



# UNIVERSITÀ DI PISA

DIPARTIMENTO DI INFORMATICA  
CORSO DI LAUREA TRIENNALE IN INFORMATICA

**zk-ABAC: un sistema di controllo degli accessi con  
garanzie di privacy basato su zero-knowledge**

**Relatori:**

Prof. Laura Ricci  
Dott. Damiano Di Francesco Maesa

**Candidato:**

Gianluca Boschi

**Sessione autunnale**  
Anno Accademico 2019/2020

## **Sommario**

Il progetto di tirocinio consiste nello studio, progettazione, implementazione e valutazione di un'estensione per un sistema di controllo degli accessi basato su blockchain Ethereum. L'idea alla base dell'estensione è l'applicazione di tecniche Zero-Knowledge per garantire la privacy agli utenti utilizzatori del sistema, introducendo un metodo per rendere private le informazioni sensibili che transitano sulla blockchain. Durante il tirocinio è stata sviluppata una proof of concept del sistema stesso.

# Indice

<b>1</b>	<b>Introduzione</b>	<b>3</b>
<b>2</b>	<b>Stato dell'arte</b>	<b>6</b>
2.1	Blockchain . . . . .	6
2.1.1	Ethereum . . . . .	7
2.1.2	Miners . . . . .	7
2.1.3	Costi: gas e fee . . . . .	7
2.1.4	La memoria . . . . .	8
2.1.5	Smart Contract . . . . .	9
2.1.6	Transazioni concorrenti: problemi . . . . .	9
2.2	Attribute Based Access Control . . . . .	10
2.2.1	ABAC e XACML . . . . .	10
2.2.2	Struttura e componenti standard XACML . . . . .	11
2.3	Un Sistema di Access Control basato su Blockchain . . . . .	13
2.3.1	Vantaggi: la trasparenza . . . . .	15
2.3.2	Punto critico: la mancanza di privacy . . . . .	15
2.4	Zero-knowledge . . . . .	15
2.4.1	Background matematico: le curve ellittiche . . . . .	16
2.4.2	zk-SNARKs . . . . .	17
2.4.3	Zokrates . . . . .	20
<b>3</b>	<b>Privacy-Based ABAC su blockchain Ethereum</b>	<b>24</b>
3.1	Integrazione di ZoKrates ed ACS: introduzione . . . . .	25
3.2	Integrazione di ZoKrates nell'ACS: la parte off-chain . . . . .	27
3.2.1	Primo modello: un verificatore per ogni politica . . . . .	28
3.2.2	Secondo modello: oracoli modulari . . . . .	29
3.3	Interazione tra ACS e Prover . . . . .	31
3.3.1	Modello Sincrono . . . . .	31
3.3.2	Modello Asincrono . . . . .	35

<b>4 zk-ABAC: Implementazione</b>	<b>38</b>
4.1 Tool utilizzati: Remix . . . . .	38
4.2 La scrittura della politica di sicurezza . . . . .	39
4.2.1 Creazione e compilazione dei file . . . . .	40
4.2.2 Il verificatore per le prove: verifier.sol . . . . .	40
4.3 L’interfaccia on-chain del nodo off-chain: Oracolo.sol . . . . .	42
4.4 La logica per il controllo degli accessi: ACS_logic.sol . . . . .	48
4.5 Esempio di utilizzo di zk-ABAC . . . . .	52
<b>5 Valutazioni</b>	<b>55</b>
5.1 ZoKrates: costi computazionali e dimensionali . . . . .	55
5.1.1 Analisi dell’occupazione di memoria . . . . .	55
5.1.2 Analisi dei Tempi . . . . .	58
5.2 Valutazioni del sistema su blockchain . . . . .	60
5.2.1 Il caso ottimo non deterministico . . . . .	62
5.2.2 Caso di funzionamento standard . . . . .	63
5.2.3 Utilizzo di una blockchain dedicata per garantire buone prestazioni	64
5.3 Considerazioni finali . . . . .	64
<b>6 Conclusioni</b>	<b>66</b>
6.1 Lavori futuri . . . . .	66
<b>Bibliografia</b>	<b>68</b>

# Capitolo 1

## Introduzione

I sistemi di controllo degli accessi [3] sono ad oggi estremamente importanti per quanto riguarda le garanzie di sicurezza e privacy che essi apportano ad aziende, organizzazioni, società, o in ambito privato. Questi sistemi consentono di limitare l'accesso ad alcune risorse, considerate sensibili, tramite l'adozione di politiche di accesso. Esistono varie tipologie di sistemi di controllo degli accessi, che si diversificano in base ai meccanismi che utilizzano per implementare le politiche di sicurezza. Nel progetto di tirocinio verrà studiato un sistema di controllo degli accessi ABAC [17], cioè basato su attributi. Un attributo è una qualsiasi caratteristica di un utente, soggetto o dell'ambiente, definito da una entità autorizzata e verificata. Un sistema ABAC permette di imporre politiche che utilizzano gli attributi per limitare gli accessi a risorse sensibili.

L'uso di sistemi di controllo degli accessi centralizzati comporta alcuni problemi, come quello delle entità “trusted” (entità verificate e affidabili). Data l'esistenza di una singola entità centrale dove risiede tutto il potere decisionale, se i suoi servizi venissero disabilitati, gli utenti non potrebbero più accedere alle risorse o le risorse non sarebbero più protette; se questa l'entità centrale venisse invece compromessa, utenti legittimi potrebbero vedersi negato l'accesso o utenti illegittimi potrebbero riuscire a guadagnare l'accesso. Il concetto di decentralizzazione riesce a risolvere queste situazioni: nascono quindi i sistemi di controllo degli accessi basati su attributi e implementati su blockchain.

Il rapido sviluppo delle tecnologie decentralizzate, come le blockchain, ne ha dettato la diffusione anche nell'ambito dei sistemi di controllo, dove tradizionalmente le architetture sono sempre state centralizzate. Le blockchain garantiscono molti vantaggi, primi tra i quali l'auditability delle operazioni e dei dati, e il controllo decentralizzato: questi sono concetti che sono in pieno accordo con le esigenze dei sistemi di controllo degli accessi. Nelle blockchain, come ad esempio Ethereum [6], una rete di macchine (nodi), tramite uno specifico algoritmo, convalida o meno un'operazione: è l'intera rete a decidere se tale operazione sia valida o meno.

Dalla natura pubblica della blockchain nasce però un altro problema: tutto ciò che viene registrato sulla blockchain è pubblico, quindi risulta difficile garantire la privacy di tutti quegli attributi o di quelle informazioni che non devono essere rivelate perché

considerate sensibili. Il problema di garantire la privacy degli attributi privati è molto comune, e nel corso del tempo sono state fornite varie soluzioni: nel progetto di tirocinio verrà studiato l'utilizzo di tecniche a Zero-Knowledge [1], tramite le quali è possibile dimostrare la conoscenza di alcuni dati senza doverli obbligatoriamente rivelare.

Verrà mostrato come sia possibile e cosa comporta l'uso di tecnologie “a conoscenza zero” su blockchain per garantire la privacy degli attributi, e verrà fornita una PoC (Proof of Concept) del funzionamento del sistema. Queste tecnologie sono nuove: la loro applicazione era molto complessa fino a pochi anni fa, in quanto richiedeva una conoscenza matematica approfondita di argomenti come l'aritmetica modulare e la crittografia su Curve Ellittiche. La tecnologia zero-knowledge (letteralmente “conoscenza zero”) si basa sul concetto di dimostrare la conoscenza di una soluzione per un certo problema senza doverla rivelare.

## Zero Knowledge | Intuitive Example

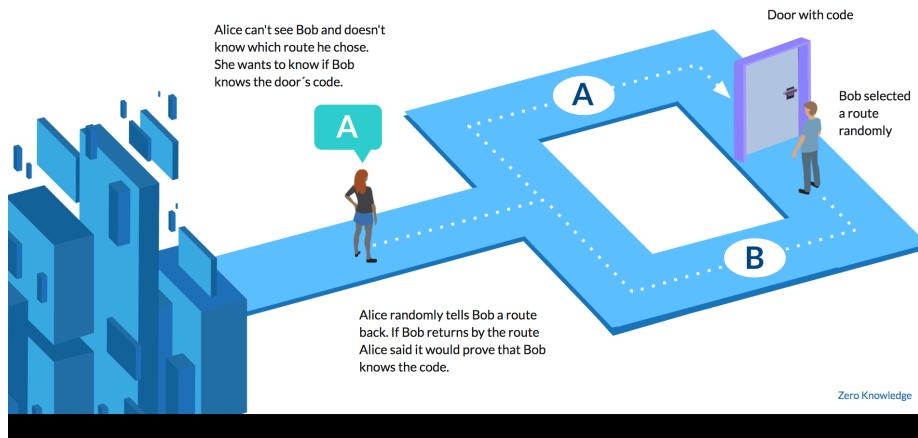


Figura 1.1: Esempio intuitivo di come funzionino le zero-knowledge

Nella figura 1.1 viene mostrato un esempio intuitivo di zero-knowledge. Bob deve dimostrare ad Alice di conoscere il codice corretto che apra la porta, senza però rivelarlo. Alice quindi sceglie uno dei due percorsi, A o B, e comunica a Bob di usare quel percorso per raggiungerla. Se Bob utilizza il percorso che Alice gli aveva suggerito, allora lei si convince che Bob conoscesse il codice per la porta. Alice potrebbe pensare che sia stata una questione di fortuna: c'era infatti una probabilità del 50% che entrambi avessero scelto la stessa strada. Tuttavia, se questo esperimento viene ripetuto più volte, la probabilità che Bob utilizzi lo stesso percorso selezionato da Alice senza avere il codice si riduce progressivamente, fino a tendere a zero. In conclusione, se Bob utilizza un numero sufficiente di volte il percorso scelto da Alice, ha dimostrato inequivocabilmente ad Alice

la veridicità della sua affermazione: conosce il codice segreto. A tal fine non è stato necessario condividere il codice.

Per il progetto è stato preso in considerazione un tool che implementa la tecnologia zero-knowledge, ZoKrates, e un sistema di controllo degli accessi ABAC in standard XACML implementato su blockchain Ethereum. Tramite analisi e studio di questi due argomenti è stato possibile estendere il sistema di controllo, fornendo un'implementazione che garantisca la privacy su blockchain pubbliche.

Il resto del documento è suddiviso in capitoli. Il capitolo 2 fornisce le conoscenze di background necessarie a capire il lavoro svolto; in particolare in questo capitolo viene introdotto inizialmente il concetto generale di blockchain, per poi entrare nel dettaglio e analizzare Ethereum, un particolare tipo di blockchain *permissionless* e pubblica; sono poi forniti i concetti sui sistemi di controllo degli accessi in standard XACML, ed è descritta la rispettiva implementazione su blockchain utilizzata in questo progetto; il capitolo 2 termina con la spiegazione di che cosa è e come funziona la tecnologia zero-knowledge.

Nel capitolo 3 viene presentato ad alto livello zk-ABAC, cioè il sistema di controllo degli accessi implementato su blockchain Ethereum esteso in modo tale da garantire la privacy degli attributi sensibili; sempre in questo capitolo verranno discussi possibili modelli di funzionamento basati su concetti differenti.

Il progetto prosegue entrando nel dettaglio del funzionamento del sistema nel capitolo 4: viene fornita una descrizione dettagliata di come sia stata implementata una Proof of Concept per i modelli discussi, e sono mostrati esempi di codice che simulano casistiche di base.

Nel capitolo 5 viene posta l'attenzione sulla valutazione del sistema: sono mostrate misurazioni, ragionamenti e stime sul funzionamento del sistema implementato. Dalle stime e dalle valutazioni fornite emergono alcuni problemi, in parte derivanti dalle scelte implementative, e in parte derivanti dalla tecnologia blockchain sottostante. Sempre nel capitolo 5 saranno vagliate possibili soluzioni ai problemi, mostrando come per alcuni si debba scendere a compromessi su sicurezza per garantire efficienza o viceversa, e proponendo l'uso di tecnologie alternative come le blockchain *permissioned*. Anticipando i risultati che saranno evidenziati in questo capitolo si noti come per garantire l'auditability, la verificabilità e la trasparenza del sistema di controllo degli accessi si debbano sacrificare le performance.

Infine il capitolo 6 presenta una panoramica finale del progetto, con riferimenti a lavori futuri interessanti da approfondire, come lo sviluppo di un parser, o l'adozione di tecnologie blockchain permissioned per implementare il sistema.

Anche se si sono evidenziate alcune problematiche in termini di costi e in termini di efficienza, i risultati a cui si è giunti nel progetto sono incoraggianti, pensando anche che le tecnologie di tipo zero-knowledge come ZoKrates sono estremamente nuove.

## Capitolo 2

# Stato dell'arte

Questo capitolo fornisce al lettore una breve introduzione sui concetti principali utilizzati o riferiti nel progetto. Per prima cosa verrà introdotto il concetto di **Blockchain**, una tecnologia recente nata nel 2008 da Satoshi Nakamoto [19]. Il capitolo continuerà con alcune nozioni su una piattaforma che, utilizzando proprio questa tecnologia, si pone l'obiettivo di creare una rete di esecuzione decentralizzata di applicazioni: Ethereum [6].

Successivamente si passerà all'approccio basato su Blockchain per la costruzione del **sistema di controllo degli accessi** riferito in [3]. Questo è il sistema che è stato alla base di questo tirocinio e che è stato esteso per consentire la privacy per gli attributi sensibili.

La parte finale di questo capitolo riguarderà un tool basato sulla tecnologia *zk-SNARKs*: **ZoKrates**. Tramite esso sarà possibile specificare programmi per generare prove off-chain e verificare la loro correttezza on-chain (su Ethereum). Per scrivere questa tipologia di programmi è necessaria un'approfondita conoscenza di nozioni matematiche e crittografiche, infatti il programma deve essere codificato in forma polinomiale, ma ZoKrates solleva il programmatore da questi passaggi complessi automatizzando tutto il processo [2].

### 2.1 Blockchain

Una blockchain è una struttura dati condivisa e immutabile: è molto simile ad un database online distribuito. La blockchain è simile ad un registro digitale le cui voci sono raggruppate in blocchi concatenati in modo cronologico a formare una lista. Il contenuto della blockchain è *condiviso*, perché tutti i nodi della rete possono accedervi, e *immutabile*, perché i vari record non possono essere modificati o eliminati a meno di invalidare l'intera struttura [19]. Inoltre la blockchain è una struttura dati decentralizzata perché non esiste un sistema centrale che permette o nega operazioni: saranno tutti i nodi della rete a farlo, tramite un meccanismo conosciuto come “*algoritmo di consenso*”. I due più famosi esempi di blockchain sono Bitcoin e Ethereum.

### 2.1.1 Ethereum

Ethereum è una piattaforma globale open-source per la creazione di applicazioni decentralizzate [6]. L'obiettivo di Ethereum è quello di creare una vasta rete di computer privati che eseguono software di terze parti in modo decentralizzato. Per fare ciò la piattaforma utilizza un linguaggio Turing-completo, con cui è possibile scrivere applicazioni che prendono il nome di Smart Contract, che saranno eseguite sui vari nodi della rete. Ethereum inoltre ha la sua criptovaluta specifica, chiamata Ether, che è possibile scambiare tramite le transazioni. L'imposizione di un limite massimo di consumo di gas sulle transazioni garantisce che i vari Smart Contract non eseguano operazioni troppo lunghe o cicli potenzialmente infiniti. I concetti fondamentali di Ethereum sono il mining, gli Smart Contract, i Messaggi, le Transazioni, e i costi in termini di gas/fee: ognuno di essi verrà presentato nei paragrafi successivi.

### 2.1.2 Miners

La definizione ufficiale di Ethereum riporta: *il mining è il processo di creazione di blocchi di transazioni da aggiungere alla blockchain Ethereum* [6]. I miners sono dunque dei computer che eseguono particolari software e che usano il loro tempo e la loro potenza di calcolo per processare transazioni e produrre blocchi per la blockchain [6].

Una definizione alternativa di mining è la seguente: *Il mining è un servizio di “record-keeping” (salvataggio dei record) effettuato tramite l’uso di potenza di calcolo. I miners mantengono consistente, completa e non alterabile la blockchain raggruppando ripetutamente le nuove transazioni all’interno di un blocco; il blocco viene quindi spedito al resto della rete per poter essere verificato dai nodi. Ogni blocco contiene una chiave crittografica del blocco precedente: in questo modo è possibile creare una lista di blocchi (da cui deriva il nome blockchain)* [21] .

Il motivo per il quale i miners esistono è che Ethereum, così come Bitcoin, si basa su un algoritmo di consenso di tipo PoW (Proof of Work) per la produzione dei blocchi. I miners sono quegli attori che competono nella produzione di nuovi blocchi, guadagnando come ricompensa criptovalute. *PoW* è un metodo per evitare che la produzione dei blocchi sia troppo economica: se così fosse infatti la blockchain sarebbe suscettibile allo “spam” di blocchi inutili da parte dei miners al solo fine di aggiudicarsi il relativo guadagno. Tramite l'adozione di PoW si richiede che la creazione di un blocco necessiti di una spesa in termini di potenza computazionale: a costo di una produzione lenta si evita il problema della creazione di blocchi inutili.

