particolare alle stesse operazioni

	Total	Avg./sol file
Contract	44893	4.40
Library	4260	0.42
Interface	662	0.06

Tabella 2.1: Numero di contract, library, interface all'interno degli smart contract

	Min	Max	Avg.	Median	Std.dev.
SLOC	1	1250	60.30	29	100.57
LLOC	1	843	38.28	18	66.07
CLOC	0	481	13.68	2	30.69
NF	0	93	5.11	3	6.43
WMC	0	522	7.58	4	14.10
NL	0	125	1.59	0	4.68
Avg. McCC	1	40.15	1.25	1	0.65
Avg. NL	0	17.86	0.17	0	0.34

Tabella 2.2: Altre informazioni analizzate tramite SolMet sugli smart contracts analizzati

matematiche. Le interfacce sono estremamente rare. Ne troviamo soltanto una ogni 20 contratti analizzati.

Come possiamo osservare dalla tabella, i contratti sono corti. Le linee di commento sono molto elevate in media, ma possiamo osservare che il valore della mediana è solo 2. Questo indica che vi sono molti contratti con meno di 2 righe di commento. In media ogni contratto ha 5 funzioni, ma la complessità media pesata è intorno a 7.58.

2.6.3 Conclusioni

Il tool permette di eseguire delle analisi sugli smart contract e/o usare queste informazioni nel modo che si ritiene più appropriato. In questo specifico caso viene usato per fare delle considerazioni generali sulla struttura degli smart contract. Questo ha permesso di capire meglio la struttura degli smart contract in tutte le sue forme.

2.7 Caratteristiche innovative del tool sviluppato rispetto allo stato dell'arte

Tutti gli strumenti permettono differenti tipologie di visualizzazioni degli smart contract. Ciò che il tool realizzato fa rispetto a tutte le visualizzazioni descritte è di utilizzare altre rappresentazioni che vanno oltre a quella sotto forma di grafo.

Questo tipo di rappresentazione è indubbiamente utile e intuitiva, ma potrebbe in certi contesti risultare un po' limitante. Ci sono infatti alcune situazioni in cui il grafo deve rappresentare smart contract molto grandi e quindi diventa difficile tenere il livello di astrazione a valori accettabili.

Il tool sviluppato invece cerca di mostrare questa tipologia di informazioni utilizzando grafici nuovi, grazie all'utilizzo di 'd3.js', e soprattutto interattivi. Non solo cercando di utilizzare i grafici per dare all'utente un'idea su come si struttura lo smart contract, ma anche delle differenze che ha rispetto ad altri.

In particolare, risulta evidente la possibilità di generare grafici che permettano di visualizzare e confrontare il gas che le funzioni di ogni contratto consumerebbero nel caso fossero eseguite sulla EVM.

Nessuno dei tool/software citati permette di ricavare certe informazioni. La diversità rispetto agli altri sistemi non avviene solo nel modo in cui le informazioni vengono visualizzate, ma anche nella tipologia delle informazioni che vengono raccolte.

Capitolo 3

Tecnologie utilizzate

3.1 Node.js

Node.js [31] è un ambiente 'runtime' (ovvero una specie di sistema operativo che mette a disposizione le funzionalità di cui un programma ha bisogno per la corretta esecuzione) e 'cross-platform' (indipendente dalla piattaforma su cui viene eseguito) e 'open-source' (libero, tutti possono accedervi e usarlo gratuitamente). Si basa su linguaggio di programmazione JavaScript ed è utilizzato moltissimo in tutto il mondo.

Un'applicazione Node.js viene avviata utilizzando un singolo processo e non si dirama in altri thread iin base al numero di richieste in arrivo. Nonostante questo essa mette a disposizione dei programmatori una serie di funzionalità I/O asincrone nella sua libreria, che permette di gestire nel modo migliore la programmazione web (back-end). Questo infatti permette al codice JavaScript di non bloccarsi durante l'esecuzione nell'attesa di risposta, ma di procedere verso altre linee che possono essere eseguite in attesa della risposta di Node.js.

Quanado Node.js esegue un'operazione di I/O, come ad esempio la lettura dei valori da un database, invece di bloccare il thread in attesa che venga elaborata la risposta, essa elaborerà la risposta nel momento in cui la richiesta verrà conclusa.

Questo permette di avere più richieste contemporanee al server senza dover stare a gestire la concorrenza in quanto viene tutto fatto in modo automatico.

Come si è detto, Node.js viene utilizzato soprattutto per gestire la parte Server delle applicazioni Web, ma dal momento che questo può essere fatto in linguaggio JavaScript, è facilmente accessibile anche per tutti i programmatori specializzati nel front-end.

3.1.1 Storia

Node.js è nato relativamente da poco, nel 2010. Ormai è impiegato in tutto il mondo per la programmazione back-end e non esistono molti altri ambienti che permettano di avere tutta la flessibilità che Node.js mette a disposizione.

Prima che nascesse però, la programmazione back-end veniva gestita completamente da JavaScript, un linguaggio creato da Netscape per la manipolazione delle pagine web all'interno del browser.

Inizialmente Netscape aveva pensato di vendere 'Web Servers' che includessero un ambiente di programmazione, chiamato Netscape LiveWire, che permettesse di creare pagine web dinamiche usando programmazione in linguaggio Javascript. Questo però fu un fallimento e la programmazione server-side mediante Javascript ha cominciato a diffondersi soltanto nell'ultimo decennio con l'avvento di Node.js.

Node.js si è diffuso grazie al miglioramento degli engines di Javascript negli ultimi anni. Questo è stato possibile grazie a un lavoro instancabile da parte del team di sviluppo che ha lavorato in modo da permettere tutto il supporto necessario a rendere Javascript più performante. Inoltre il fatto che metta a disposizione una serie di nuove funzionalità innovative gli ha fatto acquistare punti importanti rispetto a tutto quello che c'era sul mercato.

3.1.2 Differenze rispetto al Browser

Come già detto si utilizza Javascript da entrambi i lati, ma nonstante questo l'approccio alla programmazione che si ha da una parte piuttosto che dall'altra è decisamente diverso.

In particolare le differenze le troviamo nell'ecosistema. Sul browser infatti, quello che viene fatto per la maggior parte del tempo è interagire con il DOM. Questo non avviene lato con Node.js, in quanto non è possibile agli oggetti che vengono messi a disposizione dal browser.

D'altro canto nel browser non possiamo utilizzare tutte le API che invece cu vengono messe a disposizione da Node.js grazie ai suoi moduli. Stiamo parlando delle migliaia (se ne contano circa 1.000.000) di librerie open-source che possono essere utlizzare con Node.js. Verranno in seguito citate tutte quelle usate per la creazione di questo tool.

Inoltre con Node.js si è in grado di avere controllo sull'ambiente in cui viene creato. Le applicazioni Node.js per funzionare devono avere una versione stabilita, cosa che non succede nel browser. Chiunque può accedervi con browser differenti.

3.1.3 Le Promise di Javascript

Una 'promise' può essere descritta come una promessa [32] che viene fatta al programmatore che stabilisce che quando il risultato sarà disponibile, questo verrà

restituito. Bisogna in ogni caso essere consci del fatto che non è certo che questo risultato potrà mai essere disponibile.

Dunque permettono di gestire il codice asincrono in maniera fluida e visivamente più pulita. Questa tipologia di struttura è stata parte del linguaggio per molto tempo e recentemente si è evoluto con l'avvento di 'async' e 'await'.

Appena una Promise viene chiamata le si associa lo lo stato pendente. Questo indica che la funzione è in esecuzione e non ha un dato che può essere ritornato immediatamente. Nel momento in cui l'esecuzione termina la Promise ritornerà al valore calcolato durante il suo stato pendente.

I valori di ritorno però non hanno sempre esito positivo. Al termine infatti la promise potrà risolversi in esito positivo o rifiutato. Questo dipenderà dal fatto che la Promise sia riuscita o meno ad effettuare il calcolo/operazioni necessarie.

Le promise vengono usate da importanti API, come ad esempio:

- Battery API.
- Fetch API.
- Service Workers

Nel caso sel nostro tool abbiamo utilizzato le Promise per effettuare le operazione di fetch dal client al server per ottener informazioni specifiche.

3.1.4 Async e Await

Un'estensione importante che è stata fatta riguardo alle Promise è l'introduzione di async e await. Permettono di scrivere codice asincorno in modo sincrono e quindi in maniera più lineare e ordinata.

Il funzionamento che hanno è molto semplice. Le funzioni async ritornano una Promise. Quando si vogliono chiamare queste funzioni bisogna anteporre la parola await al nome della funzione.

3.2 Express

Express.js [33] è un framewoork open-source Node.js che permette la creazione di applicazioni web sul lato server. Express è molto flessibile e veloce e potenzia molto Node.js senza compromettere le sue funzionalità.

Esso permette di creare delle API di routing, necessarie per gestire le richieste del client con la risposta che deve dare il Server, e di impostare dei middleware per rispondere a specifiche richieste HTTP.

3.2.1 Routing

Routing significa determinare come un'applicazione risponde a una specifica richiesta del client. La richiesta viene identificata da uno specifico URI e dal tipo di richiesta: GET, POST, PUT e DELETE.

Ogni 'route' (che sarebbe l'unione di URI e tipo di richiesta e uniti devono essere univoci) può avere più funzioni che vengono eseguite nel momento in cui si entra nella route.

La forma che assume la route può essere sintetizzabile in questo modo:

```
app.METODO(PERCORSO, FUNZIONI)
```

In cui:

- app rappresenta un'istanza di express.
- METODO è la tipologia di richiesta HTTP che viene gestita (GET, POST, PUT, DELETE).
- PERCORSO rappresenta l'URI che identifica la route in esame.
- FUNZIONI rappresenta le funzioni che vengono eseguite quando si entra nella route.

Le route possono avere più di una funzione come argomento. In questi casi, quando si hanno più funzioni è importante utilizzare nel modo corretto il **next** come argomento, in modo che per passare alla callback successiva si utilizzi la chiamat a **next()**

Un esempio di route con METODO POST e GET potrebbe essere questo:

```
// METODO GET
app.get('/', (req, res) => {
  res.send('Richiesta GET dalla pagina web')
})

// METODO POST
app.post('/', (req, res) => {
  res.send('Richiesta POST dalla pagina web')
})
```

Esiste inoltre un METODO speciale, **app.all()**, che viene usato per incanalare qualunque tipo di richiesta HTTP (indipendente da GET, POST o altro) che abbia l'URI (PERCORSO) indicato dalla route.

3.2.2 Metodo Request della Route

I parametri di una Route sono chiamati segmenti di URL e vengono usati per catturare i valori che sono presenti nell'URL. Questi valori si trovano salvati nell'oggetto **req.params**.

```
Route path: /users/:userId/books/:bookId
Request URL: http://localhost:3000/users/34/books/8989
req.params: { `userId`: `34`, `bookId`: `8989` }
```

Come possiamo vedere avremo il nome presente nel Route path che diventa il nome dell'attributo e il suo valore è quello presente nell'URL.

Per usare degli URI che possano prendere dei parametri basta utilizzare nel PERCORSO il solito indirizzo, ma anteponendo `:` ai campi.

```
app.get('/users/:userId/books/:bookId', (req, res) => {
  res.send(req.params)
})
```

L'oggetto req può avere anche altri campi oltre al params che dipendono da come viene strutturato l'URL. In particolare, vi sono anche i **req.query** e **req.body**.

1. Il primo si presenta quando l'URL è nella forma:

   ```
   Request URL: http://localhost:3000/profile?name=Alex

   //come si presenterà la route
   app.get(`/profile`, (req, res) =>{
       console.log(`Name: ` + req.query.name);
       res.send(req.query);
   }
   ```

   In req.query.name sarà presente il parametro `Alex` che troviamo nell'URL.

2. Il secondo si presenta quando vengono fatte delle richieste POST dal client al Server in cui i parametri che il client manda non si trovano nell'URL, ma in un pacchetto JSON apposito. Risulta utile usare questa modalità quando le informazioni che il client deve mandare sono abbastanza onerose e non sarebbe quindi possibile gestirle unicamente dall'URL. Inoltre questo permette di trasmettere informazioni che sono invisibili sull'URL.

   Il modo in cui si presentano le Route è lo stesso di quello visto per **req.query**. L'unica differenza è che il metodo che viene utilizzat è POST e non GET.

3.2.3 Metodo Response della Route

Il metodo **res** viene utilizzato per mandare una risposta al client e terminare il ciclo di richiesta e risposta. E' molto importante che al termine di ogni route vi sia una risposta da mandare al client, altrimenti il client continuerà a rimanere in attesa dì qualcosa che non arriverà mai.

I metodi associati alla response sono:

- **res.download()** : Vien richiesto il download di un file.
- **res.end()** : Termina il proceso di risposta.
- **res.json()** : Manda un pacchetto JSON in risposta al client
- **res.jsonp()** : pacchetto JSON in risposta con supporto JSONP.
- **res.redirect()** : Ridirige su un'altra route la richiesta.
- **res.render()** : Renderizza un template.
- **res.send()** : Manda una risposta di vario tipo.
- **res.sendFile()** : Restituisce un file in risposta.
- **res.sendStatus()** : Setta il codice di risposta e invia la sua rappresentazione sottoforma di stringa.

3.2.4 Middleware

I Middleware sono delle funzioni che possono accedere all'oggetto **req**, all'oggetto **res** e alla funzione di **next**. Quest'ultima è una funzione che permette di terminare, se presente, il middleware in esecuzione ed eseguire il successivo.

I middleware possono eseguire queste operazioni:

- Eseuire del codice
- Modificare gli oggetti **req** e **res**
- Concludere il ciclo di chiamata e risposta.
- Richiamare il middleware successivo.

Se il middleware non termina nel ciclo di richiesta e risposta corrente, deve essere chiamata la **next()** in modo da dare il controllo al middleware successivo. In caso contrario la richiesta rimarrà pendente

Un'esempio di middleware:

```
const express = require('express')
const app = express()

const myLogger = function (req, res, next) {
  console.log('LOGGED')
  next()
}

app.use(myLogger)

app.get('/', (req, res) => {
  res.send('Hello World!')
})

app.listen(3000)
```

Ogni volta che si entra nella route '/' verrà stampato sulla console il valore 'LOGGED'.

3.3 Multer

Si tratta di un middleware di node.js per la gestione dei 'multipart/form-data', che in sostanza è ciò che permette di fare il caricamento dei file [34].

Multer permette di aggiungere un oggetto body e un oggetto file/files all'oggetto request. Il body conterrà il valore testuale del form, mentre file conterrà il file caricato nel form.

Per comprendere al meglio il funzionamento osseravere il codice lato client (HTML) e Server(JS):

```
<form action='/profile' method='post' enctype='multipart/form-data'>
  <input type='file' name='avatar' />
</form>
```

```
const express = require('express')
const multer  = require('multer')
const upload = multer({ dest: 'uploads/' })

const app = express()

app.post('/profile', upload.single('avatar'), function (req, res,
    next) {
  // req.file is the 'avatar' file
```

```
9   // req.body will hold the text fields, if there were any
10 })
```

Nelle varie route saremo quindi in grado di leggere il contenuto del file che viene caricato dal form, inoltre multer, attraverso il comando **upload.single()**, permette di caricare e salvare sul server il file. In modo da averlo il modo permanente.

Ovviamente, nel caso non si volesse salvare sul server il file basterebbe utilizzare il comando **upload.none()** invece di **upload.single()**. In questo modo potremo leggere il file senza salvarlo per forza sul nostro server.

3.3.1 API

Quando viene caricato un file con multer, possiamo all'interno dell'oggetto file accedere alle seguenti informazioni:

Key	Description	Note
fieldname	Il nome specificato nel form	
originalname	Il nome del file sul computer dell'utente	
encoding	Tipo di codifica del file	
mimetype	Mime type del file	
size	Dimensione del file in bytes	
destination	La cartella nel quale il file è stato salvato	Archiviazione su disco
filename	Il nome del file con la destinazione	Archiviazione su disco
path	Il percorso completo del file caricato	Archiviazione su disco
buffer	Un buffer dell'intero file	Archiviazione della memoria

Tabella 3.1: Rappresentazione delle informazioni presenti in file

Nel web tool sviluppato, viene usata l'opzione upload.none() in quanto non è necessario il salvataggio del file sul server. La maggior parte delle API viene usata per ottenere le informazioni necessarie a sviluppare la risposta corretta e desiderata.

3.4 sqlite3

Si tratta di un modulo di Node.js che permette di gestire i database. In particolare i comandi che sono stati usati nel web tool sono:

3.4.1 new sqlite3.Database(filename [, mode] [, callback])

Viene creato un nuovo oggetto di tipo Database. Gli input rappresentano:

- **filename** : Rappresenta il file in cui si trova salvato il database che vogliamo aprire.
- **mode** : specifica la modalità con cui si vuole aprire il database. Ad esempio in modalità di sola lettura o lettura e scrittura (default).
- **callback** : Questa funzione viene chiamata nel momento in cui il database viene aperto con successo o quando si presenta un errore.

3.4.2 sqlite3.verbose()

Viene settata la modalità di esecuzione a 'verbose'. Questo permetterà di tenere traccia di tutte le query e i relativi risultati. In particolare può essere utile in caso di errori.

L'elenco delle operazioni che possono essere fatte sul database sono:

3.4.3 close([callback])

Permette di chiudere il database. Il parametro callback rappresenta una funzione che viene chiamata nel caso in cui la chiusura del database ha successo o meno.

3.4.4 run(sql [, param, ...] [, callback])

Permette di eseguire delle query SQL con parametri specifici. La callback è una funzione che viene eseguita dopo la query e in cui si possono manipolare i risultati. In particolare:

- **sql** : La query SQL da eseguire. Se la query non fosse corretta verrebbe chiamata la callback con un oggetto che rappresenta il messaggio di errore ottenuto da SQLite.
- **param, ...** : Quando la query SQL presenta dei parametri che devono essergli assegnati, si possono specificare qui.

- **callback** : Se presente, viene chiamata a seguito dell'esecuzione della query o in caso si presenti un errore durante l'esecuzione.

In generale la **run** viene usata per effettuare delle scritture/cancellazioni/aggiornamenti di righe del database.

3.4.5 get(sql [, param, ...] [, callback])

Permette di eseguire la query SQL con i parametri specificati e richiamare la callback al termine dell'esecuzione. La callback sarà una funzione del tipo: **function(err, row)**. Se il risultato della query è vuoto, il secondo parametro è di tipo **undefined**, altrimenti è un oggetto contenente il valore della prima riga del risultato della query.

3.4.6 all(sql [, param, ...] [, callback])

Permette di eseguire la query SQL con i parametri specificati e richiamare la callback al termine dell'esecuzione. La callback sarà una funzione del tipo: **function(err, rows)** . **rows** è un array. Se il risultato è vuoto, sarà un array vuoto, altrimenti sarà presente un oggetto che contiene ogni riga restituita dalla query. Bisogna considerare il fatto che vi sia la possibilità che il numero di righe restituite sia molto alto. Dal momento che queste vengono vengono salvate in memoria sarebbe meglio, in questi casi, utilizzare l'opzione **each** seguita da numerose **get**.

3.4.7 each(sql [, param, ...] [, callback] [, complete])

Permette di eseguire la query SQL con i parametri specificati e richiamare la callback una volta per ogni riga di risultato. la callback sara una funzione del tipo: **function(err, row)**. Se il risultato è vuoto, la callback non viene chiamata, in caso contrario, la callback è chiamata per ogni riga trovata nel risultato. L'ordine delle chiamate corrisponde esattamente all'ordine delle righe del risultato.

3.5 Surya: A Solidity Inspector

Surya [35] è un tool che viene utilizzato per fare delle analisi sugli smart contract. Permette di visualizzare una serie di output e informazioni riguardo la struttura degli smart contracts. Permette inoltre di interagire con il grafo delle chiamate delle funzioni in diversi modi.

Da qui in poi verranno presentati i comandi principali del tool:

3.5.1 graph

Questo tipo di comando permette di ottenere in output un grafo che descrive il flusso dello smart contract.

