Università degli Studi di Milano - Bicocca
Scuola di Scienze
Dipartimento di Informatica, Sistemistica e Comunicazione
Corso di laurea in Informatica

Studio e Implementazione di un Mixer per Criptovalute

Relatore: Prof. Alberto Leporati

Correlatore: Prof. Claudio Ferretti

Relazione della prova finale di:
Riccardo Ambrogio Gabriele Longoni
Matricola 806853

Anno Accademico 2018-2019

"L'intelligenza non consiste soltanto nella conoscenza, ma anche nella capacità di applicare la conoscenza alla pratica." - *Aristotele*

Indice

Elenco delle figure V

Introduzione VI

1 Tipologie studiate di mixer 1

1.1 Definizione e scopi di un mixer 1

1.2 I mixer con Cryptocurrency-Pool 4

1.2.1 Presenza di un partecipante disonesto 5

1.3 LRS per aumentare l'anonimato dei mixer 7

1.4 I mixer con partecipanti 8

1.4.1 Ottimizzazione in fase di implementazione 11

1.5 Considerazioni e tipologia sviluppata 12

2 Strumenti adoperati 14

2.1 Atom come editor di testo 14

2.2 Remix come editor di testo alternativo 15

2.3 Il linguaggio di programmazione Solidity 16

2.4 Ganache e Truffle come test net 17

2.5 React per sviluppare una Dapp 17

2.6 Material UI per la parte grafica 18

3 Implementazione 19

3.1 Creazione di una Dapp base 19

3.2 Avviamento della Dapp 21

3.3 Lo sviluppo del contratto MixΞtherio 23

3.3.1 La funzione transaction 26

3.4 Implementazione del client della Dapp 28

3.4.1 Il componente PasswordManager.js 28

3.4.2 Il componente AmountManager.js 31

3.4.3 Il componente MixingManager.js 31

3.5 Difficoltà riscontrate e risoluzioni 33

3.5.1 Tipi presenti in Solidity e non in Javascript 33

3.5.2 Assenza del metodo equals per stringhe in Solidity 33

3.5.3 Creazione costosa dell'insieme degli archi 34

3.5.4 Limitazioni in Solidity 34

3.5.5 Limitazioni della test net Ganache 35

4 Esempio di utilizzo 37

4.1 Inserimento di una password 37

4.2 Inserimento di un ammontare da mixare 38

4.3 Inizio del processo di mixing 39

5 Conclusioni e sviluppi futuri 41

5.1 Studio sui costi del cammino dei token 42

5.2 Ritardare le transazioni attraverso dei timer 42

Elenco delle figure

1.1 Esempio di transazione in Etherscan 2
1.2 Esempio mixer con partecipanti 10

2.1 Esempio della interfaccia di Atom 15
2.2 Interfaccia di Remix con smart contract 16

3.1 Folder Dapp 21
3.2 Ganache con smart contract pubblicati 22
3.3 Client della Dapp di MixΞtherio 22
3.4 Funzione con il tag OnlyOwner 24
3.5 La funzione transaction 27
3.6 Funzione implementata per inviare la password al contratto 29
3.7 Cattiva e corretta gestione di una password 30
3.8 Funzione che crea l'insieme degli archi-transazioni 36

4.1 Inserimento di una password 38
4.2 Inserimento di 10 ether 39
4.3 Avvio del processo di mixing 40
4.4 Insieme degli archi-transazioni costruiti 40

Introduzione

Il mondo delle criptovalute ha assunto un ruolo sempre più importante durante il corso degli anni. Sono state introdotte nuove tecnologie e nuovi software con differenti scopi che sfruttano la tecnologia delle Blockchain. Una di queste sono i *mixer* di criptovalute, utilizzati per nascondere l'origine dei token per i pagamenti. Celare questa informazione significa anche nascondere ulteriormente l'identità di chi possiede queste monete virtuali. Lo scopo della tesi è quello di mostrare i risultati delle ricerche effettuate durante lo stage per studiare e comprendere il funzionamento dei mixer. Come ulteriore scopo si è fissato l'obiettivo di imparare un linguaggio di programmazione, con relative direttive, per poterne sviluppare una versione propria, scegliendo fra quelle studiate. È quindi scopo della tesi dimostrare lo studio e l'applicazione che sono stati fatti durante lo stage. Come sistema di Blockchain è stato adoperato Ethereum. Alla base di questa scelta ci sono differenti motivazioni. La prima si collega con l'obiettivo di imparare un linguaggio di programmazione, ed Ethereum utilizza *Solidity* per creare degli *smart contracts*. Questo porta ad un differente approccio a livello programmativo perché al posto di avere concetti come *classe*, presente nel linguaggio *Java*, vengono utilizzati dei *contratti*. La seconda motivazione riguarda la vasta scelta di sviluppo, che Ethereum offre, per poter creare un mixer. In questo sistema di Blockchain è possibile realizzare delle *applicazioni decentralizzate*, chiamate anche *Dapp*, che si collegano direttamente alla rete di Ethereum, o testnet che la simulano. Un'applicazione decentralizzata ingloba al

suo interno dei contratti che sono pubblicati o *deployed* sulla Blockchain. Inoltre, viene messo a disposizione dell'utente un client che comunica costantemente con gli smart contracts dell'applicazione. Questo particolare approccio è stato scelto per lo sviluppo di un mixer di criptovalute. Infatti è stato sviluppato un contratto chiamato *Mix*Ξ*therio*, che assieme ad un client programmato in React concretizza una tipologia di mixer di criptovalute studiata.

Nel primo capitolo verrà mostrato il risultato della ricerca effettuata sui mixer di criptovalute. Inzialmente sarà data una definizione generica dei mixer e del loro funzionamento. Successivamente verranno presentate le forme in cui i mixer si declinano: mixer con Cryptocurrency-Pool e mixer con partecipanti. Il secondo capitolo presenterà tutti gli strumenti che sono stati utilizzati per sviluppare un mixer in forma di Dapp. Il terzo capitolo mostra i passaggi più importanti che sono stati effettuati per l'implementazione del mixer. Essendo stata sviluppata un'applicazione decentralizzata, si mostreranno le differenti parti di questa. Si passerà dal mostrare lo smartcontract MixΞtherio, e le sue funzioni più importanti, fino ad arrivare a mostrare il client dell'applicazione. Il quarto capitolo presenterà un esempio di utilizzo con relative immagini. Nel quinto, ed ultimo, capitolo verranno fatte delle conclusioni in merito al lavoro svolto durante lo stage, e si presenteranno alcuni possibili sviluppi futuri.

Capitolo 1

Tipologie studiate di mixer

In questo capitolo verrà data una definizione generica dei mixer di criptovalute. Successivamente verranno presentate le due principali tipologie di Tumbler studiate durante lo stage. Per la prima tipologia, mixer con Ether-Pool, è stata inserita una sezione che descrive come questa sia migliorabile con l'utilizzo di un particolare tipo di firma digitale chiamato *Linkable Ring Signature*. In conclusione verranno fatti dei confronti fra le due tipologie, e si descriverà la scelta effettuata in fase di sviluppo, motivando opportunamente tale decisione.

1.1 Definizione e scopi di un mixer

I mixer di criptovalute o Tumbler sono dei servizi software con lo scopo di rendere irrintracciabili i propri token all'interno di una Blockchain. Questa definizione si declina in più sviluppi e approcci che si adattano a seconda del tipo di sistema e criptovaluta che si considera. Infatti, per alcune monete virtuali tali servizi non sono per nulla utili o persino non attuabili, mentre per altre sono delle soluzioni molto efficienti. Quest'ultimo è il caso del sistema Ethereum e della sua moneta virtuale Ether, in quanto le transazioni sono pubbliche rendendo attuabile una ricerca del percorso che compiono i vari token da un account all'altro. Questo consente di

effettuare anche analisi sui vari Nodi della Rete ed eventualmente sugli indirizzi presenti. Infatti utilizzando Etherscan [8] si osservano gli elementi di una qualsiasi transazione.

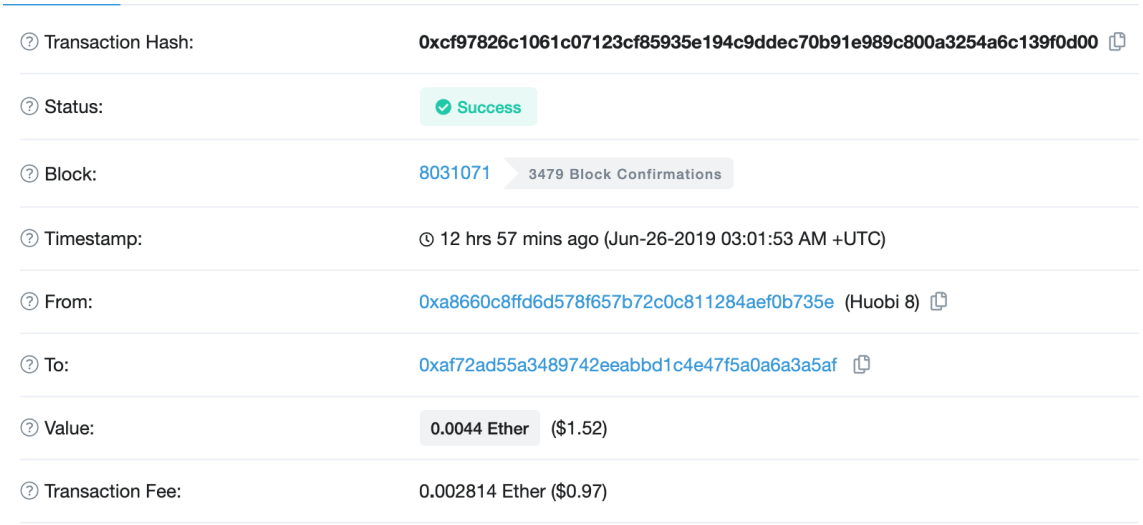

Figura 1.1. Esempio di transazione in Etherscan

Dall'esempio in Figura 1.1 si osservi come le transazioni hanno:

- Transaction Hash: Indirizzo con cui la transazione viene inserita all'interno di un Blocco.
- Status: Indicatore che riporta lo stato della avvenuta/fallita transazione.
- Block: Indica il numero di blocco in cui è stata inserita la transazione. Cliccando sul numero riportato in questo campo è possibile visionare il Blocco con tutte le sue relative proprietà ed informazioni.
- Timestamp: Segna la data e l'ora in cui è avvenuta la transazione.
- From: Indirizzo del mittente, che può aver inviato soldi virtuali oppure invocato una funzione dello smart contract.
- To: Indirizzo del ricevente.