### 2.1.3 Costi: gas e fee

Come già anticipato prima l'esistenza di costi per l'esecuzione è fondamentale per la piattaforma Ethereum: in questo modo si evita che un'unica esecuzione monopolizzi i nodi della rete, e si evita l'esecuzione di codice che potenzialmente potrebbe non terminare mai. Il gas è definito come l'ammontare da pagare per eseguire una certa istruzione del bytecode Ethereum [11]. Il tasso di cambio tra gas e Ether varia nel tempo. L'utente

quando invia una transazione deve specificare il *gasLimit*, cioè la quantità massima di gas che è disposto ad utilizzare, e il *gasPrice*, cioè il costo unitario di gas espresso in Ether. In caso di superamento del *gasLimit* l'esecuzione viene terminata e l'utente deve comunque pagare l'intero costo del gas speso fino a quel punto. La commissione calcolata in base al gas consumato va ad aggiungersi alla ricompensa in Ether del nodo che produce il blocco [11].

#### 2.1.4 La memoria

Per quanto riguarda la struttura di Ethereum è utile puntualizzare come è gestita la memoria. Essa è suddivisa in tre diverse aree, tutte formate da slot di 256 bit (32 byte): [10]

- *Stack-Based*: contiene le istruzioni EVM (Ethereum Virtual Machine). È la memoria meno costosa, ma in termini di dimensioni è molto limitata. Le operazioni base di Ethereum (ADD, PUSH, etc...) agiscono su elementi dello stack.
- *Memory*: è la memoria locale a una certa transazione. Essa è sempre inizializzata a zero all'inizio della transazione. Anche se è considerata teoricamente infinita, nella pratica il suo utilizzo è limitato da un costo in gas proporzionale alle dimensioni dei dati da memorizzare.
- *Storage*: è la memoria destinata a contenere i dati persistenti tra le varie transazioni, che definiranno lo stato del contratto. Anche in questa memoria l'uso ha un costo, molto maggiore rispetto alla *Memory*.

L'utilizzo di memoria ha un costo ben preciso, che varia in base a quanti slot (32 byte) vengono utilizzati e in base a quale area di memoria viene scelta per la memorizzazione di tali dati. Nello *Yellow Paper* [22] si trovano i costi per la memoria storage, che è quella utilizzata per memorizzare i dati persistenti:

- $G_{sset}$ : è il costo dell'uso dell'istruzione SSTORE per settare uno slot di memoria dello storage (32 byte) da 0 a un valore non-zero. Esso vale 20.000 gas.
- $G_{sreset}$ : è il costo dell'uso dell'istruzione SSTORE che si paga quando si modifica uno slot di memoria dello storage (32 byte) da un valore non-zero a 0 o da un valore non-zero a sé stesso (scrittura del solito valore nello slot). Esso vale 5000 gas.
- $R_{sclear}$ : è il rimborso per settare uno slot dello storage (32 byte) da un valore non-zero a 0. Esso vale 15000 gas.

In generale possiamo dire che la memorizzazione nello storage di  $n$  byte non-zero, su un area di memoria inizializzata a 0 richiede un certo quantitativo di gas, che può essere ottenuto con la seguente formula:

$$GasCost = \lceil \frac{n}{32} \rceil \cdot 20000$$

<b>ZONE</b>	<b>EVM OPCODE</b>	<b>GAS/WORD</b>	<b>GAS/KB</b>	<b>GAS/MB</b>
STACK	POP	2	64	65,536
	PUSHX	3	96	98,304
	DUPX	3	96	98,304
	SWAPX	3	96	98,304
MEMORY	CALLDATACOPY	3	98	2,195,456
	CODECOPY	3	98	2,195,456
	EXTCODECOPY	3	98	2,195,456
	MLOAD	3	96	98,304
	MSTORE	3	98	2,195,456
	MSTORE8	3	98	2,195,456
STORAGE	SLOAD	200	6,400	6,553,600
	SSTORE	20,000	640,000	655,360,000

Figura 2.1: Tabella dei costi di memorizzazione nelle varie aree di memoria. [14]

### 2.1.5 Smart Contract

Gli smart contract sono degli *agenti autonomi* salvati sulla blockchain, creati tramite una transazione verso l'indirizzo 0. Possono essere considerati come dei programmi memorizzati e eseguiti sulla chain. Ogni contratto è identificato da un indirizzo univoco, possiede un certo ammontare di Ether, ha uno spazio di memoria privato, ha il suo stato, ed è associato a un determinato codice eseguibile (EVM code).

Per creare uno smart contract viene utilizzato un linguaggio ad alto livello, Solidity. L'invocazione del contratto con indirizzo  $y$  avviene quando un utente manda una transazione all'indirizzo  $y$ ; la transazione include tipicamente anche i dati di input e output, e un pagamento per l'esecuzione del codice.

### 2.1.6 Transazioni concorrenti: problemi

Una transazione è un pacchetto dati che contiene al suo interno un messaggio da mandare ad un account. Un messaggio è una transazione inviata da un contratto invece che da un utente o entità esterna.

Una transazione è composta da:

- *Payload*: il contenuto del messaggio.
- *Signature*: una firma del mittente, ottenuta tramite la chiave privata dello stesso, che lo identifica univocamente.
- *Ether da trasferire*: la quantità di Ether da trasferire al destinatario.
- *STARTGAS*: il massimo numero di step computazionali che la transazione può fare. E' il gasLimit, cioè la massima quantità di gas utilizzabile per la transazione.
- *GASPRICE*: il costo per un singolo step computazionale. E' la fee che il mittente deve pagare.

### Il concetto di concorrenza

Quando vengono create delle nuove transazioni che invocano contratti, esse vengono raggruppate in blocchi dai miners per essere successivamente eseguite. Le transazioni hanno un'esecuzione atomica, cioè esse o vengono eseguite interamente o non vengono eseguite. Uno scenario particolare è quello nel quale due transazioni arrivano nello stesso momento e vengono incluse nel solito blocco: a questo punto è il miner che decide quale transazione eseguire prima. Il problema non è banale, perché in certe situazioni le transazioni portano in stati diversi a seconda dell'ordine in cui vengono eseguite. [7]

Ecco un esempio nel quale l'ordine influenza sull'ether guadagnato a seguito di una certa esecuzione.

*Si supponga di avere un contratto che richiede la soluzione per un certo puzzle. Inizialmente il proprietario del contratto imposta a 10 il valore della variabile REWARD, che rappresenta la ricompensa per chi sottomette una soluzione valida, ma dispone anche di una funzione che permette di aggiornare questo valore arbitrariamente. Un certo utente, Alice, dopo aver risolto il puzzle sottomette una transazione  $T_i$  la soluzione, ma contemporaneamente l'owner del contratto crea una transazione  $T_j$  per cambiare il valore di REWARD e abbassarlo a 5. Le transazioni arrivano insieme nel blocco: sarà quindi il miner a decidere l'ordine. Se per prima fosse eseguita  $T_i$  il guadagno per Alice sarebbe di 10, in caso fosse eseguita prima  $T_j$  questo stesso guadagno sarebbe dimezzato [11].*

## 2.2 Attribute Based Access Control

Un “Attribute Based Access Control” (ABAC) [17] è un modello per sistemi di controllo degli accessi che utilizza politiche che specificano, tramite l'uso di attributi, chi o cosa può avere accesso a una determinata risorsa; questi sistemi possono utilizzare attributi pubblici, privati, o entrambi. Un utente può utilizzare o accedere la risorsa solo se dispone dei valori degli attributi che soddisfano le condizioni della politica. Un ABAC può definire autorizzazioni che esprimono condizioni su proprietà sia del soggetto (colui che vuole accedere), sia della risorsa. È utile dare una definizione di attributo: “*Un attributo è una caratteristica, predefinita e preassegnata da un'autorità, che definisce un aspetto specifico di un soggetto, di un oggetto, di un ambiente, o delle operazioni richieste [17]. Un attributo è pubblico quando la sua diffusione a qualsiasi entità non comporta problemi di privacy, privato altrimenti.*” Gli attributi usati nelle condizioni possono essere relativi ad un soggetto (attributi del “subject”), ad una risorsa (attributi della “resource”), o all’ambiente (attributi di “environment”).

### 2.2.1 ABAC e XACML

Un sistema ABAC che utilizza lo standard XACML di OASIS [13] è formato da diverse componenti che interagiscono tra di loro per valutare una politica e restituire un risultato all’entità che richiede l’accesso alla risorsa. Con XACML si definiscono la struttura del sistema, le politiche, le richieste di accesso, e le risposte. Le richieste sono utilizzate per

fornire all'ABAC gli ID di: chi vuole accedere alla risorsa, della risorsa stessa, e delle azioni che si vogliono compiere su di essa. Le risposte invece contengono la decisione, che può essere di quattro diversi tipi [3]:

- *Permit*: l'accesso alla risorsa è garantito.
- *Deny*: l'accesso alla risorsa è negato.
- *Indeterminate*: in caso errore nel processo di valutazione.
- *Not Applicable*: la richiesta non riguarda nessuna delle politiche conosciute.

Una politica è definita come un insieme di condizioni e relazioni che governano i comportamenti ammessi in una organizzazione, basandosi su privilegi del soggetto e su come le risorse o gli oggetti devono essere protetti; i *privilegi* (anche noti come diritti, autorizzazioni) rappresentano il comportamento autorizzato di un soggetto: sono definiti da un'autorità e sono inclusi in una politica [17]. La politica è tipicamente scritta dalla prospettiva dell'oggetto che deve essere protetto. Una politica XACML è formata da [3]:

- *Target*: un insieme di condizioni semplici che riguardano il soggetto, la risorsa e l'azione. Queste condizioni devono essere applicate alla richiesta di accesso.
- *Insieme di regole*: le regole sono formate da un target interno, relativo solamente all'attuale regola, e da una condizione. La condizione è una funzione booleana su combinazioni di funzioni che prendono in input attributi.

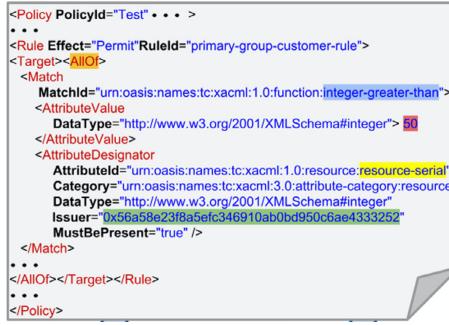


Figura 2.2: Struttura standard di una politica XACML [3].

Quando la politica viene eseguita, a seguito di una richiesta di accesso, i target e le regole sono valutati utilizzando gli attuali valori degli attributi, al fine di restituire una decisione per garantire o meno l'accesso alla risorsa; i valori restituiti saranno combinati tra di loro per creare il risultato finale [3].

### 2.2.2 Struttura e componenti standard XACML

Nella figura 2.3 viene mostrata l'architettura XACML di un sistema ABAC, che evidenzia le varie componenti e le loro interazioni.

- *Policy Enforcement Point (PEP)*: componente che si trova in diretta associazione con la risorsa da proteggere. Il suo compito è quello di intercettare le varie richieste di accesso verso la risorsa per effettuare la valutazione della politica. Il PEP avvia il processo di valutazione, chiamando in causa la giusta politica.
- *Policy Administration Point (PAP)*: si occupa della gestione delle politiche, e in particolare agisce da repository per esse, così che siano disponibili per essere recuperate quando necessario.
- *Attribute Managers (AMs)*: componenti che effettivamente gestiscono gli attributi dei soggetti, delle risorse e dell'ambiente. In Figura 2.3, gli AMs relativi agli attributi del soggetto sono indicati con “subjects”, gli AM relativi agli attributi della risorsa sono indicati con “resource”, mentre quelli relativi agli attributi che descrivono l’ambiente sono indicati con “environment”. Ogni applicazione utilizza l’insieme di AMs specifico per il suo scenario applicativo. Inoltre questo componente può far parte del sistema di controllo stesso, oppure può essere relativo a entità di terze parti, e può essere allocato sia nello stesso dominio del sistema di controllo degli accessi che in un dominio di terzi.
- *Policy Information Points (PIPs)*: dato che solitamente l’insieme di attributi necessari alla valutazione di una politica risiede su diversi AMs, ognuno con il suo protocollo o con un diverso metodo di interazione per il recupero dei valori, i PIPs agiscono da plug-in per tali AMs. Questi componenti forniscono un’interfaccia con ognuno degli AMs, permettendo il recupero e l’aggiornamento degli attributi.
- *Policy Decision Point (PDP)*: componente che serve per la valutazione vera e propria: prende come input la politica e i valori degli attributi, e restituisce in output la decisione.
- *Context Handler (CH)*: tutti i vari componenti vengono orchestrati dal context handler che si occupa di gestire il workflow dell’intero processo.

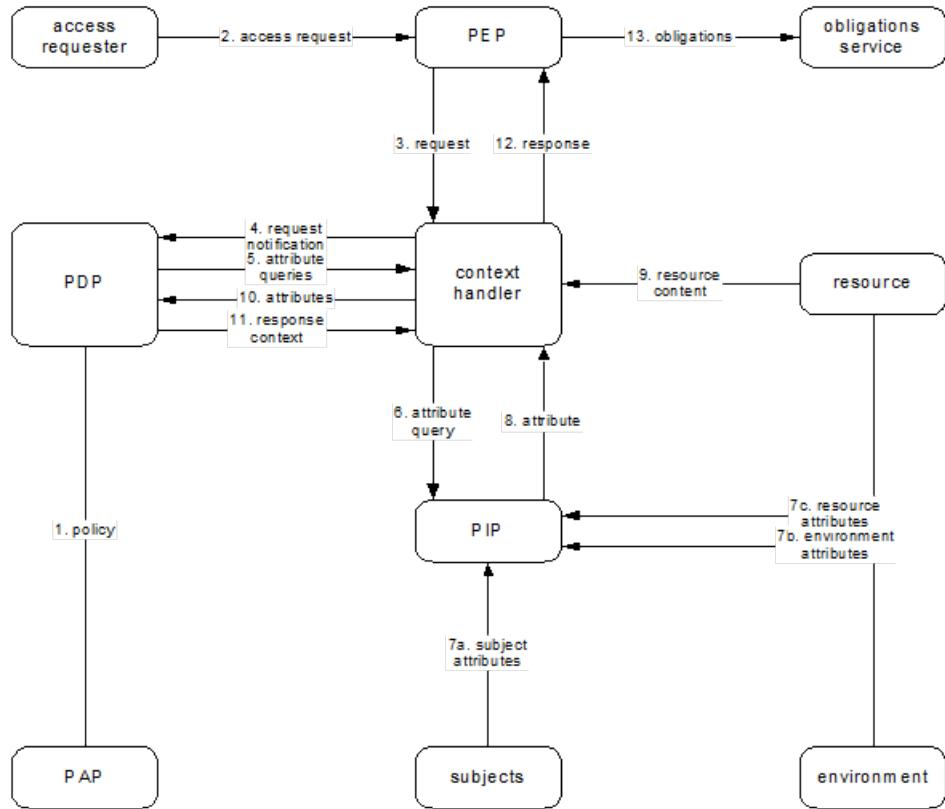


Figura 2.3: Architettura di riferimento OASIS XACML. [13]

## 2.3 Un Sistema di Access Control basato su Blockchain

Nel seguente capitolo viene preso in considerazione un sistema di controllo degli accessi basato su attributi e implementato utilizzando la tecnologia blockchain di Ethereum. Questo sistema viene proposto in [3] e nel progetto verrà riferito anche con l'abbreviazione “ACS” (Access Control System), che indicherà questa specifica implementazione tramite blockchain. Il sistema mantiene i concetti e lo standard di XACML con alcuni accorgimenti che gli permettono di sfruttare la tecnologia blockchain. Sfrutta sempre politiche basate su attributi, ed è implementato tramite smart contract memorizzati su Ethereum. In questo scenario le risorse da proteggere sono degli smart contract (o parti di essi, come ad esempio funzioni, strutture dati o variabili) e sono dunque poste on-chain.

La politica viene chiamata *Smart Policy*: è creata dal proprietario della risorsa e viene memorizzata sulla blockchain per sempre (la blockchain è immutabile). È possibile rimpiazzare logicamente una politica, ma quella precedente sarà comunque sempre presente e recuperabile. Oltre che alla memorizzazione, anche la valutazione della politica viene effettuata sulla blockchain. Dato inoltre che l'implementazione del sistema ABAC è su

blockchain, un qualsiasi utente che vuole accedere ad una risorsa protetta deve disporre di un “*wallet*” per poter pagare i costi delle transazioni in termini di gas e Ether [3].