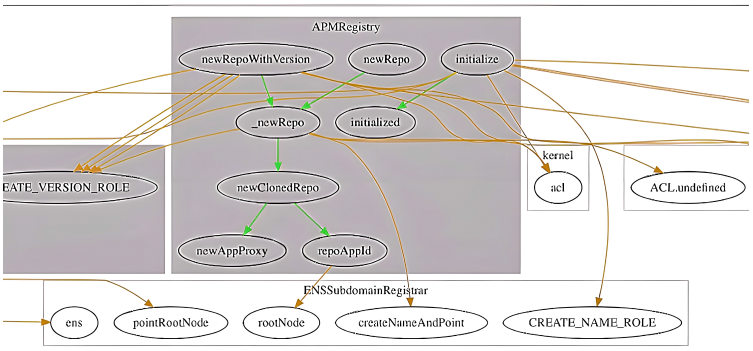

Figura 3.1: Output del comando graph

3.5.2 ftrace

Questo comando permette di visualizzare un albero delle chiamate delle funzioni. Per ciascuna avremo la possibilità di conoscere il nome del contract in cui sono definite e la loro visibilità. Inoltre se si tratta di funzioni che vengono chiamate dall'esterno saranno colorate di arancione.

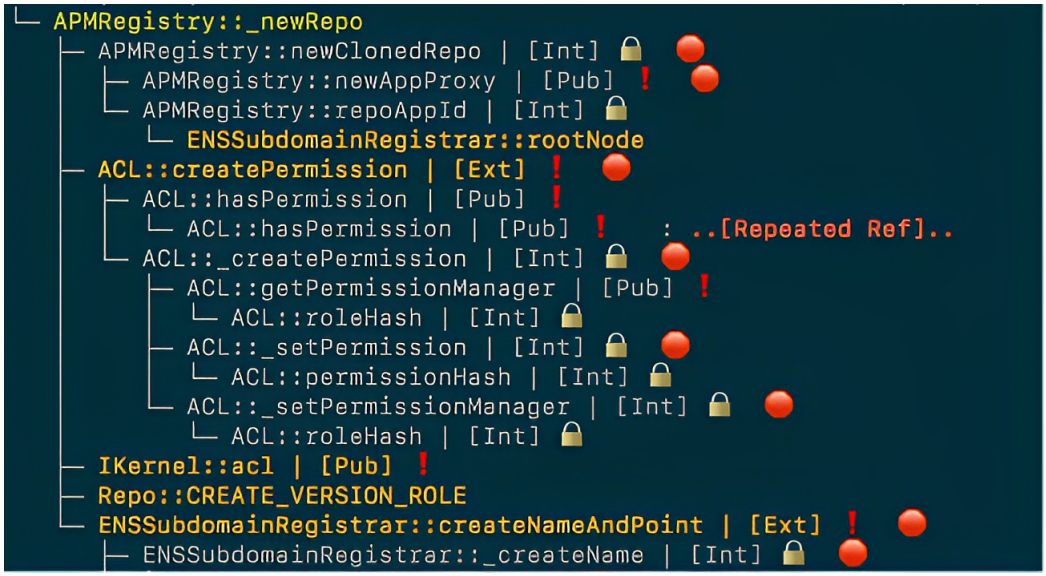

Figura 3.2: Output del comando **ftrace**

3.5.3 flatten

Questo comando permette di dare in output una versione 'appiattita' del codice sorgente, con tutti gli 'import' sostituiti dal corrispondente codice sorgente. Le righe in cui sono presenti gli 'import' vengono automaticamente commentate

3.5.4 describe

Questo comando mostra un sommario dei contratti e metodi presenti nel file. Le funzioni verranno divise in base alla loro visibilità:

- [**Pub**] public
- [**Ext**] external
- [**Prv**] private
- [**Int**] internal

Il simbolo $ indica che la funzione è payable. Il simbolo # indica che la funzione può modificare lo stato della blockchain.

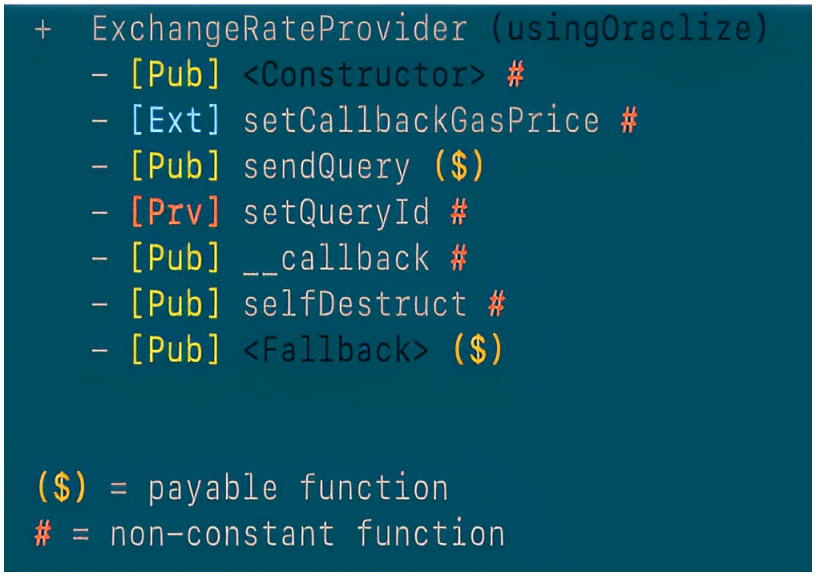

Figura 3.3: Output del comando **describe**

3.5.5 description

Si tratta di una funzione aggiunta al tool di Surya che si basa fortemente sull'algoritmo utilizzato in **describe**. In particolare, quest'ultima, si occupa di visualizzare

in console il risultato presente in Figura 3.3. La funzione **description** cerca di salvare queste informazioni in un'apposita struttura dati che possa essere utile al server per eseguire delle operazioni.

3.5.6 inheritance

Questo comando permette di visualizzare, sotto forma di grafo, l'albero delle dipendenze.

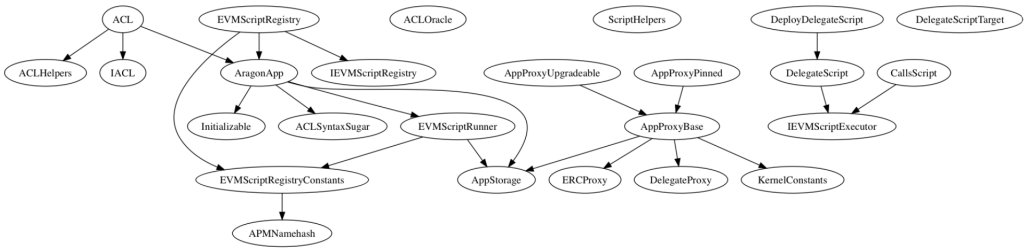

Figura 3.4: Output del comando **inheritance**

3.6 solidity-parser

Si tratta di un package JavaScript per l'analisi del linguaggio Solidity. Esso contiene il metodo **parse** che permette di creare l'AST (abstract syntax tree) del relativo smart contract passato in input.

Oltre allo smart contract il metodo **parse** può ricevere un secondo parametro:

Key	Type	Default	Description
tolerant	Boolean	false	Quando viene impostato a true esso salverà tutti gli errori di sintassi in una lista.
loc	Boolean	false	Quando viene impostato a true esso aggiungerà informazioni sulla posizione di ogni nodo, con chiavi di inizio e di fine che contengono il corrispondente numero di linea e colonna
range	Boolean	false	Quando viene impostato a true, esso aggiungerà informazioni sull'intervallo di ogni nodo, che consiste in un array di due elementi con i caratteri di inizio e fine

Tabella 3.2: Possibili parametri in input al metodo **parse**

La vera forza di questo modulo però risiede in particolar modo sul fatto che si possa andare a visitare l'albero che viene creato grazie al metodo **parse**.

```
var ast = parser.parse('contract test { uint a; }')

// output the path of each import found
parser.visit(ast, {
  ImportDirective: function(node) {
    console.log(node.path)
  }
})
```

In questo caso ad esempio si potrà visitare l'albero in base a ogni import che viene fatto nel contratto. Nel caso della funzione description (e anche describe) si cerca di visitare l'albero del contratto in base ai contract/library/interface e in base alle funzioni presenti in ciascuno di essi.

3.7 conceptNet

Si tratta di una rete semantica totalmente gratuita, che ha lo scopo di aiutare i computers a capire il significato delle parole usate dalle persone [36]. La conoscenza che ha conceptNet delle parole deriva da una serie di risorse, come ad esempio Wiktionary [37], Open Mind Common Sense, giochi vari (Verbosity e nadya.jp), WordNet [38] e JMDict [39].

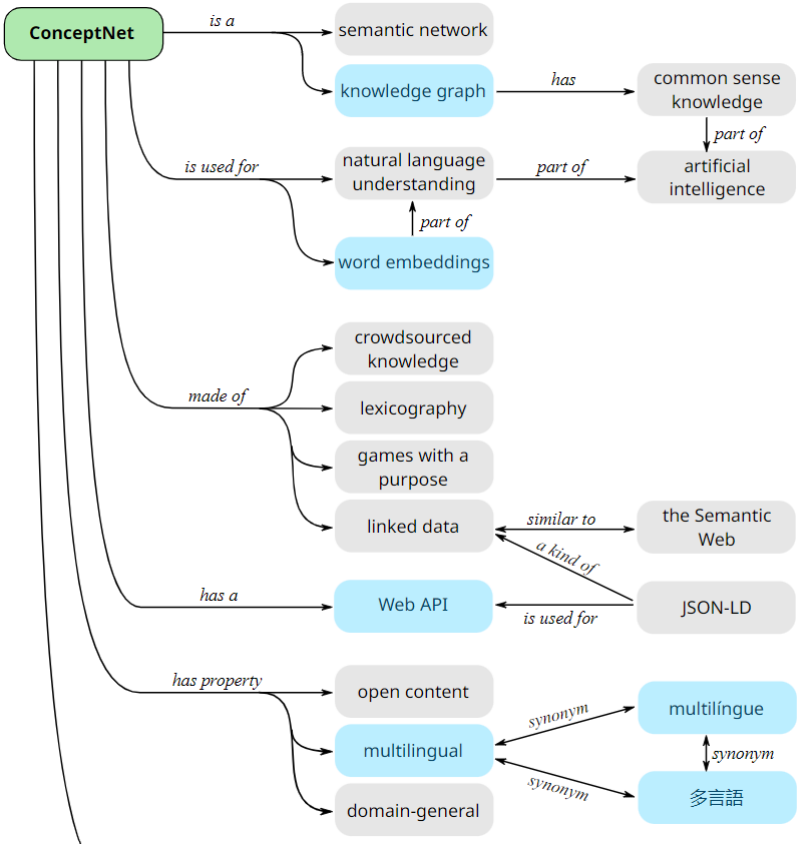

Figura 3.5: Cosa si può ottenere utilizzando ConceptNet

3.7.1 API

ConceptNet ha un'API REST all'indirizzo **api.conceptnet.io** da cui si possono prendere i dati in formato JSON-LD. Questo è il metodo più semplice per interagire con ConceptNet.

L'API [40] è accessibile in sola lettura e si può interagire con essa tramite specifici URL attraverso il protocollo HTTP GET. Si può fare direttamente dal proprio browser o nel linguaggio di programmazione che si preferisce.

I nodi di ConceptNet sono parole e frasi del linguaggio naturale (che si possono trovare su un dizionario). Ogni nodo è identificato da un URI che incomincia con **/c/**, seguita dall'abbreviativo di una lingua e infine il nome della parola di cui si vogliono avere informazioni. Ad esempio se volessi capire meglio la parola 'tesi' basterebbe scrivere **/c/it/tesi**.

I nodi si presentano in questa forma:

```
{
  '@context': [
    'http://api.conceptnet.io/ld/conceptnet5.7/context.ld.json'
  ],
  '@id': '/c/it/tesi',
  'edges': [...],
  'version': '5.8.1',
  'view': {
    '@id': '/c/it/tesi?offset=0&limit=20',
    '@type': 'PartialCollectionView',
    'comment': 'There are more results. Follow the 'nextPage' link
   for more.',
    'firstPage': '/c/it/tesi?offset=0&limit=20',
    'nextPage': '/c/it/tesi?offset=20&limit=20',
    'paginatedProperty': 'edges'
  }
```

Le informazioni realmente importanti si trovano in 'edges', ma quella parte verrà analizzata in seguito. Questo oggetto che viene ritornato è un JSON abbastanza standard. Le uniche differenze riguardano il fatto che alcune proprietà presentano il carattere @ all'inizio. Si tratta di proprietà specifiche del formato JSON-LD.

In particolare **@context** permette di dare un riferimento a un file che spiega tutte le proprietà presenti nel JSON in modo da poter essere capite meglio e analizzate dagli utenti che utilizzano l'API.

id specifica l'URI che si riferisce all'oggetto (il suo identificativo). Andando a osservare gli edges all'interno se ne trovano diversi e possono essere usati per analizzare l'arco specifico.

3.7.2 Struttura di un arco

All'interno della lista di 'edges' si troveranno gli oggetti che rappresentano gli archi del grafo. Questi archi permettono di collegare il nodo corrente con altri nodi.

```
{
      '@id': '/a/[/r/DerivedFrom/,/c/it/tesista/,/c/it/tesi/]',
      '@type': 'Edge',
      'dataset': '/d/wiktionary/en',
      'end': {
        '@id': '/c/it/tesi',
        '@type': 'Node',
        'label': 'tesi',
        'language': 'it',
        'term': '/c/it/tesi'
      },
      'license': 'cc:by-sa/4.0',
```

```
13        'rel': {
14          '@id': '/r/DerivedFrom',
15          '@type': 'Relation',
16          'label': 'DerivedFrom'
17        },
18        'sources': [
19          {
20            '@id': '/and/[/s/process/wikiparsec/2/,/s/resource/
    wiktionary/en/]',
21            '@type': 'Source',
22            'contributor': '/s/resource/wiktionary/en',
23            'process': '/s/process/wikiparsec/2'
24          }
25        ],
26        'start': {
27          '@id': '/c/it/tesista',
28          '@type': 'Node',
29          'label': 'tesista',
30          'language': 'it',
31          'term': '/c/it/tesista'
32        },
33        'surfaceText': null,
34        'weight': 1.0
35      }
```

L'**id** dell'arco è /a/[/r/DerivedFrom/,/c/it/tesista/,/c/it/tesi/. Questo URI descrive quali nodi sono collegati e in che modo. Non è necessario estrarre informazioni direttamente da questo URI, è preferibile utilizzare le proprietà **start, end e rel** presenti nel relativo arco.

```
1 'start': {
2           '@id': '/c/it/tesista',
3           '@type': 'Node',
4           'label': 'tesista',
5           'language': 'it',
6           'term': '/c/it/tesista'
7         }
```

3.7.3 start e end

start e **end** si riferiscono ai nodi di ConceptNet. Essi contengono un **id** da cui si possono andare a prelevare tutte le informazioni riguardo al nodo. Inoltre permettono di:

- un campo **label** che può essere capito dagli utenti, che può essere anche una frase più complessa rispetto a quella evidenziata nell'esempio.

- **language**, indica la lingua con cui la **label** è scritta.
- **term**, un collegamento a una versione più generale di questo termine.

3.7.4 rel

```
'rel': {
        '@id': '/r/DerivedFrom',
        '@type': 'Relation',
        'label': 'DerivedFrom'
      }
```

rel serve a individuare una delle 40 relazioni possibili che connettono i nodi di ConceptNet. Le relazioni sono identificate con nomi 'artificiali' come ad esempio 'DerivedFrom' in questo caso. Cerca di dare informazioni su come i due nodi sono collegati.

3.7.5 sources

```
'sources': [
        {
          '@id': '/and/[/s/process/wikiparsec/2/,/s/resource/
    wiktionary/en/]',
          '@type': 'Source',
          'contributor': '/s/resource/wiktionary/en',
          'process': '/s/process/wikiparsec/2'
        }
      ]
```

sources indicano perché ConceptNet reputa affidabili queste informazioni. Ogni arco deriva da una o più risorse. Ognuna di queste risorse è un oggetto con il suo identificativo **id**. Una sorgente può descrivere vari elementi che descrivono da dove deriva la conoscenza di una certa parola. Questi elementi sono:

- **contributor**, rappresenta una persona che ha partecipato all'analisi della parola.
- **activity**, indica cosa hanno fatto nello specifico.
- **process**, indica il processo che ha estratto la conoscenza riguardo alla parola.

3.7.6 license

```
1 'license': 'cc:by-sa/4.0'
```

La seguente proprietà indica come le informazioni ottenute possono essere eventualmente riutilizzate. Si tratta di un link che si riferisce al 'Creative Common's linked-data API'. Accedendo all'indirizzo **http://creativecommons.org/licenses/by-sa/4.0** è possibile recuperare la licenza che è leggibile sia da computer che dagli utenti.

3.7.7 weight

```
1 'weight': 1.0
```

Il peso indica quanto L'informazione è attendibile. Un peso abbastanza buono è 1.0. Questo tenderà ad aumentare nel caso in cui l'informazione deriva da più sorgenti o da sorgenti più credibili.

L'utilizzo di queste informazioni sono state essenziali per capire quali argomenti erano correlati con i nomi delle funzioni presenti negli smart contract.

In particolare il web tool si concentra sull'analisi della relazione **'HasContext'** per cercare di capire a quali macro argomento il nome della funzione fosse correlato.

3.8 d3.js

D3.js [41] è una libreria JavaScript che permette di manipolare i documenti basati su dati. Può essere uno strumento interessante per dare una forma ai proprio dati usando HTML, SVG, CSS. D3 si concentra molto sugli standard offerti dal web offrendone anche tutte le funzionalità. Questo permette di creare 'potenti' visualizzazioni basandosi sulla manipolazione del DOM.

In particolare, D3 permette di unire i dati al DOM (Document Object Model), e successivamente di applicare una serie di trasformazioni al documento. Quindi ad esempio si possono creare delle tabelle HTML partendo da un array di numeri, oppure, con gli stessi dati creare un SVG interattivo.

D3 può essere visto come un framework che cerca di risolvere il punto cruciale di un problema attraverso la manipolazioni dei documenti dipendenti da dati. Si tratta di un framework incredibilmente flessibile e veloce, in grado di supportare grandi quantità di dati e comportamenti dinamici per interazione e animazione.

Di seguito verranno elencate alcune operazioni basilari del framework per la creazioni di grafici più complessi come quelli realizzati nel web tool in esame.

3.8.1 Selezioni

D3 utilizza un approccio dichiarativo, operando su dei sets di nodi chiamati ‘selections‘. Ad esempio per selezionare tutti i paragrafi del documento e modificare il colore in blu avremo:

```
d3.selectAll('p').style('color', 'blue');
```

I nodi possono essere selezionati in base al tag name, valore degli attributi, classe, ID e molto altro. D3 ha numerosi metodi per la manipolazione dei nodi: settare gli attributi o lo stile; creare degli ‘event listeners‘; aggiungere, rimuovere o ordinare i nodi; cambiare l'HTML o il testo.

3.8.2 Proprietà dinamiche

In D3 gli stili, attributi e altre proprietà possono essere considerate delle funzioni di dati, non solo delle semplici costanti. Nonostante possano apparire molto semplici, queste funzioni possono essere molto potenti per il comportamento che possono assumere.

Ad esempio per colorare i paragrafi randomicamente:

```
d3.selectAll('p').style('color', function() {
  return 'hsl(' + Math.random() * 360 + ',100%,50%)';
});
```

Oppure per alternare ombre di grigio per nodi pari e dispari:

```
d3.selectAll('p').style('color', function(d, i) {
  return i % 2 ? '#fff' : '#eee';
});
```

I dati che vengono passati alla funzione rappresentano un array di valori, ogni valore viene passato come primo argomento della funzione. In particolare ogni elemento dell'array verrà associato a un singolo elemento partendo dal primo fino all'ultimo. Ovvero, il primo elemento del vettore sarà associato al primo nodo, il secondo elemento del vettore sarà associato al secondo nodo e cosi via fino alla fine.

3.8.3 Enter e Exit

Usando questi due comandi è possibile creare nuovi nodi in base ai dati in entrata e rimuovere i nodi in uscita che non sono più necessari.

Quando un dato è legato a una selezione, ogni elemento dell'array è accoppiato col corrispondente nodo della selezione. Se ci sono meno nodi rispetto ai dati, quelli in eccesso possono essere usati per stanziare nuovi nodi.

```
d3.select('body').selectAll('p').data([4, 8, 15, 16, 23, 42]).enter()
    .append('p').text(function(d){ return 'I m number ' + d + '!';});
```

Se la enter e la exit vengono omesse verranno automaticamente selezionati solo gli elementi per cui si riesce a creare una corrispondenza coi dati.