- Value: Ammontare di criptovaluta inviato nella transazione, con la sua conversione in USD.

- Transaction Fee: In Ethereum, ogni volta che viene effettuato un pagamento o viene eseguita una funzione che cambia lo stato generale del contratto adoperato si deve pagare una tassa chiamata *gas*.

Oltre a questi campi, possono essere visionati nello specifico lo smart contract utilizzato, con relativi indirizzo e blocco su cui esso è stato pubblicato, ed anche il codice sorgente, se disponibile. Nonostante la presenza di tutte queste informazioni, Ethereum si fa garante del completo anonimato delle persone che detengono suddetti dati.

Come detto precedentemente, si possono adottare differenti tipologie di implementazione e sviluppo di un mixer, ad esempio si può progettare uno smart contract che si occupi di "mescolare" i token che riceve assieme ad altri già presenti, i quali possono essere sia dello stesso smart contract sia di altri utenti. Il mixer in questo caso andrebbe successivamente a restituire in egual quantità l'ammontare di token al mittente, garantendo che quelli che l'utente riceve non siano gli stessi che ha precedentemente immesso. Così facendo si ostacola in maniera efficiente una potenziale ricerca sulle transazioni antecedenti.

Un secondo esempio di sviluppo è quello dell'utilizzo di un'Applicazione Decentralizzata (Dapp), che si può collegare alla Blockchain e con la quale si sviluppa un client che permetta di avere un'interfaccia grafica per l'utente e diverse funzionalità che consentano la fase di mixing. L'applicazione, collegandosi alla Blockchain avrà una costante comunicazione con uno o più smart contract presenti su quest'ultima. Anche in questo modo l'utente avrà, al termine del processo di mixing, il totale che lui ha immesso, ma con un differente *Id* e la conseguente certezza di essere meno rintracciabile.

In generale, affidarsi ad un servizio di mixing può comportare dei seri rischi. Si ha sempre un contatto con terze parti che possono mascherare intenti malevoli per l'utente, con lo scopo di sottrargli permanentemente le sue criptovalute.

1.2 I mixer con Cryptocurrency-Pool

La prima categoria di mixer studiata è quella che prevede l'utilizzo di un unico smart contract, particolarmente consistente e ricco di codice. Un esempio concreto è presente, sul sito Github, nominato: "Decentralized Ether Mixer"[5]. In questa implementazione di mixer si è studiato come il processo di rimescolamento di token sia frutto di un versamento di questi ultimi all'interno di un unico fondo che fa da *pool di monete virtuali*. Inoltre, essendo concettualmente semplice il processo, si cerca di rendere anonimi e irrintracciabili il più possibile gli account che riceveranno i nuovi token e per conseguire questo scopo si implementa il tutto su più fasi. Essendo Ethereum il sistema preso in considerazione in questa tesi, i rispettivi mixer con *Cryptocurrency-Pool* saranno nominati mixer con *Ether-Pool*. Lo sviluppo e il relativo funzionamento di un mixer con Ether-Pool verrà illustrato con un esempio.

Si considerino per semplicità una coppia di utenti che si chiamano: *Alice* e *Bob*. Entrambi gli utenti detengono una coppia di account identificati con degli indirizzi, uno pubblico e uno privato. Tali coppie sono indicate nel seguente modo: $A = (P_{\mathrm{A}}, S_{\mathrm{A}})$ e $B = (P_{\mathrm{B}}, S_{\mathrm{B}})$. Entrambi gli utenti hanno a disposizione sul rispettivo account pubblico un fondo di 100 Ether e decidono di voler inviare questa quantità ai corrispondenti indirizzi privati. Per essere maggiormente sicuri, scelgono di utilizzare uno smart contract che prevede il mixing con Ether-Pool. Il contratto suddivide il processo in tre fasi:

1. Nella prima fase tutti gli indirizzi pubblici devono inviare al contratto la quantità di Ether che vogliono mixare. Deve essere effettuato un solo versamento per indirizzo pubblico. Quindi Alice e Bob effettuano un pagamento allo smart

contract di 100 Ether con gli account P_A e P_B.
È in questa prima fase che tutti i fondi ricevuti sono inseriti nel conto del contratto in un'unica quantità di criptovalute. Una volta raggiunto un minimo di partecipanti, stabilito dallo smart contract, si passa alla fase successiva.

2. Nella seconda fase è richiesta un'iscrizione da parte di tutti gli account privati, e per farlo devono pagare un ammontare di Ether che viene stabilito dal contratto. Alice e Bob provvedono a pagare tale somma con i rispettivi account S_A e S_B. Per sapere quante criptovalute inviare successivamente ai vari account privati, durante l'iscrizione è richiesto di esplicitare l'account pubblico tramite quello privato.

3. Durante la terza ed ultima fase si effettuano dei controlli. In particolare il contratto verifica che il numero di versamenti della prima fase sia **uguale** al numero di iscrizioni. Se l'uguaglianza viene rispettata allora si procede alla distribuzione dei token, sia quelli della prima fase sia quelli usati per l'iscrizione. Il contratto provvede a consegnare agli account privati delle criptovalute che non sono quelle versate inizialmente. Alice e Bob ricevono su S_A e S_B 100 ether più quelli della iscrizione.

Da osservare come venga correttamente consegnata la giusta quantità di criptovalute agli account privati. Questo è dovuto alla fase di iscrizione in cui è stato esplicitato l'indirizzo pubblico. Il contratto potrà quindi visionare la transazione del versamento effettuato nella prima fase e quindi vedere quanti token ha ricevuto e vericare l'effettivo indirizzo pubblico.

1.2.1 Presenza di un partecipante disonesto

Lo scenario precedentemente descritto è quello in cui viene dato per scontato che **tutti** i partecipanti del contratto siano onesti. Si supponga il caso in cui ci sia un

utente nominato *Charlie* il quale ha trovato l'indirizzo pubblico di Alice P_{A}, e che possiede un indirizzo privato S_{C}. Charlie ha lo scopo di rubare i token di Alice e per farlo procede a iscriversi durante la seconda fase, pagando la tassa imposta per l'iscrizione e presentando P_{A} come suo indirizzo Pubblico. Durante la terza fase, il contratto osserva che ci sono più iscrizioni rispetto al numero dei versamenti effettuati inizialmente. Pertanto viene eliminata la fase di mixing e quindi, tramite un processo inverso, vengono restituiti agli indirizzi pubblici i rispettivi token, sia quelli inviati nella prima fase sia quelli usati per l'iscrizione. Il contratto effettua questa operazione basandosi solo sulle transazioni che sono state effettuate durante la prima fase. Così facendo Charlie non riceve l'ammontare di token che lui aveva utilizzato per l'iscrizione dato che si è inserito solamente durante la seconda fase, e questi verranno aggiunti alla Ether-Pool dello smart contract. Alice, invece, riceve tutti i 100 ether assieme a quelli utilizzati per l'iscrizione.

Questo genere di sviluppo non incentiva l'inserimento di utenti malevoli e *man in the middle*, in quanto porterebbe ad una perdita definitiva delle criptovalute inserite per iscriversi e passare alla terza fase. Si possono progettare differenti controlli durante questa fase, per esempio si può inserire un numero prestabilito di iscrizioni, magari lo stesso di quelli che inviano allo smart contract i token durante la prima fase. Un'ulteriore soluzione potrebbe essere la registrazione degli account che si iscrivono durante la seconda fase. Per evitare di perdere l'anonimato degli indirizzi segreti si potrebbe implementare una funzione che registra l'hash dell'indirizzo anziché l'indirizzo stesso.

1.3 LRS per aumentare l'anonimato dei mixer

Allo scopo di migliorare l'anonimato e di aumentare ulteriormente la segretezza, è possibile introdurre il concetto di firma ad anello o ring signature [17]. Questo particolare tipo di firma digitale [12] consente di implementare un mixer che permette il prelievo di token dalla *Ether-Pool* senza far conoscere l'identità di chi compie questa azione. La firma ad anello viene generata da un gruppo di persone, dette *signers*, ciascuna delle quali possiede una coppia di chiavi:

- P_{i} è la chiave pubblica che specifica lo schema della firma e il modo per verificarne l'identità, dove i indica un qualsiasi *signer* dell'anello.
- S_{i} è la rispettiva chiave segreta, con cui l'utente firmerà i messaggi che vorrà inviare.

La proprietà principale di una firma ad anello è che ciascun componente del gruppo che la genera può firmare un messaggio, ma non può essere in nessun modo riconosciuto o identificato. Questo perché la firma viene generata da tutte le chiavi pubbliche dei partecipanti assieme alla chiave segreta del firmatario.

È possibile estendere la firma ad anello alla cosiddetta Linkable Ring Signature [13] (LSR). In questa tipologia di firma digitale, oltre a essere valide le proprietà della firma ad anello, è consentito che una persona del gruppo firmi per conto di esso. Viene richiesto un tag con il quale un utente non partecipante del gruppo può avere la certezza che il firmatario faccia parte dell'anello, senza però conoscerne l'identità. Così facendo si può migliorare il mixer con *Ether-Pool*, utilizzando la Linkable Ring Signature, nel seguente modo:

1. Si crea uno smart contract che:
    - stabilisce un numero minimo di partecipanti che formeranno l'Anello;
    - implementa una serie di funzioni per la distribuzione di criptovalute;

- implementa una funzione per il versamento dei token da Mixare.

2. I partecipanti devono avere un indirizzo pubblico ed uno segreto che saranno rispettivamente associati ad un account Mittente (M) e uno Ricevente (R).

3. I partecipanti devono creare una chiave Pubblica e una Privata. La chiave Pubblica deve essere inviata allo smart contract.

4. Lo smart contract attende che il numero delle chiavi pubbliche e i rispettivi account Mittenti siano sufficienti per iniziare il processo di mixing. Una volta raggiunto si passa alla fase successiva.

5. Gli account Mittente inviano l'ammontare di criptovalute che vogliono Mixare. Lo smart contract associa ad ogni account Mittente il totale che esso ha versato.

6. Per prelevare e confermare la propria identità, i partecipanti con l'account Ricevente inviano una firma con un tag al contratto. Quest'ultimo abbinerà successivamente il tag ricevuto alla chiave pubblica associata e al rispettivo account Mittente. Di conseguenza invierà l'ammontare di criptovalute mixate all'account Ricevente.