Le varie componenti dell’ACS sono quelle standard di XACML, mostrate nella figura 2.3, ma adattate in modo tale da sfruttare la tecnologia blockchain:

- *Smart AMs*: in questo tipo di sistema gli attributi sono direttamente salvati sulla blockchain e sono gestiti da contratti. Gli smart contract che memorizzano e gestiscono gli attributi sono chiamati Smart AMs. Tramite la memorizzazione su blockchain gli attributi diventano inalterabili e pubblici [3].
- *Smart Policy*: le politiche hanno la stessa struttura di quelle utilizzate in qualsiasi sistema ABAC XACML, ma in questo caso esse vengono trasformate in smart contract. La creazione di una politica passa per tre fasi. La prima fase è quella di scrittura, dove il proprietario della risorsa utilizza il linguaggio XACML; la politica entra successivamente nella fase di traduzione, dove viene trasformata dal PTP (policy translation point, un modulo del PAP) in uno smart contract, chiamato “*Smart Policy*”. La smart policy non è il semplice risultato di una traduzione, in quanto contiene anche la logica per la valutazione della politica stessa: per ogni attributo riferito nella politica viene inserita una relativa funzione che permette il suo recupero tramite gli Smart AMs. Le Smart Policy sono pubbliche e immutabili, come qualsiasi altro contratto sulla chain: possono essere solamente “disabilitate”, ma non distrutte completamente [3].
- *PEP*: è integrato all’interno dello smart contract che rappresenta la risorsa da proteggere. In questo modo ogni richiesta di accesso o utilizzo dello smart contract protetto viene prima intercettata dal PEP che inizia il procedimento di valutazione: questo componente deve quindi già conoscere l’indirizzo del contratto relativo alla giusta smart policy che protegge la risorsa.
- *PAP*: si occupa di gestire il mapping tra risorse e relative smart policy per permettere la loro invocazione on-chain. Nell’attuale implementazione dell’ACS il PAP è un componente che si trova off-chain ed è utilizzato direttamente dal proprietario della risorsa, e comunica con la chain tramite il CH.
- *CH*: è rappresentato tramite due diverse componenti. La prima è il *CH<sub>D</sub>*, posizionato off-chain, che si occupa di gestire le comunicazioni tra PAP e blockchain: quando viene creata una nuova policy il *CH<sub>D</sub>* la riceve dal PAP e la invia sulla chain per farne il *deployment* tramite una transazione. La seconda componente è il *CH<sub>E</sub>*, che gestisce il workflow dei componenti on-chain.
- *PDP and PIPs*: i compiti che avevano questi due componenti sono integrati nella smart policy. Il processo decisionale (relativo al PDP) è implementato come una chiamata ad una certa funzione “*evaluate*”, mentre il processo di recupero attributi (relativo ai PIPs) è implementato tramite chiamate di funzioni degli Smart AMs.

### **2.3.1 Vantaggi: la trasparenza**

L'approccio basato su blockchain, usato per sviluppare l'ACS, apporta vantaggi e miglioramenti rispetto agli approcci tradizionali (server centrale). Il principale vantaggio è la trasparenza: su blockchain tutte le varie operazioni sono pubbliche e visibili ai nodi della rete, e non più sotto l'esclusivo controllo del proprietario della risorsa. Chiunque può accedere ai risultati o ai file di log che tengono traccia delle transazioni per effettuare verifiche o controlli: questo garantisce la verificabilità e *l'auditability* delle varie operazioni. Un utente che si vede negato l'accesso a una risorsa in modo fraudolento potrà utilizzare i dati salvati sulla chain, relativi alla sua richiesta, per dimostrare che in realtà possedeva i diritti di accedere. Inoltre la proprietà di immutabilità della blockchain garantisce una verificabilità a lungo termine: ad esempio le politiche memorizzate sulla chain possono essere recuperate anche dopo la loro revoca. Grazie alla proprietà della trasparenza, la blockchain può essere considerata un ambiente di esecuzione affidabile per le varie politiche. Nell'ACS preso in esame le risorse da proteggere sono esse stesse degli smart contract (o loro parti), e il PEP è posizionato sulla chain: questo garantisce che anche ignorare fraudolentemente le richieste di accesso diventi praticamente impossibile, in quanto l'utente potrebbe dimostrare l'effettiva ricezione della sua richiesta.

### **2.3.2 Punto critico: la mancanza di privacy**

Se da una parte l'utilizzo dell'ACS basato su blockchain garantisce trasparenza e verificabilità dei risultati, dall'altra introduce limiti e potenziali minacce alla privacy degli utenti. Tutte le transazioni e tutti gli oggetti che vengono salvati sulla blockchain sono visibili pubblicamente a qualsiasi nodo, così come i parametri di una certa richiesta di accesso. Tutti gli attributi utilizzati per un accesso a una risorsa protetta saranno resi pubblici: una grande limitazione se venissero utilizzate politiche che comprendono la valutazione di dati sensibili quali stipendio, numeri di telefono, indirizzi, etc... L'ACS utilizza solamente attributi pubblici, e non considera quindi il problema della privacy, ma l'utilizzo di informazioni private è fondamentale per creare politiche più precise, complete o più complesse. Data la natura pubblica di Ethereum, il problema non può essere affrontato on-chain, e si deve ricorrere a tecniche che mascherino e proteggano i dati. A questo scopo è possibile utilizzare diversi approcci: una prima idea potrebbe essere quella di cifrare i dati, ma questa soluzione non funzionerebbe molto bene perché sulla chain un numero consistente di nodi miners avrebbe comunque bisogno di decifrare i dati per effettuare la valutazione delle politiche, e il problema iniziale persisterebbe. In questo progetto, per risolvere il problema della privacy, verrà utilizzata la tecnologia zero-knowledge, basata su tecniche crittografiche.

## **2.4 Zero-knowledge**

Le blockchain sono costruite in modo da essere completamente pubbliche: tutte le informazioni che transitano o che vengono salvate sulla chain sono accessibili da tutti i nodi

che fanno parte della rete [2]. Questo può essere visto come punto di forza se si pensa alla verificabilità dei risultati o dei procedimenti on-chain, ma allo stesso tempo crea anche un grave problema riguardante la privacy. Attualmente la quasi totalità delle applicazioni o dei software richiede un certo livello di privacy, perché gli utenti non vogliono esporre i loro dati sensibili. L'adozione della tecnologia blockchain in questo campo, quindi, è molto limitata dalla trasparenza dei dati. È necessario un sistema che renda possibile il mantenimento di livelli di privacy anche sulla chain: problemi critici sono ad esempio nascondere il link tra indirizzi di account e identità reali, passaggio di dati sensibili tramite transazioni etc. Per l'ACS basato su blockchain sarebbe interessante utilizzare attributi privati: un utente che vuole accedere una risorsa protetta vorrebbe non dover rivelare alcune informazioni sensibili.

L'adozione di tecnologie basate sulle *zero-knowledge proofs*, come ad esempio *zk-SNARKs*, si propone di aumentare la privacy in sistemi aperti come le blockchain. Con questi meccanismi è possibile verificare la correttezza di una computazione senza dover eseguire la computazione stessa, e in particolare senza la necessità di conoscere i parametri. [2]

#### 2.4.1 Background matematico: le curve ellittiche

In questa sezione viene fornita al lettore una brevissima panoramica del background matematico relativo alle tecnologie delle zero-knowledge proofs. Data la grande complessità dell'argomento, il lettore verrà introdotto solamente ai principali concetti espressi ad alto livello.

Le curve ellittiche furono proposte da Victor Miller e Neal Koblitz [1] come standard per la crittografia già nel 1985. La robustezza di questo metodo è garantita dalla difficoltà di calcolo del logaritmo discreto.

Una curva ellittica è definita come l'insieme dei punti  $(x,y)$  che soddisfano l'equazione:

$$C : y^2 = x^3 + ax + b$$

La curva non deve essere singolare, cioè non deve avere cuspidi o intersezioni con se stessa; per verificare questa proprietà deve essere soddisfatta la seguente equazione:

$$\Delta = -16(4a^3 + 27b^2)$$

$$\Delta \neq 0$$

Sia  $P$  un punto qualsiasi, e  $O$  il punto all'infinito (l'elemento neutro o l'identità), allora le principali operazioni sulle curve ellittiche sono:

- $P + O = P$
- $O + O = O$
- $P + (-P) = O$
- $k \cdot P = R$

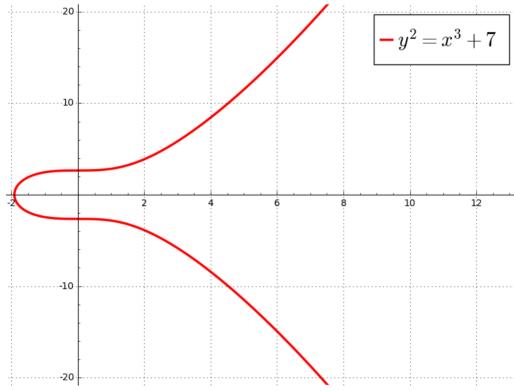


Figura 2.4: Curva ellittica rappresentata sui reali

### Logaritmo discreto su curve ellittiche

La crittografia a curve ellittiche si basa sulla difficoltà di risolvere il logaritmo discreto.

Il problema del Logaritmo Discreto su curva ellittica (ECDLP) è definito nel seguente modo [9]:

*data una curva ellittica  $\xi$  definita su un campo finito  $\Gamma_q$ , un punto  $P \in \xi(\Gamma_q)$  di ordine  $n^1$  e un punto  $Q \in \langle P \rangle$ , trovare l'intero  $k \in 0, \dots, n-1$  tale che  $Q = kP$ .*

*k viene chiamato logaritmo discreto di Q in base P e viene indicato come  $k = \log_P(Q)$ .* [9]

- Conoscendo  $P$  e  $k$  è computazionalmente facile calcolare  $Q$ .
- Conoscendo  $Q$  e  $P$  è computazionalmente difficile trovare  $k$ .

### Crittografia omomorfica

Per finire viene accennato il concetto di tecniche di crittografia omomorfica.

**Def.** La crittografia omomorfica è un tipo di crittografia basata su tecniche che permettono la manipolazione di dati cifrati. [20]

Per comprendere meglio di cosa si tratta viene fornito un semplice esempio che spiega bene il concetto alla base di queste tecniche.

Si supponga di avere due numeri  $A$  e  $B$ , che vengono cifrati tramite crittografia omomorfica rispettivamente in  $X$  e  $Y$ . È possibile, avendo utilizzando questa tecnica, calcolare il risultato cifrato di  $A + B$  semplicemente calcolando  $X + Y$ .

### 2.4.2 zk-SNARKs

Prima di introdurre il tool ZoKrates viene introdotta la tecnologia su cui esso si basa. La tecnologia zk-SNARKs ha molti campi di applicazione e la sua funzionalità principale è quella di permettere di verificare la correttezza di una computazione senza doverla eseguire e quindi senza conoscere i dati di input [2].

Viene illustrato che cosa è e come funziona zk-SNARKs partendo dal significato dell'acronimo.

- **zk - Zero Knowledge:** il verifier non conosce alcuna informazione sulla computazione o sui suoi dati. Egli conoscerà solamente se la proof (prova della computazione) è valida o meno. In particolare il verifier non conosce nulla del *witness* (oggetto che sarà introdotto nel quinto punto di questa lista) [2] [1].
- **S - Succinct:** la dimensione della proof è molto piccola, anche per computazioni grandi o complesse. Questa proprietà è fondamentale se messa in relazione con le performance sulla blockchain: la verifica può essere fatta sempre in modo molto veloce ed efficiente [2] [1].
- **N - Non interactive:** non c'è interazione tra prover e verifier, se non per l'invio della prova. Esiste una fase iniziale di setup. La proof è un singolo messaggio inviato da prover a verifier che permette di effettuare la verifica senza dover scambiare ulteriori messaggi. L'assenza di interazioni tra verificatore e prover aumenta le performance generali [2] [1].
- **AR - Arguments:** grazie alle proprietà di *Completeness* e *Soundness* (introdotte nel paragrafo seguente) il verifier è protetto da prover disonesti. Dato che le proof rispecchiano una computazione “*true*” (cioè corretta), creare proof incorrecte è computazionalmente molto difficile, e al contrario crearne corrette è molto semplice [1].
- **K - Knowledge:** per il prover è impossibile creare delle proof senza conoscere il *witness*, cioè una serie di input validi su cui viene eseguita la computazione. È importante tenere segreto il witness, perché comprenderà le informazioni sensibili (gli input privati) che si vogliono nascondere. Grazie alla proprietà di Zero-Knowledge il verificatore non conosce il witness ed esso non può essere ricavato neppure dall'analisi della proof.

### Zero-knowledge proofs

In astratto una zero-knowledge proof -ZKP- (letteralmente “prova a conoscenza zero”), può essere definita come un metodo crittografico di interazioni tra due entità che vogliono scambiarsi informazioni [1]. In particolare le due entità sono un *prover*, colui che fornisce una proof (prova), e un *verifier*, colui che verificherà la correttezza della proof tramite un apposito programma. Il punto fondamentale di questi meccanismi è che il verifier non può ottenere alcuna informazione dalla proof che gli viene fornita: essa infatti attesta solamente la corretta esecuzione di una computazione senza rivelare ulteriori dettagli.

Ci sono tre proprietà che ogni ZKP deve soddisfare [1]:

1. *Completeness:* se il prover è onesto e fornisce una proof valida, il verifier sarà sempre in grado di attestarne la correttezza.

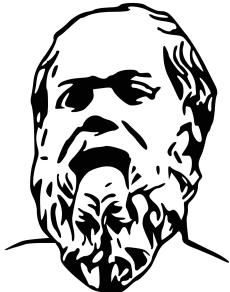
2. *Soundness*: se il prover è disonesto e tenta di ingannare fornendo una proof non valida, il verifier, *con un'alta probabilità*, non accetterà la proof.
3. *Zero-Knowledge*: è la proprietà fondamentale su cui si basa il metodo crittografico. Se il prover fornisce una proof corretta, il verifier esaminandola non potrà ottenere alcuna informazione oltre alla correttezza della computazione.

### L'implementazione della tecnologia

In questa sezione viene discusso come zk-SNARKs venga effettivamente implementato. Ci sono 4 punti alla base della tecnologia [2]:

1. *Succinctness*: avere delle prove di dimensioni molto ridotte anche per problemi di grosse dimensioni è possibile grazie alla riduzione del problema a una semplice moltiplicazione e uguaglianza di polinomi, del tipo  $t(x)h(x) = w(x)v(x)$ , per un punto  $x$  casuale.
2. *Encoding*: per verificare la computazione di un certo programma, esso viene ridotto ad un'uguaglianza tra polinomi, che garantisce la proprietà di “Succinctness”. In questi termini il prover dovrà convincere il verifier di essere a conoscenza di punto che soddisfi l'uguaglianza. Per effettuare l'encoding sono necessarie varie trasformazioni del problema. Esso viene trasformato in un R1CS [4] (Rank-1 Constraint System, un sistema di vincoli matematici), del quale il verifier deve verificare tutti i vincoli. Dato che queste verifiche sono molto costose, è possibile trasformare il R1CS in un QAP [4] (Quadratic Arithmetic Program), dove il controllo può essere effettuato su un singolo vincolo.
3. *Encryption*: viene utilizzata una funzione crittografica *Omomorfica*, sia questa  $\xi$ , per criptare i dati. Grazie alle proprietà della crittografia omomorfica essa può operare su dati cifrati senza doverli decifrare e ottenendo gli stessi risultati ottenuti usando i dati in *plain-text*.
4. *Zero-Knowledge*: il trasferimento dei dati e la creazione della proof avvengono tramite un modello a zero-knowledge. Il prover deve dimostrare al verifier di conoscere certi valori (rappresentati dal witness) che soddisfano delle proprietà, senza rivelarli. I parametri del witness devono rimanere nascosti e privati.

### 2.4.3 Zokrates



Zokrates [23] è un tool open source che permette di semplificare e automatizzare tutta la fase di definizione, setup, e esportazione di un verificatore Solidity che sia in grado di accettare o rifiutare prove di computazione di specifici programmi. Il tool usa schemi di verifica di tipo zk-SNARK.

L'uso del tool è particolarmente semplice, infatti una volta scaricato e installato, tramite comandi da CLI è possibile effettuare le fasi necessarie per arrivare a creare un verificatore e una proof valida per esso.

#### Programmazione in ZoKrates

Il linguaggio di programmazione ZoKrates è un semplice linguaggio di tipo imperativo e comprende i comandi base per il controllo del flusso, gli operatori aritmetici comuni e di confronto, gli operatori logici AND, OR e XOR, e la definizione di struct per creare tipi complessi.