3.8.4 Transizioni

Le transizioni permettono di interpolare stili e attributi nel tempo. L'interpolazione può essere controllata attraverso l'utilizzo di funzioni come 'elastic', 'cubic-in-out' e 'linear'. Gli interpolatori di D3 supportano entrambe le primitive, come numeri, numeri sotto forma di stringa e valori composti.

Per rendere nero lo sfondo di una pagina avremo:

```
d3.select('body').transition()
    .style('background-color', 'black');
```

Per ridimensionare i cerchi presenti su una mappa di simboli con un ritardo avremo:

```
d3.selectAll('circle').transition()
    .duration(750)
    .delay(function(d, i) { return i * 10; })
    .attr('r', function(d) { return Math.sqrt(d * scale); });
```

Modificando solo gli attributi che effettivamente cambiano, D3 riduce permette di avere una complessità grafica elevate con elevato frame rates.

3.9 web3.js

Quando si sviluppano applicazioni su blockchain tramite Ethereum bisogna considerare due possibili approcci:

- Sviluppo di smart contract: Scrittura di codice Solidity che verrà caricato sulla blockchain.

- Sviluppo di siti web o client che interagiscono con la blockchain: in questo caso la scrittura del codice si concentrerà nella lettura e scrittura di valori presenti sulla blockchain attraverso l'utilizzo di smart contract

Web3.js [42] permette la realizzazione del secondo punto precedentemente descritto, ovvero permettere di interagire con l'Ethereum blockchain da parte degli utenti. In sostanza si tratta di un insieme di librerie che permettono di eseguire operazioni tipiche della blockchain.

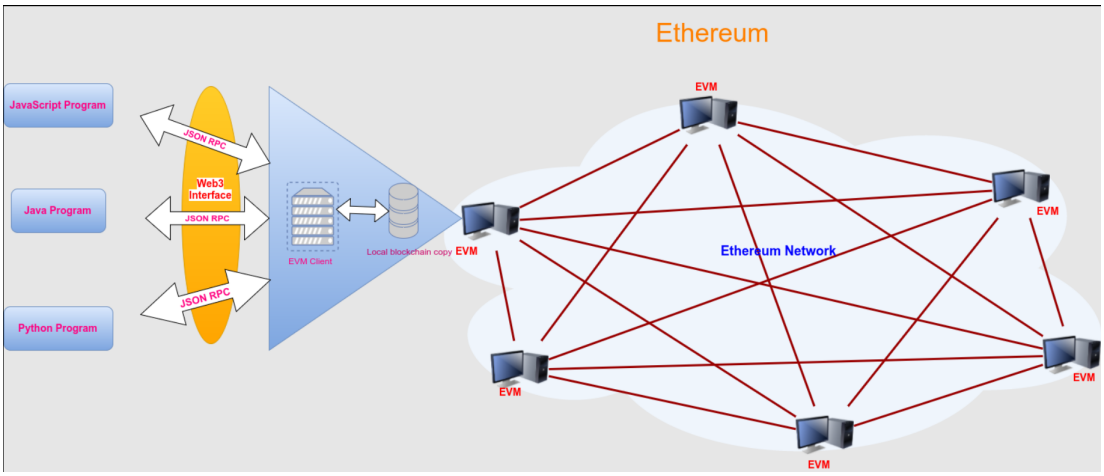

Figura 3.6: Interazione tra Web3.js e la blockchain Ethereum [43]

Web3.js riesce a dialogare con l'Ethereum blockchain attraverso JSON RPC, che significa protocollo 'Remote Procedure Call'. Ethereum è una rete peer-to-peer di nodi che salvano una copia di tutti i dati e codice (smart contract) presente sulla rete (blockchain). Web3.js ci permette di effettuare interazioni con un singolo nodo della rete a nostra scelta attraverso il protocollo JSON RPC.

3.9.1 Infura

Per potersi collegare a un nodo della blockchain sarà però necessario averne l'accesso. Principalmente esistono due modi per poter fare questo:

- Avviare il proprio nodo Ethereum con tool specifici (Geth o Parity), ma questo richiede di scaricare una grande quantità di dati dalla blockchain e non è un metodo consigliato

- Utilizzare Infura per accedere a un nodo Ethereum senza doverne avviare uno personale.

Infura è un servizio che permette l'accesso gratuito a un nodo della rete Ethereum. Tutto ciò che serve è registrarsi per ottenere una chiave e l'RPC URL relativo alla rete a cui ci si vuole connettere.

Nel web tool realizzato si procede proprio con l'utilizzo di Infura per collegarsi a un node dalla blockchain Ethereum.

3.9.2 Leggere dati dagli smart contract

Per leggere i dati dagli smart contract con Web3.js sono necessarie due cose:

- Una rappresentazione JavaScript dello smart contract con cui vogliamo interagire.
- Un modo per richiamare le funzioni dello smart contract quando stiamo leggendo i dati.

Per ottenere la rappresentazione JavaScript di uno smart contract sarà necessario utilizzare la funzione:

```
new web3.eth.Contract(jsonInterface[, address][, options])
```

Questa funzione permette di creare una nuova istanza del contratto con tutti i suoi metodi ed eventi definiti nella sua 'json interface', meglio conosciuta come ABI del contratto.

L'ABI di uno smart contract significa 'Abstract Binary Interface', e si tratta di un array JSON che descrive come uno smart contract 'lavora'.

Analisi dei parametri in input:

1. **jsonInterface/ABI** - **Object**: Interfaccia json per l'istanza del contratto.
2. **address** (opzionale) - **String**: L'indirizzo dello smart contract da richiamare.
3. **options** (opzionale) - **Object**: Le opzioni del contratto.

A questo punto dobbiamo cercare di leggere i dati dallo smart contract chiamando le sue funzioni. Tutte le funzioni si trovano nel seguente metodo:

```
myContract.methods.myMethod([param1[, param2[, ...]]])
```

In questo modo si può accedere alle funzioni di uno smart contract ed eseguire una serie di operazioni tra cui:

- called

- send
- estimated
- createAccessList
- ABI encoded

Si può accedere alle funzioni dello smart contract in base al:

- nome: **myContract.methods.myMethod(123)**
- il nome dei parametri: **myContract.methods['myMethod(uint256)'](123)**
- la firma della funzione: **myContract.methods['0x58cf5f10'](123)**

Le operazioni che possono essere eseguite sono già state citate nel capitolo 4.6.4 con particolare attenzione alla **estimateGas** che è appunto la funzione utilizzata per calcolare il gas delle funzioni dello smart contract.

Tra le varie operazioni che si possono effettuare ricordiamo le più importanti, tra cui:

1.
```
1  myContract.methods.myMethod([param1[, param2[, ...]]]).
   estimateGas(options[, callback])
2
```

   Questa funzione è stata già ampiamente discussa e analizzata nel capitolo 4.6.4.

2.
```
1  myContract.methods.myMethod([param1[, param2[, ...]]]).call(
   options [, defaultBlock] [, callback])
2
```

   Viene chiamato ed eseguito un metodo costante sulla EVM senza inviare alcuna transazione. La chiamata non altera lo stato dello smart contract.

   Parametri

   - **options**: si tratta di un oggetto contenente le informazioni riguardo a:
     - **from - String**: L'indirizzo da cui la 'transazione' dovrebbe essere eseguita.
     - **gasPrice - String**: Il gas da utilizzare per questa 'transazione' (chiamata della funzione).

  - **gas - Number**: Il massimo gas che può essere utilizzato per questa chiamata.

- **defaultBlock**: Se si passa qua un parametro significa che non verrà utilizzato il blocco di default (contract.defaultBlock) per eseguire il calcolo.
- **callback**: Questa funzione verrà invocata a seguito del risultato ottenuto dall'esecuzione del metodo dello smart contract.

3.
```
1  myContract.methods.myMethod([param1[, param2[, ...]]]).send(
   options[, callback])
2
```

Viene mandata una transazione allo smart contract ed esegue il metodo specificato. Questa operazione può alterare lo stato dello smart contract.

Parametri

- **options**: Contiene le informazioni di:
  - **from - String**: L'indirizzo da cui deve essere inviata la transazione.
  - **gasPrice - String**: Il gas da usare per questa transazione. **gas - Number**: Il massimo gas previsto per l'esecuzione della transazione.
  - **value - Number|String**: Il valore trasferito per la transazione.
  - **nonce - Number**: Il 'nonce' della transazione (numero univoco che si assegna alla transazione).
- **callback**: Funzione che verrà richiamata a seguito dell'esecuzione della funzione.

La callback ritornerà l'hash su 32 bytes della transazione.

4.
```
1  myContract.methods.myMethod([param1[, param2[, ...]]]).
   encodeABI()
2
```

Codifica l'ABI per il metodo specificato. Il risultato sarà l'hash della firma su 32 bit più i parametri passati in Solidity. Questo può essere usato per inviare una transazione, chiamare un metodo, o passarlo in un metodo di un altro smart contract come argomento.

5.
```
myContract.methods.myMethod([param1[, param2[, ...]]]).
createAccessList(options, blockHashOrBlockNumber [, callback])
```

Permette di creare una lista di accesso a cui l'esecuzione del metodo può accedere quando viene eseguito sulla EVM.

Parametri

- **options**: Contiene le informazioni di:
  - **from - String**: L'inidirizzo da cui deve essere fatta la transazione.
  - **gas - Number**: Il massimo gas che può essere usato per la 'transazione' (chiamata della funzione dello smart contract).
- **block**: Il numero o l'hash del blocco.
- **callback**: La callback verrà richiamata al termine dell'esecuzione della funzione.

Come valore di ritorno della Promise avremo la lista riferita alla transazione.

Capitolo 4

Sistema realizzato

4.1 Vista generale e architettura

Il sistema in Figura 4.1 è stato realizzato mediante l'utilizzo combinato dei linguaggi: Javascript, SQL, HTML5, CSS. In particolare, si tratta di un'applicazione Web in grado di ricevere in input dall'utente uno o più file in formato Solidity per analizzarli nel dettaglio e restituire le informazioni estratte sotto forma di grafici e/o altro.

L'analisi è resa possibile grazie allo scambio di informazioni con un server personale, in cui sono effettuate tutte le operazioni necessarie, per giungere alla creazione di apposite strutture utilizzando alcune librerie di nodeJS.

Lo scambio di informazioni avviene mediante API REST (Representational State Transfer) con scambio di dati in formato JSON (Javascript Object Notation). Inoltre, per questioni di efficienza, per non dover ripetere le stesse operazioni più volte su uno stesso file, si è creato un database per salvare le informazioni.

A seconda del numero di file che l'utente vorrà inserire l'applicazione risponderà con differenti comportamenti.

In particolare:

- Un solo file in input (Figura 4.1, ***1s***);
- Più file in input (Figura 4.1, ***1m***).

4.2 Singolo file in input

In questo caso il testo del contratto verrà visualizzato per intero nella schermata della pagina Web. Mediante un menù nella parte sovrastante sarà possibile evidenziare alcune parti specifiche del contratto in base ad alcune sue proprietà.

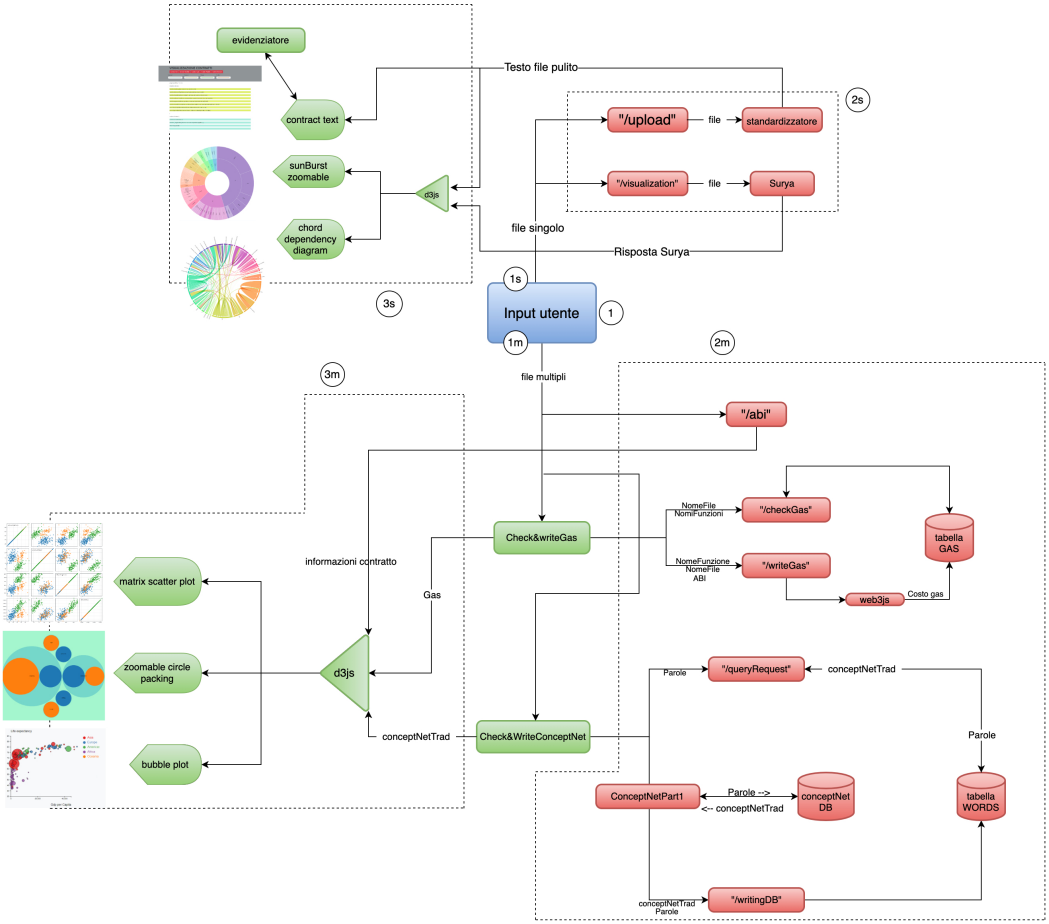

Figura 4.1: Architettura del sistema realizzato

In alternativa sarà possibile la visualizzazione di alcuni grafici che riassumano la struttura gerarchica del contratto;

4.2.1 Fase di '/upload'

Nel momento in cui l'utente inserisce un singolo file in input, la pagina web manderà una richiesta POST al server privato tramite una fetch all'indirizzo di '. . . /upload'. Il body della richiesta conterrà il file inserito dall'utente e inglobato all'interno di una struttura apposita nota come 'FormData'.

Il file verrà ricevuto dal server, anche grazie all'utilizzo di una libreria di NodeJs: 'multer', un middleware utilizzato per gestire il caricamento dei file sul server (in particolare i 'multipart/form-data'). In base alla struttura del progetto non è stato necessario salvare i file sul server, ma semplicemente fare una lettura 'on the fly'

per ricavare le informazioni necessarie, per questo motivo si è utilizzata l'opzione 'upload.any()'. Quest'ultima, infatti, permette la lettura del file senza per forza doverlo salvare sul server.

4.2.2 Standardizzatore

A questo punto il server esegue una serie di operazioni che permettono di estrapolare il contenuto testuale del file ma in maniera leggermente modificata (la semantica rimane la stessa) per facilitare l'esecuzione degli algoritmi lato client. Questo perché non esiste una regola specifica per programmare in solidity, o meglio, tutti possono scrivere il codice ma in maniera leggermente diversa. Per questo motivo si è trovata la necessità di conformare ogni singolo file ad alcuni standard di livello testuale, in modo da poter utilizzare algoritmi comuni per tutti i file.

In particolare, si è cercato di:

- Non permettere che vi fosse altro testo oltre al carattere '}' . Quest'ultimo deve occupare da solo una riga intera. Per cui tutte le volte in cui oltre al carattere '{' ci si trova ad avere altro, si devono separare le due parti e portarle su righe diverse;
- Evitare qualsiasi considerazione sulle righe di commento: in particolare, nel momento in cui si dovesse trovare il carattere '/**' si prosegue alle righe successive ignorando tutto, fino a quando non si trova la terminazione del commento stabilita dal carattere '*/';
- Cercare di unire righe che normalmente non dovrebbero essere spezzate. In particolare, se una riga non termina con il carattere ';' (viene dato per scontato il fatto che il contratto sia scritto e compilato in maniera corretta) significa che il carattere si troverà su una riga successiva, per cui sarà necessario riunire tutte le righe fino a quando non si troverà il carattere precedentemente citato.

Queste modifiche verranno salvate in un'apposita struttura dati (un vettore in cui ogni elemento rappresenterà una riga) che potrà essere restituita al client tramite formato JSON.

4.2.3 Contract Text

Una volta ottenuta la risposta lato Client, un vettore con all'interno tutte le righe del file, si dovrà visualizzare all'interno della pagina Web il contenuto testuale del file. Dato che tutte le operazioni sono state eseguite lato server si dovranno semplicemente inserire dei nuovi elementi nel DOM. In particolare, basterà creare degli elementi di tipo ' ', uno per ogni riga del file (o per ogni elemento del vettore).

Per questioni di semplicità ogni paragrafo avrà uno specifico id che sarà il numero di riga a cui fa riferimento.

A seguito di questo verrà abilitato il menù nella parte superiore della pagina. Si tratta di un menù che permette di effettuare delle evidenziazioni di testo in base a delle specifiche proprietà.

In particolare, permetterà di evidenziare parti di testo in base a:

- Visibilità delle funzioni: public, private, external, internal;
- Tipi delle funzioni: Fallback, Receive Ether, Constructor;
- Funzioni di tipo payable o mutating;

Gli algoritmi che permettono l'evidenziazione di queste parti sono molto simili tra loro e sono tutti basati sull'utilizzo di alcune informazioni fornite da Surya, una libreria di NodeJs.

4.2.4 Surya

L'utilizzo di Surya avviene tramite una fetch all'indirizzo '. . . /visualization'. La richiesta GET al server permette di richiamare una specifica funzione di Surya (leggermente modificata) chiamata 'description'.

Questa funzione riceve una stringa come parametro, che rappresenterà l'intero contratto posto in input dall'utente, e utilizzando le funzionalità di 'Solidity-parser' [44] (permetterà di creare un albero del codice visitabile —> Abstract Syntax Tree (AST)), andrà a creare una struttura dati con tutte le informazioni riguardo il codice Solidity.

Nello specifico verrà creata una struttura contenente due differenti tipologie di oggetti:

- Oggetti di tipo contract: verranno creati in seguito alla lettura nell'albero (ast) di specifiche strutture di tipo: contract, library e interface. Quest'ultimi saranno caratterizzati da: nome, tipologia di struttura, bases (tutti i contratti o librerie o interfacce che estendono/implementano) e un vettore contenente tutte le funzioni che hanno al loro interno.
- Oggetti di tipo method: verranno creati in seguito alla visita nell'albero (ast), di strutture di tipo function. Per ognuna di esse si avranno salvate le informazioni riguardo a: nome, se si tratta di un costruttore o fallback o receive Ether, la visibilità della funzione (internal, external, private, public), se la funzione è payable o mutating, i modifiers della funzione.

Gli oggetti di tipo method verranno poi inseriti all'interno del vettore dell'oggetto contract di cui fanno riferimento.

4.2.5 Evidenziatore

Ottenuta client-side la struttura appena descritta si potrà procedere nell'evidenziazione del codice in base alle proprietà specifiche selezionate nel menù sovrastante di cui si era precedentemente parlato.

Dal momento che si devono evidenziare manualmente le righe interessate, si dovrà scorrere su tutti i paragrafi e capire se sono da evidenziare oppure no.

In particolare, l'algoritmo procede seguendo dei punti fondamentali:

- Quando si incontrano delle righe di commento, ovvero caratteri del tipo '/**', si devono ignorare e si deve proseguire oltre fino alla riga successiva a quella che li termina, ovvero dove si presenta il carattere '*/'. Questo perché sicuramente non presentano nessuna informazione rilevante a livello di codice.

- Si cerca di capire se la riga corrente presenta una dichiarazione di strutture di tipo: contract, library o interface. Se così fosse, se ne cerca e verifica la presenza all'interno della struttura ottenuta da Surya. Questa fase è importantissima in quanto, in seguito, è molto probabile che incontreremo dichiarazioni e implementazioni di funzioni. Per capire, ad esempio, la loro visibilità, sarà necessario leggere la struttura fornita in precedenza da Surya, ma per farlo dovremo sapere a quale contract o library o interface la funzione in esame fa riferimento.