In questo modo si evita di dover implementare dei controlli per evitare *man in the middle* perché tutto viene regolato dalla firma ad anello.

1.4 I mixer con partecipanti

La seguente tipologia di mixer è quella che meglio sviluppa e simula il concetto di mescolamento di criptovalute. In questo caso non vi è solo un unico smarcontract, ma si utilizzano dei partecipanti che assieme compongono il mixer stesso. Questi ultimi possono essere account, con indirizzi associati, oppure altri smart contract nella Blockchain.

L'idea è quella di inviare l'ammontare di criptovalute che si vuole rendere anonime allo smart contract che funge da facciata del mixer. Questo provvede a stabilire in quante parti suddividere il fondo che ha ricevuto e ad inviarlo al numero, stabilito sempre dal contratto, di partecipanti. Ciascuno di essi provvede a dividere ulteriormente, oppure a mantenerla intatta, la quantità di token che riceve. Prima di compiere questa azione, si provvede a inserire l'ammontare ricevuto nell'insieme di criptovalute che già aveva, tenendo conto di quanti token dovranno essere mixati. Questo passaggio definisce il processo di mixing delle monete virtuali; ciascun partecipante può essere visto come una sorta di piccola Ether-Pool nella quale vengono inserite le criptovalute. Una volta partizionato il totale da mixare, ciascuna parte invia ad altrettanti partecipanti le nuove quantità. Tale processo crea un insieme di transazioni fra i vari partecipanti che possono scambiarsi ripetutamente più partizioni dello stessa quantità da mixare. Maggiore è l'insieme di queste transazioni e più saranno mixate le criptovalute dell'utente. Durante lo studio e l'implementazione di tale mixer è stato utile astrarre questo processo osservandolo come un grafo orientato: G. Un grafo [3] G è composto da un insieme di nodi chiamato V ed un insieme di archi orientati, chiamato E. In questo caso i nodi rappresentano tutti i partecipanti del mixer e gli archi sono tutte transazioni che avvengono nel processo di mescolamento dei token. Per poter rendere un grafo l'astrazione corrispondente ad un mixer con partecipanti devono valere le seguenti proprietà:

i. Il grafo deve essere privo di cappi [3].

ii. Nel grafo possono essere presenti dei cicli [3]. In questo caso si deve avere uno o più archi uscenti, da uno o più nodi del ciclo, che collegano ad altri nodi non appartenenti al ciclo. Così da evitare uno stallo nel flusso di criptovalute da mixare.

iii. L'utente di partenza, l'utente di arrivo e lo smart contract facciata devono appartenere a V. Pertanto l'utente di partenza verrà indicato come nodo di

partenza, mentre l'utente di arrivo come nodo di arrivo.

iv. Nel grafo deve esistere **almeno** un cammino dal nodo di partenza al nodo di arrivo.

v. Il nodo di partenza può essere il nodo di arrivo.

vi. Solamente il nodo di arrivo può essere un nodo pozzo [3]. Se questo coincide con il nodo di partenza allora deve esistere almeno un arco entrante ad esso.

Nella Figura 1.2 viene presentato un esempio di mixer con partecipanti che rispetta le proprietà elencate. In questo esempio abbiamo che i nodi del mixer sono sia account con relativi indirizzi, sia smart contract.

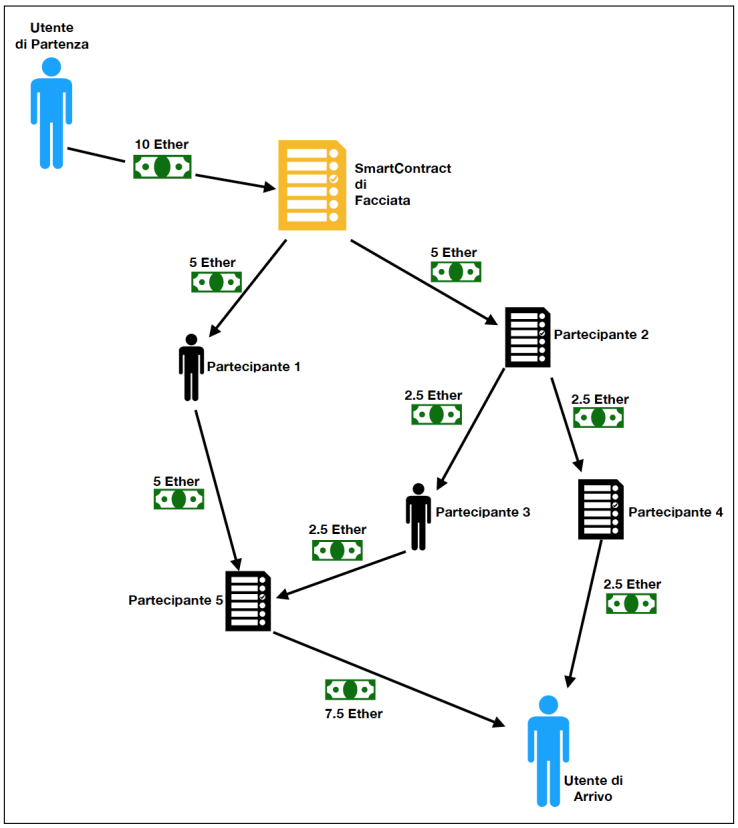

Figura 1.2. Esempio mixer con partecipanti

1.4.1 Ottimizzazione in fase di implementazione

Avendo scelto di implementare questa tipologia di mixer, sono state introdotte ulteriori proprietà ed elementi in fase di sviluppo che verranno elencate in questa sottosezione. Le proprietà elencate precedentemente devono essere rispettate dagli archi che rappresentano le transazioni che effettivamente trasportano le criptovalute da mixare e che sono dell'utente di partenza/arrivo. Tuttavia, così facendo, in E avremmo solo l'insieme degli archi che rappresentano le transazioni del processo di mixing, e quindi sarebbe facilmente rintracciabile l'utente di arrivo, il quale vuole rimanere il più anonimo possibile. Per questo è stata introdotta la possibilità di effettuare delle *false transazioni* o *fake transactions*. Queste sono sempre delle transazioni fra i partecipanti del mixer, ma con la caratteristica che le quantità di token che vengono scambiate non sono quelle dell'utente di partenza, e quindi non sono quelle da mixare, ma sono dei partecipanti.

Così è possibile costruire un nuovo grafo $G_{\mathrm{f}} = (V_{\mathrm{f}}, E_{\mathrm{f}})$ che **non** rispetta alcune proprietà di G. In particolare per G_{f} deve valere che:

i. $V_{\mathrm{f}} = V$ - {NodoPartenza, NodoArrivo}. Vengono coinvolti solo i partecipanti del mixer.

ii. E_{f} è l'insieme degli archi rappresentanti le false transazioni.

iii. Nel grafo è possibile avere uno o più cicli.

iv. Non devono esserci dei cappi nel grafo. Questo perché si andrebbe ad effettuare una transazione in più che risulterebbe inutile e dispendiosa in Ethereum.

v. $\exists a \in V_{\mathrm{f}}$ che è un nodo pozzo. Deve esistere almeno un partecipante che faccia da nodo pozzo per le false transazioni. Questo per aggiungere ulteriore confusione.

Queste proprietà elencate sono state anche sviluppate. Ne possono essere aggiunte ulteriori per aggiungere ulteriore confusione al processo di mixing. Per il

momento, sono stati considerati i due grafi G e G_f, e si è supposto che questi fossero completamente isolati. Tutta una Blockchain, come quella di Ethereum, è composta da indirizzi con account associati, smart contract che interagiscono fra di loro o con altri utenti e transazioni. Quindi se si osservano gli account e smart contracts come nodi, e le transazioni come archi orientati, allora G e G_f potrebbero essere dei sottografi [3] di un unico grafo che rappresenta l'intera Blockchain. Questa estensione ad un grafo più ampio è volta a sottolineare come possa essere ulteriormente difficile rintracciare il cammino delle criptovalute che vengono mixate nel momento in cui si implementa una tipologia di mixer con partecipanti.

L'ultima caratteristica dei mixer è la loro possibilità, a livello programmativo, di cambiare tutti i partecipanti.

1.5 Considerazioni e tipologia sviluppata

In questo capitolo sono state presentate due tipologie mixer:

- Mixer con Ether-Pool. Si presenta principalmente come un unico smart contract che provvede a prendere e a mixare il quantitativo di criptovalute inviate dall'utente. Essendo un unico contratto, ci sarà molto codice, tuttavia si potranno gestire in tre fasi tutto il processo di mixing delle monete virtuali. Si evitano intrusi e *man in the middle*. Inoltre, sono adoperabili le Linkable Ring Signature per poter mantenere l'anonimato degli utenti e diminuire i controlli durante l'ultima fase.

- Mixer con partecipanti. Esso è composto da uno smart contract, più esiguo di codice rispetto al precedente mixer, che funge da facciata per l'utente. Le restanti componenti del mixer sono una serie di partecipanti, smart contract o indirizzi o entrambi, che si scambiano di continuo le criptovalute da mixare

e alla fine restituiscono all'utente destinatario dei token completamente differenti. Questa tipologia è facilmente confrontabile con un modello astratto come quello dei grafi. Infine, si presta bene ad una implementazione tramite applicazioni decentralizzate (Dapp).

Fra queste è stata scelta la seconda tipologia di mixer. La motivazione principale è stata la possibilità di poter sviluppare e sperimentare un'applicazione decentralizzata. Inoltre, i partecipanti sono facilmente interscambiabili con altri, specialmente se questi sono solo indirizzi.

Capitolo 2

Strumenti adoperati

In questo capitolo si elencheranno tutti gli strumenti software utilizzati per sviluppare il mixer con partecipanti. Si elencheranno gli editor, i linguaggi di programmazione adoperati ed infine le librerie software utilizzate.

2.1 Atom come editor di testo

Atom [2] è un IDE (ambiente di sviluppo integrato). È un ambiente programmabile, infatti mette a disposizione una serie di packages che aggiungono funzionalità all'ambiente. Per esempio è stato installato un package che consentiva l'utilizzo di un terminale direttamente nell'editor, velocizzando il lancio dei comandi per compilare o fare deployment di smart contract sulla testnet. Il terminale è visibile nella Figura 2.1 sotto al codice. Una funzionalità ulteriore, presente di default, è quella di avere un pannello per effettuare commit/push/pull da Git o Github. Un esempio è visibile nella Figura 2.1 a destra.