La struttura di un programma ZoKrates è pressoché standard: è sempre presente una funzione principale, chiamata “main”, ed è possibile definire al suo esterno o al suo interno funzioni accessorie. Tra i parametri del *main* si deve specificare, tramite le apposite keyword del linguaggio, quali sono quelli pubblici (“public”) e quelli privati (“private”), che ZoKrates tratterà in modo diverso: i valori dei primi saranno memorizzati in chiaro nei file di output, mentre i valori dei secondi rimarranno nascosti. Nei programmi ZoKrates è fondamentale l'utilizzo della keyword “*assert*”: essa asserisce una certa proprietà su una variabile, che, se non rispettata, provoca la terminazione con errore dell'esecuzione. Tutti gli input del programma (cioè gli input della funzione “main”) devono avere su di essi almeno una condizione imposta tramite “*assert*”, altrimenti il compilatore restituirà un errore.

Sono ammessi solo tipi numerici nel linguaggi di programmazione: i programmi devono basarsi su operazioni aritmetiche, logiche o di confronto su tali tipi (non sono, ad esempio, ammesse stringhe). I tipi delle variabili definiti nel linguaggio sono due: interi senza segno nel range  $[0, 2^n]$  con  $n = 8/16/32$ , o interi senza segno nel range  $[0, p]$ , dove  $p$  è un numero primo molto grande. Il secondo metodo è generalmente più efficiente, per questo tutti i programmi utilizzati negli esempi fanno uso di tale tipo di interi; l'utilizzo di interi con valori da 0 a  $2^n$  viene consigliato solo se sono necessarie operazioni binarie tra numeri.

#### L'architettura di ZoKrates

Per poter capire le fasi del processo di creazione della proof e del suo verificatore viene descritta l'architettura di ZoKrates, i suoi componenti e il loro funzionamento.

- *Compiler*: il compilatore è formato dal parser e dal flattener. Esso prende come input il programma scritto in linguaggio ZoKrates e lo trasforma nel codice ”flat-

tened”, cioè un codice composto da una serie di definizioni di variabili e asserzioni. Si utilizza questo tipo codice perché è facilmente e efficientemente traducibile nelle astrazioni utilizzate dagli schemi di verifica, come ad esempio R1CS o QAP per quanto riguarda zk-SNARKs. [4]

- *Witness-Generator*: l’obiettivo di un prover è quello di generare una prova che attesti il possesso di una soluzione (corretti valori dei parametri di input) per il programma. Il prover deve trovare un witness, cioè una computazione corretta del programma, ottenuta utilizzando input corretti. Si richiama il concetto di witness già visto precedentemente: esso è un *assegnamento di valori a variabili che soddisfa i vincoli di un certo flattened-code specifico* [4]. Il witness-generator è il componente che prende gli input pubblici e privati per un programma e li usa per generare un witness, eseguendo il flattened-code.
- *Contract-Generator*: questo componente ha il compito di creare un programma Solidity che accetti proof e che ne verifichi la correttezza restituendo un risultato booleano. Il programma creato è un contratto utilizzabile su Ethereum.

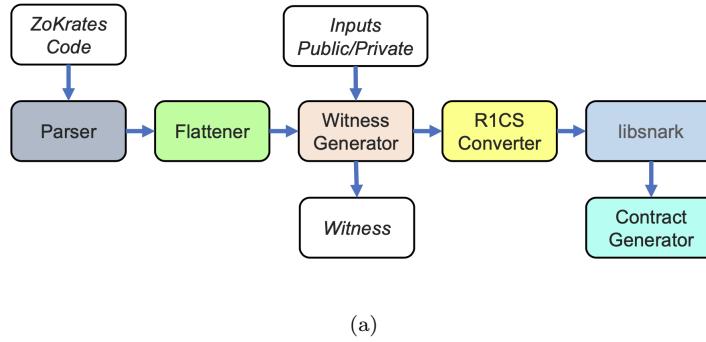


Figura 2.5: Architettura ZoKrates [4]

## ZoKrates: un esempio di utilizzo

Per illustrare i vari procedimenti nel funzionamento di ZoKrates, dalla creazione di un verificatore alla creazione e verifica di una prova, il lettore consideri due entità: Peggy e Victor. Peggy è un prover, cioè un utente che vuole dimostrare a Victor, che è un verifier, di possedere certi attributi che soddisfano certe proprietà. Lo scopo di Peggy è quello di convincere Victor senza rivelargli alcun dettaglio sui valori degli attributi. Il seguente esempio, ripreso pagina GitHub di ZoKrates, illustra le varie operazioni che le due entità devono eseguire [24].

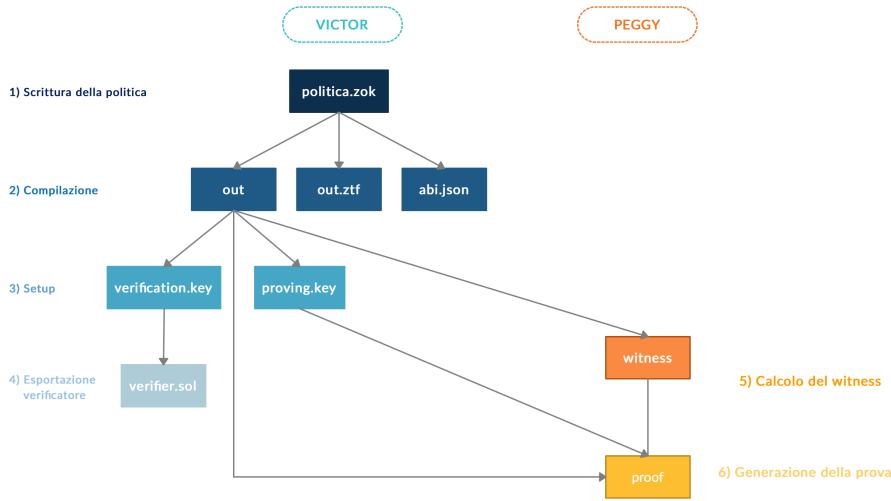


Figura 2.6: Fasi di utilizzo di ZoKrates. Nelle tonalità di blu sono rappresentate le operazioni che svolge Victor, mentre nelle tonalità arancio sono rappresentate le operazioni di Peggy. Le frecce esprimono la dipendenza tra file: ad esempio il file “proof” necessita dei file “out”, “proving.key”, “witness” per essere generato.

### VICTOR:

1. *Definizione del programma:* Victor scrive in ZoKrates un programma che prende in input i valori di alcuni attributi. Il programma in questione è da considerare come un insieme di regole o condizioni sugli input, e sarà salvato con estensione “.zok”.
2. *Compilazione:* una volta scritto il programma Victor lo compila tramite il comando:

```
zokrates compile -i nomeFile.zok
```

e ottiene così tre diversi file: “out”, “out.ztf”, “abi.json”. Il file “out” e “out.ztf” sono, rispettivamente, il compilato in binario e il compilato in *human-readable-code* [1]. “abi.json” invece è un file json che riporta per ogni input del programma il tipo e la visibilità (pubblica o privata).

3. *Fase di Setup:* Victor esegue la fase di setup eseguendo, all'interno della cartella con i file precedentemente creati, il comando:

```
zokrates setup
```

Questa fase genera due chiavi: *proving.key* e *verification.key*. La chiave di prova serve a Peggy per poter creare una proof per lo specifico programma, mentre la chiave di verifica serve a Victor per generare il contratto in linguaggio Solidity che verificherà le prove.

4. *Esportazione del Verificatore:* Victor crea il verificatore in linguaggio Solidity da usare sulla blockchain Ethereum per verificare le varie proof che verranno sottomesse al contratto utilizzando il seguente comando:

```
zokrates export-verifier
```

#### PEGGY:

1. *Recupero del programma:* Peggy deve avere a disposizione o recuperare il programma “.zok” che aveva creato Victor o, alternativamente, il suo compilato, cioè il file “out”. Nel primo caso Peggy stessa dovrà compilare il file “.zok”.
2. *Generazione del witness:* Peggy deve eseguire il programma con i giusti input (supponendo che li conosca) e generare il witness della computazione, che attesta la sua conoscenza della soluzione al problema. Per fare ciò Peggy esegue il comando:

```
zokrates compute-witness -a input1 input2 ... inputn
```

3. *Creazione della Proof:* una volta ottenuto il witness Peggy deve generare una proof da inviare a Victor. Per fare ciò ha bisogno che Victor le invii o le fornisca un modo di recuperare la chiave di prova (*proving.key*), necessaria in questa fase. Una volta recuperata la chiave può procedere eseguendo il seguente comando nella stessa cartella dove si trovano il witness e il programma compilato, per generare la proof:

```
zokrates generate-proof
```

Viene generato il file *proof.json* che rappresenta la proof vera e propria. Nel file inoltre sono anche specificati gli input pubblici utilizzati per il generare il witness; non sono invece riportati gli input che erano stati etichettati come privati.

## Capitolo 3

# Privacy-Based ABAC su blockchain Ethereum

Il problema che viene affrontato in questo capitolo è quello della definizione di meccanismi per riuscire a garantire la privacy di attributi privati del soggetto nell’implementazione su blockchain dell’ACS descritto nella sezione 2.3. L’uso di questo sistema basato su attributi e su blockchain apporta grandi vantaggi, quali la verificabilità delle transazioni e la verificabilità delle politiche: un utente può sempre controllare le varie operazioni eseguite al fine di riscontrare la correttezza dei procedimenti. In questo modo si limitano le azioni fraudolente, come ad esempio negare arbitrariamente l’accesso a un entità legittima. Tuttavia, questa soluzione potrebbe introdurre problemi di privacy nel caso in cui si utilizzassero attributi privati, come descritto in dettaglio in seguito.

L’ACS preso in esame in 2.3 utilizza politiche che vengono codificate in smart contract, distribuiti sulla blockchain, e generate tramite un parser. Tali politiche sono valutate usando valori di attributi recuperabili tramite i PIPs e gli AMs; il sistema è stato ideato per lavorare con attributi pubblici, che possono essere salvati direttamente sulla chain senza alcun problema di privacy. Questo tirocinio affronta il problema di introdurre l’utilizzo di politiche che prevedono anche *attributi privati*: in questo modo il potere espressivo delle politiche aumenta, ed è possibile imporre nuovi e più precisi controlli di accesso. Lo svantaggio di utilizzare gli attributi privati è la necessità di introdurre un meccanismo per garantire la privacy. In letteratura sono presenti varie soluzioni, più o meno efficienti, per risolvere questo problema: quella che verrà studiata e analizzata nelle seguenti sezioni riguarda l’uso della tecnologia *zero-knowledge* tramite il tool *ZoKrates* che la implementa.

Il capitolo inizierà analizzando il comportamento di ZoKrates nel dettaglio, per capire se esso sia effettivamente integrabile nell’ACS. Da queste analisi sono sorte alcune problematiche interessanti in relazione ai costi di Ethereum: saranno proposte soluzioni o approcci che possano risolverle. Dopo aver parlato dei punti deboli e dei limiti del tool basato su zk-SNARKs, verrà analizzato il sistema di controllo di accessi e il suo workflow:

saranno esaminati quindi alcuni modelli astratti per poter implementare l'utilizzo degli attributi privati. I modelli principali per l'ACS basato su zero-knowledge prendono in considerazione l'uso di richieste e operazioni “*sincrone*” o l'uso di *dimostrazioni di interesse* “*asincrone*”. Nei paragrafi seguenti sarà esposto il lavoro condotto, soffermandosi sulle problematiche e le relative soluzioni.

### 3.1 Integrazione di ZoKrates ed ACS: introduzione

In questa sezione vengono descritte le potenzialità di ZoKrates e discussa l'integrazione del suo funzionamento nell'ACS, al fine di garantire la privacy degli attributi ritenuti sensibili.

#### Uso di ZoKrates per la definizione di politiche “privacy preserving”

Come spiegato nella sezione 2.2.1, una politica ha una determinata struttura: è composta da regole, ognuna delle quali contiene condizioni. Quando, in questo capitolo, parleremo di “scrittura di politiche” in realtà faremo riferimento alla scrittura di condizioni “privacy preserving” che andranno a comporre una politica. ZoKrates nasce proprio come linguaggio di programmazione ideato per scrivere programmi composti da condizioni, come ad esempio puzzle o giochi matematici con soluzioni numeriche. Scrivere in ZoKrates le condizioni che definiscono una politica è estremamente semplice: è sufficiente utilizzare il comando assert per specificare uno o più vincoli sui parametri del programma, che rappresentano gli attributi coinvolti nella politica. È possibile imporre tutti i più comuni vincoli sugli input, come ad esempio quelli di uguaglianza e di confronto; inoltre è anche possibile combinare tra loro più vincoli tramite gli operatori logici AND, OR e XOR.

Di seguito è riportato un esempio di politica che regola l'accesso a una risorsa in base allo stipendio dei dipendenti (gli utenti che vogliono accedere la risorsa). L'attributo “*sal*” rappresenta il salario ed è un'informazione privata sensibile, da non diffondere o rendere pubblica: per questo viene etichettato come “*private*”. Al contrario il salario di un amministratore è un dato di pubblico dominio: non c'è necessità che venga nascosto, e infatti viene etichettato come *public*.

```

//sal_a: salary
//role: role, 1=clerk, 2=supervisor, 3=admin

//This is a simple policy that allows users to access a resource:
// 1) if user is a clerk, then his salary has to be
//     lower than admin's salary
// 2) if user is a supervisor, then his salary has to
//     be lower than admin's
//     salary, but higher than 1500 (medium clerk salary)
// 3) if user is a admin, then his salary has to be the
//     admin's salary

def main(
    private field sal,
    private field role,
    public field sal_admin
):
    assert(role==1 && sal<sal_admin)
    assert(role==2 && sal<sal_admin && sal>1500)
    assert(role==3 && sal==sal_admin)

    return

```

### Un protocollo per l'integrazione con l'ACS

In questa sezione viene definito ad alto livello un protocollo astratto per integrare il tool ZoKrates con il sistema di controllo degli accessi implementato su blockchain.

Gli attori principali di questo protocollo sono Victor, il verifier, e Peggy, il prover. Victor sarà un nuovo componente aggiuntivo che dovrà far parte dell'ACS, mentre Peggy verrà rappresentata da un nodo off-chain: esso avrà il compito di creare la proof, e memorizzarla sulla blockchain. L'AM è il componente che meglio si presta a rappresentare Peggy. Nell'ACS riferito nella sezione 2.3 queste componenti erano on-chain perché si supponeva che tutti gli attributi fossero pubblici. Gli AMs che gestiscono attributi privati devono però essere spostati off-chain: saranno entità, anche di terze parti, che comunicheranno con il resto del sistema e che opereranno da “oracoli” per il recupero di attributi privati e file. Anche se vengono definiti off-chain, i nodi in questione avranno un’interfaccia verso la blockchain, che permetterà loro di comunicare con il resto del sistema.

Definiti questi ruoli si passa all’analisi delle varie fasi del protocollo per l’accesso a una risorsa protetta, facendo riferimento all’uso di politiche con attributi privati.

1. *Fase di Setup*: in questa fase il proprietario della risorsa da proteggere scrive una politica in linguaggio ZoKrates, la compila, ottiene i vari file, ottiene un verificatore in linguaggio Solidity e ne effettua il deploy sulla blockchain: da questo momento è possibile verificare su Ethereum le proof per quella politica.
2. *Richiesta di Accesso*: un utente che vuole accedere a una risorsa protetta manda una richiesta di accesso, che deve essere intercettata dal PEP per poter iniziare la valutazione della specifica politica. Dalla richiesta il sistema deve essere in grado di recuperare l'identità dell'utente e la politica interessata: per questo nella richiesta si suppongono inclusi degli identificativi appositi.
3. *Creazione della proof*: una volta recuperate le informazioni citate sopra può partire una comunicazione verso la parte off-chain che richiede la creazione di una proof per la politica e relativa agli attributi dell'utente.
4. *Recupero file e attributi*: il nodo off-chain deve recuperare i file che servono per creare la proof della specifica politica: questi sono il file “proving.key” e il file “out” (o il file “politica.zok”). Il nodo provvede quindi a cercare i valori degli attributi pubblici e privati dell'utente con cui crea il witness, necessario per la creazione della proof. Infine, una volta creata, la proof viene memorizzata sulla blockchain.
5. *Verifica*: sulla chain si effettua la verifica della proof tramite il contratto verificatore appropriato, utilizzando la funzione “*verifyTx*”, che restituirà un risultato true o false. Il risultato attesterà se le condizioni della politica sono state rispettate o meno, e verrà utilizzato per valutare la politica (che potrebbe contenere anche altre condizioni).

### 3.2 Integrazione di ZoKrates nell'ACS: la parte off-chain

Il punto di forza delle zero-knowledge proofs che utilizzano schemi di tipo zk-SNARKs risiede nella proprietà “*Succinctness*”: essa garantisce che le prove generate siano sempre di dimensioni ridotte. Sono proprio le piccole dimensioni delle prove, unite alla loro struttura standard, che rendono la verifica un'operazione molto veloce e poco costosa, adatta all'esecuzione sulla blockchain.