- Si cerca di capire se la riga corrente presenta una dichiarazione della struttura di tipo function. Se così fosse, se ne cerca e verifica la presenza all'interno della struttura ottenuta da Surya e si preleva da essa le informazioni che permettono di capire se e come va evidenziata la riga in base alle proprietà della funzione.

- Nel caso ci si trovasse ad analizzare una riga che non fa parte di nessuna delle condizioni esposte nei punti precedenti, si sta analizzando una riga che fa parte del corpo di una funzione. È importante in questo caso evidenziarla in base alle informazioni che si sono ottenute dalla dichiarazione delle funzione corrente e si controlla se nella riga vi è la presenza del carattere '}'. Questo permette di capire che la riga successiva non farà più parte del corpo della funzione. Esistono però una serie di problematiche legate a questo punto. In particolare è possibile che vengano utilizzati i caratteri '}' per indicare il termine del corpo di un costrutto if(){} else {} ad esempio. Per questo motivo si deve cercare di tenere traccia di tutti i caratteri '{'. Ad ogni '{' dovrà corrispondere una '}'. Solo così si potrà considerare la terminazione di una funzione in modo corretto e quindi evitare di continuare a evidenziare righe successive.

4.3 Chord diagram functions

Il Chord diagram functions è un grafico, realizzato con d3.js, che permette una visualizzazione grafica delle chiamate delle funzioni del contratto. In particolare, si tratta di un diagramma circolare in cui ogni spicchio della circonferenza rappresenta una singola funzione del contratto. Per ogni singolo spicchio di circonferenza potremmo trovare delle frecce in entrata o in uscita di diverso spessore.

Le frecce mettono in relazione le funzioni. Nello specifico, una freccia uscente da uno spicchio A ed entrante in uno spicchio B indica che la funzione A richiama dentro di sé la funzione B.

Lo spessore della linea invece sta ad indicare quante volte la funzione a cui la freccia punta è richiamata.

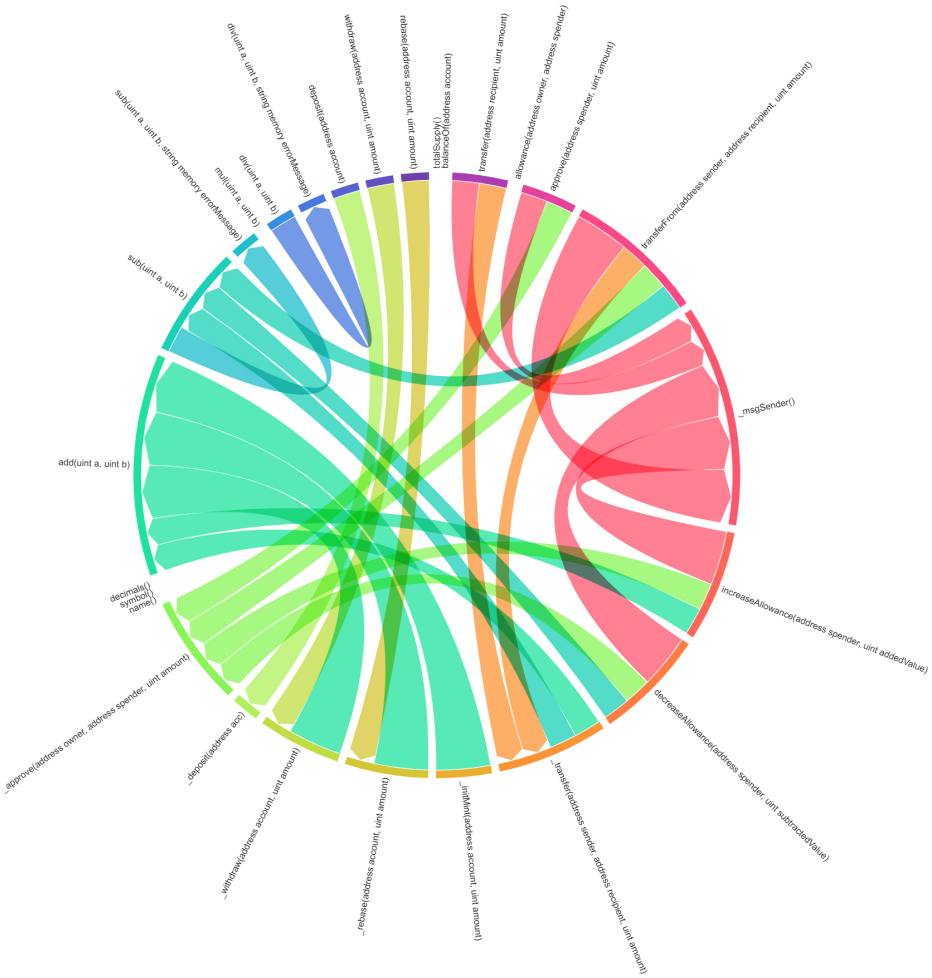

Figura 4.2: Chord diagram functions

4.3.1 Elaborazione e creazione dati

Prima della creazione del grafico bisogna occuparsi di creare e gestire nel modo corretto i dati che devono essere utilizzati per tale scopo. Non è necessario effettuare alcuna richiesta al Server in quanto tutte le informazioni sono già presenti client-side. In particolare, il testo dello smart contract.

La prima cosa che si è dovuta creare è stata una matrice quadrata che contenesse le dipendenze tra tutte le funzioni. Ogni cella (i, j) rappresenta il numero di volte che la funzioni 'i' richiama la funzione 'j'.

Per creare la matrice si è dovuto quindi andare a rileggere l'intero smart contract riga per riga. Una prima lettura viene fatta per trovare i nomi di tutte le funzioni presenti e salvarli in un apposito vettore. Quest'ultimo, oltre a elencare tutte le funzioni dello smart contract, permetterà l'associazione di ogni funzione con gli indici all'interno della matrice che verrà successivamente creata. Una volta creata tale struttura si può passare alla creazione della matrice.

Vanno fatte alcune considerazioni importanti sull'organizzazione dell'algoritmo:

- Si è ignorata completamente ogni riga che contenesse dei commenti. Come precedentemente detto nel capitolo 4.2.4 è importante evitare di considerare queste parti, che altrimenti potrebbero causare errori non da poco. In questo caso però si sono dovute ignorare anche le strutture denominate come 'interface' e 'abstract'. Questo perché, da definizione, le interfacce non implementano nulla, ma sono solo dei contenitori di dichiarazione per le funzioni, che poi verranno implementate dai contract che estenderanno tali interfacce. Discorso simile può essere fatto per gli abstract contract che a livello pratico non hanno nessun tipo di implementazione. Dal momento che avevamo bisogno di leggere il corpo delle varie funzioni, per evitare letture multiple e possibili problemi, si è deciso di ignorare tali strutture.

- Nel momento in cui si presenta una riga con la dichiarazione di una funzione, ovvero si ha la presenza della keyword 'function' all'interno della riga, bisogna innanzitutto cercare di isolare il nome della funzione. Una volta fatto questo sarà necessario capire l'indice che tale funzione ha all'interno della matrice (ovvero l'indice a cui si trova nel vettore dei nomi delle funzioni);

  - A questo punto si cominciano a leggere tutte le righe successive alla presente fino a quando non si giunge al termine del corpo della funzione ('}', facendo attenzione a tutte le varie '{');
  - Per ognuna delle righe successive si deve cercare di capire se vi sono delle chiamate ad altre funzioni che vengono fatte. Il principio con cui ho stabilito se vi fosse una chiamata a una funzione del contratto è la presenza delle parentesi tonde. Ogni funzione infatti, per essere

chiamata, indipendentemente dalla quantità di parametri (possono anche non essercene), ha bisogno delle parentesi tondo dopo il nome. Partendo da questo presupposto, il numero di '(' corrisponderà al numero di chiamate alle funzioni. (Questo generalmente non è sempre vero. In Solidity Ci sono delle keyword specifiche come ad esempio 'require' o 'assert' o 'revert' che hanno la parentesi tonda, ma questo non porta problemi in quanto nessuna funzione potrà mai chiamarsi con il nome di una keyword).

- Per ogni funzione che viene chiamata sulla riga si deve recuperare il nome originale, trovare il suo indice di riferimento all'interno della matrice e a quel punto aggiornare cella della matrice relativa;
- Possono esserci dei casi, isolati e rari, in cui due funzioni all'interno dello stesso smart contract abbiano il medesimo nome. Solidity permette questa cosa a patto che il numero di parametri differisca tra loro. Per questo motivo ho dovuto implementare un ulteriore controllo per arginare questa problematica e permettere di aggiornare nel modo corretto la matrice. In particolare, quando mi trovo in questa situazione, prendo la funzione che ha lo stesso numero di parametri richiesti da quella che viene chiamata.

4.3.2 d3js - Creazione grafico

Per la creazione del grafico viene utilizzato d3.js. Si tratta di una libreria che permette di creare gli elementi del DOM in maniera molto più dinamica e intuitiva.

In particolare, l'approccio utilizzato è di creare ogni singolo elemento del grafico e poi assemblare tutto quanto insieme.

Ecco la lista degli elementi:

- Color: per visualizzare e distinguere bene ogni funzione dalle altre è importante dare a ogni spicchio della circonferenza (che rappresenta una funzione specifica) un colore particolare. D3js mette a disposizione una funzione che permette di fare tutto questo automaticamente e si tratta di ***d3.scaleOrdinal()***. Si tratta di una funzione che permette di creare una scala di colori in cui ogni funzione è associata in modo univoco a un unico colore.

- Svg (***Scalable Vector Graphics***): si tratta di un elemento fondamentale in quanto sarà il contenitore del nostro grafico. In generale l'elemento svg è un formato di grafica vettoriale basato su XML e fornisce opzioni per disegnare forme differenti. Essa inoltre è un'immagine che si adatta alla grandezza del contenitore in cui si trova. Tramite l'attributo ***ViewBox*** abbiamo impostato le dimensioni dell'immagine.

  Alcune delle caratteristiche salienti di SVG sono:

1. SVG è un formato immagine basato su vettore e testo;
2. SVG ha una struttura simile all' HTML;
3. SVG può essere rappresentato come un modello a oggetti del documento;
4. Le proprietà SVG possono essere specificate come attributi;
5. SVG ha posizioni assolute rispetto all'origine;
6. SVG può essere incluso così com'è nel documento HTML.

- Chords: è una funzione di d3js che, presa in input la matrice delle chiamate delle funzioni, restituisce un array di chords dove ogni chord rappresenta il flusso bidirezionale combinato tra due nodi i e j e presenta una serie di proprietà:

  1. Source - il sottogruppo di origine;
  2. Target – il sottogruppo di destinazione

  Ogni sottogruppo di origine e destinazione è anche un oggetto con le seguenti proprietà:

  1. startAngle - l'angolo iniziale in radianti;
  2. endAngle – l'angolo finale in radianti;
  3. value – la matrice del valore di flusso [i][j];
  4. index – l'indice del nodo i;

  Questa struttura è fondamentale per creare le relazioni all'interno del grafico.

- Function fade: si tratta di una funzione che permette di visualizzare il grafico con maggiore chiarezza. In pratica quando si passa col cursore sopra uno spicchio rappresentante una funzione, tutte le frecce non riguardanti la funzione vengono opacizzate. In questo modo risulteranno visibili solo le frecce relative alla funzione su cui il cursore si trova. Nel momento in cui il cursore si allontana dalla zona riservata alla funzione tutte le frecce tornano allo stesso livello di opacità. Questo può risultare molto utile nel visualizzare in modo più chiaro solo le informazioni che ci servono;

In seguito alla creazione di questi elementi ho utilizzato la logica di d3js per combinarli insieme e andare a creare il grafico vero e proprio.

4.4 Zoomable Sunburst diagram

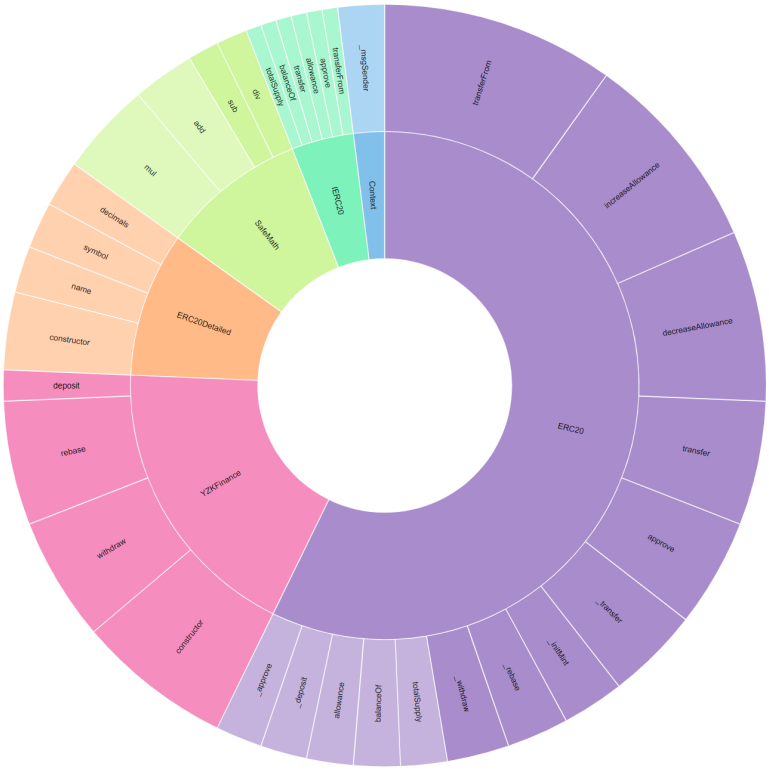

Figura 4.3: Zoomable Sunburst Diagram

Questo grafico permette di rappresentare la correlazione tra le funzioni di uno smart contract ma in modo più dinamico e interattivo rispetto al metodo precedente.

In particolare, il diagramma è strutturato come un cerchio in cui vengono disposte all'interno tutte le strutture (contract, interface, library, function) di cui lo smart contract è composto seguendo un andamento ad anello. La parte centrale dell'anello rimarrà vuota e fungerà da reset del grafico, ovvero permetterà di tornare alla configurazione standard iniziale.

Il primo anello del grafico (quello più interno) conterrà tutte le strutture basilari dello smart contract (contract, interface e library). Ognuna di queste avrà riservata una porzione di corona circolare che dipenderà dalla quantità e dimensione delle strutture interne di cui è composta (functions/variabili generiche).

Il secondo anello invece presenterà tutte le strutture (functions) di cui ogni porzione del primo è composta. Anche in questo caso la porzione di corona circolare riservata a una singola struttura dipenderà dalla sua dimensione e lunghezza.

Come detto però il grafico permette una certa interattività, questo perché le singole porzioni degli anelli rispondono agli input dell'utente. Nello specifico l'utente può, cliccando sulla porzione di suo interesse, entrare più in profondità in quella struttura. Questo significa che la composizione degli anelli cambia completamente portando ad avere come primo anello il contenuto della struttura selezionata (probabilmente le sue functions) e come secondo anello le strutture presenti in ogni struttura del primo anello.

Questo ha portato ad avere una grande quantità di informazioni da dover gestire, soprattutto nei casi in cui gli smart contract hanno dimensioni molto grandi in cui appaiono numerose strutture dati.

4.4.1 Elaborazione e creazione dati

In questo caso, rispetto al precedente, si è dovuto interrogare il server per prelevare alcune informazioni che erano necessarie. E' stata quindi eseguita una fetch all'indirizzo '. . . /visualization' in modalità GET senza nessun passaggio di parametri nella richiesta. L'informazione che verrà restituita è già spiegata nel capitolo 4.2.4 e si tratta di utilizzare una funzionalità di Surya.

La struttura necessaria per la creazione di un grafico di questo tipo è molto complessa. Si tratta infatti di una struttura ad albero in cui i nodi padri (i progenitori da cui tutti i figli discendono) sono rappresentati da contract o library o interface, mentre tutti i successivi nodi figli sono rappresentati dalle functions.

Si tratta quindi di una struttura nota solo per i primi due livelli, ovvero i padri e i relativi figli, ovvero le funzioni da cui i padri sono composti. Per avere una struttura completa dello smart contract si è dovuto utilizzare un algoritmo ricorsivo che venisse attivato ogni volta che una funzione ne chiamava un'altra.

Nello specifico, l'algoritmo presenta alcuni punti fondamentali:

- I primi due livelli della struttura, come già detto, vengono creati in modo abbastanza semplice utilizzando le informazioni ottenute da Surya;
- Per i livelli successivi sarà necessario andare a leggere e analizzare riga per riga il codice Solidity dello smart contract:
    1. Se la riga contiene commenti, si ignorano e si prosegue oltre;
    2. Se la riga contiene la dichiarazione di interface o abstract contract si prosegue portandosi alla riga in cui queste strutture terminano. È bene evidenziare il fatto che queste strutture vengono ignorate in quanto le loro informazioni sono già state precedentemente salvate utilizzando Surya.
    3. Se la riga contiene la dichiarazione di una library o contract si cerca di trovare l'indice a cui questa fa riferimento all'interno dell'albero. Essa dovrà per forza essere presente nel primo anello dell'albero.

4. Se la riga contiene la dichiarazione di una funzione (e quindi sappiamo anche a quale contract appartiene) si cerca di trovare l'indice a cui questa fa riferimento all'interno dell'albero. Essa dovrà per forza essere presente nel secondo anello dell'albero in riferimento al contract di cui fa parte.

- Nel caso in cui si stesse analizzando una riga che corrisponde al corpo di una funzione è necessario capire se avvengono chiamate ad altre funzioni. Questo è possibile utilizzando il metodo già spiegato nel capitolo 4.3.1 in cui, per riassumere, ad ogni carattere '(' si fa coincidere la chiamata a una funzione presente nello smart contract. A questo punto, dopo avere trovato gli indici della funzione all'interno dell'albero, verrà eseguita una ricorsione. Questa funzione ricorsiva cercherà nel testo l'ultima funzione che è stata chiamata e si andrà ad analizzare riga per riga per vedere se anch'essa presenta delle chiamate ad altre funzioni. In caso affermativo si esegue ulteriormente la ricorsione, altrimenti si ritorna dalla ricorsione andando a salvare nell'albero le informazioni così ottenute.
- Una volta che si giunge al termine della ricorsione si sarà creata una parte di albero completa per una certa funzione. Ovviamente questa operazione avrà bisogno di essere ripetuta per ogni singola funzione presente nel contratto.
- Nel caso in cui nella riga fosse presente il carattere '}' vengono fatte una serie di considerazioni per capire se il carattere è: parte del costrutto if(){} else {} o simili; se indica il termine di una funzione; se indica il termine di un contratto (o library o interface). Questo è importante in quanto in alcuni dei casi esposti si dovranno fare delle operazioni sulla struttura ad albero che si sta creando.

Ogni funzione sarà caratterizzata da un attributo 'value' oppure 'children'. Il value è presente nelle funzioni che non richiamano nessun'altra funzione, mentre children rappresenta un vettore presente nelle funzioni che richiamano altre funzioni.

Il value è fondamentale perché permette al grafico di assumere dimensioni specifiche in base alla dimensione delle funzioni e nel nostro caso corrisponde al numero di righe di cui la funzione si compone.

4.4.2 d3js - Creazione grafico

Per la creazione del grafico si utilizza la libreria d3.js. Alcuni degli elementi utilizzati sono comuni o molto simili a quelli del grafico precedente per cui non verranno ulteriormente analizzati ma semplicemente citati:

- Color: permette di marcare con uno stesso colore le funzioni che fanno parte dello stesso contratto, con sfumature differenti in base alla profondità che hanno nell'albero. (funzionamento spiegato nel capitolo 4.3.2);

- partition: è una funzione di D3js e produce diagrammi di adiacenza, ovvero una variante che permette di riempire lo spazio di un nodo di un diagramma ad albero. Invece di tracciare un collegamento tra genitore e figlio nella gerarchia, i nodi vengono disegnati come aree solide (archi o rettangoli) e la loro posizione rispetto ad altri nodi rivela la loro posizione nella gerarchia. La dimensione dei nodi codifica una dimensione quantitativa che sarebbe difficile da mostrare in un normale diagramma ad albero. Nel nostro caso specifico forniamo in input alla funzione la nostra struttura creata al punto precedente. In questo modo verrà restituito l'array di nodi associati al nodo radice specificato.