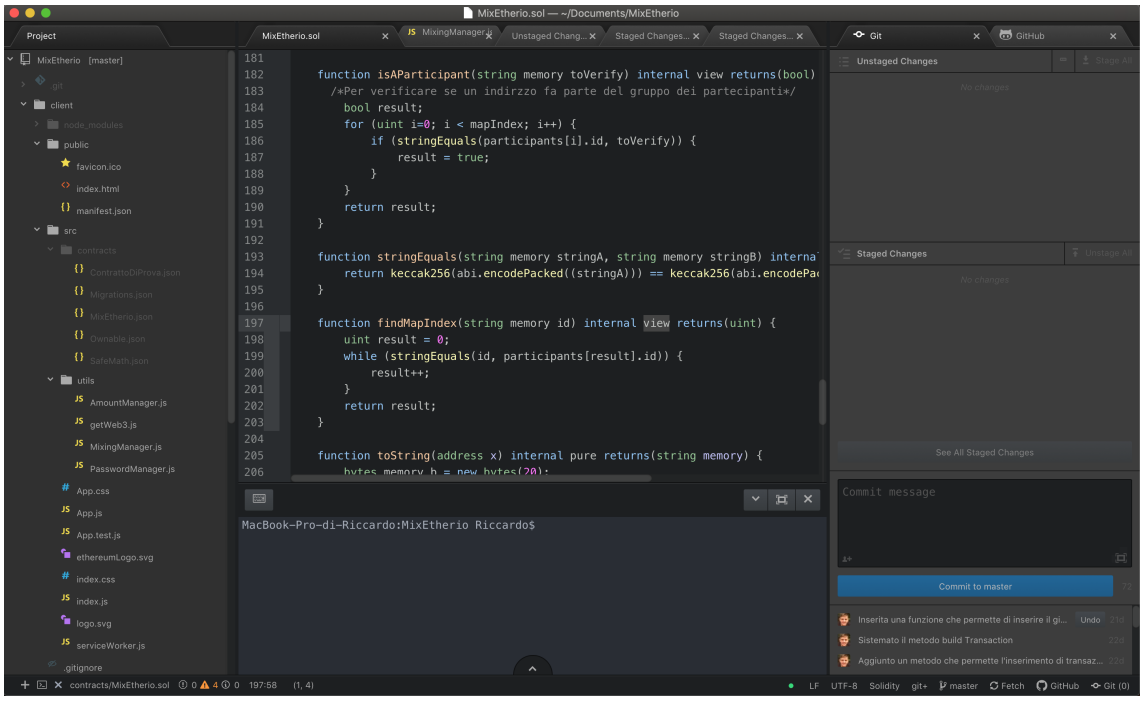

Figura 2.1. Esempio della interfaccia di Atom

2.2 Remix come editor di testo alternativo

Remix [16] è un IDE online che mette a disposizione del programmatore non solo un'ottima interfaccia grafica, ma permette anche di compilare direttamente i vari smart contract direttamente online. È possibile anche scegliere il tipo di Virtual Machine che si vuole adoperare, stando ovviamente attenti al tipo di versione che si adotta poichè ciascuna ha le sue direttive. Implementa anche una test net privata, una che può collegarsi a MetaMask e quindi direttamente alla Blockchain di Ethereum. Un esempio è illustrato nella Figura 2.2, dove si può notare sulla destra, in basso, lo smart contract già pubblicato sulla test net. Inoltre è visibile l'account in uso con il quantitativo di Ether che possiede, e una serie di ulteriori funzionalità.

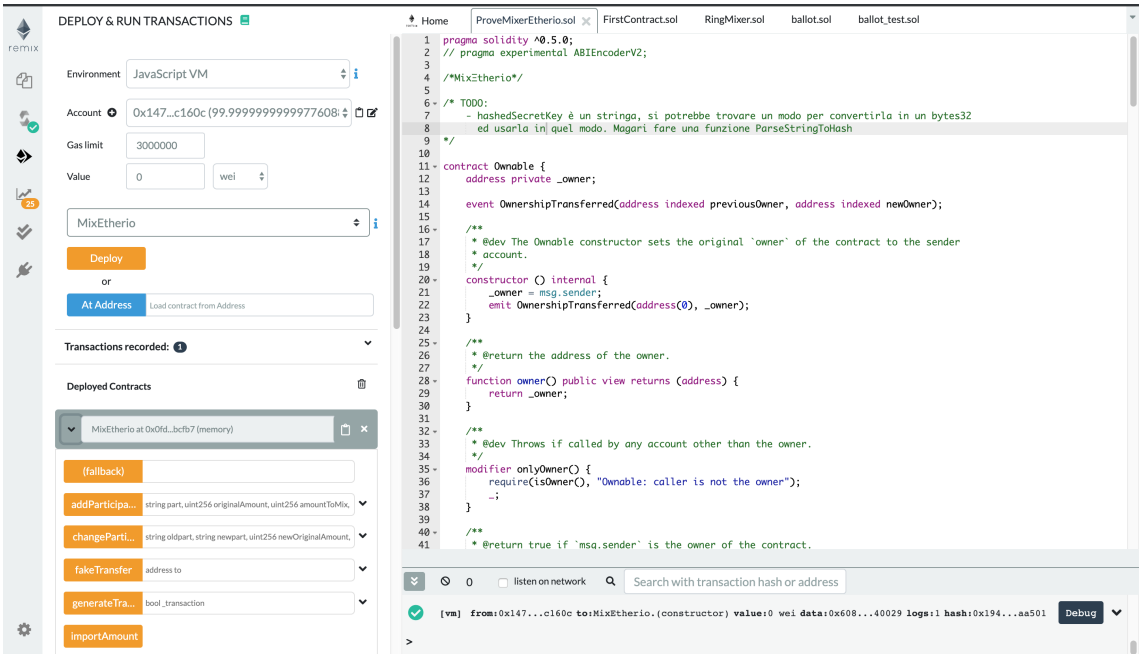

Figura 2.2. Interfaccia di Remix con smart contract

2.3 Il linguaggio di programmazione Solidity

Solidity [1] è un linguaggio di programmazione imperativo con una sintassi molto simile a linguaggi come Javascript, C# o C++ e Java. È il linguaggio maggiormente utilizzato in Ethereum per programmare e sviluppare smart contract. Questo viene definito come un programma che viene eseguito su Ethereum. In particolare può essere visto come l'equivalente di una classe in Java o C#, solo che possiede particolari tipi di tag per la firma dei metodi. Questa caratteristica non deve essere sottovalutata in Solidity per esempio, se una funzione non cambia degli attributi globali del contratto, ma si occupa solo di mostrare un determinato valore allora questo dovrà essere segnato come un metodo *view* oppure *pure*. In questo modo non verranno richieste da Ethereum dei costi in *gas* [1] [4] per l'esecuzione di quella funzione.

2.4 Ganache e Truffle come test net

Ganache [18] è un'interfaccia grafica che mette a disposizione un'ottima test net su cui pubblicare e testare i propri smart contract. Inoltre si può stabilire un numero di account utilizzabili sulla test net. Questo è stato un elemento fondamentale per riuscire a simulare il funzionamento di un mixer con partecipanti.

Truffle [11] è un framework che offre delle librerie Javascript tra cui Web3js, che permettono la comunicazione tra il client della Dapp e la test net, con i relativi smart contract pubblicati su di essa.

2.5 React per sviluppare una Dapp

React [10] è un framework Component-Based scritto in Javascript. Le componenti [10] in React sono oggetti incapsulati che gestiscono il loro stato e permettono la gestione del client di una Dapp. In React un qualsiasi client è composto da un insieme di componenti che comunicano costantemente fra loro, ciascuna delle quali ha un proprio stato e i propri compiti. La caratteristica principale a livello programmativo è che React utilizza Javascript per renderizzare le proprie componenti, che vengono scritte in HTML5 e CSS. Quindi tutto viene compattato come se fosse un unico linguaggio.

```
render() {
    const var1 = 'esempio';
    const React = 'React';

    return (
        <div>
        {'Questo è un ' + var1 + 'di come ' + React + ' Funziona'}
        </div>
    )
}
```

L'esempio precedente è una parte del metodo *render()* presente su tutti i componenti di React. Nelle prime due righe interne al metodo vengono dichiarate e assegnate due variabili; si osservi come la sintassi sia quella di Javascript. Successivamente, alla chiamata del metodo return(), come parametro riceve un tag in HTML5 che verrà renderizzato in una pagina web. Tuttavia si usa **sempre** Javascript per programmare tutto.

2.6 Material UI per la parte grafica

React consente la personalizzazione dei vari componenti. È presente in rete una libreria di componenti React: pulsanti, AppBar, Dialog, Steppers ed altri elementi che vengono utilizzati per il funzionamento della Dapp. Tale libreria è chiamata MaterialUI [9]. La sua peculiarità è la grafica che esso adotta, inoltre è una libreria OpenSource. Tutti i pulsanti, campi e ulteriori parti del client sono stati sviluppati con la grafica di MaterialUI.

Capitolo 3

Implementazione

A seguito dello studio delle tipologie di mixer, si è successivamente passato alla parte di programmazione e sviluppo. In questo capitolo seguono differenti passaggi che sono stati effettuati durante lo stage, e verranno presentati i principali elementi che compongono la Dapp che implementa un mixer con partecipanti.

3.1 Creazione di una Dapp base

Grazie al Framework Truffle [11], è possibile scaricare una Dapp molto semplice, ma funzionante, sulla quale si possono effettuare modifiche e inserire ulteriori file per ottenere la Dapp che ci serve. Si procede con l'apertura del terminale del computer, e si digitano i seguenti comandi:

```
cd /PathCheSiDesidera // scelgo un percorso.
mkdir progetto        // creo la cartella per il progetto.
cd progetto           // mi sposto nella cartella creata.
truffle unbox react   // comando per creare una Dapp,
                      // assieme ai moduli richiesti.
```

Il progetto è stato nominato: MixΞtherio (da leggersi MixEtherio), pertanto sarà utilizzato tale nome per indicare la cartella contenente il progetto, e lo stesso, che

è stato realizzato durante lo stage. A seguito del caricamento, una volta eseguito il comando *'truffle unbox react'*, sarà possibile aprire da Atom [2] il folder MixΞtherio. Nella Figura 3.1 si possono vedere tutti i folder e file che sono stati caricati; per praticità e chiarezza, in fase di programmazione è stata mantenuta la disposizione di tutti questi. Nello specifico:

- **client**: è la cartella dove si trova il client della Dapp. In particolare sono presenti al suo interno una cartella contenente i *node modules* o moduli di NodeJs che permettono l'avvio e il funzionamento dell'applicazione. Inoltre, in questo folder, sono da inserire tutte le componenti che costituiscono il client della Dapp.