Il modello proposto nel corso di questo progetto utilizza uno Smart Contract al quale l'utente fa una richiesta di accesso (proprio come accadeva nell'ACS discusso nella sezione 2.3), e un nodo off-chain, dotato di un'interfaccia on-chain, che si occupa delle operazioni più pesanti in termini di costi computazionali e/o di memorizzazione, e che avrà il compito di produrre le prove a conoscenza zero tramite il tool ZoKrates. Questa sezione analizza due modelli possibili per l'organizzazione della parte off-chain. Il nodo off-chain, oltre a mantenere gli attributi privati degli utenti, dovrà anche memorizzare i file necessari per creare una proof: è proprio questo nodo a ricoprire il ruolo di “Peggy”, cioè il generatore e fornitore di prove. Gli attributi privati, che il nodo mantiene memorizzati sono utilizzati esclusivamente per la creazione della proof e non saranno mai

inviai in chiaro su Ethereum. Per la memorizzazione dei file necessari al tool ZoKrates si distinguono due scelte: mantenere sul nodo il file “*out*”, cioè il compilato del programma, o il file *programma.zok*, cioè il programma da compilare. Mantenendo *programma.zok* si limita l’uso della memoria, ma si aumenta il tempo richiesto per la creazione della proof, dato che la compilazione andrà eseguita ogni volta. Un utilizzo maggiore di memoria per mantenere il file “*out*” risulta essere la scelta migliore per eliminare i tempi di compilazione (che per politiche complesse potrebbero essere lunghi): questo è ragionevole anche perché ogni volta che l’ACS deve valutare una politica è necessario creare una nuova proof, utilizzando i compilati. Compilare i file ogni volta sarebbe dunque uno spreco di tempo, eliminabile al costo di un maggiore uso di memoria. Per quanto riguarda i file, anche in questo caso abbiamo due diverse opzioni, entrambe valide. Un primo approccio vede la memorizzazione sul nodo off-chain dei vari file, più precisamente di un insieme di file per ogni diversa politica. Un secondo approccio invece vede l’utilizzo di verificatori che garantiscono solo delle operazioni standard con cui comporre operazioni logiche più complesse. In ogni caso, dato il funzionamento di ZoKrates, descritto nella sezione 2.4.3, i file salvati off-chain (*proving.key* e *out*), specifici per una politica, dovranno essere recuperabili in coppia, perché sono entrambi necessari e indispensabili. È importante notare come ad ogni politica sia associato un suo specifico verificatore: è necessario che il nodo sappia individuare quali file utilizzare per creare la proof, altrimenti essa non potrà essere considerata valida dallo Smart Contract verificatore. Per fare ciò si può associare un identificatore (ad esempio un hash dell’indirizzo del richiedente e del numero progressivo della richiesta) alla politica, e associare i vari file online in una mappa chiave-valore tramite quell’identificatore: in questo modo la ricerca sarà efficace e veloce. Lo Smart Contract on-chain dovrà anche conoscere a quale nodo inoltrare la richiesta di creazione proof: ci saranno più nodi off-chain e la richiesta deve essere indirizzata a quello che possiede gli attributi privati dell’utente che vuole accedere.

Nelle seguenti due sezioni vengono discussi due diversi approcci per l’implementazione della computazione off-chain. La parte off-chain sarà composta dal nodo off-chain e dalla sua interfaccia verso Ethereum.

### 3.2.1 Primo modello: un verificatore per ogni politica

Il primo modello è molto immediato: la parte off-chain ha a disposizione tutti i file ZoKrates necessari per la creazione delle proof. In particolare, per ogni diversa politica, la parte off-chain ha un insieme di file, e sulla chain è presente uno specifico verificatore. In questo modo se la parte on-chain richiedesse una proof per una certa politica, quella proof verrebbe generata tramite i suoi file e inviata quando disponibile: viene memorizzata ed utilizzata sulla blockchain una sola proof per richiesta. Il contratto che richiedeva una proof potrà verificare quest’ultima ottenendo direttamente il risultato finale, senza dover eseguire ulteriori manipolazioni. I vantaggi di questo approccio sono la velocità di computazione delle proof e l’immediatezza nell’utilizzo della proof stessa. Inoltre, per ogni politica, è memorizzata sulla chain una singola proof. C’è però un punto critico che sta nel numero di file generati e nel numero di contratti di cui si deve fare il deploy. Per

ogni politica, che differisce dalle altre anche solo per una condizione di confronto in più o in meno, si devono salvare sul nodo off-chain i rispettivi file, associandoli con la parte on-chain tramite degli identificatori appositi. Inoltre su Ethereum sarà necessario, per ognuna di queste politiche, effettuare il deploy di un diverso smart contract “verifier.sol”. Infine un altro punto critico è l’uso stesso dell’oracolo (AM): esso è un entità con interfaccia on-chain ma di natura off-chain, che deve conoscere le associazioni tra politiche e file. Utilizzando questo approccio, ogni AM dovrebbe conoscere i file associati a tutte le politiche presenti per poter creare proof al momento della richiesta.

### 3.2.2 Secondo modello: oracoli modulari

Il secondo modello nasce da un’idea completamente diversa: esso si basa sul concetto di definire degli oracoli “modulari” e “universali”. Ricordando che gli oracoli sono entità di terze parti, che in un caso reale potrebbero essere forniti da un’azienda esterna, la modularità rappresenta sicuramente un grande vantaggio, così come l’universalità. In questo modello gli oracoli gestiscono solamente pochi file ZoKrates; in particolare essi possono generare proof solamente per operazioni base: il confronto tra due oggetti (numerici) mediante operazioni di maggiore, minore, uguaglianza o disuguaglianza. Queste sono le uniche operazioni per le quali un oracolo può generare proof, perché esso memorizzerà i file ZoKrates solamente di programmi che implementano tali funzionalità (*ad esempio:  $x > y$ ,  $x < y$ ,  $x = y$ ,  $x! = y$ , con  $x$  e  $y$  input dei programmi*).

Cambia anche la modalità di interazione della politica con l’oracolo: ogni politica è rappresentata da un contratto, che sa come sono formate le varie condizioni. Non verranno inviate all’oracolo le richieste di proof per la politica nel suo intero o per la condizione, ma saranno richieste proof singole di una parte della politica. Le proof ricevute saranno verificate e i loro risultati saranno manipolati tramite operatori logici per ottenere il risultato finale. Il concetto è estremamente semplice da capire se si considera il seguente esempio:

```

sal: salario
role: ruolo in azienda (1: admin, 2: supervisore)

```

Si consideri la seguente politica:  
 $(\text{sal} > 3000 \ \&\& \text{role} == 1) \ || \ (\text{sal} == 2000 \ \&\& \text{role} == 2)$

Lo smart contract non chiederà la valutazione dell'intera politica all'oracolo, ma effettuerà diverse richieste, componendo quindi le risposte:

```

proof R1 = op_maggiore(sal, 3000)
proof R2 = op_uguale(role, 1)
proof R3 = op_uguale(sal, 2000)
proof R4 = op_uguale(role, 2)

boolean a = verifyTx(R1)
boolean b = verifyTx(R2)
boolean c = verifyTx(R3)
boolean d = verifyTx(R4)

resultTOT = (a && b) || (c && d)

```

Come è possibile vedere dall'esempio, il contratto che rappresenta la logica dell'ACS su blockchain, che conosce perfettamente come è composta la politica, invia all'oracolo delle richieste di proof per le parti della politica. In queste richieste saranno incluse anche altre informazioni come l'identità dell'utente, per poter recuperare i valori degli attributi, e gli identificativi per associare le proof alla richiesta. L'oracolo riceverà le richieste, e utilizzerà i file ZoKrates relativi alle operazioni base per creare le proof. Infine il contratto dell'ACS verifica i risultati "parziali" e successivamente li compone propriamente tramite operatori logici per ottenere il risultato vero e proprio. I vantaggi sono evidenti: sulla chain, per quanto riguarda i verificatori, viene effettuato il deploy solamente di  $n$  contratti (dove  $n$  è il numero di operazioni elementari definite), mentre sull'oracolo si salvano solamente i file di  $n$  programmi ZoKrates. Un risparmio considerevole in termini di spazio, perché saranno sufficienti questi pochi file per poter verificare tutte le possibili politiche. Non si perde di espressività, in quanto ZoKrates stesso ammetteva solo operazioni base (maggiore, uguale etc...). Con questo modello gli oracoli vengono definiti in modo universale e modulare: la loro integrazione diventa più semplice con il resto del sistema. Ci sono però anche importanti aspetti negativi di questo approccio, in primis la gestione delle richieste e il salvataggio delle proof. Mentre nel modello definito nella sezione 3.2.1 per una richiesta era generata una singola proof, adesso per valutare una singola politica saranno create più proof: potrebbe non essere semplice ritrovare le proof specifiche per una certa

politica, per questo si dovrebbero mantenere delle strutture dati aggiuntive con le giuste associazioni proof-politica che identifichino quale parte della politica una certa proof sta rappresentando. Tutto ciò è possibile al prezzo di un maggiore uso della memoria e di una logica più complessa lato ACS su blockchain. Inoltre il punto critico è il salvataggio delle proof per garantire auditability: per ogni politica dovranno essere memorizzate più proof. Ad esempio, una politica con 10 condizioni potrebbe arrivare ad avere 10 proof da salvare (ogni proof ha una dimensione di 256 byte). Questo potrebbe diventare una grande spesa se si pensa al costo di utilizzo della memoria *Storage* di Ethereum. Nella proof of concept mostrata nel capitolo 4 utilizzeremo il modello con un verificatore per ogni politica.

### 3.3 Interazione tra ACS e Prover

Vengono proposti due diversi modelli di interazione tra ACS implementato su blockchain, riferito in 2.3, e parte off-chain (il nodo che assume il ruolo di Prover): un “*modello sincrono*” e un “*modello asincrono*”. Nelle relative sezioni verrà spiegato il motivo di tale denominazione. Entrambi si basano sull’interazione tra tre diverse entità: l’utente, uno smart contract che rappresenta l’ACS, e il nodo off-chain. Con il termine “Smart Contract” nel paragrafo viene inteso un contratto che regola le operazioni delle componenti introdotte nel punto 2.3. Può essere utile al lettore avere anche un’idea di come internamente si svolgono le transazioni, le richieste e le risposte, per questo viene proposto uno schema di funzionamento che riporta tutte le varie componenti (relativo solo al “modello sincrono”).

#### 3.3.1 Modello Sincrono

Il termine *sincrono* deriva dal fatto che questo modello fa riferimento ad un modello di interazione in stile “domanda e risposta”: l’utente aspetta una risposta (esito) alla richiesta di accesso e lo smart contract aspetta una proof in risposta alla richiesta verso il nodo off-chain. Questo modello implica una condizione fondamentale da garantire: dato che l’utente sta aspettando un risultato, il tempo di attesa deve essere accettabile, perché in una situazione reale l’utente sta cercando di accedere una risorsa, e richiederà di attendere il minor tempo possibile. Vengono di seguito introdotte le interazioni tra le varie componenti dell’ACS, elencate nella sezione 2.3.

Come già precisato, il modello qui descritto utilizza l’approccio con un verificatore per ogni politica per la parte off-chain, descritto nel paragrafo 3.2.1, e il modello di interazione “sincrono” per gestire le varie richieste. Nello schema innanzitutto sono evidenziati tre diversi colori per le interazioni: essi rappresentano tre diverse transazioni ideali, la prima (in verde) dall’utente verso lo smart contract è la richiesta di accesso, la seconda (in azzurro) è la richiesta di creazione proof inviata dallo smart contract al nodo off-chain, e la terza (in arancione) è il processo di verifica della proof e di comunicazione del risultato. C’è da precisare che le diverse operazioni passano attraverso il Context Handler (CH), che non è riportato nella figura per semplicità: esso agisce da orchestratore e fa comunicare tra

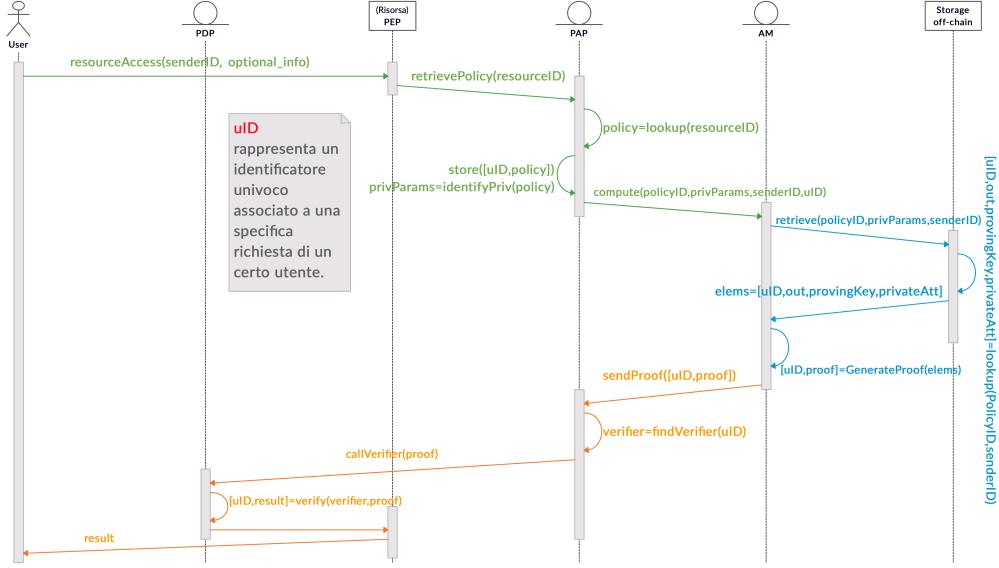


Figura 3.1: Interazioni ad alto livello del modello sincrono. PEP, PDP e AM sono entità on-chain (nello specifico in questa figura AM fa riferimento all’interfaccia verso Ethereum dell’Attribute Manager), mentre il PAP è una componente off-chain. Lo schema omette il componente CH per semplicità, ma è da tenere presente che tutte le varie interazioni passano attraverso di esso.

di loro le varie entità. Inoltre nel diagramma in figura 3.1 PIP e AM sono rappresentati, sempre per semplicità, come una sola entità: ogni accesso agli attributi è comunque sempre mediato da un PIP che gestisce la comunicazione con il relativo AM. Infine si noti che nel modello mostrato non viene considerato il caso in cui un utente voglia fornire personalmente degli attributi al momento della richiesta: gli unici attributi considerati sono eventualmente quelli pubblici, già sulla blockchain, e quelli privati, memorizzati in un dispositivo off-chain.

Viene fornita al lettore una breve spiegazione dello schema in figura 3.1, che riporta l’architettura di zk-ABAC con riferimento alle componenti di un ACS classico:

1. L’utente effettua una richiesta di accesso ad una risorsa protetta, inviando la sua identità (rappresentata da *senderID*) e, se necessario, altre informazioni aggiuntive. La richiesta è un tentativo di accesso alla risorsa che viene intercettato dal PEP, dando inizio al processo di verifica.
2. Il PEP, intercettata la richiesta, comunica con il PAP, che effettuerà il recupero della giusta politica (codificata come smart contract) specifica per la risorsa (a sua volta identificata da *resourceID*). Al termine della ricerca si crea un *uID*, cioè un identificatore univoco creato sulla base dell’identità utente e della richiesta stessa,

per tenere traccia della richiesta e per creare un'associazione tra proof che verrà creata e richiesta stessa. uID identificherà una richiesta di un particolare utente.

3. Vengono successivamente identificati i parametri privati necessari alla politica, e si richiede all'AM di recuperare i valori di quei parametri per l'utente e di creare una proof.
4. Sul nodo off-chain a seguito della richiesta vengono recuperati i valori degli attributi privati dell'utente e i file necessari alla generazione della proof, che una volta pronta viene associata all'uID, per "ricordare" a quale richiesta essa corrisponda.
5. Il nodo off-chain invia all'AM la coppia uID-proof, tramite la quale sarà selezionato il giusto contratto verificatore.
6. Infine la proof viene valutata sul PDP, che invierà l'esito al PEP, unico componente in grado di comunicare con l'utente. Il PEP trasmetterà l'esito della richiesta di accesso all'utente.

Ricordando che i vari componenti dell'ACS, cioè PDP, PAP, AM etc., saranno rappresentati tutti da un singolo smart contract, possiamo vedere questo modello sincrono anche con lo schema in figura 3.2, meno dettagliato ma più corrispondente alla realtà.