  Diversi attributi sono popolati su ciascun nodo:

  1. parent – il nodo padre o null per la radice;
  2. children – l'array dei nodi figli o null per le foglie;
  3. value – il valore del nodo, come restituito dalla funzione apposita;
  4. depth – la profondità del nodo, partendo da zero per la radice;
  5. x – la coordinata x minima della posizione del nodo;
  6. y – la coordinata y minima della posizione del nodo;
  7. dx – l'estensione x della posizione del nodo;
  8. dy – l'estensione y della posizione del nodo;

  Le coordinate x e y servono per dare una dimensione al layout, ma rappresentano solo una sistema di coordinate arbitrarie. Si possono infatti trattare x e y come rispettivamente raggio e angolo per produrre una visualizzazione radiale piuttosto che cartesiana. Nel nostro caso, infatti, abbiamo utilizzato una rappresentazione radiale.

- Transition: è una funzione d3js che permette di rendere interattivo e dinamico il grafico. Grazie ad essa, infatti, il diagramma è in grado di mutare in base agli input dell'utente. La funzione non applica le modifiche istantaneamente, bensì interpola il DOM dal suo stato attuale allo stato di destinazione desiderato per una determinata durata. In particolare, essa restituisce una nuova transizione partendo da quella presa in input. Se il parametro in input è un'istanza di transizione, la transizione che viene restituita avrà lo stesso ID e nome della transizione specificata. Se esiste già una transizione con lo stesso ID su un elemento, la transizione esistente viene restituita per quell'elemento. In caso contrario, la tempistica della transizione restituita viene ereditata dalla transizione esistente dello ID sull'antenato più vicino di ciascun elemento selezionato. Per questo motivo si può anche utilizzare transition per sincronizzare per sincronizzare una transizione tra più selezioni o per riselezionare una transizione per elementi specifici e modificarne la configurazione.

- Svg: Come descritto per il grafico precedente sarà l'elemento contenente l'intero grafico.

4.5 Multipli file in input

In questo caso verranno automaticamente visualizzati nella pagina web alcuni grafici che mettano a confronto alcune proprietà dei contratti di cui si è richiesta l'analisi.

4.5.1 Preparazione richiesta al Server - ABI (Clilent Side)

A seguito dell'inserimento in input dei file dell'utente, dovrà essere preparata la struttura da inviare al server come body. Questo può essere fatti inglobando i file in una apposita struttura nota come FormData.

Però la richiesta al server non viene subito eseguita. Per fornire il servizio, infatti, avremo bisogno di altre informazioni, in particolare, l'ABI di ogni singolo contratto ricevuto in input. Per questo motivo, al click dell'utente sul bottone submit, viene creato dinamicamente un apposito form contenente tante caselle di testo quanti i file in input per permettere di inserire le ABI di ciascun file.

ABI (Ethereum Binary Interface) è la rappresentazione binaria dell'interfaccia di uno smart contract. Questa rappresentazione verrà fornita con un formato molto simile a JSON.

L'ABI di uno smart contract sarà necessario in alcune situazioni:

- Deploy di uno smart contract;
- Interazione con Web3;

Noi ci troviamo proprio nel secondo caso. Vorremo infatti interagire con la blockchain di Ethereum, grazie appunto a Web3 (una libreria di nodeJS), per verificare il gas delle funzioni dei nostri contratti.

Gli attributi che descrivono la forma di una ABI sono:

- type: definisce il tipo della funzione. Può essere una delle seguenti, 'function', 'conctructor', 'receive' (per le funzioni receive ether), 0 'fallback' (per le funzioni di default);
- name: definisce il nome della funzione;
- inputs: è un array di oggetti che ne definisce i seguenti parametri:
  - name: definisce il nome dei parametri;

  - type: definisce i tipi canonici dei parametri. Ad esempio, uint256;

- outputs: è un array di oggetti di output simile a quello di input;

- stateMutability: definisce se una funzione è mutable. Può assumere uno dei seguenti valori: 'pure' (non può leggere o modificare lo stato della blockchain), 'view' (lo stato della blockchain può essere letto, ma non modificato), 'nonpayable' (la funzione non accetta Ether, ma può leggere e modificare lo stato della blockchain), 'payable' (la funzione accetta Ether e può leggere e modificare lo stato della blockchain).

Cliccato ulteriormente il tasto di submit dopo aver inserito le ABI di ciascun file viene eseguita la richiesta al server.

4.5.2 Fase di '/abi'

Dopo aver inserito nel FormData del punto precedente anche le ABI dei singoli contratti viene fatta la richiesta POST al Server, tramite l'utilizzo di una fetch, all'indirizzo '.../abi' e come body il FormData in cui sono inglobate tutti i file e le ABI.

Il server dovrà elaborare il body in arrivo e restituire al client una struttura che possa permettere di creare le visualizzazioni successive.

In particolare, le strutture avranno questo formato:

```
var info = {
      name: '',
      length: 0,
      size: 0,
      contracts: [],
      library: [],
      interface: [],
      gasPrice: 0,
      abi: 'nulla',
    }
var smart = {
      name: '',
      methodsName: [],
      visibility: {private: 0, public: 0, internal: 0, external: 0},
      payable: 0,
      mutating: 0,
      fallback: 0,
      receiveEther: 0,
}
```

L'oggetto 'info' rappresenta tutte le informazioni di un singolo file.

I vettori presenti in info rappresentano tutti i contract, library e interface presenti nel file. Ogni cella di questi vettori rappresenterà un contract, una library o un interface e sarà rappresentata dall'oggetto smart.

L'oggetto 'smart' conterrà i nomi di tutti i metodi presenti nel relativo contract (o interface o library) e tutte le informazioni globali su di essi. In particolare: il numero di funzioni private, pubbliche, interne, esterne; il numero di funzioni payable, mutating, fallback e receive ether.

Per creare una simile struttura dati si è nuovamente dovuto utilizzare Surya su ogni file in input. Questo ha poi permesso di prelevare quelle informazioni e gestirle ed elaborarle in modo tale da giungere alla struttura desiderata.

Avremo quindi un vettore di oggetti info (i file) che a sua volta conterrà 3 vettori (contract, library, interface) di oggetti smart che conterrà le informazioni su tutte le funzioni.

4.6 Check&writeGas - Gas delle funzioni

Questo blocco, che troviamo in Figura 4.1 avrà la responsabilità di calcolare e/o reperire il gas delle funzioni presenti negli smart contract inseriti dagli utenti. Esso presenta all'interno, due funzioni specifiche che si occupano della gestione del gas.

4.6.1 CheckGasOnDb()

La richiesta fatta al capitolo 4.5.2 non ha però calcolato il gas dei contratti e delle singole funzioni.

Dal momento che questo calcolo è abbastanza oneroso in termini di tempo e risorse si cerca di salvare le informazioni del gas di uno smart contract su un apposito database presente sul server.

Per questo motivo si utilizza una funzione di controllo:

```
1 async function checkGasOnDb()
2
```

Questa funzione esegue, per ogni file, una richiesta POST al server all'indirizzo '.../checkGas' e inserisce, come body, un oggetto contenente il nome del file e un array contenente i nomi delle funzioni presenti nel file di cui è possibile calcolare il gas.

4.6.2 Fase di '/checkGas'

Lato server verrà eseguita una query sul database per verificare se il gas delle funzioni presenti nel file sono già state calcolate o meno.

La tabella chiamata GAS che conserva queste informazioni nel nostro database è così strutturata:

	functionName	fileName	gasCost
	Filtro	Filtro	Filtro
1	allowance	contract1.sol	0.001841718662083
2	approve	contract1.sol	0.00182245665038782
3	balanceOf	contract1.sol	0.00181062909934692
4	decimals	contract1.sol	0.00177953953661084
5	decreaseAllowance	contract1.sol	0.00182245665038782
6	deposit	contract1.sol	0.00181062909934692
7	financer	contract1.sol	0.00181062909934692
8	increaseAllowance	contract1.sol	0.00182245665038782
9	name	contract1.sol	0.00177953953661084
10	rebase	contract1.sol	0.00182245665038782

Figura 4.4: Tabella GAS presente nel database

A seguito della nostra query SQL che sarà del tipo:

```
1 let sql = 'SELECT functionName, gasCost FROM GAS WHERE fileName = ?
    AND functionName = ? OR functionName = ? OR ... '
2
```

Nella query saranno presenti tanti functionName quante saranno le funzioni di cui si vuole conoscere il gas.

La query avrà come valore di ritorno un oggetto contenente:

- un array di oggetti con: nome e gas di una funzione;
- un array con tutte le funzioni di cui non è presente il gas all'interno del database;
- il nome del file di cui si sta facendo la query;

Questa struttura verrà restituita direttamente al client e servirà per capire di quali funzioni ancora è necessario calcolare il gas.

4.6.3 writeGasOnDb(request)

Questa funzione viene eseguita subito dopo la checkGasOnDb( ). Riceve in input un parametro che rappresenta l'output della checkGasOnDb. Quindi sarà un array di oggetti del tipo appena descritto.

Essa avrà il compito di andare a calcolare il gas delle funzioni di cui non vi era alcuna informazione sul nostro database. Per fare questo, dovremo creare una struttura dati apposita, da poter dare in input, a una funzione di web3.js che eseguirà il calcolo del gas di una specifica funzione.

I parametri di cui si ha necessità per svolgere queste operazioni sono gli input delle varie funzioni, per ottenerli basterà cercarli all'interno dell'ABI relativa al file in cui si trova la funzione di cui si vuole calcolare il gas.

In solidity esistono i seguenti tipi di variabili (e quindi di possibili input):

- bool: si tratta della variabile booleana che può assumere i valori true o false;
- int, int8, int16, int32, ..., int256: interi con segno di determinata lunghezza;
- uint, uint8, uint16, uint32, ..., uint256: interi senza segno di determinata lunghezza;
- address: è un tipo di dato dalla dimensione di 20 byte (lo stesso di un indirizzo in ethereum) e serve appunto per memorizzare un indirizzo di ethereum;
- bytes1, bytes2, bytes3, ..., bytes32: sono array con dimensione fissa;
- string: sono stringhe di dimensione variabile;

Il nostro algoritmo dovrà creare degli input standard in base a quelli richiesti dalla funzione in esame. Quindi ad esempio se per la funzione A sono presenti due parametri in input di tipo address e uint256 dovremo creare un vettore in cui avremo, in ordine, i seguenti elementi:

```
1 address = `0x5B38Da6a701c568545dCfcB03FcB875f56beddC4`;
2 uint256 = 100;
3
```

Sono valori totalmente casuali che fungono da input per la nostra funzione. Questo vettore di input, il nome della funzione in esame e il nome del file in cui si trova la funzione verranno passati al server attraverso una richiesta POST all'indirizzo `.../writeGas`.

4.6.4 Fase di '/writeGas'

Lato Server, dopo aver ricevuto queste informazioni nella body della richiesta, si utilizza una funzione di web3js per interagire con il nostro smart contract. In particolare viene stanziato un nuovo oggetto:

```
let myContract = new web3.eth.Contract(JSON.parse(req.body.ABI));

```

Per la creazione di questo oggetto è necessario fornire la JSON interface del relativo smart contract, in questo modo web3 convertirà automaticamente tutte le chiamate attraverso l'ABI di basso livello. Questo permetterà di interagire con gli smart contract come se fossero oggetti JavaScript.

Si procede nella modifica manuale dell'indirizzo associato a questa specifica istanza (in quanto si tratta di un oggetto) del contratto. Questo permetterà di effettuare delle transazioni sullo specifico smart contract.

Si prosegue con l'utilizzo di un'ulteriore funzionalità che è possibile utilizzare su uno smart contract in web3js: methods.

```
myContract.methods.myMethod([param1[, param2[, ...]]])

```

Esso permette di creare un oggetto di tipo transazione per la specifica funzione presente sullo smart contract di cui è stata creata l'istanza, su cui in seguito potranno essere effettuate delle operazioni specifiche.

Si può accedere ai metodi dello smart contract in tre differenti modi:

- Utilizzando il nome:

```
myContract.methods.myMethod(123)

```

- Utilizzando il nome e i parametri:

```
myContract.methods['myMethod(uint256)'](123)
//metodo che si utilizza in questo caso

```

- Utilizzando la firma:

```
myContract.methods['0x58cf5f10'](123)

```

Le informazioni riguardo al nome della funzione e i relativi parametri sono tutti presenti all'interno del body presente nella richiesta, come specificato in precedenza.

L'oggetto transazione che viene ritornata sarà cosi strutturato:

- Array - argomenti passati al metodo precedente, possono essere cambiati;
- Function - call: chiamerà i metodi costanti ed eseguirà i metodi dello smart contract sulla EVM senza eseguire nessuna transazione;
- Function - send: Manderà una transazione allo smart contract ed eseguirà i metodi;
- Function - estimateGas: Restituirà una stima del gas utilizzato se il metodo venisse eseguito nel momento in cui lo smart contract fosse su blockchain;
- Function - econdeABI: codifica l'ABI per questo metodo.

Per il calcolo del gas della funzione viene utilizzata la Promise estimateGas:

```
myContract.methods.myMethod([param1[, param2[, ...]]]).estimateGas(
    options[, callback])

```

Eseguirà il metodo specificato per stimare il gas richiesto per un'esecuzione sulla EVM. Le stime possono variare significativamente da una volta all'altra a causa dello stato del contratto e della blockchain stessa.

Tra le possibili options (nessuna obbligatoria) che possono essere passate alla Promise troviamo:

- from - String: L'inidirizzo da cui dovrebbe essere fatta la transazione;
- gas - Number: Il gas massimo previsto per questa chiamata;
- value - Number|String|BN|BigNumber: il valore trasferito in wei per la transazione

All'interno della Promise, dopo che sarà stato trovato il gas della funzione, verrà eseguita una specifica funzione che interagirà con il database per salvare il valore appena calcolato.

4.6.5 contractDao.writeGasOnDb()

La funzione si occuperà di gestire tre parametri in input:

1. il Gas della funzione;
2. il nome della funzione;
3. il nome dello smart contract a cui la funzione appartiene.

Una semplice query SQL permetterà di salvare l'informazione sulla tabella GAS del database:

```
const sql = 'INSERT INTO GAS(functionName, fileName, gasCost) VALUES
    (?, ?, ?)';

```

Si ritornerà al Client un JSON con le informazioni calcolate grazie a web3js.

La fetch all'indirizzo '.../writeGas' verrà eseguita per tutte le funzioni di tutti gli smart contract di cui ancora non si conosce il gas.

Un'apposita struttura di questo tipo permetterà di salvare tutte le informazioni:

```
let result = {
    result: [{name: el.functionName, gas: el.gasCost}, {...}], //
    array con le informazioni delle funzioni su cui si conosce già il
    gas
    toCalculate: toCalculate, //array con le i nomi delle funzioni di
     cui si deve ancora calcolare il gas
    file: gas.file //nome del file a cui l'array delle funzioni fa
    riferimento
}
```

4.7 Check&writeConceptNet - Contesti dei nomi delle funzioni

ConceptNet è una rete semantica libera che permette di capire il significato di una parola. Si è utilizzato questo tool, per fare in modo che si analizzassero nello smart contract i nomi delle funzioni.

Questo perché logicamente quando si programma qualunque cosa si cerca di dare nomi a funzioni e o variabili in modo tale da permettere a tutti di sapere cosa una struttura dati fa o ha dentro di sé direttamente dal nome.

Non si tratta in realtà di un regola assoluta, ma come buon metodo di programmazione risulta essere un'idea condivisa da molte persone del settore.

In tal senso nasce l'idea di poter analizzare i nomi delle funzioni per ritrovare un argomento specifico di cui il contratto si occupa.

Questo blocco, presente in Figura 4.1, sarà dunque composto da una serie di funzioni che gestiranno strutture dati e query alle API di conceptNet:

4.7.1 async function queryToDatabase(wrapp)

Questa funzione permette di verificare se una specifica parola è già stata ricercata e analizzata con una query ai database di conceptNet. Se cosi fosse, significherebbe che i dati che la riguardano si trovano già salvati in un'apposita tabella del nostro database provato.

Prima di far questo però la funzione svolge un compito importantissimo che nasce da una considerazione che è necessario fare. Come detto prima, spesso durante la programmazione, si cerca di chiamare le varie strutture in modo da rendere noto il loro scopo.

Questo però non implica che il nome debba effettivamente essere reale, ovvero una parola che abbia un reale significato, che può essere trovata all'interno del dizionario italiano.

Per fare un esempio: se ho una funzione che deve mandare del denaro a un destinatario con un indirizzo specifico la funzione potrebbe chiamarsi 'sendTransactionOfMoney'. Se andassimo a chiedere a conceptNet il significato di questa parola, verrebbe ritornato un codice di errore che dice che la parola, giustamente, non esiste. Non è infatti presente in nessun dizionario.

Da qui sorge una riflessione riguardo al fatto che spesso, i nomi che i programmatori usano per le loro strutture, sono composti da più parole di senso compiuto.

L'algoritmo, per l'appunto, cerca di fare proprio questo: partendo dal nome di una funzione tenta di separare tutte le parole di cui è composta seguendo alcune specifiche:

- Se il nome della funzione è 'constructor' o 'receive ether' significa che siamo di fronte a funzioni che non hanno nessuna valenza in merito all'analisi che si sta svolgendo. Motivo per cui si dovranno ignorare.
- Se il nome della funzione presenta il carattere '_' significa che ci sono più parole all'interno del nome. Si procede con la loro separazione e analisi individuale per verificare l'eventuale presenza di lettere maiuscole.

  Questo perché potrebbero indicare l'eventuale presenza di parole composte all'interno della stessa, in cui il marcatore per separarle sarebbe indicato proprio da questo tipo di lettera maiuscola.

Questa analisi viene eseguita solo nel caso in cui la singola parola, ottenuta dividendo in base al carattere '_', abbia una lunghezza adeguata a poter essere ulteriormente separata (in particolare, se la lunghezza è almeno di 4 caratteri).

Una volta verificato questo, ogni parola viene dunque separata e salvata in un apposito vettore con tutti i caratteri in minuscolo.

- Se il nome della funzione non presenta alcun carattere del tipo '_' significa che la parola potrebbe avere separazioni che derivano dall'utilizzo delle lettere maiuscole.

  Viene dunque ripetuto il procedimento del punto sovrastante.

- Se il nome della funzione non ha né lettere maiuscole, né caratteri '_', semplicemente potrebbe voler dire che la parola è una unica e di senso compiuto. Viene restituita da sola.

Eseguita l'operazione per tutte le funzioni di uno specifico file, si potrà proseguire con la richiesta al database privato, per verificare la presenza delle parole trovate.

Si eseguirà quindi una richiesta POST all'indirizzo '.../queryRequest' passando come body della richiesta l'array contenente le parole estratte da tutte le funzioni.

4.7.2 Fase di '/queryRequest'

Lato Server verrà eseguita una query sul database che ci permetterà di capire e prelevare le informazioni sulle parole di cui già si hanno informazioni. La query verrà eseguita sulla tabella '***WORDS***' del nostro database

	wordName	relationType	relationValue
	Filtro	Filtro	Filtro
1	msgsender	hasContext	
2	total	hasContext	us,slang,mathematics
3	supply	hasContext	
4	balance	hasContext	Balance,watchmaking
5	transfer	hasContext	
6	allowance	hasContext	minting,commerce
7	approve	hasContext	archaic,english law,legal
8	initmint	hasContext	
9	rebase	hasContext	source control,computing,dentistry

Figura 4.5: Tabella WORDS presente nel database

La nostra query SQL si presenterà in questo modo:

```
1 const sql = 'SELECT relationValue, wordName FROM WORDS WHERE wordName
      = ? OR wordName = ? OR ...'
2
```

La query presenterà tanti campi 'wordName' quanti quelli all'interno del vettore passato come input. Esso infatti avrà tutte le parole di cui si vogliono avere informazioni.

Al termine della query verrà ritornato un oggetto JSON al client con le seguenti informazioni:

- **vettConcept** - array: si tratta di un array contenente tutte le parole che non erano presenti sul database e di cui si dovranno avere informazioni da conceptNet;
- **resultQuery** - array: si tratta di un array contenente il risultato ottenuto leggendo il database.