- **contracts**: è la cartella nella quale andranno inseriti tutti i contratti scritti in Solidity [4].

- **migrations**: è una cartella molto importante, perché contiene dei programmi Javascript che consentono di caricare sulla test net i contratti in
  Solidity che vengono compilati.

- **test**: è la cartella nella quale devono essere inseriti i programmi utilizzati per effettuare suite di test sui relativi smart contracts. Di default si trovano due file, uno è in Javascript l'altro è in Solidity.

3.2 Avviamento della Dapp

Una volta preparato l'ambiente di sviluppo, e dopo aver preso visione delle differenti cartelle della Dapp, verranno elencati i comandi che sono fondamentali per avviare l'applicazione. Prima di tutto si deve aprire e tenere attivo Ganache [18], in modo da avere la test net online. Successivamente, tramite Truffle si compilano e pubblicano sulla test net i contratti presenti nella cartella *contracts*, tramite i comandi:

```
truffle compile // compilare i contratti
truffle migrate // deployment dei contratti
```

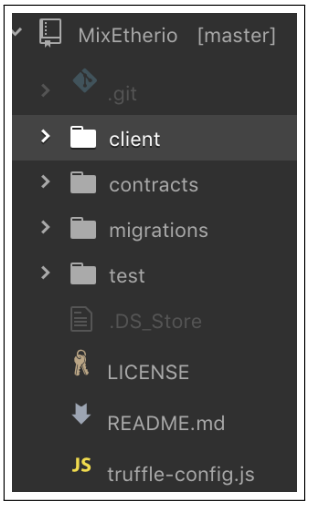

Figura 3.1. Folder Dapp

Se non sono presenti errori nei contratti e non ci sono problemi relativi all'applicazione Ganache, allora gli smart contract saranno pubblicati sulla test net e si potrà interagire con essi. In Figura 3.2 si può osservare l'insieme degli smart contracts compilati, tutti quanti, e "deployed" che sono solo *"Migrations"* e *"MixEtherio"*. Solamente questi due sono utilizzati nella Dapp: il primo è, dal nome, un contratto di prova e i restanti vengono utilizzati solo per far ereditare al contratto *"MixEtherio"* dei metodi.

Finora è stata avviata solamente la prima parte della Dapp. Quello che è stato precedentemente descritto è anche l'aspetto fondamentale che caratterizza una applicazione decentralizzata. Infatti, l'elemento che rende l'effettiva "decentralizzazione" è il collegare l'applicazione ad una blockchain, che per sua definizione è un sistema distribuito e decentralizzato.

Nella seconda ed ultima parte di avviamento è necessario avviare il client della Dapp. Essendo programmato in React [10], si deve navigare da terminale fino alla cartella *client*, vista nella sezione precedente, e digitare il comando:

```
npm run start // per avviare il client della Dapp
```

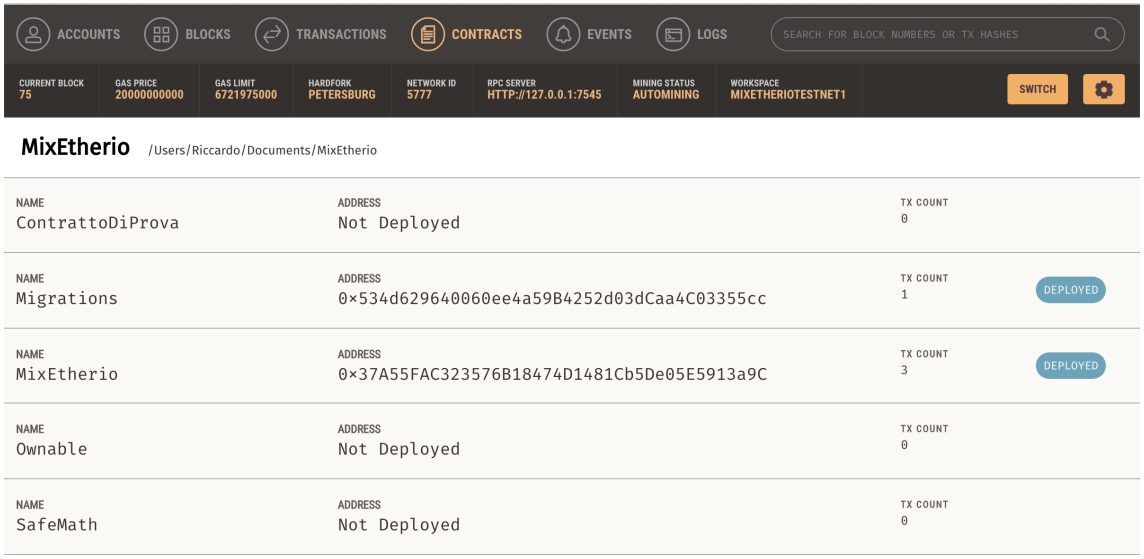

Figura 3.2. Ganache con smart contract pubblicati

Anche in questo caso, se non sono presenti errori di programmazione e se l'applicazione Ganache è attiva in background allora verrà visualizzato sul proprio browser predefinito il client dell'applicazione decentralizzata. Si riporta in Figura 3.3 il client della Dapp sviluppata durante lo stage.

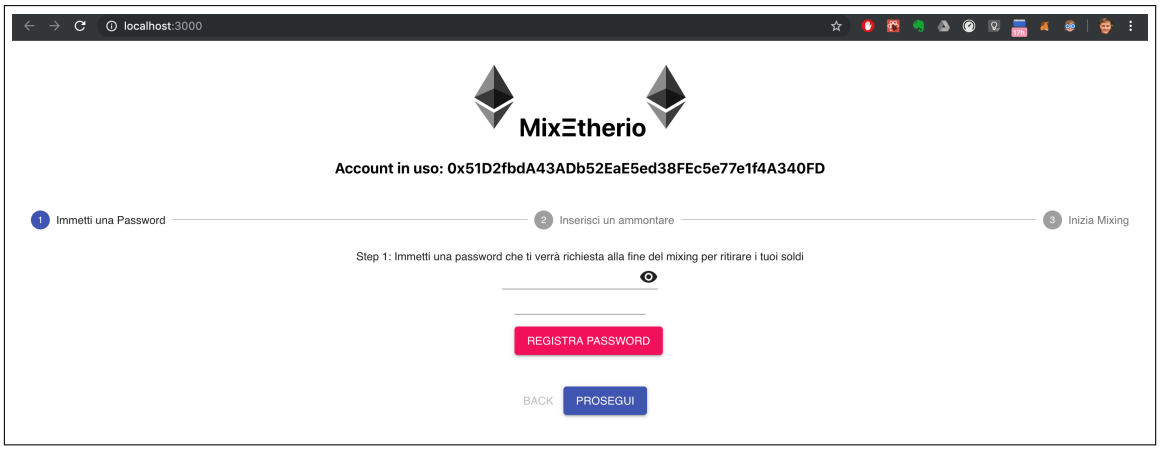

Figura 3.3. Client della Dapp di MixΞtherio

Dopo aver definito i passaggi necessari all'avvio della Dapp verranno presentati i principali elementi che sono stati sviluppati.

3.3 Lo sviluppo del contratto MixΞtherio

Il contratto MixΞtherio è molto importante per il funzionamento dell'intera Dapp, e per riprodurre al meglio un mixer con partecipanti. Prima di tutto è stata scelta la versione 0.5.0, o superiori, del compilatore di Solidity. Si è scelta tale implementazione in quanto durante lo stage era la versione più recente e allo stesso tempo più stabile che si potesse utilizzare. Inoltre è presente un'ottima documentazione [7], che è stata largamente utilizzata per sviluppare il contratto.
Il contratto MixΞtherio, come prima cosa, è stato sviluppato in modo tale che ereditasse dal contratto *Ownable.sol* [14] una serie di metodi che consentono di creare una gerarchia tra gli utilizzatori del contratto. Infatti, alcune funzioni rese pubbliche, nella loro firma, sono state siglate con il tag: *onlyOwner*. Così facendo queste sono visibili da tutti quelli che utilizzano lo smart contract, ma non possono eseguirle a meno che non siano i proprietari del contratto. Il proprietario è la persona o l'ente che pubblica sulla blockchain il contratto.

Nel contratto è stata importata anche una libreria: *SafeMath* [15]. Questa consente di effettuare le operazioni di Somma, Differenza, Prodotto e Divisione senza avere rischi di bugs, legati ad esempio, agli overflow. Successivamente sono state definite e dichiarate una serie di variabili globali, con un ruolo centrale per rendere lo smart contract *"MixEtherio"* un contratto "facciata", ma non solo. In fase di sviluppo i partecipanti sono stati implementati come account e non come smart contract. Tale scelta è stata effettuata per praticità: infatti con Ganache si potevano avere a propria disposizione una serie di account già presenti sulla test net; questo nella realtà si traduce anche in un risparmio in fatto di soldi. Infatti, tutte le volte che si pubblica sulla blockchain un contratto è richiesto un costo in ether [1]. Creare

```
function addParticipant(string memory ids,
                        bool isWaterWell,
                        uint index)
                        public
                        onlyOwner
{
    idParts[index] = ids;
    originalAmountParts[index] = 100000000000000000000;//  default
    mixableAmountParts[index] = 0;
    isWaterWellParts[index] = isWaterWell;
}
```

Figura 3.4. Funzione con il tag OnlyOwner

un account, invece, non richiede alcun pagamento. Per suddette ragioni, sono state implementate anche una serie di funzioni che fossero a disposizione degli account partecipanti del mixer. Inoltre gli account sono facilmente rimpiazzabili con altri, quindi si potrebbe pensare di cambiarli ciclicamente dopo una serie di processi di mixing. Le variabili globali con maggiore importanza sono:

- **contractAmount**: Variabile intera, privata. Essa contiene un valore "fittizio" che indica l'ammontare di ether che il contratto detiene. Si può vedere come la quantità di token contenuta nella ether-pool del contratto.
- **etherToMix**: Variabile intera, privata. Essa contiene un valore "fittizio" che indica l'ammontare di criptovalute da mixare. Su questo valore vengono adottate delle strategie per la partizione del totale da inviare ai partecipanti.
- **hashedSecretKey**: Variabile, privata, che memorizza una stringa. Quest'ultima sarà una password stabilita dall'utente, che verrà inviata al contratto tramite il client, attraverso la chiamata di una funzione del contratto:

```
function deployTheKey(string memory privKey)
                      public
                      onlyOwner
{
/*Funzione utilizzata dal client per memorizzare
  l'hash della chiave dell'utente nel contratto.
  Lo scopo è quello di avere un ulteriore strumento
  per la verifica dell'invio all'utente di partenza.
*/
 hashedSecretKey = privKey;
}
```

A causa di un problema riscontrato in fase di sviluppo, è stata implementata una serie di variabili pubbliche di tipo mapping. In Solidity questo tipo indica una lista dinamica di elementi identificata da una coppia chiave-valore. Per semplicità la chiave è sempre un intero, mentre il valore cambia a seconda della variabile scelta. Di seguito si riportano tali variabili:

- **idParts**: Insieme degli indirizzi delle parti che compongono il mixer. Questi indirizzi vengono memorizzati come stringhe.
- **originalAmountParts**: Insieme che contiene tutti i fondi di ciascun partecipante.
- **mixableAmountParts**: Insieme che contiene tutte le quantità che ciascun partecipante deve mixare.
- **isWaterWellParts**: insieme che contiene i booleani che indicano se un partecipante è un nodo pozzo oppure no.
- **participants**: Insieme dei partecipanti, questo insieme lo si ha solamente quando le precedenti variabili elencate sono state instanziate.

A seguire sono state sviluppate alcune funzioni per il corretto funzionamento dello smart contract. Un esempio è riportato in Figura 3.4, dove vengono inseriti

i dati all'interno delle variabili globali pubbliche. È stata implementata anche una funzione per cancellare un partecipante, nel caso in cui si volesse aggiungerne altri oppure no.

```
function deleteParticipant(uint index)
                           public
                           onlyOwner
{
/*
  Conoscendo l'indice della mappa
  che contiene il partecipante
*/
    delete (participants[index]);
}
```

3.3.1 La funzione transaction

Una delle funzioni più importanti è stata quella che permette al mixer di effettuare delle transazioni. Nella Figura 3.5 viene riportato il codice. La funzione, nella sua firma, presenta diversi tag dichiarativi. Prima di tutto con *public* si esplicita che la funzione è visibile ed invocabile da tutti. Il tag *payable* indica che con questa funzione avviene un trasferimento di ether.

I successivi tag: *isParts()* con parametri *to* e *from*, sono delle chiamate, esterne alla funzione, al modificatore isParts il quale controlla che il parametro passato, di tipo string, sia un associato ad un partecipante del mixer. Nel caso questa condizione non fosse verificata allora la funzione non verrebbe eseguita, così si ha la certezza che la funzione venga eseguita **solamente** dai partecipanti, evitando che qualcuno di esterno si intrometta.

La funzione prende come parametri:

- **to**: un parametro stringa, nel quale deve esserci l'indirizzo del partecipante che invia delle criptovalute.

- **from**: un parametro stringa, nel quale deve essere presente l'indirizzo del partecipante che riceve delle criptovalute.
- **foo**: un parametro booleano, il quale è utilizzato dalla funzione per stabilire se la transazione da effettuare sarà:
  - una transazione di token da mixare;
  - una falsa transazione.

```
function transaction(string memory to, string memory from, bool foo)
                    public
                    payable
                    isParts(to)
                    isParts(from)
{
    uint indexTo = findMapIndex(to);
    uint indexFrom = findMapIndex(from);
    uint amountTo = 0;
    uint amountFrom = 0;
    address(this).transfer(msg.value);
    contractAmount = contractAmount.add(msg.value);
    if (foo) {
        amountTo = participants[indexTo].mixableAmount;
        amountFrom = participants[indexFrom].mixableAmount;
        participants[indexFrom].mixableAmount = amountFrom.sub(msg.value);
        participants[indexTo].mixableAmount = amountTo.add(msg.value);
    }else {
        amountTo = participants[indexTo].originalAmount;
        amountFrom = participants[indexFrom].originalAmount;
        participants[indexTo].originalAmount = amountTo.add(msg.value);
        participants[indexFrom].originalAmount = amountFrom.sub(msg.value);
    }
}
```

Figura 3.5. La funzione transaction

Come mostrato in Figura 3.5, il corpo della funzione *transaction* è composto essenzialmente da un controllo if-then-else. A seconda del valore che viene passato

al parametro *foo* si prenderà una decisione piuttosto che l'altra. Se *foo* fosse *true* allora verrebbe effettuata una transazione di token da mixare. Al contrario, se fosse *false* allora verrebbe effettuata una falsa transazione. Questo è reso possibile grazie a due attributi che ogni variabile *Partecipante* ha: *mixableAmount* è il totale che deve essere mixato, *originalAmount* è l'ammontare che il partecipante ha in partenza, e che deve utilizzare per effettuare una falsa transazione.

3.4 Implementazione del client della Dapp

Il client dell'applicazione è stato sviluppato tramite una serie di componenti React e MaterialUI. Il primo fra questi è stato nominato *App.js*, il quale mette a disposizione un'interfaccia grafica nella quale viene visualizzato il nome dell'applicazione che si sta utilizzando, l'indirizzo dell'utente e uno stepper. Si veda la Figura 3.3 che mostra gli elementi elecanti precedentemente assieme al componente *PasswordManager.js*, di cui si parlerà successivamente. Lo stepper non è altro che un ulteriore componente React, preso da MaterialUI [9], che consente all'utente di effettuare una serie di passaggi. Per ciascuno step sono stati sviluppati dei componenti, che verranno mostrati e spiegati di seguito.

3.4.1 Il componente PasswordManager.js

Nella prima fase, viene utilizzato il componente *PasswordManager.js*. Il suo compito è quello di mettere a disposizione dell'utente due campi in cui inserire la password, che può essere visualizzata in chiaro tramite un pulsante a lato del campo, e un pulsante da cliccare nel momento in cui si vuole inviare la password allo smart contract. Per poter inviare la password è stata implementata nel contratto una funzione apposita, che viene invocata dal componente tramite l'utilizzo di Truffle e Web3.js [6]. Dato che questo passaggio può essere effettuato solo tramite una transazione

che avviene nel momento in cui si invoca, nella funzione del componente, la procedura presente nel contratto, si deve prestare particolare attenzione alla gestione della password stessa. Infatti, nel caso non vi fossero controlli adeguati si rischierebbe di inviare la password in chiaro e leggibile direttamente nella transazione stessa. Pertanto è stata utilizzata una funzione di hash che converte la password inserita in una stringa alfanumerica, che viene poi inviata al contratto. Vengono presentati due esempi nella Figura 3.7, dove prima si mostra come una cattiva gestione della password porti ad una visibilità totale della password, che in questo esempio è *prova*, e poi un esempio dove invece si ha un hash della chiave che ne nasconde la parola d'origine, che nell'esempio rimane sempre *prova*.

```
importKey = () => {
  const { accounts, contract } = this.state;
  const keccak256 = require('keccak256');
  let password1 = this.state.psw1;
  let password2 = this.state.psw2;

  if(this.isMatching(password1,password2)){
      let hashedKey = keccak256(this.state.psw1).toString('hex');
      contract.methods.deployTheKey(hashedKey).send( { from: accounts[0] } );
      this.setState({ psw1:'', psw2:''}); // Elimino le password dalla Dapp
      this.setState({isImported: this.state.psw1 === this.state.psw2});
  }else{
      console.log("Le Password non coincidono");
  }
}
```

Figura 3.6. Funzione implementata per inviare la password al contratto

Per rendere effettuabili delle chiamate dal componente al contratto sulla test net viene invocata una funzione, presente su questo e tutti i componenti presenti nella Dapp, chiamata *componentDidMount*, che ha la seguente forma:

Figura 3.7. Cattiva e corretta gestione di una password

```
componentDidMount = async () => {
  try {
    const web3 = new Web3("http://127.0.0.1:7545"
                          || Web3.givenProvider);
    const accounts = await web3.eth.getAccounts();
    const user = accounts[0];
    const networkId = await web3.eth.net.getId();
    const deployedNetwork = MixEtherioContract.networks[networkId];
    const instance = new web3.eth.Contract(
        MixEtherioContract.abi,
        deployedNetwork && deployedNetwork.address,
      );
    this.setState({ web3, accounts, contract: instance},
                  this.startMixing
                  );
    } catch (error) {
      alert(
        ‘Failed to load web3, accounts, or contract.
        Check console for details.‘,
      );
      console.error(error);
    }
};
```

Questa funzione è *asincrona* poiché effettua una serie di chiamate ad un servizio che potrebbe non rispondere istantaneamente. Successivamente vengono dichiarate e

istanziate una serie di variabili che sono successivamente memorizzate nello stato del componente e quindi sono rese accessibili da tutte le restanti funzioni del componente stesso. Tali variabili consentono di effettuare una serie di chiamate a funzioni della libreria di Web3.js e quindi di richiamare anche delle funzioni presenti nel contratto *MixEtherio*.

3.4.2 Il componente AmountManager.js

Una volta completata la prima fase, si passa al successivo *step*. In questo è stato sviluppato un componente che mette a disposizione dell'utente un campo in cui inserire un valore numerico. È stato inserito a lato del campo un menù a tendina nel quale l'utente può selezionare l'unità di ether che vuole inviare. Una volta selezionato l'ammontare da inviare, l'utente può effettuare una transazione verso il contratto tramite un apposito pulsante.

3.4.3 Il componente MixingManager.js

Nell'ultima parte dello stepper è stato sviluppato un componente che racchiude in sé una funzione che completa la Dapp rendendola un vero e proprio mixer con partecipanti. Se nel contratto sono stati definiti i partecipanti, componendo l'insieme dei Nodi V, nel client, grazie al componente *MixingManager.js*, viene costruito l'insieme degli archi E che sono le transazioni da effettuare. Nella Figura 3.8 viene riportato il codice della funzione *buildTransaction()* che consente la creazione dell'insieme degli archi.