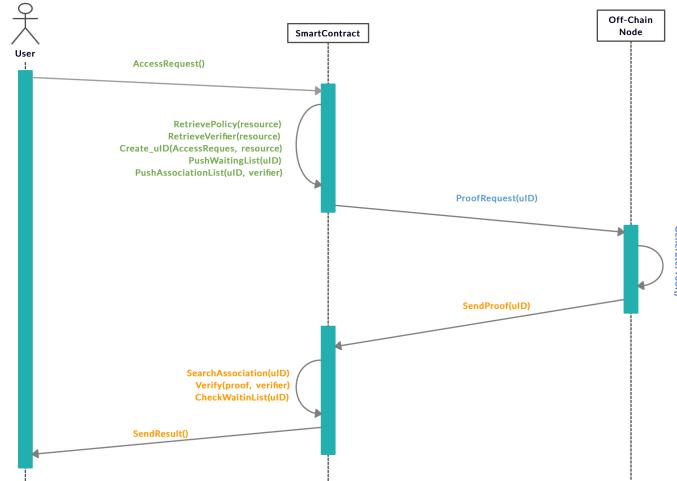


Figura 3.2: Diagramma del workflow con componenti del sistema rappresentate come unico Smart Contract. Il diagramma è equivalente a quello in figura 3.1. L'off-chain node avrà "un'interfaccia" verso la chain, chiamata "Oracolo", che permetterà le varie interazioni con il sistema.

Il modello presentato in questa sezione introduce alcuni problemi e limitazioni:

- Quanto deve attendere al massimo un utente per la risposta? Questo tempo potrebbe anche dipendere dal carico presente sulla blockchain, quindi quale è un valore massimo di tempo da attendere significativo?
- Quanto attende lo smart contract per ricevere una proof dal nodo-off chain? In questo lasso di tempo il contratto sarebbe “fermo”, dato che aspetterebbe i risultati, e non potrebbe eseguire ulteriori operazioni.

### **Il problema della concorrenza delle transazioni**

Il modello “sincrono” appena presentato soffre del problema di concorrenza tra transazioni diverse. Per risolvere tale problema è stata necessaria l’introduzione dell’identificatore per le richieste “*uID*”. Facendo un confronto con l’ACS descritto in 2.3, esso non soffriva di questo problema perché esisteva un’unica transazione, che, essendo atomica, veniva eseguita correttamente e interamente in ogni caso. Per utilizzare gli attributi privati si rende necessaria l’adozione di tecniche di computazione off-chain (la generazione della proof non può avvenire su Ethereum), che implicano la divisione del flusso di lavoro in più transazioni. In particolare, nel modello presentato in questa sezione, sono presenti 3 diverse transazioni. Senza l’uso dell’identificatore per le richieste, sarebbe impossibile riuscire a risalire all’associazione proof-richiesta e si creerebbero situazioni inconsistenti: le prove relative a una determinata richiesta potrebbero essere utilizzate per verificare l’accesso di un altro utente e compromettere la sicurezza del sistema. L’identificatore per le richieste risolve i problemi di concorrenza, eliminando le possibilità che le prove e le richieste non corrispondano. Nel capitolo 4 saranno forniti ulteriori dettagli sulla creazione e sulla gestione dell’identificatore univoco per le richieste.

### **Il problema della sincronizzazione delle componenti**

Uno dei problemi fondamentali è quello dell’attesa dei risultati, sia da parte dell’utente che da parte dello Smart Contract. Questo problema nasce proprio dalla natura sincrona di questo modello. Questo overhead è presente anche nei sistemi di controllo degli accessi tradizionali (non implementati su blockchain e senza zero-knowledge), ma assume rilevanza in questo caso specifico perché la creazione e la verifica della prova richiedono molti passaggi, alcuni dei quali svolti off-chain.

Una soluzione potrebbe essere quella di considerare dei timeout per le richieste dell’utente e dello Smart Contract. L’utente dovrebbe quindi aspettare una risposta fino allo scadere di un certo timer *wait\_time*, e considerare negativo l’esito della richiesta allo scadere di quest’ultimo. Persiste comunque un ulteriore problema, cioè il fatto che l’utente non sappia se la sua richiesta sia stata persa, se la generazione delle proof abbia impiegato troppo tempo, o se la comunicazione del risultato non sia avvenuta entro i limiti temporali stabiliti. L’utente potrebbe essere costretto a continuare a richiedere l’accesso ripetutamente.

Il problema del tempo di attesa è presente anche dal lato dello Smart Contract: esso deve sapere quanto tempo è passato dalla richiesta di un utente, per capire se, in caso

avesse la risposta, contattarlo o interrompere il procedimento senza ulteriori operazioni. Un’idea potrebbe essere quella di far mantenere allo smart contract una lista di utenti in attesa, controllata e aggiornata periodicamente per togliere le richieste che si sono “esaurite” per il timer o per l’effettiva riuscita del processo decisionale. Questa lista non risolverebbe comunque completamente il problema: potrebbe arrivare un’altra richiesta di accesso ed essere considerata prima rispetto a quella del primo utente (ricordiamo la concorrenza delle transazioni 2.1.6), e far esaurire quindi il timer del primo utente. La scelta della durata del timer, inoltre, è una decisione molto complessa: dovrebbe essere effettuata in modo empirico tramite esperimenti per valutare sperimentalmente il comportamento del sistema. Anche nello Smart contract è presente un ulteriore problema di temporizzazione, che riguarda l’interazione con il nodo off-chain: in particolare il problema è relativo a quanto deve aspettare il contratto prima di decidere che la prova è andata persa o che il nodo non riesce a crearla. Si può introdurre un’ulteriore timer anche per questo aspetto, ma ci sono molte variabili da considerare, quali il carico di lavoro che caratterizza il nodo off-chain in quel determinato momento, o la complessità della politica. Oltre a questi problemi va ricordato che sulla blockchain anche l’uso della memoria ha un costo, e mantenere liste di attesa, modificarle, e aggiornarle sono tutte operazioni che vengono pagate in termini di gas o fee.

### Il problema della memorizzazione

Questo problema si ripresenterà anche con il “modello asincrono”, ed è da tenere in considerazione, in quanto riguarda le spese, in termini di gas o Ether (e quindi soldi) che si è disposti a sostenere per il controllo di un accesso. Nel modello sincrono va mantenuta traccia sia degli utenti in attesa, che delle associazioni richiesta-proof, con tutti i costi che essi comportano. Salvare una proof sarà fondamentale se si vuole garantire l’auditability nel sistema, mentre vedremo nella prossima sezione che il modello asincrono consente di evitare la memorizzazione degli utenti in attesa.

### 3.3.2 Modello Asincrono

Il secondo modello pensato per il nuovo sistema di controllo degli accessi viene definito asincrono. Si basa su un “protocollo” abbastanza comune nell’ambiente blockchain, basato su “*dimostrazioni di interesse*”: piuttosto che effettuare una richiesta e attendere la risposta, si sottoscrive l’interesse relativo a una certa operazione, e, successivamente, in modo periodico o dopo un certo intervallo di tempo, si controlla la presenza dei risultati di tale operazione.

Una dimostrazione di interesse è un messaggio verso un’entità che comunica l’interesse nell’avviare una certa operazione e nell’ottenerne i relativi risultati.

Di seguito viene esposto uno schema di funzionamento del modello:

1. L’utente invia allo SmartContract una dimostrazione di interesse per l’accesso a una risorsa protetta, non attende di ricevere i risultati, ma sa che dopo un certo periodo di tempo la risposta potrebbe essere disponibile. La dimostrazione di interesse

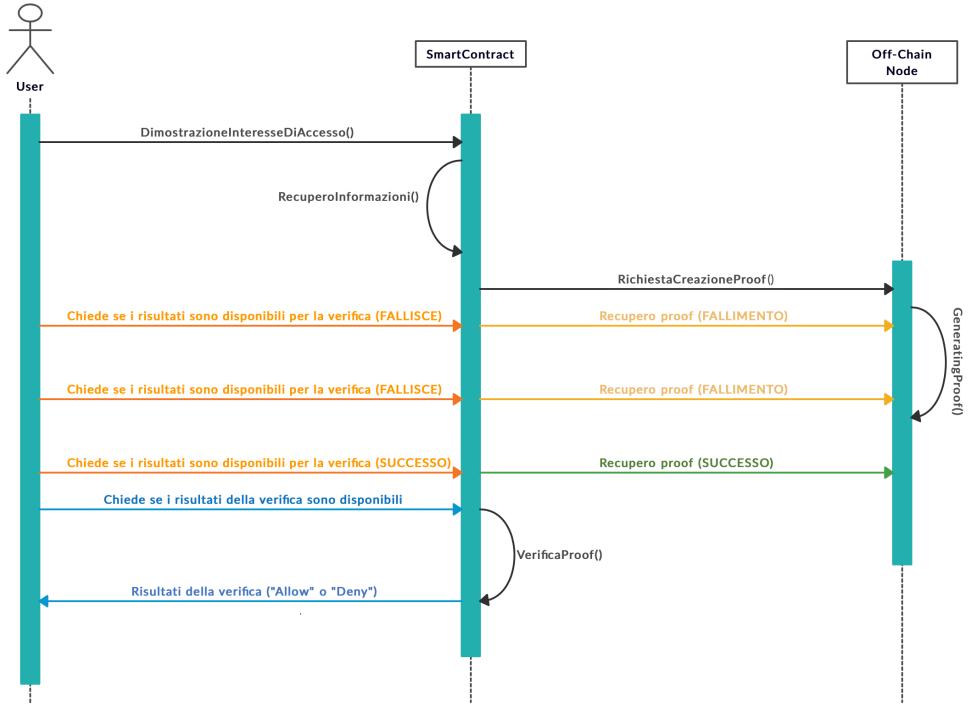


Figura 3.3: Diagramma del workflow del modello asincrono.

avviene chiamando una funzione dello SmartContract, che restituisce all’utente l’identificatore univoco della sua richiesta: tale oggetto servirà in seguito per le altre operazioni e per verificare la disponibilità dei risultati.

2. Lo SmartContract riceve la dimostrazione di interesse, effettua le operazioni che permettono di creare un identificatore univoco per la richiesta; una volta generato tale identificatore lo SmartContract lo invia all’utente. Il contratto a questo punto invia una richiesta (è anche questa una dimostrazione di interesse) all’oracolo, cioè all’interfaccia sulla blockchain del “nodo off-chain”, per richiedere la creazione di una proof relativa all’utente e alla politica che protegge la risorsa da accedere. Il contratto non attende la risposta del nodo, ma sa che la sua richiesta è stata presa in carico.
3. Il compito di verificare se i risultati sono pronti è dell’utente: è lui che può chiamare due diverse funzioni dello SmartContract, una per verificare la presenza della proof sulla blockchain, e una per recuperare il risultato della valutazione della proof. L’utente utilizza la prima funzione per comunicare allo SmartContract di contattare l’oracolo per richiedere una proof: ne ottiene una valida solo quando tutte le operazioni off-chain si sono concluse. Tramite la seconda funzione l’utente comunica allo

SmartContract di verificare la prova, se presente; lo SmartContract a seguito della verifica trasmetterà il risultato finale all'utente.

4. Nella figura: le frecce arancioni rappresentano l'utente che chiede se i risultati sono disponibili per essere valutati, le frecce gialle evidenziano le conseguenti azioni dello SmartContract che tenta di recuperare tali risultati dall'oracolo, e le frecce blu rappresentano l'utente che richiede il risultato finale.

Il modello così proposto introduce un grande miglioramento: è l'utente a decidere quanto tempo spendere in “attesa” di una risposta e quindi quando terminare l’interrogazione dello SmartContract. Inoltre lo SmartContract, una volta inviata la dimostrazione di interesse all’Oracolo, può continuare a svolgere altre operazioni, senza dover rimanere in attesa.

## Capitolo 4

# zk-ABAC: Implementazione

In questo capitolo verrà mostrato lo Smart Contract per la simulazione dell'ACS con garanzie di privacy, descrivendone dettagliatamente tutti gli aspetti, i passaggi, le operazioni e i compromessi che sono stati adottati.

Il modello di interazione scelto tra le proposte viste nel paragrafo 3.3 per sviluppare gli Smart Contract è quello che considera operazioni asincrone: esso è molto più intuitivo e facile da implementare su blockchain, oltre che essere più efficiente. Per l'interazione con la parte off-chain è stato scelto il modello dove ad ogni singola politica è associato un diverso verificatore e dei diversi file ZoKrates (modello descritto nella sezione 3.2.1).

Il capitolo sarà diviso in sezioni, ognuna delle quali descriverà una parte dei contratti implementati: come sono stati creati, le operazioni, le interazioni, e il relativo codice. Sarà quindi mostrata una semplice simulazione del sistema, nella quale un utente tenta di accedere a una risorsa protetta.

### 4.1 Tool utilizzati: Remix

Per la scrittura e la simulazione dei contratti e dell'esempio sono stati utilizzati due tool: ZoKrates, già descritto nel capitolo di background 2.4.3, e l'IDE Remix. Remix è l'IDE ufficiale di Ethereum per la scrittura, il deploy e la simulazione di contratti: esso è online [15] e gratuito ed è dotato di moltissime funzioni. Il tool è utilizzabile via web: nella pagina iniziale, sulla sinistra, una barra di navigazione permette di spostarsi tra le varie sezioni; quelle utili al progetto sono state: File Explorer, Solidity Compiler, Deploy & Run. In *File Explorer* è possibile vedere tutti i file presenti e salvati dall'utente, creare cartelle, creare o rinominare o spostare file. Tra più sessioni di utilizzo il File Explorer mantiene i file anche se viene chiuso il browser. Le altre due sezioni, *Solidity Compiler* e *Deploy & Run* servono rispettivamente per compilare i file Solidity, per effettuare il deploy dei contratti, e per utilizzare i contratti una volta depositati su Ethereum. Scrivere programmi Solidity è molto semplice grazie a questo IDE, che provvede anche a fornire un'interfaccia per la scrittura dei file “.sol”, evidenziando automaticamente le keyword o

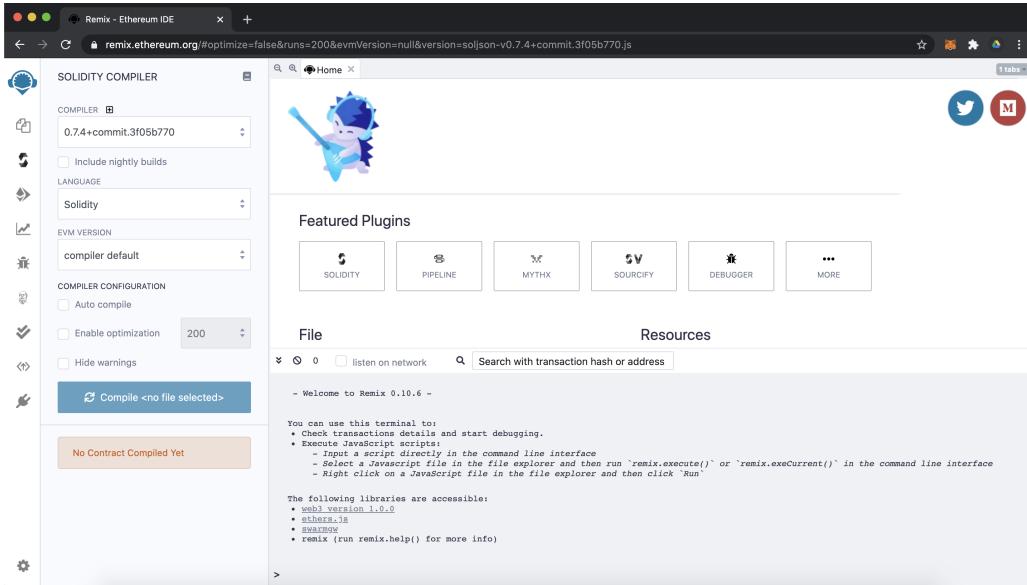


Figura 4.1: Schermata iniziale dell’IDE Remix su browser Google Chrome. Sulla sinistra è possibile vedere le icone delle varie sezioni.

gli errori. Nella sezione *Deploy & Run* è presente un terminale di comando tramite il quale è possibile analizzare gli input, i risultati, l’esito e i vari costi delle azioni effettuate sulla chain, come la chiamata di funzioni dei contratti. È possibile utilizzare più account con “address” diversi e impostare un costo personalizzato per l’uso di gas.

Remix è stato scelto anche per un’altra funzione interessante: è possibile scaricare e installare un pacchetto aggiuntivo per l’uso del tool ZoKrates. Il pacchetto funziona molto bene, è intuitivo e tramite pulsanti e textbox automatizza tutto il procedimento di generazione del verificatore e della proof in ZoKrates. Per utilizzare questo pacchetto è necessario, per prima, cosa scrivere un programma con estensione “.zok”; una volta creato il file, navigando nella sezione ZoKrates dalla barra laterale, è possibile effettuare la compilazione, la generazione witness inserendo gli input nelle apposite textbox, la generazione della proof, e la generazione del verificatore “.sol”. Remix velocizza tutto il procedimento di scrittura di contratti che usano zero-knowledge.