4.7.3 async function conceptNetPart1(success)

Questa funzione si occuperà di prendere in input l'oggetto terminativo della query precedente e, per le parole non presenti in Figura 4.5, effettuare una richiesta direttamente ai server di conceptNet.

La richiesta viene eseguita all'indirizzo:
'https://api.conceptnet.io/c/en/'+vett+'?offset=0&limit=600'

conceptNet infatti viene reso disponibile anche in formato JSON-LD API, un formato il cui scopo è di creare dati linkati in modo facile da capire e utilizzare. Il formato JSON-LD è un formato molto simile a JSON con alcuni meta dati in più.

L'URL con cui facciamo la richiesta a conceptNet si struttura nel seguente modo:

- vett: rappresenta la parola che si vuole analizzare;
- offset=0: rappresenta l'offset dell'informazione che si vuole visualizzare. In particolare partendo da 0 prenderemo l'oggetto JSON restituito da conceptNet a partire dall'inizio;
- limit=600: rappresenta la quantità di informazioni che si vuole considerare. Di default sarebbe pari a 20, ma si è aumentato il parametro per avere una maggior quantità di informazioni a cui fare riferimento e analizzare.

Una volta che conceptNet restituirà il JSON in risposta, dovremo dovranno essere fatte alcune considerazioni prima di procedere nell'analisi. ConceptNet

restituisce tutti gli archi in cui si presenta la parola da noi inserita durante la richiesta all'url specifico.

Inoltre verranno analizzati tutti gli archi in tutte le lingue presenti nei loro database. Questo porterà ad avere un JSON molto esteso, denso di informazioni. E' dunque necessario cercare di considerare solo ed esclusivamente le informazioni che sono effettivamente rilevanti.

Per tale motivo si utilizza un'apposita funzione che prende in input il JSON-LD restituito da conceptNet e lo analizza:

- cerchiamo di analizzare tutti gli 'edges' dell'oggetto, ovvero tutti i possibili collegamenti della parola con altre simili o derivate;
- ciascun edges è di interesse se:
  1. il campo 'label' dell'attributo 'relation' è definito come 'HasContext': è importante ricordare che tutte queste operazioni che stiamo facendo servono per capire se le parole che stiamo analizzando sono riferite a un contesto specifico. Motivo per cui sarà necessario che tale valore sia 'HasContext';
  2. il campo 'language' dell'attributo 'start' (parola di inizio del collegamento) deve essere 'en': Non si vogliono avere informazioni riguardo a parole in altre lingue. Si analizzano quindi solo le parole inglesi;
  3. il campo 'language' dell'attributo 'end' (parola di fine del collegamento) deve essere 'en': stessa motivazione del punto precedente.
- cerchiamo di capire dove risiede l'informazione che abbiamo trovato: in 'end' o in 'start'. Questo è dovuto al fatto che la parola di cui vogliamo avere informazioni non è obbligatoriamente all'inizio del nostro arco, può infatti anche trovarsi al fondo. Nel caso appena descritto l'informazione la troveremo in 'start'.

  In seguito si prosegue al salvataggio sul database delle informazioni ritenute rilevanti tra quelle appena ottenute. Per fare questo verrà eseguita una richiesta POST all'indirizzo '.../writingDB', passando come body i dati presi da conceptNet.

4.7.4 Fase di '/writingDB'

Lato Server verranno eseguiti multipli accessi al database per scrivere le informazioni ottenute. In particolare a ogni accesso verrà scritta l'informazione riguardo a una specifica parola.

La tabella in cui le informazioni vengono salvate sono mostrate in Figura 4.5.

Verrà eseguita una query in linguaggio SQL per eseguire l'operazione:

```
1 const sql = 'INSERT INTO WORDS(wordName, relationType, relationValue)
      VALUES(?, ?, ?)';
2
```

- wordName: rappresenta il nome o una parte del nome di una funzione di uno smart contract;
- relationType: rappresenta il tipo di relazione, in questo caso troveremo solo 'hasContext', perchè è l'unica relazione considerata;
- relationValue: rappresenta tutti i collegamenti/contesti di uno specifico 'wordName'. Dato che relationValue potrebbe non essere un valore univoco, perchè una parola può essere usata in differenti contesti, si è deciso di mettere tutto in un'unica colonna separando con delle virgole.

Se tutto è andato a buon fine si conclude questa fase con una struttura dati quasi completa per la creazione del grafico.

A questo punto avremo tantissimi contesti che si riferiscono al nostro contratto. Facendo alcune analisi, a meno che i contratti non siano molto brevi, è molto comune che per ogni contratto si abbiano di media una cinquantina di contesti di argomenti diversi.

Per stabilire una gerarchia viene utilizzata un'apposita funzione che, in base al numero di ripetizioni dello stesso contesto, ne farà aumentare l'importanza in percentuale.

Questo serve sia per fare una selezione dei contesti con maggiore presenza in uno smart contract sia per eliminare dalla struttura la ripetizione degli stessi contesti.

Nell'analisi dei contesti, verranno considerati, per ogni smart contract, solo i primi 4 per percentuale più alta di affinità.

Una volta eseguite queste operazioni si giunge all'effettiva creazione della struttura dati necessaria per la creazione del grafico.

4.8 Scatter Plot Matrix

Lo scatter Plot Matrix è una matrice di scatter plots in cui ogni scatter plot viene creato utilizzando il valore di differenti combinazioni di variabili.

In questo specifico caso si sono messi in evidenza tra loro i seguenti parametri:

- lunghezza - int: si intende il numero di righe utilizzato dallo smart contract;
- dimensione - int: si intende la dimensione, in termini di spazio occupato sull'hard disk , dallo smart contract;

- gas price - Number: si intende il gas price totale del contratto se venissero eseguite tutte le sue funzioni;
- numero di funzioni - int: si intende il numero di funzioni presenti nello smart contract;
- numCIL - int: si intende il numero di strutture di tipo contract, interface e library sommate insieme;

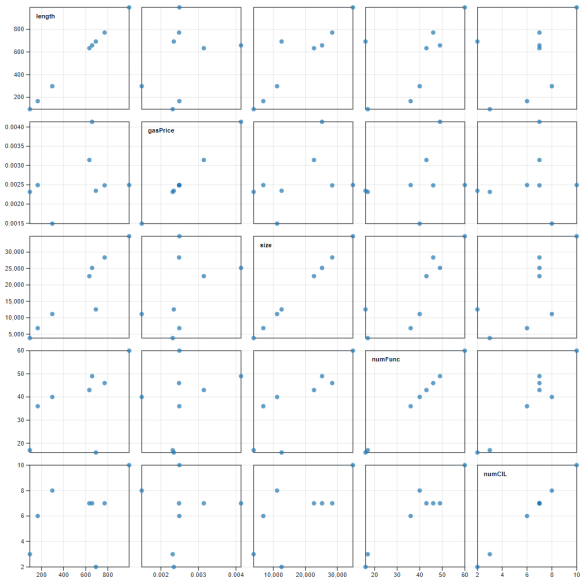

Figura 4.6: Scatter Plot Matrix

4.8.1 Elaborazione e creazione dati

Prima dell'effettiva creazione della struttura dati, necessaria per la creazione del grafico, devono essere unite le informazioni riguardo al gas delle funzioni e le informazioni sulla struttura dello smart contract che erano state prelevate come descritto nel Capitolo 4.5.2.

Per trovare il gas complessivo, relativo a un singolo contratto, si utilizzerà un'apposita funzione che, per ogni contratto, sommerà il gas delle singole funzioni da cui è composto.

Dopodiché il risultato verrà salvato nell'apposita variabile (gasPrice) presente nella struttura descritta nel Capitolo 4.5.2

La struttura dati effettiva per la creazione del grafico verrà invece gestita dalla funzione dataScatterPlot(data). Si tratta di una funzione molto minimalista e

semplice, in quanto la struttura passata in input possiede già tutte le informazioni necessarie.

In particolare, si occupa di ordinare e considerare solo le informazioni necessarie per la creazione del grafico.

4.9 Zoomable circle packing

Figura 4.7: Zoomable Circle Packing

Si tratta di un grafico in cui ogni file viene rappresentato da una o più bolle. Le bolle vengono inserite all'interno di altre bolle 'contenitori'. Quest'ultime rappresentano gli argomenti a cui, gli smart contract all'interno, presentano una percentuale di affinità. Ognuna di queste bolle presenterà, sotto forma di scritta, la descrizione dell'argomento che rappresenta.

Le bolle che rappresentano gli smart contract, infatti, hanno dimensioni diverse che variano in base a quanta affinità possiedono su un certo argomento. Più l'affinità è alta e maggiore sarà la grandezza della bolla, viceversa, più l'affinità è bassa e minore sarà la grandezza della bolla.

Per distinguere tra loro le bolle che si riferiscono a smart contract diversi, si è utilizzata una serie di colori differenti con la relativa legenda.

Il grafico presenta la possibilità di interagire dinamicamente con l'utente. In particolare, è possibile, cliccando su una bolla 'argomento' specifica, avere uno zoom di tutte le bolle (rappresentanti i differenti smart contract) presenti all'interno della bolla selezionata. In questa visualizzazione non avremo più la scritta dell'argomento all'interno della bolla, ma per ogni bolla contratto, troveremo il nome del contratto a cui si riferisce. Per la creazione del grafico si sono dovute reperire una serie di informazioni specifiche sul contratto tramite l'utilizzo di conceptNet.

Per questioni di ottimizzazione a livello di tempo e velocità si sono salvate queste informazioni su un database privato.

4.9.1 Elaborazione e creazione dati

Anche in questo caso si ha la necessità di creare una struttura ad albero ma di soli due livelli rispetto a quella creata nel capitolo 4.4.

Questo è dovuto al fatto che avremo solo due tipi di bolle:

- bolle che rappresentano i contesti: potranno solo contenere bolle che rappresentano 'smart contract';
- bolle che rappresentano 'smart contract': potranno trovarsi solo all'interno delle bolle che rappresentano un contesto e non potranno mai loro stesse contenere nulla.

La struttura che si creerà apparirà in questa forma:

- primo livello: avremo un oggetto con due attributi:
  1. name: 'flare';
  2. children: un vettore contenente tutti i contesti.
- secondo livello: si svilupperà a partire dal vettore children del livello precedente, qui saranno contenuti degli oggetti che rappresenteranno i contesti:
  1. name: nome del contesto;
  2. children: vettore che conterrà tutti gli smart contract che presentano una certa affinità con il contesto 'name'.
- terzo livello: si svilupperà a partire dal vettore children del livello precedente, qui saranno contenuti degli oggetti che rappresentano gli smart contract:
  1. name: nome dello smart contract;
  2. value: valore numerico in percentuale che rappresenta l'affinità dello smart contract rispetto al contesto (name del livello precedente).

4.10 Bubble Plot diagram

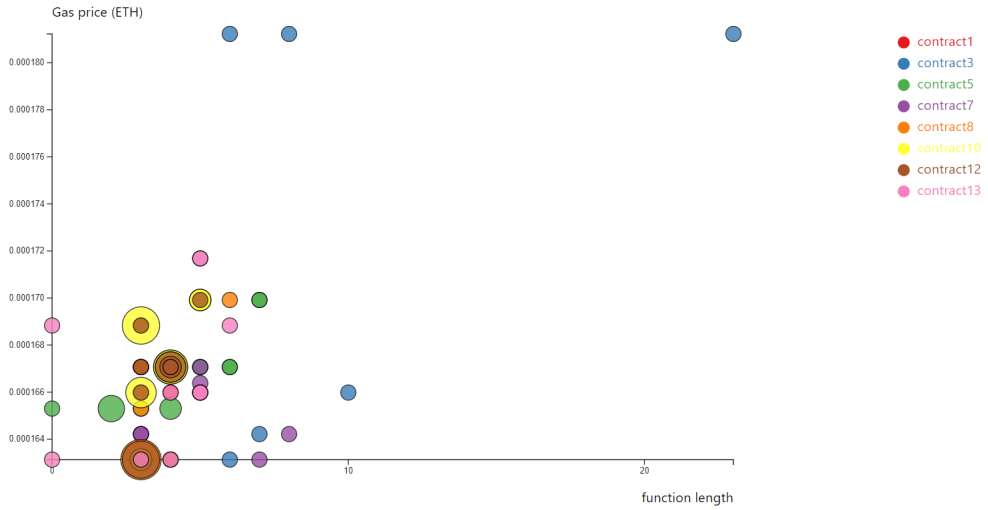

Figura 4.8: Bubble Plot Diagram

Si tratta di un grafico molto simile allo 'scatter plot' in termini di implementazione, in cui però, l'unica differenza è che abbiamo un'ulteriore variabile che si basa sulla grandezza dei punti.

Le bolle presenti all'interno del grafico rappresenteranno le funzioni di ogni smart contract, queste saranno caratterizzate dalle seguenti caratteristiche:

- Asse y - gas Price: rappresenta il gas per l'esecuzione sulla EVM della funzione;
- Asse x - lunghezza: rappresenta la lunghezza, in termini di linee di codice, della funzione;
- Colore - tipo di contratto: il colore caratterizza a quale smart contract la funzione fa riferimento;
- Grandezza - richiami: La grandezza della bolla si riferisce a quante volte la funzione viene richiamata da altre all'interno dello smart contract.

Questo ci permetterà quindi di avere una visione completa di tutti gli smart contract presi in input con un confronto mirato sulle variabili precedentemente citate.

E' presente una sorta di interazione con l'utente che permette un'analisi più concentrata dei particolari di interesse.

A lato del grafico, infatti, è presente una legenda in cui a ogni colore viene associato uno specifico smart contract. Passando sopa uno di questi col cursore

vi sarà la possibilità di visualizzare nel grafico soltanto le bolle (funzioni) che si riferiscono allo smart contract selezionato.

Questo può essere molto interessante dal momento che, di primo impatto, alcune bolle potrebbero trovarsi al di sotto di altre e non essere quindi visualizzabili.

4.10.1 Richiesta al Server (Server Side)

Per la creazione della nostra struttura dati e quindi successivamente per il grafico, avremo bisogno di ottenere alcune informazioni dal server.

Per questo rifaremo una richiesta POST al server a un indirizzo già utilizzato in passato per la creazione del grafico nel capitolo 4.3.1.

In particolare verrà eseguita una fetch all'indirizzo '.../upload' e nel body della richiesta troveremo, in questo caso, più di un file.

Per questo motivo, il codice lato Server è flessibile riguardo a questa duplice possibilità. Se in input è presente un solo file, verrà fatto ciò che è descritto nel capitolo 4.3.1, altrimenti se sono presenti più file in input, oltre a quello che veniva fatto per il singolo file, viene aggiunta l'informazione prelevata da Surya descritta nel capitolo 4.2.4.

Il tutto verrà ritornato al client in formato JSON.

4.10.2 Elaborazione e creazione dati

I valori necessari per la creaione del grafico sono quelli che abbiamo descritto nel capitolo 4.10.

In particolare:

- gas price: è già presente in quanto precedentemente calcolato per lo scatter plot matrix nel capitolo 4.6;
- function length: da calcolare;
- function call (numero di chiamate a una certa funzione): da calcolare;

La funzione si occupa quindi di calcolare questi parametri necessari proseguendo nel seguente modo:

- L'analisi verrà fatta confrontando ogni funzione di ogni contratto col testo dello smart contract. Questo risulta essere l'unico modo per poter prelevare le informazioni necessarie;
- Durante lo scorrimento vengono totalmente ignorati i commenti al codice dello smart contract. Non sono minimamente rilevanti, anzi, potrebbero essere causa di errori nella rilevazioni delle informazioni. Vi è un'apposita funzione che gestisce queste situazioni ed è la **goAhead**;

- Nel caso in cui nella riga del contratto che si sta analizzando vi è la presenza del nome di una funzione, significa che la funzione è richiamata una volta da altre strutture. La variabile relativa a quante volte la funzione viene richiamata si aggiorna e aumenta.

  Bisogna però prendere in considerazione il fatto di non considerare una riga in cui si ha la dichiarazione della stessa. Per cui si escluderà il caso in cui il nome della funzione è presenta sulla riga di dichiarazione della medesima.

- Nel caso in ci si trovasse nella riga di dichiarazione della funzione in esame, ne vengono calcolate le linee di codice in base a un algoritmo che prosegue riga per riga nel testo dello smart contract fino a quando non si incontra il carattere '}'.

  Come già precedentemente esposto si deve fare attenzione a non considerare le '}' di alcuni costrutti particolari.

Al termine di queste operazioni si calcolano tutte le informazioni necessarie alla creazione del grafico.

Capitolo 5

Valutazione

La parte di Valutazione del tool è stata effettuata facendo provare il software a dei possibili utilizzatori futuri cercando un riscontro finale che potesse far emergere problematiche o migliorie da apportare. Ritengo sia utile far presente che il 'target' di persone scelte per la valutazione del tool è molto giovane, nella quasi totalità sono studenti universitari o che da poco hanno terminato i loro studi. Questa fase di valutazione con gli utenti è stata suddivisa in 4 parti principali:

- Tutorial: fase studiata per permettere agli utenti di poter conoscere meglio il tool e il suo funzionamento generale
- Competenze: fase in cui gli utenti, rispondendo ad alcune specifiche domande, hanno espresso le loro competenze in campo informatico e sulla Blockchain.
- Uitilizzo tool: fase in cui sono stati presentati dei task da risolvere agli utenti.
- Giudizio finale: Al termine è stato chiesto agli utenti se potessero esprimere alcuni giudizi sul tool. In particolare, se ci fossero degli elementi particolarmente negativi che secondo loro avessero bisogno di essere migliorati o elementi che sarebbe stato meglio inserire all'interno del tool

5.1 Tutorial e competenze degli utenti

Durante questa fase si è cercato di insegnare e mettere a proprio agio il più possibile l'utente riguardo all'utilizzo del tool in questione. In un primo momento è stata data una spiegazione generale del campo di ricerca e analisi in questione, ovvero la Blockchain. Per quanto il tool possa essere la parte fondamentale, non avrebbe avuto senso porre un utente inesperto subito a contatto con esso senza prima una spiegazione che potesse porlo nel giusto contesto. E' risultata fondamentale per molti questa fase, in quanto non tutti erano esperti sulla materia e altri non ne

avevano neanche mai sentito parlare come si può notare dalla tabella (i valori possono andare da un minimo di 1, bassa competenza, a un massimo di 5, alta competenza). Dalla tabella risulta che tutti gli utenti abbiano una certa esperienza

Id	Età	Esperienza nella programmazione	Tempo che passi a programmare	Esperienza sulla Blockchain	Tipo di Questionario
1	24	3	2	2	A
2	25	2	1	1	B
3	24	3	2	2	A
4	24	1	1	1	A
5	25	5	5	2	B
6	23	4	2	1	B
7	25	5	5	1	B
8	26	5	5	3	A
9	26	5	5	3	A
10	24	5	5	4	B
11	22	3	3	1	B
12	23	4	3	2	A
13	25	5	5	3	A
14	25	5	5	3	B
Avg	24.4	3.9	3.5	2.1	NaN

Tabella 5.1: Competenze degli utenti in valutazione del tool

nel campo dell'informatica, ma in pochi abbiano conoscenze sulla Blockchain, sintomo che si tratta di una tecnologia ancora molto nuova e poco diffusa.

In un secondo momento gli utenti sono stati introdotti al tool vero e proprio e al suo funzionamento. In particolare, sono stati preparati una serie di appositi smart contract [45] che potessero fungere da input per il tool in modo da mostrare i grafici all'utente con la relativa spiegazione. Una specie di tutorial si potrebbe dire. Non si è entrati troppo nello specifico nell'analisi del grafico per non influenzare la fase in cui avrebbero dovuto rispondere a task tramite l'utilizzo del software.

5.2 Utilizzo del tool

Questa fase, che risulta essere anche la principale, è stata organizzata e suddivisa in ulteriori due parti:

- La prima parte si basa sulla presentazione di undici task a cui l'utente deve rispondere attraverso l'utilizzo del tool
- nella seconda l'utente dovrà rispondere ad altri undici task senza l'utilizzo del tool (in particolare attraverso Remix IDE o con qualunque altro strumento che l'utente riterrà più utile) e con smart contract differenti rispetto ai precedenti.