Inizialmente, nella funzione, vengono definite una serie di variabili:

1. **Transaction**: Una variabile oggetto, che rappresenta un arco-transazione. In questo caso, oltre ad avere l'indirizzo di partenza e arrivo, è stato inserito anche un intero che indica l'insieme di criptovalute che vengono scambiate tra le parti.

2. **transactionsNum**: Una variabile che rappresenta il numero di transazioni che si vogliono effettuare. Quindi il numero di archi di E. Questa variabile viene utilizzata come contatore per un ciclo *for* che seguirà la dichiarazione delle variabili.

3. **transactions**: Un array di oggetti di tipo Transaction. Questo array a livello programmativo è l'insieme E degli archi.

4. **rdnNum1** e **rdnNum2**: Due variabili che contengono due numeri prodotti in maniera casuale da una funzione definita nel componente.

Successivamente, nella funzione viene eseguito un ciclo *for* che basa le sue iterazioni sul numero di transazioni che viene indicato nella variabile *transactionsNum*, la quale fino a quando è strettamente positiva fa iterare il ciclo. Nel ciclo, vengono generati due numeri casuali che saranno utilizzati come indici per pescare due indirizzi con cui creare la transazione. Per evitare cappi e andare a violare una proprietà del mixer con partecipanti, descritta nel primo capitolo, viene eseguito un ciclo dentro al quale fintanto che i due numeri casuali sono uguali fra loro allora si generano ulteriori numeri casuali. A seguito di questo controllo, vengono utilizzate due variabili che memorizzano un valore booleano. Questo viene calcolato in base ad un predicato che prevede l'utilizzo della variabile *isWaterWellParts*, per capire se ci sono nodi pozzo oppure no, e quale dei due lo è. A seconda della condizione verificata vengono poi inseriti nell'array *transactions* le transazioni costruite seguendo e rispettando le proprietà di un mixer con partecipanti, per convenzione si è supposto che l'ammontare di soldi da scambiare fosse solo 1 ether. Questo tipo di algoritmo e di funzione consente la costruzione di un qualsiasi insieme di transazioni. Pertanto basta modificare opportunamente l'insieme dei nodi, indirizzi dei partecipanti, e l'insieme *isWaterWellParts*, che specifica quali nodi sono nodi pozzo oppure no, per poter ottenere alla fine un grafo delle transazioni e uno di false transazioni.

Grazie a questi componenti React sviluppati durante lo stage è stato possibile concretizzare un mixer con partecipanti.

3.5 Difficoltà riscontrate e risoluzioni

Durante lo sviluppo del mixer sono sorti differenti difficoltà che verranno presentate in questa sezione. Per alcune sono state adottate delle soluzioni che possono essere rese durature, per altre è necessario attendere degli aggiornamenti software dai differenti framework e per altri problemi invece non è stata trovata alcuna soluzione.

3.5.1 Tipi presenti in Solidity e non in Javascript

La difficoltà maggiormente riscontrata è stata la presenza di tipi di variabili in Solidity che non erano utilizzabili in Javascript. Per esempio in Solidity si può dichiarare una variabile di tipo Address (contenente ad esempio il valore 0x1234ABCD000108) e in Javascript un simile valore viene riportato come una stringa. Nel concreto non è stato possibile avere sia sullo smart contract sia nel client della Dapp la possibilità di avere un unico tipo Address per entrambi.

È stato quindi scelto di convertire nel contratto tutte le variabili Address a variabili Stringhe.

3.5.2 Assenza del metodo equals per stringhe in Solidity

Non è possibile in Solidity confrontare due stringhe con un metodo equals(). Per verificare che due stringhe siano uguali è stato adoperata una funzione hash molto utilizzata in Solidity: keccak256 [7], se due stringhe sono uguali allora lo sono anche la loro codifica in hash.

```
function stringEquals(string memory stringA,
                      string memory stringB)
                      internal
                      pure
                      returns(bool)
{
return (keccak256(abi.encodePacked((stringA))) ==
       keccak256(abi.encodePacked((stringB))));
}
```

3.5.3 Creazione costosa dell'insieme degli archi

Per i contratti scritti in Solidity, tutte le volte che vengono invocate delle funzioni in cui nella loro firma non è presente il tag *pure* o *view* si deve pagare un costo chiamato *gas*. In particolare si deve effettuare un pagamento per l'invocazione di tutte le funzioni che cambiano lo stato del contratto, quindi se viene cambiata una variabile globale dello smart contract. Per questo motivo sarebbe stato assai costoso cercare di costruire l'insieme degli archi nel contratto, in quanto si sarebbe dovuto tenere un array di archi-transazioni come una variabile globale, con una struttura uguale a quella presentata nella sezione precedente. Se nel contratto ci sono dei costi di *gas*, nel client della Dapp no. Quindi l'algoritmo è stato sviluppato e inserito nel componente che si occupa della gestione di avvio mixing.

3.5.4 Limitazioni in Solidity

In Solidity sono presenti delle forti limitaizoni a livello programmativo. La più importante che è stata riscontrata durante lo sviluppo del contratto è stata l'impossibilità di far restituire alle funzioni un valore complesso come un array o un oggetto. Questo problema non ha ancora trovato una soluzione definitiva, dato che si tratta di una limitazione del linguaggio Solidity e della versione del compilatore. Tuttavia, si può utilizzare una particolare libreria: *pragma experimental ABIEncoderV2;*. Questa libreria consente di effettuare delle operazioni, come la restituzione di un

oggetto più complesso da parte di una funzione, ma è fortemente sconsigliata da utilizzare per sviluppare contratti che devono poi essere pubblicati sulla blockchain. Infatti nel momento in cui si compila il contratto con questa libreria, da console viene visualizzato un messaggio: ***Warning****: Experimental features are turned on. Do not use experimental features on live deployments. pragma experimental ABIEncoderV2;* . In questo caso, si deve attendere che vengano rilasciate nuove versioni di Solidity che consentano l'utilizzo sicuro di queste funzionalità.

3.5.5 Limitazioni della test net Ganache

Ganache, pur essendo molto utile, presenta una forte limitazione. Infatti arrivati ad una certa quantità di transazioni effettuate il *gas* spendibile sulla test net finisce, rendendo impossibile la visione delle transazioni che sono effettuate durante il processo di mixing, sia quelle false sia quelle normali. Anche in questo caso si deve attendere una nuova release di Ganache, oppure cercare un nuovo programma che implementi una test net più potente e migliorata.

```
buildTransactions = () => {
  const { accounts, contract } = this.state;
  let Transaction = {
    from: '',
    to: '',
    amount: 0,
  }
  let transactionsNum = 10; // può essere generato casualmente purchè non superi |VxV|
  let transactions = [];
  let rdnNum1 = 0;
  let rdnNum2 = 0;
  for(; transactionsNum>0;transactionsNum--){
    rdnNum1 = this.randomInt(6);
    rdnNum2 = this.randomInt(6);
      while (rdnNum1 === rdnNum2) {
        rdnNum1 = this.randomInt(6);
        rdnNum2 = this.randomInt(6);
      }
      let condition1 = this.state.isWaterWellParts[rdnNum1]
                      && !this.state.isWaterWellParts[rdnNum2];
      let condition2 = this.state.isWaterWellParts[rdnNum2]
                      && !this.state.isWaterWellParts[rdnNum1];
      if(condition1){
        transactions.push({
          from: this.state.IdParts[rdnNum2],
          to: this.state.IdParts[rdnNum1],
          amount: 1,
        });
      }else if(condition2){
        transactions.push({
          from: this.state.IdParts[rdnNum1],
          to: this.state.IdParts[rdnNum2],
          amount: 1,
        });
      }else {
          (rdnNum1 > rdnNum2)
          ? (transactions.push({ from: this.state.IdParts[rdnNum1],
                                 to: this.state.IdParts[rdnNum2],
                                 amount: 1,
                               }))
          : (transactions.push({ from: this.state.IdParts[rdnNum2],
                                 to: this.state.IdParts[rdnNum1],
                                 amount: 1,
                               }));
    }
  }
}
```

Figura 3.8. Funzione che crea l'insieme degli archi-transazioni

Capitolo 4

Esempio di utilizzo

In questo capitolo sono presentate le tre fasi che l'utente deve effettuare per avviare il processo di mixing. Nella prima dovrà creare una chiave segreta, nella seconda scegliere un ammontare da inviare e mixare, nella terza e ultima avviare il processo di mixing.

4.1 Inserimento di una password

Nella prima fase, illustrata in Figura 4.1, è stata inserita una password debole che contiene solo caratteri minuscoli. Il componente durante questa fase effettua un controllo su entrambi i valori che vengono inseriti nei due campi e nel momento in cui questi coincidono e l'utente clicca il pulsante "registra password" queste vengono inviate al contratto. Per evitare che la password fosse messa in chiaro nella transazione effettuata, è stata utilizzata una funzione hash nel client che codificasse la chiave e mandasse il risultato al contratto. Come presentato nella seconda transazione della Figura 3.7, viene inviato un hash della password "prova" inserita nell'esempio.