## 4.2 La scrittura della politica di sicurezza

Il primo file da creare è quello relativo alla politica di sicurezza (per la precisione, relativo ad alcune condizioni di una ipotetica politica di sicurezza) che proteggerà una risorsa. La politica deve essere scritta in linguaggio ZoKrates per poter sfruttare la tecnologia delle zero-knowledge, e successivamente dovrà essere compilata per generare i vari file.

```

1 //ruolo:
2 //      1-admin
3 //      2-supervisore
4 //      3-impiegato
5
6 def main(private field guadagno, private field ruolo):
7     assert(ruolo == 1 || ruolo == 2 || ruolo == 3)
8     assert(guadagno > 0)
9     assert((ruolo == 1 && guadagno > 2000) || (ruolo == 2 && guadagno > 1500) || (ruolo == 3 && guadagno < 1500))
10    return

```

Figura 4.2: Politica ZoKrates

La politica nella figura 4.2 è quella utilizzata dai contratti che implementano il sistema ACS nell'esempio che stiamo fornendo. Il “ruolo” potrebbe essere un attributo non necessariamente privato ma viene comunque considerato come tale; “guadagno” sicuramente lo è: questa informazione non deve essere pubblica. Le prime 4 righe del programma ZoKrates sono dei commenti che descrivono la corrispondenza tra un numero e un ruolo: in ZoKrates infatti non si possono utilizzare stringhe. In riga 6 è possibile vedere la definizione del metodo “main” e l'uso della keyword “private” per etichettare i campi ruolo e guadagno, che sono gli input privati della politica. Dato che i campi sono privati, è assicurato che i valori passati al programma non saranno visibili o deducibili dalla proof o dai file generati. Le righe 7, 8 e 9 sono gli *assert* della politica, cioè i vincoli logici che devono essere rispettati per la creazione di una proof valida: i primi due assert sono dei “controlli sugli input”, mentre il terzo è la condizione vera e propria. Usando questo programma l'accesso viene garantito a tre diverse categorie di utenti: amministratori con un guadagno maggiore di 2000€, supervisori con guadagno maggiore di 1500€ e impiegati con guadagno inferiore a 1500€. Infine il codice si conclude con la keyword *return*, che fa terminare la funzione main correttamente.

### 4.2.1 Creazione e compilazione dei file

Il file “.zok” deve quindi passare attraverso le varie fasi del processo di ZoKrates illustrate nella sezione 2.4.3: compilazione, setup, esportazione verificatore. Queste operazioni sono state svolte sempre su Remix, con il pacchetto specifico di ZoKrates.Terminate queste operazioni il *resource owner* avrà ottenuto il contratto verificatore *verifier.sol*, di cui effettuare il deploy sulla chain, e altri file, tra cui: il programma compilato “out”, la chiave di prova *proving.key* e la chiave di verifica *verification.key*. Tali file andranno resi disponibili anche agli oracoli, a cui servono per creare le proof. L'idea è quella di mettere online i file, salvati su una repository condivisa, a cui tutti gli oracoli possono attingere: per fare ciò è necessario inoltre assegnare ad ogni politica un suo identificativo, in modo tale che per una richiesta di creazione proof un oracolo riesca a recuperare i corrispondenti file.

### 4.2.2 Il verificatore per le prove: *verifier.sol*

Una volta effettuati tutti i passaggi di ZoKrates si otterrà il programma per la verifica delle prove della nostra politica, cioè *verifier.sol*. Questo programma viene generato

automaticamente da ZoKrates ed ha una struttura di base sempre uguale per tutte le politiche. Ottenuto il verificatore, il contratto in esso contenuto, “*Verifier*”, deve essere registrato su Ethereum pagando un costo di deploy in termini di gas. Il contratto sulla blockchain avrà un suo indirizzo, necessario per poter utilizzare le sue funzioni, e sarà visibile dagli altri nodi della rete. Quello che interessa del file “*Verifier.sol*” è una singola funzione, interna al contratto “*Verifier*” che permetterà di ottenere un risultato booleano in base alla validità di una prova. L’analisi della funzione *verifyTx(...)* mette in evidenza la minima dimensione della proof da salvare sulla chain: questa funzione infatti prende in input una prova per effettuarne la verifica, considerandola come 8 interi (memorizzati nei vettori *a*, *b*, *c*) da 32 byte ciascuno (in Solidity *uint256*), come è possibile vedere nella firma della funzione stessa. Da questo si deduce che per salvare sulla chain una proof è sufficiente salvare  $32 \text{ byte} \cdot 8 = 256$  byte, per una spesa di circa 160000 gas.

```

583     function verifyTx(
584         uint[2] memory a,
585         uint[2][2] memory b,
586         uint[2] memory c
587     ) public view returns (bool r) {
588     Proof memory proof;
589     proof.a = Pairing.G1Point(a[0], a[1]);
590     proof.b = Pairing.G2Point([b[0][0], b[0][1]], [b[1][0], b[1][1]]);
591     proof.c = Pairing.G1Point(c[0], c[1]);
592     uint[] memory inputValues = new uint[](0);
593
594     if (verify(inputValues, proof) == 0) {
595         return true;
596     } else {
597         return false;
598     }
599 }
600 }
```

Figura 4.3: Codice della funzione *verifyTx(...)* del contratto *Verifier* presente nel file *Verifier.sol*

Dalla figura 4.3 è possibile vedere che la funzione accetta in input una prova, o meglio le sue componenti *a*, *b*, *c* descritte come array di interi a 256 bit (righe 584-585-586-587-588). Si noti che la funzione è di tipo *view*, ovvero non apporta alcuna modifica a variabili di stato: questo implica che non viene pagato alcun costo di esecuzione in termini di Ether per questa funzione (cioè non viene pagata la *fee*), ma viene solo conteggiato il gas utilizzato. La funzione salva gli 8 interi in una struttura “*proof*”: tale struttura viene salvata nella *memory*, cioè nella memoria “vocale” della blockchain. Infine il contratto richiama un’altra funzione, che svolge le operazioni matematiche per la verifica dei dati, che restituisce un booleano a seconda della correttezza della proof.

### 4.3 L'interfaccia on-chain del nodo off-chain: Oracolo.sol

Oracolo.sol è il programma Solidity che contiene uno dei due smart contract creati per la simulazione di alcuni casi base dell'utilizzo di *zk-ABAC*. Lo scenario generale vede un utente, dotato di alcuni attributi privati mantenuti off-chain da un certo nodo (che verrà chiamato *StorageOffChain*), che tenta di accedere una risorsa on-chain protetta da una politica.

Lo StorageOffChain ha una parte off-chain, rappresentata da dispositivi di memorizzazione che serviranno per mantenere, aggiornare e salvare gli attributi privati di un utente e i file ZoKrates delle policy, e da nodi per il calcolo delle proof per le politiche. Per integrare con il sistema di controllo degli accessi lo StorageOffChain dovrà avere anche una parte su Ethereum: questa è un'interfaccia che verrà chiamata “*oracolo*”. Tramite l'oracolo sarà possibile per lo StorageOffChain ricevere comandi (richieste) dalla blockchain e effettuare le relative azioni sulla parte off-chain. Il contratto “*Oracolo.sol*” rappresenta questa interfaccia.

La struttura dell'oracolo è molto semplice e verrà presentata nelle figure 4.4 - 4.10. In figura 4.4 vengono mostrate le strutture dati principali.

```

11  contract Oracolo{
12
13      /// @notice Struct che rappresenta una proof di ZoKrates
14      struct proof{
15          bool presente;
16          uint[2] a;
17          uint[2][2] b;
18          uint[2] c;
19      }
20
21      /// @notice Alcune informazioni relative ad una richiesta
22      /** @dev Queste informazioni sono a scopo di esempio per dimostrare e evidenziare l'avvenuta ricezione di una richiesta
23      * Per controllare la ricezione basta controllare che address sia diverso da address(0)
24      */
25      struct infos{
26          address user;
27          bytes32 req_ID;
28      }
29
30      /// @notice Owner del contratto
31      address private owner;
32
33      /// @notice mapping per le prove generate e per le richieste ricevute. Entrambi hanno come key l' uID della richiesta
34      mapping (bytes32 => proof) private mapp_proofGenerate;
35      mapping (bytes32 => infos) private received_requests; //usato per la funzione "generateProof"
36
37      /// @notice evento emesso per poter svolgere le operazioni off-chain
38      /// @dev questo evento dovrà essere catturato dalla parte off-chain, che capirà che deve iniziare la computazione di una proof
39      event offChainComputation(address addUtente, uint32 policy_ID, bytes32 req_ID);
40
41      /// @notice modificatore per le varie funzioni. Ammette che solo il proprietario possa fare certe operazioni.
42      modifier onlyOwner{
43          require (
44              owner == msg.sender,
45              "Sender not authorized!"
46          );
47      }
48  }
```

Figura 4.4: Variabili di stato, struct e strutture di appoggio del contratto Oracolo.sol

Partendo dall'alto, nelle righe 14-19 troviamo la struct che rappresenta una prova di ZoKrates. I campi a, b, c rappresentano gli 8 interi a 256 bit utilizzati dal tool per verificare le proof; il campo booleano “presente” servirà per capire se una determinata

prova è stata generata o meno. Questo booleano è fondamentale, in quanto il valore di default (lo zero) per gli altri campi interi è utilizzato in una situazione particolare.

La seconda struct (righe 25-28) rappresenta alcune informazioni di base riguardo a una specifica richiesta effettuata da un utente. Queste informazioni in un'applicazione reale non sarebbero necessarie, ma nella simulazione che verrà fornita sono utili per avere un riscontro dell'avvenuta ricezione di una certa richiesta. Ogni volta che si riceve una richiesta, un oggetto di tipo *infos* viene creato e salvato nel mapping corretto; tramite un metodo “getter” sarà possibile recuperare i dati e verificare quindi l'avvenuta ricezione della richiesta. Questa struttura è presente solo a scopo di debug, per avere un riscontro delle richieste lato oracolo.

Sono poi utilizzati due diversi mapping. Ricordiamo che un mapping è definito dalla documentazione di Solidity nel seguente modo: *“I mappings possono essere visti come tabelle hash virtualmente inizializzate in modo tale che ogni possibile chiave esista e sia mappata su un valore la cui rappresentazione è formata da tutti zeri (il valore di default per ogni tipo). Le similarità con le hash table finiscono qui: le key non sono infatti memorizzate nel mapping, ma viene invece usato il loro hash keccak256 per la ricerca dei valori”* [18]. Il primo mapping serve per associare ad ogni ID di una richiesta (rappresentato da 32 byte) una determinata proof creata appositamente: in questo modo ogni prova viene associata correttamente alla rispettiva richiesta e non si commettono errori di verifica. Il secondo mapping invece è necessario per mantenere l'associazione tra ID della richiesta e le varie informazioni utilizzate per il debug: sarà possibile, tramite questo mapping, recuperare le informazioni associate a una certa richiesta. Entrambi i mapping sono memorizzati sulla memoria *storage* di Ethereum, e le informazioni che contengono sono dunque persistenti: le proof generate potranno essere recuperate anche successivamente.

Alla riga 39 è definito un evento, necessario per la parte off-chain: questo evento viene utilizzato come “trigger” delle operazioni off-chain. L'evento emesso contiene i dati necessari alle operazioni off-chain: l'indirizzo dell'utente per poter recuperare i valori dei suoi attributi privati, l'identificativo della politica per poter recuperare i corrispondenti file ZoKrates, l'identificativo della richiesta per poter associare la prova che verrà generata alla richiesta.

Infine viene definito un modificatore (riga 42) utilizzato per limitare l'uso di alcune funzioni solamente al creatore del contratto (che sarà colui che gestisce l'oracolo): ad esempio la funzione che memorizzerà una prova non può essere richiamata da chiunque, altrimenti potrebbero essere salvate prove fasulle e inconsistenti.

### Funzione: memorizzazione della proof

In figura 4.5 viene descritta la funzione per la memorizzazione delle prove on-chain.

La funzione memorizza gli 8 interi, ognuno da 256 bit, che rappresentano una prova. In una situazione reale questa funzione dovrebbe essere chiamata ed utilizzata in modo automatico dal nodo StorageOffChain, quando esso dispone di una prova da memorizzare su Ethereum.

```

54     /// @notice Funzione usata per salvare una proof. Se fosse stata già presente una prova per req_ID, la funzione la sovrascrive.
55     /// @dev La proof dovrebbe essere generata off-chain dallo StorageOffChain, ma nel mio esempio sarò io che simulerò questa creazione off-chain
56     /** @param aa componente di una proof di zokrates
57     *  @param bb componente di una proof di zokrates
58     *  @param cc componente di una proof di zokrates
59     *  @param req_ID: identificativo univoco relativo a una richiesta
60     */
61     //return true
62     function storeProof(uint[2] memory aa, uint[2][2] memory bb, uint[2] memory cc, bytes32 req_ID) public onlyOwner returns (bool){
63         proof memory p;
64         p.a=aa;
65         p.b=bb;
66         p.c=cc;
67         p.presente=true;
68
69         mapp_proofGenerate[req_ID]=p; //salvo la proof nella chain
70         return true;
71     }

```

Figura 4.5: Funzione storeProof del contratto Oracolo.sol

La funzione prende in input 8 interi a 256 bit ciascuno, raggruppati in 2 vettori (“*a*” e “*c*”) ciascuno da due posizioni, e in una matrice “*b*” da 4 posizioni. L’ultimo input rappresenta l’ID della richiesta per la quale quella prova è stata generata. Per prima cosa viene dichiarata una variabile di tipo “*proof*” temporanea (salvata sulla memoria *memory*), che non verrà memorizzata sulla chain ma sarà relativa solo all’esecuzione della funzione. Tale variabile viene inizializzata con i valori passati in input. Il campo “*presente*” viene impostato a *true*. La prova viene salvata sulla memoria persistente della blockchain, lo *storage*, inserendola all’interno del mapping “*mapp\_proofGenerate*” e associando ad essa come chiave l’ID della richiesta: in questo modo è creata l’associazione univoca richiesta-prova.

#### Funzione: generazione della proof

Questa funzione “comunica” con la parte off-chain ed è necessaria per iniziare il processo di creazione di una proof. Per prima cosa emesso un evento: questo servirà come “trigger” del calcolo off-chain. A seguito della ricezione dell’evento, infatti, la parte off-chain inizierà il procedimento di creazione di una proof per una determinata richiesta. Tutti i dati necessari alla creazione sono specificati nell’evento e saranno quindi recuperabili dal nodo StorageOffChain. Dalla riga 92 alla 95 invece il codice effettua delle operazioni a scopo di debug: vengono salvate informazioni relative alla richiesta di un utente sulla blockchain per avere un riscontro dell’inizio della fase di generazione della prova.

I parametri di questa funzione sono particolarmente interessanti, in quanto servono per recuperare i vari file e dati off-chain:

- *address\_utente*: è l’indirizzo dell’utente che serve per recuperare i valori dei suoi attributi privati. Si assume che off-chain esista una mappatura tra indirizzi utente e relativi attributi, in particolare l’indirizzo utente deve essere la chiave per accedere agli attributi.
- *id\_politica*: è l’oggetto che identifica una determinata politica. È necessario in quanto per creare una proof sono necessari i file specifici della politica presa in considerazione: tali file devono essere salvati off-chain a causa della loro dimensione.

```

74     /// @notice Funzione che serve per dire allo StorageOffChain di generare una proof per un certo utente per una certa richiesta
75     /** @dev Questa emette un evento che dovrà essere catturato dalla parte off-chain. Tramite la cattura dell'evento inizieranno i processi off-chain.
76     */
77     /** @param address_utente indirizzo dell'utente che richiede la proof, necessario per recuperare attributi
78     *  @param id_politica identificativo della politica, necessario per recuperare i giusti file per zokrates
79     *  @param req_identifier identificativo della richiesta, necessario per associare la proof alla richiesta
80     */
81     function generateProof(address address_utente, uint32 id_politica, bytes32 req_identifier) public{
82
83         //AZIONI OFF-CHAIN A SEGUITO DELL'EVENTO:
84         // /identificazione parametri privati della politica "id_politica"
85         // /recupero dei valori relativi ai parametri privati della politica appena trovati
86         // /creazione proof per l'utente
87         // /salvataggio proof on-chain
88
89         //ATTENZIONE: è tutto asincorno, non si sa quando la computazione off-chain effettivamente sarà avviata
90         emit offChainComputation(address_utente, id_politica, req_identifier);
91
92         infos memory datas;
93         datas.user=address_utente;
94         datas.req_ID=req_identifier;
95         received_requests[req_identifier]=datas;
96
97     }

```

Figura 4.6: Funzione generateProof del contratto Oracolo.sol

Tramite l'ID della politica è possibile recuperare i corretti file, assumendo che, come per gli indirizzi utente, esista una mappatura tra id\_politica e file. Per questo campo è stato scelto il tipo uint32, pensando di creare l'identificatore delle politiche mediante un contatore intero progressivo e univoco; alternativamente si potrebbe anche pensare di creare l'ID della politica tramite l'utilizzo di una funzione hash.