Vengono utilizzati due gruppi di contratti diversi [46] [47] per diversificare le risposte tra il primo gruppo (con tool) e il secondo gruppo di task (senza tool). A ogni utente, prima dell'inizio del test, è stato dato un apposito file, tra i due disponibili [48] [49], per segnare le risposte e il tempo impiegato a portare a termine ogni task. L'unica differenza consiste nel fatto che gli utenti a cui è stato assegnato il Questionario A dovranno utilizzare gli Smart contract presenti in contractGruppo1 per rispondere ai task attraverso l'utilizzo del tool e dovranno utilizzare gli smart contract presenti in contractGruppo2 per rispondere ai task senza l'utilizzo del tool. Gli utenti a cui è stato assegnato il Questionario B faranno al contrario rispetto agli altri.

Nel momento in cui si procede alla risoluzione di un task viene fatto partire un cronometro che misura il tempo necessario per la sua risoluzione. E' importante specificare che il limite massimo che viene imposto per poter risolvere un task è di 5 minuti. Se il task non viene risolto in quel limite di tempo è considerato non portato a termine. Si è deciso di imporre questo limite in quanto nel caso della risoluzione dei task senza l'utilizzo del tool i tempi si dilatavano eccessivamente.

5.2.1 Task per singolo smart contract in input

Tutti i task sono stati pensati per far interagire l'utente con tutte le varie tipologie di grafici presenti. Dal momento che i task presenti nei due questionari sono pressoché identici verrà analizzata solo la procedura di risoluzione dei task del Questionario A.

Analizzando singolarmente il procedimento di risoluzione dei task avremo:

- Per il task 1 veniva chiesto di trovare la funzione che richiamava più volte altre funzioni all'interno di **contract1.sol**: Gli utenti, per risolvere correttamente il task, hanno dovuto ispezionare il *Chord diagram functions*. In questo caso specifico possiamo notare che la funzione **transferFrom(...)** è quella da cui troviamo più frecce, quattro nello specifico, in uscita verso altre funzioni.
- Per il task 2 veniva chiesto di trovare la funzione che veniva richiamata più volte da altre funzioni all'interno di **contract1.sol**: Gli utenti hanno dovuto agire esattamente come per il task 1, ma osservando la funzione verso cui confluivano il maggior numero di frecce. In questo caso specifico è molto intuitivo notare che per la **__msgSender()** abbiamo ben cinque frecce in entrata.
- Per il task 3 veniva chiesto di trovare quante funzioni venivano richiamate dalla funzione **__transfer(...)**. Gli utenti, utilizzare sia il *Chord diagram functions* sia il *Zoomable sunburst diagram*. Era necessario trovare la funzione e contare il numero di frecce uscenti da essa, ovvero il numero di funzioni che vengono chiamate, in questo caso esattamente due

- Per il task 3 veniva chiesto di trovare quante funzioni venivano richiamate dalla funzione **_approve(...)**. Gli utenti, utilizzare sia il *Chord diagram functions* sia il *Zoomable sunburst diagram*. Era necessario trovare la funzione e contare il numero di frecce entranti in essa, ovvero il numero di funzioni che la richiamano, in questo caso esattamente quattro

Per quanto sia semplice rispondere a questi task attraverso l'utilizzo del tool, non è affatto scontato rispondere senza. Il fatto è che su Remix IDE l'unico modo per rispondere a queste domande è di cercare la risposta manualmente, andando a ispezionare direttamente il codice. La soluzione può essere fattibile per un contratto molto breve, ma all'aumentare della complessità e lunghezza del contratto aumenta esponenzialmente il tempo per indicare la risposta corretta. Come possiamo osservare nella Tabella 5.2 la differenza di tempo nei primi due task, con e senza l'utilizzo del tool, è estremamente elevata.

Inoltre si può notare che la difficoltà nel rispondere a questi task non dipende minimamente dall'esperienza e capacità dell'utente. Possiamo infatti notare che senza l'utilizzo del tool è praticamente impossibile rispondere in tempi ragionevoli. E' giusto e opportuno precisare che i task sono stati pensati e modellati appositamente in funzione delle informazioni che il tool è in grado di fornire. Non bisogna quindi stupirsi se il tempo impiegato è decisamente inferiore nel risolvere il task con il tool piuttosto che con altro.

	con tool				senza tool			
Id	1.Trova la funzione che richiama più volte altre funzioni nel contratto 'x'	2.Trova la funzione che viene richiamata più volte nel contratto 'x'	3.Quante funzioni vengono chiamate dalla funzione 'x'	4.Quante volte viene richiamata la funzione 'x'	5.Trova la funzione che richiama più volte altre funzioni nel contratto 'x'	6.Trova la funzione che viene richiamata più volte nel contratto 'x'	7.Quante funzioni vengono chiamate dalla funzione 'x'	8.Quante volte viene richiamata la funzione 'x'
1	00:47	01:03	00:26	00:14	> 05:00	> 05:00	00:33	00:47
2	00:53	00:59	00:25	00:16	> 05:00	> 05:00	00:31	00:42
3	01:08	00:41	00:25	00:23	> 05:00	> 05:00	00:26	00:39
4	00:43	00:34	00:22	00:14	> 05:00	> 05:00	00:39	00:51
5	00:49	00:42	00:18	00:09	> 05:00	> 05:00	00:23	00:41
6	00:29	00:34	00:17	00:15	> 05:00	> 05:00	00:26	00:42
7	00:28	00:37	00:19	00:07	> 05:00	> 05:00	00:21	00:31
8	00:19	00:28	00:14	00:08	> 05:00	> 05:00	00:20	00:30
9	00:20	00:36	00:20	00:12	> 05:00	> 05:00	00:25	00:36
10	00:21	00:15	00:12	00:12	04:37	> 05:00	00:35	00:43
11	00:24	00:45	00:10	00:21	> 05:00	> 05:00	00:30	00:39
12	00:31	00:39	00:11	00:08	> 05:00	> 05:00	00:31	00:38
13	00:34	00:29	00:18	00:08	> 05:00	> 05:00	00:19	00:35
14	00:25	00:28	00:14	00:10	> 05:00	> 05:00	00:17	00:30
Avg	00:35	00:38	00:18	00:13	> 05:00	> 05:00	00:27	00:39

Tabella 5.2: Tempo impiegato nella risoluzione dei task di un singolo smart contract

Per completezza è bene dare un valore alle 'x' all'interno della tabella:

- Per le domande (1), (2), (5) e (6):
  - Coloro a cui è stato assegnato il Questionario A: **contract1.sol**
  - Coloro a cui è stato assegnato il Questionario B: **contract21.sol**
- Per la domanda (3):
  - Coloro a cui è stato assegnato il Questionario A: **__transfer()**
  - Coloro a cui è stato assegnato il Questionario B: **increaseAllowance()**
- Per la domanda (4) e (8):
  - Sia per il Questionario A sia per il Questionario B: **__approve()**
- Per le domande (7): Abbiamo la stessa cosa presente nella domanda (3) ma invertendo i due Questionari

5.2.2 Task per multipli smart contract in input

Per quel che riguarda i task con più smart contract in input si sono ottenuti quasi gli stessi risultati appena elencati. In particolare, si tratta in questo caso di task che mirano a identificare delle proprietà univoche di alcuni smart contract rispetto ad altri, come ad esempio il contesto del loro utilizzo, differenze nel gas totale e delle singole funzioni. Per visualizzare al meglio i risultati si è deciso di dividere i task su due tabelle differenti dove nella Tabella 5.3 si analizzano i risultati ottenuti mediante l'uso del tool e nella Tabella 5.4 si analizzano i risultati ottenuti senza l'utilizzo del tool.

Attraverso l'utilizzo del tool la risoluzione di ciascun task necessitava di un tempo relativamente breve. In particolare:

- Per il task numero 1 era necessario osservare il grafico in Figura 4.6 e cercare il corretto quadrante in cui queste informazioni venivano mostrate. Dal momento che veniva chiesto il più semplice e il più complesso si doveva cercare di soppesare nella maniera appropriata il numero di funzioni, gas necessario e lunghezza dello smart contract. Facendo riferimento al Questionario A e osservando il grafico è possibile notare che il **contract3.sol** è sia il più corto e il meno oneroso in termini di costo di gas. E' molto intuitivo quindi additarlo come il meno complesso tra tutti.
- Per il task numero 4 era necessario osservare il grafico in Figura 4.7. Il grafico permette infatti di riunire i contratti all'interno di specifiche bolle che rappresentano un argomento con cui i contratti presentano una certa affinità.

Per risolvere il task è necessario individuare un contratto che si trovi da solo in specifiche bolle. E' importante specificare che per il seguente task non è corretta una sola risposta dal momento che è possibile che più contratti siano presenti in bolle isolate. Facendo riferimento al Questionario A e osservando il grafico è possibile notare che il **contract12.sol** ha affinità nell'ambito 'commerce', mentre il **contract8.sol** ha affinità nell'ambito 'finance'. Risultano essere gli unici due contratti ad avere affinità con questi contesti e per questo sono anche la risposta corretta del task.

- Per i task numero 2, 3, 5, 6, 7 era necessario analizzare il grafico in Figura 4.8. In particolare, per le domande 2 e 3 era necessario contare il numero di bolle presenti per ciascun contratto. In questo modo si risaliva facilmente al contratto con più e meno funzioni a pagamento. Per le domande 6 e 7 invece bastava controllare la bolla più in alto a destra e più in basso a sinistra per trovare rispettivamente la funzione che consuma più gas e quella che ne consuma meno.

con tool							
Id	1.Tra tutti gli smart contract presenti nel gruppo 'x', identificare il più semplice e il più complesso	2.Smart contract che ha il maggior numero di funzioni che consumano gas	3.Smart contract che ha il minor numero di funzioni che consumano gas	4.Identificare il contratto che si differenzia dagli altri in base al suo scopo/utilizzo	5.Nomi degli smart contract che possiedono la funzione più lunga e corta che è a pagamento	6.Nome della funzione che consuma più gas	7.Nome della funzione che consuma meno gas
1	02:42	00:25	00:29	01:42	00:42	00:20	00:10
2	02:17	00:24	00:35	01:25	01:13	00:19	00:07
3	02:19	00:32	00:31	00:52	01:01	00:23	00:05
4	03:23	00:27	00:20	01:48	00:52	00:22	00:08
5	01:37	00:43	00:19	01:04	00:37	00:13	00:06
6	01:19	00:22	00:22	01:28	00:39	00:15	00:07
7	02:13	00:54	00:38	00:49	00:51	00:12	00:10
8	00:58	00:42	00:29	00:43	00:23	00:10	00:12
9	01:39	01:03	00:37	00:55	00:26	00:16	00:13
10	00:52	00:38	00:25	01:23	00:17	00:14	00:12
11	03:28	00:41	00:28	01:17	00:39	00:14	00:07
12	01:42	00:44	00:36	01:04	00:28	00:10	00:06
13	01:54	01:05	00:41	01:13	00:25	00:12	00:06
14	01:31	00:40	00:23	00:48	00:28	00:15	00:08
Avg	01:59	00:40	00:29	01:11	00:39	00:15	00:08

Tabella 5.3: Tempo impiegato nella risoluzione dei task di molteplici smart contract con l'utilizzo del tool

Per rispondere alle domande senza l'utilizzo del tool l'utente poteva arbitrariamente utilizzare qualunque strumento ritenesse utile. La maggior parte però ha utilizzato Remix IDE [50].

Il fatto di utilizzare un IDE ha presentato molte problematiche per gli utenti. Indipendentemente dal fatto che fossero esperti del settore o meno hanno impiegato un tempo decisamente più elevato per risolvere i task. Questo perché l'IDE non è un tool studiato per rispondere a queste domande, ma più per supportare lo sviluppatore nella scrittura del codice più che nell'analisi dello smart contract. Questo ha portato ad una dilatazione importante del tempo necessario e alcuni utenti hanno trovato molta difficoltà per approcciarsi nel modo corretto al tool per risolvere alcuni task specifici.

Indipendentemente dall'IDE però è bene precisare che non c'è un tool o software che possa essere utilizzato per rispondere in maniera diretta alla domanda. Questo rende molto evidente quanto il tool realizzato permetta di fare analisi che in altro modo non sarebbero possibili in tempi accettabili

senza tool							
Id	1.Tra tutti gli smart contract presenti nel gruppo 'x', identificare il più semplice e il più complesso (senza tool)	2.Smart contract che ha il maggior numero di funzioni che consumano gas (senza tool)	3.Smart contract che ha il minor numero di funzioni che consumano gas (senza tool)	4.Identificare il contratto che si differenzia dagli altri in base al suo scopo/utilizzo (senza tool)	5.Nomi degli smart contract che possiedono la funzione più lunga e corta che è a pagamento (senza tool)	6.Nome della funzione che consuma più gas (senza tool)	7.Nome della funzione che consuma meno gas (senza tool)
1	> 05:00	> 05:00	04:34	> 05:00	> 05:00	> 05:00	> 05:00
2	> 05:00	> 05:00	04:17	> 05:00	> 05:00	> 05:00	> 05:00
3	> 05:00	> 05:00	04:15	> 05:00	> 05:00	> 05:00	> 05:00
4	> 05:00	> 05:00	03:46	> 05:00	> 05:00	> 05:00	> 05:00
5	> 05:00	03:32	02:17	> 05:00	> 05:00	> 05:00	> 05:00
6	> 05:00	04:20	02:39	> 05:00	> 05:00	> 05:00	> 05:00
7	> 05:00	03:20	02:41	> 05:00	> 05:00	> 05:00	> 05:00
8	> 05:00	02:49	02:40	> 05:00	> 05:00	> 05:00	> 05:00
9	> 05:00	02:34	02:00	> 05:00	> 05:00	> 05:00	> 05:00
10	> 05:00	02:52	01:45	> 05:00	> 05:00	> 05:00	> 05:00
11	> 05:00	04:40	03:22	> 05:00	> 05:00	> 05:00	> 05:00
12	> 05:00	04:23	03:10	> 05:00	> 05:00	> 05:00	> 05:00
13	> 05:00	02:23	01:49	> 05:00	> 05:00	> 05:00	> 05:00
14	> 05:00	02:27	01:55	> 05:00	> 05:00	> 05:00	> 05:00
Avg	> 05:00	$\approx 04:05$	$\approx 02:54$	> 05:00	> 05:00	> 05:00	> 05:00

Tabella 5.4: Tempo impiegato nella risoluzione dei task di molteplici smart contract senza l'utilizzo del tool

5.3 Giudizio finale

Al termine dei task sono state poste delle domande (con risposta sempre compresa tra 1 e 5) per giudicare il tool e su come gli utenti si fossero trovati. Inoltre è stata data la possibilità a ciascuno di loro di proporre, in apposite caselle di testo, eventuali proposte per migliorare il tool o far emergere delle mancanze.

Id	1.Quanto il tool riesce a mostrare le informazioni in maniera completa senza dover necessariamente consultare il codice?	2.Quanto è utile il tool nel portare a termine i task richiesti?	3.Quanto sei concorde con questa frase: Penso che lo strumento mostri tutte le informazioni utili e non abbia mancanze	4.Quanto pensi che i grafici utilizzati siano appropriati per rappresentare le informazioni?	5.Quanto pensi che il tool sia utile?	6.Quanto pensi che il tool sia utile per i non esperti?	7.Se pensi che il tool sia utile, lo useresti: da solo o affiancato ad altri strumenti come ad esempio IDE?
1	5	4	4	4	5	3	3
2	5	3	3	2	3	2	3
3	5	4	4	4	4	4	3
4	5	5	4	5	4	2	1
5	5	5	4	5	3	4	2
6	5	5	5	2	4	4	2
7	5	5	4	3	3	5	3
8	4	4	3	3	4	5	4
9	5	4	2	4	4	5	4
10	4	5	3	4	5	4	3
11	5	2	2	3	4	2	4
12	5	5	4	5	4	3	2
13	5	3	3	3	3	5	3
14	5	4	4	4	5	4	2
Avg	4.86	4.14	3.5	3.64	3.92	3.71	2.78

Tabella 5.5: Valutazione del tool da parte degli utenti

Possiamo notare che i risultati medi su quanto il tool riesca ad essere utile senza l'ausilio del codice dello smart contract è molto alto. Si avvicina infatti a 5 che rappresenta il valore massimo. Leggermente più basso il punteggio sulla tipologia di grafici utilizzati per mostrare all'utente le informazioni. Il punteggio leggermente sotto la media di alcuni utenti è stato causato dalla dimensione del grafico. Il loro problema non era rivolto più di tanto alla tipologia quanto più al fatto che il grafico superava leggermente la dimensione della pagina. Questa problematica li ha obbligati a non riuscire mai a cogliere il grafico nella sua interezza,

causando aumento di tempo per la risoluzione dei task. Per quel che riguarda la valutazione sull'utilità del tool per i non esperti si nota un punteggio medio altamente confortante.

Capitolo 6

Conclusioni

6.1 Aspetti generali

Questo studio utilizza diversi metodi per condurre un'analisi degli smart contract basata sull'analisi visiva. Lo studio è stato motivato da quattro domande:

- Come sarebbe possibile, a monte di differenti implementazioni di uno smart contract con il medesimo scopo, decidere quale caricare sulla Blockchain senza valutarne il codice, ma attraverso il confronto di loro specifiche caratteristiche?
- Come sarebbe possibile comprendere il campo di utilizzo di uno smart contract senza dover obbligatoriamente analizzare il codice, ma utilizzando una visualizzazione grafica?
- Come si potrebbe permettere a chiunque, comprese persone che non hanno esperienza riguardo a Solidity e la Blockchain, di analizzare uno smart contract valutando alcune sue intrinseche ed estrinseche (legate allo stato della Blockchain in un certo momento) caratteristiche?
- Il framework Javascript, in quanto potente linguaggio di programmazione, può contribuire allo sviluppo del sistema in considerazione anche grazie all'utilizzo di D3.js, ConceptNet e Web3.js?

Il primo passo per risolvere la decisione su quale smart contract può essere caricato sulla Blockchain è la valutazione del suo Gas Price strettamente correlata, ma non sempre, alla sua dimensione, lunghezza e complessità generale. Si è consci del fatto che questo elemento di valutazione presupponga che gli smart contract siano già stati valutati positivamente in fase di test. Sviluppare uno smart contract sicuro è l'aspetto più importante che dev'essere fatto a monte dell'utilizzo del tool. Per rispondere al secondo problema viene utilizzato un dizionario semantico

'open-source' che si occupa di definire il significato di una parola in tutte le sue forme, anche i contesti in cui questa è maggiormente utilizzata. Il tool, per ogni smart contract, preleva un insieme di parole all'interno del codice (nomi delle funzioni) ed effettua per ciascuna una specifica chiamata alle API di ConceptNet. Quest'ultimo presenterà in risposta un oggetto contenente, tra le tante informazioni, alcune riguardanti il contesto in cui la parola in input viene utilizzata. In una fase successiva queste informazioni verranno analizzate e visualizzate tramite un grafico. Il terzo problema viene risolto mediante la scorrelazione tra il codice e le informazioni che il tool propone. Questo è possibile grazie all'utilizzo di differenti framework che permettono di analizzare le informazioni presenti nel codice dello smart contract (web3.js e Surya) e renderle visibili all'utente tramite grafici (D3.js) senza l'ausilio del codice (che il tool mette comunque a disposizione dell'utente). Infine questa ricerca ha confermato la capacità del framework JavaScript, assieme a Node.js, D3.js e web3.js, di fornire un sistema User-Friendly e dinamico che consente a tutti i ricercatori Blockchain di accedere e analizzare lo strumento con diversi livelli di conoscenza.

Questa ricerca è in grado di mostrare i seguenti risultati:

- Lo strumento web sviluppato fornisce un'interfaccia utente molto 'User friendly'. Infatti, in base ai dati raccolti nella valutazione del tool in Tabella 5.5, i risultati riguardanti la tipologia di grafici scelti per mostrare le informazioni sono confortanti e ampiamente approvati da parte degli utenti.

- Il tool ha permesso di analizzare lo smart contract da un punto di vista completamente nuovo e innovativo, scorrelandolo sotto certi aspetti, dall'ispezione del codice (benché se ne offra la visualizzazione).