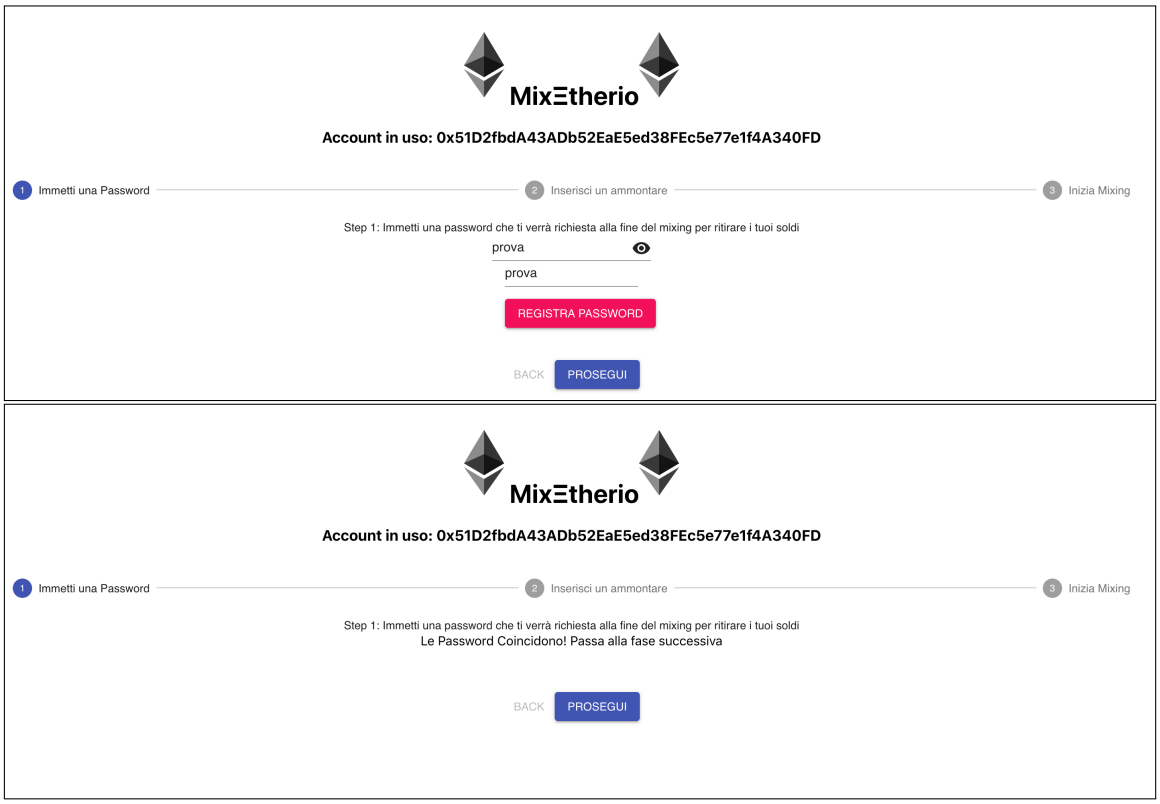

Figura 4.1. Inserimento di una password

4.2 Inserimento di un ammontare da mixare

Durante la seconda fase è richiesto all'utente di immettere una quantità di criptovalute da mixare, e successivamente di inviarle allo smart contract tramite il pulsante "invia ether" posto a lato. È stata resa possibile anche la scelta dell'unità di misura che si vuole utilizzare per inviare i token, come viene mostrato nella Figura 4.2. Nell'esempio vengono inviati 10 ether al contratto, e nella Figura 4.3 si mostra come questi siano tracciati nella transazione registrata nella test net.

Figura 4.2. Inserimento di 10 ether

4.3 Inizio del processo di mixing

Una volta inviati 10 ether e la password *"prova"*, si passa all'ultima fase, mostrata in figura Figura 4.3. In questa l'utente non deve fare altro che cliccare il pulsante "inizio mixing" oppure se dovesse cambiare idea può annullare l'intero processo che porterebbe alla restituzione dei fondi inviati meno una parte di tasse pagate per le transazioni effettuate precedentemente.

Per problemi legati alla test net di Ganache non è stato possibile effettuare un vero e proprio esempio. Tuttavia nel momento in cui si preme il pulsante di inizio mixing viene invocata la funzione *buildTransaction* del componente *MixingManager.js*, che costruisce un insieme di archi-transazioni. Nella Figura 4.4 si mostra il risultato delle transazioni avendo a disposizione un predefinito numero di parti, tra cui anche l'utente che risulta essere il primo dell'array *IdParts*. Nell'array *isWaterWellParts* viene identificato come nodo pozzo solo il primo elemento, cioè anche il

Figura 4.3. Avvio del processo di mixing

primo elemento dell'array *IdParts* che è l'utente, e di conseguenza viene costruito un insieme di transazioni dove l'utente non sarà mai un nodo che invia token agli altri partecipanti, ma li riceve e basta.

Figura 4.4. Insieme degli archi-transazioni costruiti

Capitolo 5

Conclusioni e sviluppi futuri

In conclusione, si è mostrato come sia possibile sviluppare concettualmente un mixer di criptovalute. È stato mostrato il modello di un mixer con Ether-Pool, assieme ai suoi punti di forza e limiti. È stato introdotto il concetto di *Linkable Ring Signatures* per rafforzare la sicurezza e aumentare l'anonimato del mixer con Ether-Pool. Successivamente si è mostrata la definizione e il funzionamento del mixer con partecipanti, il quale è stato poi scelto come modello da sviluppare in sede di stage. Dopo aver introdotto gli strumenti software utilizzati in fase di programmazione, è stata mostrata l'intera struttura dell'applicazione decentralizzata. In particolare si è posta l'attenzione su come creare una Dapp di base da cui partire e porre delle modifiche. Sono stati mostrati i comandi che servono per poter avviare l'applicazione. Successivamente sono stati elencati le principali funzioni e attributi del contratto MixΞtherio, smart contract utilizzato per ricevere e inviare ai partecipanti del mixer le criptovalute dell'utente. Questo contratto ha una serie di attributi che memorizzano **tutti** gli indirizzi dei partecipanti, e predispone alcune funzioni che solo il proprietario dello smart contract può effettivamente eseguire. È stata mostrata e commentata la funzione *transaction* del contratto che consente di effettuare lo scambio di criptovalute **solamente** fra i componenti del mixer. Assieme allo smart contract MixΞtherio è stato riportato lo sviluppo dei principali componenti

del client della Dapp. Il componente che si occupa della gestione della password, evita che questa possa essere resa pubblica sulla transazione. Il secondo componente mostrato invia al contratto un ammontare di token, stabilito dall'utente. L'ultimo componente riportato è forse quello più importante, perché in esso è stata programmata una funzione *buildTransactions* che costruisce un insieme di archi-transazioni fra un insieme di nodi. Dato l'insieme dei partecipanti, assieme all'indirizzo dell'utente, è quindi possibile, con tale funzione, costruire un percorso che i token devono effettuare per realizzare il processo di mixing, fino a quello dell'utente di arrivo. In particolare, l'utente di arrivo viene inserito come unico nodo pozzo. Oltre a poter costruire un cammino dei token da mixare, è possibile costruire così un insieme di false transazioni. Dopo la presentazione dello sviluppo della Dapp si è passati ad un esempio di utilizzo. A seguito di questi risultati si possono individuare degli sviluppi futuri su questo progetto:

5.1 Studio sui costi del cammino dei token

Avendo a disposizione nell'applicazione decentralizzata, l'array che contiene tutte le transazioni da effettuare per mixare e poi inviare i token nuovi all'utente, si potrebbe pensare di effettuare uno studio del costo totale che queste transazioni richiedono. A seguito di questo studio si potrebbe pensare di adottare un determinato tipo di soluzione. Per esempio si potrebbe cercare di capire se sia meglio generare un numero elevato di transazioni fasulle ed effettuare il minor numero possibile di quelle per il processo di mixing, oppure se sia meglio fare l'opposto.

5.2 Ritardare le transazioni attraverso dei timer

Per aggiungere ulteriore confusione si potrebbe implementare un algoritmo che tenga conto anche di un potenziale ritardo delle transazioni. A livello programmativo,

soprattutto in Javascript sono presenti differenti funzioni, come la funzione *setInterval(function() alert("Hello"); , 3000);*, che in questo esempio ritarda di 3 secondi la visualizzazione del messaggio "Hello". Essendo realizzabile tramite il linguaggio Javascript una funzione del genere, si potrebbe adattare alla funzione sviluppata durante lo stage. Inoltre si farebbe carico il client di questo compito, evitando costi aggiuntivi e facendo in modo di mantenere senza modifiche aggiuntive lo smart contract MixΞtherio. In aggiunta si potrebbe modificare l'insieme delle transazioni, aggiungendo un attributo a tutte le transazioni che ne andrebbe ad indicare il ritardo che quella transazione deve avere rispetto al processo intero di mixing.

Bibliografia

[1] Andreas M. Antonopoulos e Gavin Wood. *Mastering Ethereum*. O'Reilly Media, Inc., 2018.

[2] *Atom IDE*. URL: `https://atom.io/`. (Ultimo accesso: 01.03.2019).

[3] Thomas H. Cormen et al. *Introduzione agli algoritmi e strutture dati*. Mc-Graw Hill, 2010.

[4] Chris Dannen. *Introducing Ethereum and Solidity*. Apress, 2017.

[5] *Decentralized Ether Mixer*. URL: `https://github.com/yaronvel/smart_contracts`. (Ultimo accesso: 01.07.2019).

[6] *Documentazione Framework Web3.js*. URL: `https://web3js.readthedocs.io/en/1.0/`. (Ultimo accesso: 19.05.2019).

[7] *Documentazione Solidity*. URL: `https://solidity.readthedocs.io/en/v0.5.0/`. (Ultimo accesso: 30.05.2019).

[8] *Etherscan*. URL: `https://etherscan.io/`. (Ultimo accesso: 26.06.2019).

[9] *Framework MaterialUI*. URL: `https://material-ui.com/`. (Ultimo accesso: 21.05.2019).

[10] *Framework React*. URL: `https://reactjs.org/`. (Ultimo accesso: 21.05.2019).

[11] *Framework Truffle*. URL: `https://www.trufflesuite.com/truffle`. (Ultimo accesso: 22.05.2019).

[12] Alessandro Languasco e Alessandro Zaccagnini. *Manuale di Crittografia*. Hoepli, 2015.

[13] Prof Bill Buchanan OBE. *Linkable Ring Signatures, Stealth Addresses and Mixer Contracts*. URL: (https://medium.com/asecuritysite-when-bob-met-alice/linkable-ring-signatures-stealth-addresses-and-mixer-contracts-cff7057a457). (Ultimo accesso: 01.07.2019).

[14] OpenZeppelin. *Contratto Ownable*. URL: https://github.com/OpenZeppelin/openzeppelin-solidity/tree/master/contracts/ownership/Ownable.sol. (Ultimo accesso: 19.05.2019).

[15] OpenZeppelin. *Libreria SafeMath*. URL: https://github.com/OpenZeppelin/openzeppelin-solidity/tree/master/contracts/math/SafeMath.sol. (Ultimo accesso: 19.05.2019).

[16] *Remix IDE*. URL: https://remix.ethereum.org/. (Ultimo accesso: 29.05.2019).

[17] Ron Rivest, Adi Shamir e Yael Tauman. «How to Leak a Secret». In: (2001), pp. 1–5. URL: (https://people.csail.mit.edu/rivest/pubs/RST01.pdf).

[18] *TestNet Ganache*. URL: https://www.trufflesuite.com/ganache. (Ultimo accesso: 22.05.2019).