- Consente di confrontare più smart contract tra loro in base ad alcune proprietà legate allo smart contract (dimensione, numero di funzioni, lunghezza, ambito di utilizzo dello smart contract) e altre in parte correlate ad esso (Gas price):

6.2 Lavori futuri

Sulla base dei risultati di questo studio, si prevede che in futuro un numero significativo di ricercatori di vari campi sarà interessato a conoscere la tecnologia Blockchain e il suo funzionamento. Saranno dunque necessari strumenti che permettano ai neofiti di avere un approccio che possa essere scorrelato dal codice ma che permetta loro di comprendere la sostanza racchiusa all'interno di uno smart contract. Il tool potrà essere un valido mezzi non solo per questa categoria di persone, ma anche per tutti coloro che vorranno avere un confronto rapido tra numerosi smart contract senza dover per forza analizzarli tutti singolarmente.

Ci sono comunque alcune ottimizzazioni che potrebbero essere fatte su questa prima versione del tool per un suo miglioramento. In particolare, si dovrebbero studiare delle tecniche che permettano di ridurre la latenza nei tempi di comunicazione tra il front-end e il back-end quando il numero di contratti da analizzare cresce in misura elevata. L'utilizzo di ConceptNet è altamente complesso. Nel tool sono stati implementati algoritmi di analisi specifici che protessero gestire il JSON-LD ottenuto in risposta, ma si tratta di un oggetto complesso e con dimensione non trascurabile. Oltre al fatto che alcuni risultati che vengono restituiti non sono rilevanti e andrebbero scartati. Sarebbe necessaria quindi, da questo punto di vista, un'ulteriore analisi nei meccanismi con cui l'informazione viene analizzata.

Appendice A

Questionari per gli utenti

Questionario A

1.Informazioni generali:

Età: ____

Id: ____

Gruppo test: __

Tempo tutorial: _____

Livello di istruzione più alto: _____

Occupazione attuale: _____

2.Conoscenze informatiche:

Inerisci un valore compreso tra 1 e 5 per rispondere alle seguenti domande:

Quanto ti senti confidente nella programmazione in generale?	
Quanto tempo spendi nella programmazione (qualisasi linguaggio) durante la settimana?	
Quanta esperienza possiedi riguardo alle tecnologie Blockchain, smart contract e Solidity?	

3.Valutazione sui task:

Ogni cella verrà compilata con il tempo impiegato a risolvere il task (con l'utilizzo del tool):

	Tempo utilizzando il tool	Risposta
Trovare la funzione che viene richiamata più volte nel file "contract1.sol"		
Trovare la funzione che richiama più volte altre funzioni nel file "contract1.sol"		
Quante funzioni vengono chiamate dalla funzione "_transfer()"?		

Quante volte viene richiamata la funzione "_approve()"? E da quali funzioni?		
Tra tutti gli smart contract presenti nel gruppo 1, identificare il più semplice e il più complesso. (Valuta in base a lunghezza, gas e numero di funzioni)		
Identificare gli smart contract con gas più alto e più basso.		
Definire lo smart contract che ha il maggior numero di funzioni che consumano gas.		
Definire lo smart contract che ha il minor numero di funzioni che consumano gas.		
Identificare il contratto che si differenzia dagli altri in base al suo scopo/utilizzo.		
Identificare i nomi degli smart contract che possiedono la funzione più lunga che è a pagamento e la funzione più corta che è a pagamento.		
Indicare il nome della funzione che consuma più gas. A quale smart contract appartiene?		
Indicare il nome della funzione che consuma meno gas. A quale smart contract appartiene?		

Ogni cella verrà compilata con il tempo impiegato a risolvere il task (senza l'utilizzo del tool):

	Tempo impiegato senza l'utilizzo del tool	Risposta
Trovare la funzione che viene richiamata più volte nel file "contract21.sol"		
Trovare la funzione che richiama più volte altre funzioni nel file "contract21.sol"		
Quante funzioni vengono chiamate dalla funzione "increaseAllowance()"?		
Quante volte viene richiamata la funzione "_approve()"?		

Tra tutti gli smart contract presenti nel gruppo 2, identificare il più semplice e il più complesso. (Valuta in base a lunghezza, gas e numero di funzioni)		
Identificare gli smart contract con gas più alto e più basso.		
Definire lo smart contract che ha il maggior numero di funzioni che consumano gas.		
Definire lo smart contract che ha il minor numero di funzioni che consumano gas.		
Identificare il contratto che si differenzia dagli altri in base al suo scopo/utilizzo.		
Identificare i nomi degli smart contract che possiedono la funzione più lunga che è a pagamento e la funzione più corta che è a pagamento.		
Indicare il nome della funzione che consuma più gas. A quale smart contract appartiene?		
Indicare il nome della funzione che consuma meno gas. A quale smart contract appartiene?		

4.Valutazione tool:

Inserisci un valore compreso tra 1 e 5 per rispondere alle seguenti domande:

Quanto il tool riesce a mostrare le informazioni in maniera completa senza dover necessariamente consultare il codice?	
Quanto è utile il tool nel portare a termine i task richiesti?	
Esprimere un giudizio riguardo a quanto sei concorde con la seguente frase: Penso che lo strumento mostri tutte le informazioni utili e non abbia delle mancanze rilevanti.	
Quanto pensi che i grafici utilizzati siano appropriati per rappresentare le informazioni?	

Quanto pensi che il tool sia utile?	
Quanto pensi che il tool sia utile per i non esperti?	
Se pensi che il tool sia utile, lo useresti: da solo o affiancato ad altri strumenti come ad esempio IDE?	

5.Proposte

Se hai delle proposte su miglioramenti o modifiche da apportare al tool compila nello spazio sottostante.

Se hai trovato delle difficoltà nell'utilizzo del tool compila nello spazio sottostante

Questionario B

1.Informazioni generali:

Età: _____

Id: _____

Gruppo test: __

Livello di istruzione più alto: _____

Occupazione attuale: _____

2.Conoscenze informatiche:

Inerisci un valore compreso tra 1 e 5 per rispondere alle seguenti domande:

Quanto ti senti confidente nella programmazione in generale?	
Quanto tempo spendi nella programmazione (qualisasi linguaggio) durante la settimana?	
Quanta esperienza possiedi riguardo alle tecnologie Blockchain, smart contract e Solidity?	

3.Valutazione sui task:

Ogni cella verrà compilata con il tempo impiegato a risolvere il task (con l'utilizzo del tool):

	Tempo utilizzando il tool	Risposta
Trovare la funzione che viene richiamata più volte nel file "contract21.sol"		
Trovare la funzione che viene richiamata meno volte nel file "contract21.sol"		
Quante funzioni vengono chiamate dalla funzione "increaseAllowance()"?		

Quante volte viene richiamata la funzione "_approve()"?		
Tra tutti gli smart contract presenti nel gruppo 2, identificare il più semplice e il più complesso. (Valuta in base a lunghezza, gas e numero di funzioni)		
Identificare gli smart contract con gas più alto e più basso.		
Definire lo smart contract che ha il maggior numero di funzioni che consumano gas.		
Definire lo smart contract che ha il minor numero di funzioni che consumano gas.		
Identificare il contratto che si differenzia dagli altri in base al suo scopo/utilizzo.		
Identificare i nomi degli smart contract che possiedono la funzione più lunga che è a pagamento e la funzione più corta che è a pagamento.		

Ogni cella verrà compilata con il tempo impiegato a risolvere il task (senza l'utilizzo del tool):

	Tempo impiegato senza l'utilizzo del tool	Risposta
Trovare la funzione che viene richiamata più volte nel file "contract1.sol"		
Trovare la funzione che viene richiamata meno volte nel file "contract1.sol"		
Quante funzioni vengono chiamate dalla funzione "_transfer()"?		
Quante volte viene richiamata la funzione "_approve()"?		
Tra tutti gli smart contract presenti nel gruppo 1, identificare il più semplice e il più complesso. (Valuta in base a lunghezza, gas e numero di funzioni)		
Identificare gli smart contract con gas più alto e più basso.		

Definire lo smart contract che ha il maggior numero di funzioni che consumano gas.		
Definire lo smart contract che ha il minor numero di funzioni che consumano gas.		
Identificare il contratto che si differenzia dagli altri in base al suo scopo/utilizzo.		
Identificare i nomi degli smart contract che possiedono la funzione più lunga che è a pagamento e la funzione più corta che è a pagamento.		

4.Valutazione tool:

Inserisci un valore compreso tra 1 e 5 per rispondere alle seguenti domande:

Quanto il tool riesce a mostrare le informazioni in maniera completa senza dover necessariamente consultare il codice?	
Quanto è utile il tool nel portare a termine i task richiesti?	
Esprimere un giudizio riguardo a quanto sei concorde con la seguente frase: Penso che lo strumento mostri tutte le informazioni utili e non abbia delle mancanze rilevanti.	
Quanto pensi che i grafici utilizzati siano appropriati per rappresentare le informazioni?	
Quanto pensi che il tool sia utile?	
Quanto pensi che il tool sia utile per i non esperti?	
Se pensi che il tool sia utile, lo useresti: da solo o affiancato ad altri strumenti come ad esempio IDE?	

5.Proposte

Se hai delle proposte su miglioramenti o modifiche da apportare al tool compila nello spazio sottostante.

Se hai trovato delle difficoltà nell'utilizzo del tool compila nello spazio sottostante

Bibliografia

[1] Xiaoying Ma, Meiying Wei, Xiaoli Li e Xiaofei Zhang. «Analysis of Blockchain Technology and its Application in the Field of Radio Monitoring». In: *2021 International Conference on Computer, Blockchain and Financial Development (CBFD)*. 2021, pp. 450–453. DOI: 10.1109/CBFD52659.2021.00097 (cit. a p. 1).

[2] Yun Cheng. «Music Information Retrieval Technology: Fusion of Music, Artificial Intelligence and Blockchain». In: *2020 3rd International Conference on Smart BlockChain (SmartBlock)*. 2020, pp. 143–146. DOI: 10.1109/SmartBlock52591.2020.00033 (cit. a p. 1).

[3] «IEEE Approved Draft Standard for the Use of Blockchain in Supply Chain Finance». In: *IEEE P2418.7/D3.0, March 2021* (2021), pp. 1–24 (cit. a p. 1).

[4] Nikitha Abigail Muday e Geetanjali Ramesh Chandra. «Blockchain in the Legal Profession: A Boon or a Bane?» In: *2020 8th International Conference on Reliability, Infocom Technologies and Optimization (Trends and Future Directions) (ICRITO)*. 2020, pp. 717–722. DOI: 10.1109/ICRITO48877.2020.9198000 (cit. a p. 1).

[5] Chao Liu, Han Liu, Zhao Cao, Zhong Chen, Bangdao Chen e Bill Roscoe. «ReGuard: Finding Reentrancy Bugs in Smart Contracts». In: *2018 IEEE/ACM 40th International Conference on Software Engineering: Companion (ICSE-Companion)*. 2018, pp. 65–68 (cit. a p. 2).

[6] Jiaming Ye, Mingliang Ma, Yun Lin, Yulei Sui e Yinxing Xue. «Clairvoyance: Cross-contract Static Analysis for Detecting Practical Reentrancy Vulnerabilities in Smart Contracts». In: *2020 IEEE/ACM 42nd International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)*. 2020, pp. 274–275 (cit. a p. 2).

[7] Sarwar Sayeed, Hector Marco-Gisbert e Tom Caira. «Smart Contract: Attacks and Protections». In: *IEEE Access* 8 (2020), pp. 24416–24427. DOI: 10.1109/ACCESS.2020.2970495 (cit. a p. 2).

[8] Santeri Paavolainen, Tommi Elo e Pekka Nikander. «Risks from Spam Attacks on Blockchains for Intemet-of-Things Devices». In: *2018 IEEE 9th Annual Information Technology, Electronics and Mobile Communication Conference (IEMCON)*. 2018, pp. 314–320. DOI: 10.1109/IEMCON.2018.8614837 (cit. a p. 2).

[9] Pradnya Patil e M. Sangeetha. «Blockchain based Double Spending Prevention for Invoice Financing». In: *2021 Sixth International Conference on Wireless Communications, Signal Processing and Networking (WiSPNET)*. 2021, pp. 41–43. DOI: 10.1109/WiSPNET51692.2021.9419394 (cit. a p. 3).

[10] Mubashar Iqbal e Raimundas Matulevičius. «Exploring Sybil and Double-Spending Risks in Blockchain Systems». In: *IEEE Access* 9 (2021), pp. 76153–76177. DOI: 10.1109/ACCESS.2021.3081998 (cit. a p. 3).

[11] P. Rajitha Nair e D. Ramya Dorai. «Evaluation of Performance and Security of Proof of Work and Proof of Stake using Blockchain». In: *2021 Third International Conference on Intelligent Communication Technologies and Virtual Mobile Networks (ICICV)*. 2021, pp. 279–283. DOI: 10.1109/ICICV50876.2021.9388487 (cit. a p. 3).

[12] Anjaneyulu Endurthi e Akhil Khare. «Two-Tiered Consensus Mechanism Based on Proof of Work and Proof of Stake». In: *2022 9th International Conference on Computing for Sustainable Global Development (INDIACom)*. 2022, pp. 349–353. DOI: 10.23919/INDIACom54597.2022.9763215 (cit. a p. 3).

[13] Carmen Di Nardo. *Blockchain: Cos'è, come funziona, Quando investire*. Dic. 2021. URL: https://deltalogix.blog/2021/04/09/blockchain-qual-e-il-momento-migliore-per-investire/ (cit. a p. 4).

[14] Satoshi Nakamoto. *Bitcoin: A peer-to-peer electronic cash system*. Ott. 2008. URL: https://bitcoin.org/en/bitcoin-paper (cit. a p. 3).

[15] Vitalik Buterin. *Ethereum Blockchain*. 2013. URL: https://ethereum.org/it/whitepaper/ (cit. a p. 4).

[16] Gavin Wood. *Ethereum: a secure decenstralised generalised transaction ledger*. Apr. 2014. URL: https://ethereum.github.io/yellowpaper/paper.pdf (cit. a p. 4).

[17] Giuseppe Antonio Pierro. «Smart-Graph: Graphical Representations for Smart Contract on the Ethereum Blockchain». In: *2021 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*. 2021, pp. 708–714. DOI: 10.1109/SANER50967.2021.00090 (cit. a p. 6).

[18] Ruth Breu, Ursula Hinkel, Christoph Hofmann, Cornel Klein, Barbara Paech, Bernhard Rumpe e Veronika Thurner. «Towards a formalization of the unified modeling language». In: *European Conference on Object-Oriented Programming*. Springer. 1997, pp. 344–366 (cit. a p. 6).

[19] Jan Hendrik Hausmann e Stuart Kent. «Visualizing model mappings in UML». In: *Proceedings of the 2003 ACM symposium on Software visualization*. 2003, pp. 169–178 (cit. a p. 6).

[20] Kenneth Baclawski, Mieczyslaw K Kokar, Paul A Kogut, Lewis Hart, Jeffrey Smith, Jerzy Letkowski e Pat Emery. «Extending the Unified Modeling Language for ontology development». In: *Software and Systems Modeling* 1.2 (2002), pp. 142–156 (cit. a p. 6).

[21] Silvia Crafa, Matteo Di Pirro e Elena Zucca. «Is solidity solid enough?» In: *International Conference on Financial Cryptography and Data Security*. Springer. 2019, pp. 138–153 (cit. a p. 6).

[22] Giuseppe Antonio Pierro, Henrique Rocha, Roberto Tonelli e Stéphane Ducasse. «Are the gas prices oracle reliable? a case study using the ethgasstation». In: *2020 IEEE International Workshop on Blockchain Oriented Software Engineering (IWBOSE)*. IEEE. 2020, pp. 1–8 (cit. a p. 6).

[23] Josselin Feist, Gustavo Grieco e 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)*. 2019, pp. 8–15. DOI: 10.1109/WETSEB.2019.00008 (cit. a p. 8).

[24] Ting Chen, Xiaoqi Li, Xiapu Luo e Xiaosong Zhang. «Under-optimized smart contracts devour your money». In: *2017 IEEE 24th international conference on software analysis, evolution and reengineering (SANER)*. IEEE. 2017, pp. 442–446 (cit. a p. 8).

[25] ConsenSys. *Mythril: Security Analysis Tool for EVM bytecode*. URL: https://github.com/ConsenSys/mythril (cit. a p. 9).

[26] Sergei Tikhomirov, Ekaterina Voskresenskaya, Ivan Ivanitskiy, Ramil Takhaviev, Evgeny Marchenko e Yaroslav Alexandrov. «SmartCheck: Static Analysis of Ethereum Smart Contracts». In: *2018 IEEE/ACM 1st International Workshop on Emerging Trends in Software Engineering for Blockchain (WETSEB)*. 2018, pp. 9–16 (cit. a p. 10).

[27] R. Revere. *Visualize Solidity control flow for smart contract security analysis*. https://github.com/raineorshine/solgraph.git. 2019 (cit. a p. 11).

[28] Péter Hegedus. «Towards Analyzing the Complexity Landscape of Solidity Based Ethereum Smart Contracts». In: *2018 IEEE/ACM 1st International Workshop on Emerging Trends in Software Engineering for Blockchain (WETSEB)*. 2018, pp. 35–39 (cit. a p. 12).

[29] Chicxurug. *solmet-solidity-parser: A static analysis tool for calculating OO-style source code metrics for solidity smart contracts*. URL: `https://github.com/chicxurug/SolMet-Solidity-parser` (cit. a p. 12).

[30] Thomas J McCabe. «A complexity measure». In: *IEEE Transactions on software Engineering* 4 (1976), pp. 308–320 (cit. a p. 12).

[31] Node.js. *Documentation*. URL: `https://nodejs.org/en/docs/` (cit. a p. 15).

[32] Keheliya Gallaba, Quinn Hanam, Ali Mesbah e Ivan Beschastnikh. «Refactoring Asynchrony in JavaScript». In: *2017 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. 2017, pp. 353–363. DOI: `10.1109/ICSME.2017.83` (cit. a p. 16).

[33] *Node.js web application framework*. URL: `https://expressjs.com/` (cit. a p. 17).

[34] Expressjs. *Expressjs/multer: Node.js middleware for handling 'multipart/form-data'*. URL: `https://github.com/expressjs/multer` (cit. a p. 21).

[35] ConsenSys. *Surya: A set of utilities for exploring solidity contracts*. URL: `https://github.com/ConsenSys/surya` (cit. a p. 24).

[36] Commonsense. *ConceptNet: an open, multilingual knowledge graph*. URL: `https://conceptnet.io/` (cit. a p. 28).

[37] *Wiktionary Free Dictionary*. URL: `https://www.wiktionary.org/` (cit. a p. 28).

[38] Christiane Fellbaum e George Miller. «Applications of Wordnet». In: *WordNet: An Electronic Lexical Database*. 1998, pp. 197–197 (cit. a p. 28).

[39] J.W Breen. In: *Practical Issues and Problems in Building a Multilingual Lexicon* (lug. 2002) (cit. a p. 28).

[40] Commonsense. *API Conceptnet5 Wiki*. URL: `https://github.com/commonsense/conceptnet5/wiki/API` (cit. a p. 29).

[41] Mike Bostock. *Data-driven documents*. URL: `https://d3js.org/` (cit. a p. 33).

[42] *Web3.js - Ethereum javascript API*. URL: `https://web3js.readthedocs.io/` (cit. a p. 36).

[43] Parthasarathy Ramanujam. *Ethereum and blockchain*. Giu. 2019. URL: `https://iotbl.blogspot.com/2017/03/ethereum-and-blockchain-2.html` (cit. a p. 36).

[44] Solidity-Parser. *Solidity-parser: A solidity parser for JS built on top of a robust ANTLR4 grammar*. URL: `https://github.com/solidity-parser/parser` (cit. a p. 44).

[45] s278024. *contractTutorial*. URL: `https://github.com/s278024/ThesisFiles/tree/main/contractTutorial` (cit. a p. 74).

[46] s278024. *contractGruppo1*. URL: `https://github.com/s278024/ThesisFiles/tree/main/contractGruppo1` (cit. a p. 75).

[47] s278024. *contractGruppo2*. URL: `https://github.com/s278024/ThesisFiles/tree/main/contractGruppo2` (cit. a p. 75).

[48] s278024. *QuestionaryA*. URL: `https://github.com/s278024/ThesisFiles/blob/main/QuestionarioA.docx` (cit. a p. 75).

[49] s278024. *QuestionaryB*. URL: `https://github.com/s278024/ThesisFiles/blob/main/QuestionarioB.docx` (cit. a p. 75).

[50] *Ethereum IDE*. URL: `https://remix-project.org/` (cit. a p. 79).