- *req\_identifier*: è l'identificatore di una determinata richiesta utente. È necessario per associare la proof generata off-chain con la richiesta, in modo tale da non scambiare erroneamente prove e permettere accessi altrimenti non autorizzati, o negare accessi legittimi.

Nel codice in figura 4.6 sono presenti commenti che illustrano le azioni precedenti.

Si noti che, dato l'utilizzo di un modello “asincrono”, non è possibile sapere con certezza quando le azioni off-chain saranno eseguite: sarà compito dell'utente controllare la disponibilità dei risultati, anche ripetutamente se necessario.

### Funzione: verificare la ricezione della richiesta

```

101    /// @notice Funzione che serve per controllare se fosse arrivata veramente la richiesta.
102    /// @param uID identificativo della richiesta
103    /// @return true se la richiesta uID è arrivata all'oracolo, false altrimenti
104    function isArrived(bytes32 uID) public view returns (bool){
105        if(received_requests[uID].user != address(0)){return true;}
106        else{return false;}
107    }

```

Figura 4.7: Funzione isArrived del contratto Oracolo.sol

Questa funzione accetta come input l'ID di una richiesta e controlla se tale richiesta è stata registrata nel mapping “received\_requests”: a seconda dell'esito di questo controllo il risultato della funzione sarà true o false. La funzione è utile per verificare, lato blockchain, se l'Oracolo ha ricevuto e memorizzato una richiesta.

### Funzione: recupero e invio della prova

La funzione più importante del contratto è quella relativa al recupero di una prova e all'invio della stessa al contratto che rappresenta la logica del sistema di controllo, cioè “ACS\\_logic.sol”. Questa funzione verrà invocata dal contratto ACS\\_logic per tentare il recupero di una prova per una richiesta: non è sicuro che il recupero vada a buon fine (ad esempio perché la proof non è ancora stata generata), quindi dovranno essere effettuati controlli appositi.

```

111   /// @notice Funzione per il recupero della proof. Usata dallo SmartContract quando vuole ottenere la proof
112   /// @dev Dato che Zokrates non crea proof "errate", se la creazione della proof non va a buon fine restituisco un valore di default
113   /// @param req_ID identificativo della richiesta
114   /// @return La proof relativa alla richiesta o il valore di default se la proof non esiste
115   function retrieveResults(bytes32 req_ID) public view returns (uint256[9] memory){
116     uint256[9] memory vtt;
117
118     if(mapp_proofGenerate[req_ID].presente == false){ //la prova non è disponibile, non è ancora stata creata e salvata sulla chain
119       return [uint256(0),0,0,0,0,0,0,0,2]; //l'ultimo elemento è:
120                                         //          0: proof valida
121                                         //          1: proof non valida (quella formata da tutti zeri di default)
122                                         //          2: proof non ancora disponibile
123     }
124     else{ //la prova è presente nel mapping
125       proof memory prova=mapp_proofGenerate[req_ID];
126       if(prova.a[0]==0 & prova.a[1]==0 & prova.b[0][0]==0 & prova.b[0][1]==0 & prova.b[1][0]==0 & prova.b[1][1]==0 & prova.c[0]==0 & prova.c[1]==0){
127         //siamo nel caso in cui la prova è stata generata ma è una prova non valida, cioè una prova standard (tutti 0) creata
128         //di default perché zokrates non ha potuto creare prove reali perché gli assert della politica non erano rispettati
129         return [uint256(0),0,0,0,0,0,0,0,1];
130       }
131       else{
132         vtt=[prova.a[0], prova.a[1], prova.b[0][0], prova.b[0][1], prova.b[1][0], prova.b[1][1], prova.c[0], prova.c[1], 0];
133       }
134     }
135
136   }
137
138   return vtt;
}

```

Figura 4.8: Funzione retrieveResults del contratto Oracolo.sol

Come input questa funzione accetta solamente l'ID della richiesta per la quale l'utente vuole recuperare una proof. Viene inizialmente creato in *memory* un vettore di 9 interi ognuno a 256 bit, dove sarà salvata la proof e un dato aggiuntivo per la gestione degli errori. Il dato aggiuntivo è un intero a 256 bit come gli altri, per il quale però sono usati solo 3 diversi valori, con il seguente significato:

- “0”: la proof restituita è valida.
- “1”: la proof restituita non è valida perché *non è stato possibile soddisfare gli assert della politica*. La proof avrà un risultato finale negativo al momento della verifica.
- “2”: la proof restituita non è valida perché *non è ancora stata generata* dalla parte off-chain dello StorageOffChain.

Alla riga 118 viene controllato se esiste o meno una proof per la richiesta specificata in input, distinguendo due casi:

1. *Proof non ancora generata*: se il valore del campo “presente” sulla struct proof corrispondente alla key attuale (ID della richiesta) è “false”, per quella richiesta non è ancora stata generata nessuna prova off-chain. In questo caso viene inserito nel vettore di interi creato precedentemente il valore 0 per le prime 8 posizioni, e il valore 2 nell’ultima posizione.

2. *Prova generata*: se il valore del campo “valida” sulla struct proof corrispondente alla key attuale (ID della richiesta) è “true”, il passo di generazione è stato compiuto, ma si distinguono due possibili alternative.

Nella prima alternativa la parte off-chain ha provato a generare la proof, ma questo non è stato possibile perché alcuni assert non sono stati rispettati dagli input del programma ZoKrates relativo alla politica. In questo caso nel vettore di interi viene salvato il valore 0 per le prime 8 posizioni e il valore 1 per l’ultima posizione.

Nella seconda alternativa la prova è stata generata correttamente: saranno copiati i valori di tale prova nel vettore creato inizialmente, e in ultima posizione sarà memorizzato il valore 0.

### Funzione: recupero di una prova esistente

Per garantire l’auditability delle proof nel sistema di controllo degli accessi basato su blockchain le prove devono essere mantenute sulla memoria persistente di Ethereum. L’approccio utilizzato nel codice descritto in questo capitolo è quello di salvare le proof sull’oracolo. Il salvataggio rende le prove disponibili a chiunque voglia recuperarle per verificarne la validità; per il recupero prove sull’oracolo è presente un’apposita funzione. Il recupero è utile nel caso si voglia esaminare la proof, o nel caso ci sia la necessità di controllare la correttezza del procedimento di verifica.

```

140  /// @notice Funzione per il recupero di una proof relativa a una richiesta. Garantisce auditabilità.
141  /// @param u_ID: identificatore univoco della richiesta
142  /// @return Ritorna la prova relativa alla richiesta. Termina con errore se non ci sono prove relative alla richiesta
143  function retrieveExistingProof(bytes32 u_ID) public view returns (proof memory){
144      if(!mapp_proofGenerate[u_ID].presente == true){ //la proof esiste
145          proof memory pp;
146          pp.a=mapp_proofGenerate[u_ID].a;
147          pp.b=mapp_proofGenerate[u_ID].b;
148          pp.c=mapp_proofGenerate[u_ID].c;
149          pp.presente=true; //campo superfluo in questo caso, ma non voglio creare una nuova struct identica che cambia solo di un campo
150          return pp;
151      }
152      else{
153          revert("Nessuna prova esistente per la richiesta specificata");
154      }
155  }

```

Figura 4.9: Funzione per il recupero di prove salvate del contratto Oracolo.sol

Ogni prova può essere recuperata tramite l’ID della richiesta per la quale era stata generata. L’oracolo controlla l’esistenza di tale proof e se essa fosse stata salvata in memoria: in caso negativo l’esecuzione della funzione termina con un errore. Se la prova esiste essa viene restituita in output.

### Distruzione del contratto

Come ogni contratto anche “Oracolo” è dotato di una funzione per effettuarne l’eliminazione. Questa funzione può essere utilizzata solo dal proprietario di “Oracolo”, a cui verranno anche trasferiti tutti i fondi rimanenti a seguito dell’eliminazione.

```

158  /// @notice Funzione che permette di distruggere il contratto e mandare i suoi rimanenti fondi al resource owner
159  function kill() public onlyOwner{
160      selfdestruct(msg.sender);
161  }

```

Figura 4.10: Distruzione del contratto Oracolo.sol

#### 4.4 La logica per il controllo degli accessi: ACS\_logic.sol

ACS\_logic.sol è il programma in linguaggio Solidity che contiene la definizione del contratto “ACS\_logic”: esso rappresenta la logica del sistema di controllo degli accessi relativa ad una politica. Un utente che vuole accedere a una risorsa protetta dovrà effettuare una richiesta a questo contratto, che a sua volta comunicherà con l’Oracolo e verificherà le prove, restituendo un risultato positivo o negativo sull’accesso. Ricordando che il modello è di tipo “asincrono”, l’utente dovrà controllare (anche ripetutamente se necessario), tramite funzioni del contratto ACS\_logic, se i risultati fossero disponibili o meno.

```

20  /** @notice RequestResult è il risultato finale della richiesta di accesso.
21  *          0 - Allow: accesso permesso
22  *          1 - Deny: accesso negato
23  *          2 - NotReady: accesso non ancora valutato */
24  ///@ dev L'uso di un campo enum al posto di un campo string consente di essere più efficienti in termini di memoria
25  enum RequestResult { Allow, Deny, NotReady } //0:Allow - 1:Deny - 2:NotReady
26
27  /** @notice Struttura base di una richiesta di accesso di un certo utente
28  *          uID: identificatore univoco della richiesta. Hash calcolato tramite la funzione sha256 sull'indirizzo dell'utente e sul numero della richiesta
29  *          result: risultato finale della richiesta
30  *          ready: booleano per capire se la richiesta è pronta per la valutazione o no */
31  struct richiesta{
32      bytes32 uID;
33      RequestResult result;
34      bool ready;
35  }
36
37  /** @notice Identifica univocamente una politica
38  uint32 policyID;
39
40  /** @notice Struttura di una proof generata dall'oracolo
41  struct proof{
42      bool valida;
43      uint[2] a;
44      uint[2][2] b;
45      uint[2] c;
46  }
47
48  /** @notice mapping ("hash table") che contiene le varie richieste degli utenti e i risultati
49  ///@dev Il mapping è persistente perché salvato sulla memoria storage
50  mapping (bytes32 => richiesta) mapp_richieste;
51
52  /** @notice Owner della risorsa, che creerà il contratto
53  address private owner;
54
55  /** @notice Contatore di richieste d'accesso
56  ///@dev Il numero della richiesta è uno dei due parametri sui quali si calcola l'hash tramite sha256 che serve per come uID
57  uint256 private requestCounter;
58
59  /** @notice Contratti esterni di cui ci servono le funzioni
60  Oracolo oracolo;
61  Verifier verifier;
62
63  /** @notice Modificatore per permettere solo al proprietario della risorsa di usare certe funzioni
64  modifier resourceOwner{
65      require (owner == msg.sender,"Sender not authorized!");
66      _;
67  }

```

Figura 4.11: Architettura del contratto ACS\_logic.sol

La struttura di questo contratto è molto simile a quella del contratto “Oracolo”. All’inizio del codice (righe 31-35) viene definita la struttura di una richiesta, composta da: un campo “uID”, un identificativo univoco della richiesta da 32 byte, un campo “result”, che rappresenta il risultato della richiesta di accesso (Allow, Deny, NotReady) e un capo

“ready” necessario per capire se la richiesta sia pronta o meno alla valutazione. Il campo “policyID” viene utilizzato per il salvataggio dell’ID della policy in considerazione: questo contratto, infatti, rappresenta la smart policy che protegge una determinata risorsa, e quindi la politica a cui si riferisce è una soltanto. Alle righe 41-46 viene effettuata la dichiarazione del tipo struct “proof”, che rappresenta una prova che verrà generata dall’Oracolo: la struttura ha la stessa forma di quella con il medesimo nome definita in 4.4. Tra le variabili di stato sono presenti: il mapping per salvare le richieste di accesso ricevute, un campo per salvare l’indirizzo del proprietario del contratto, un modificatore per limitare l’uso di alcune funzioni, e i due oggetti “oracolo” e “verifier” che rappresentano i riferimenti agli omonimi contratti per poter chiamare le loro funzioni.

## Costruttore

```

71  /// @notice Costruttore che inizializza i vari campi del contratto.
72  /// @dev Sono necessari gli indirizzi dei due contratti Oracolo.sol e verifier.sol per poter successivamente richiamare delle loro funzioni
73  /** @param addOracolo: indirizzo del contratto Oracolo.sol
74  *      zokratesAddress: indirizzo del contratto verifier.sol, generato automaticamente dal tool ZoKrates e relativo a una specifica politica
75  */
76  constructor(address addOracolo, address zokratesAddress) public{
77
78      // Owner del contratto, il proprietario della risorsa
79      owner=msg.sender;
80
81      requestCounter=0;
82
83      /** @dev L'inizializzazione viene fatta a 0 perchè questo è un contratto di esempio e in versione di prova.
84      *      Nel concreto si dovrebbe scegliere un identificatore per la politica che poi va passato anche all'oracolo.
85      */
86      policyID=0;
87      oracolo = Oracolo (addOracolo);
88      verifier = Verifier (zokratesAddress);
89
90  }

```

Figura 4.12: Costruttore del contratto ACS\_logic

Per questo contratto il costruttore richiede due input: l’indirizzo del contratto “Oracolo” e l’indirizzo del giusto contratto “Verifier”. Questi indirizzi saranno necessari per poter utilizzare le funzioni dei due contratti. Il codice del corpo del costruttore è molto semplice: l’indirizzo di chi sta chiamando la funzione (che sarà quindi il proprietario) viene memorizzato, così come gli indirizzi dei contratti passati come input. Il contatore delle richieste viene inizializzato a 0. Alla riga 86 l’ID relativo alla politica viene inizializzato a 0: questo valore stabilisce la corrispondenza tra politica e relativi file ZoKrates off-chain, e per questo deve essere impostato in accordo con il nodo off-chain.

## La dimostrazione di interesse verso l’Oracolo

Il codice nella figura 4.13 contiene due funzioni, necessarie per effettuare le dimostrazioni di interesse, da parte dell’utente verso il contratto ACS\_logic stesso e da parte del contratto ACS\_logic verso il contratto Oracolo.

La prima funzione, *InterestDemonstrationProofOracolo*, serve a ACS\_logic per richiedere, tramite dimostrazione di interesse, una prova per una certa richiesta. Viene effettuata una chiamata alla funzione del contratto Oracolo esaminata in 4.6, passando

```

107     function InterestDemonstrationProofOracolo(address address_utente, uint32 id_politica, bytes32 req_identifier) private{
108         oracolo.generateProof(address_utente, id_politica, req_identifier); //funzione dell'oracolo che serve per iniziare la creazione proof
109     }
110
111
112
113     /** @notice Funzione che inizia la fase di verifica per accedere a una risorsa. Manda una "dichiarazione di interesse" all'oracolo
114     * per dirgli di creare una proof per verificare se è possibile accedere o meno. E' la funzione che crea l'identificatore
115     * univoco della richiesta.
116     */
117     /// @dev Per creare l' uID viene usato l'indirizzo utente e il numero della richiesta.
118     /** @return Ritorna l'identificatore univoco che l'utente dovrà utilizzare per riferirsi a questa precisa richiesta di accesso.
119     * Questo oggetto serve per tutte le operazioni future.
120     */
121     function InterestDemonstration() public returns (bytes32){
122
123         requestCounter=requestCounter+1; //incremento il numero di richieste totali
124
125         address senderUser=msg.sender; //utente che richiede la verifica
126
127         bytes32 request_uID = sha256(abi.encode(senderUser, requestCounter)); //creazione identificatore univoco
128
129         mapp_richieste[request_uID].uID=request_uID;
130         mapp_richieste[request_uID].result=RequestResult.NotReady;
131         mapp_richieste[request_uID].ready=false; //ALL'inizio il risultato non è disponibile
132         InterestDemonstrationProofOracolo(senderUser, policyID, request_uID);
133
134         return request_uID;
135     }
136 }
```

Figura 4.13: Funzioni per le dimostrazioni di interesse del contratto ACS.logic

tutti i parametri necessari. Ricevuta l’invocazione di questa funzione, l’Oracolo sarà a conoscenza che è necessario produrre una certa proof.

La seconda funzione, *InterestDemonstration*, è utilizzata direttamente dall’utente, e richiede al contratto ACS.logic l’accesso alla risorsa protetta. L’utente, una volta chiamata la funzione, otterrà come risultato un identificativo per la sua richiesta: questo oggetto sarà necessario per tutte le operazioni future. Il codice della funzione è composto da poche operazioni. La prima operazione è l’incremento del contatore delle richieste ricevute; successivamente viene creato l’identificatore univoco della richiesta calcolando l’hash del contatore e dell’indirizzo utente. La funzione hash utilizzata è la *sha256*: essa è una funzione deterministica (a stessi input corrispondono stessi output) che genera in output 256 bit. La funzione *sha256* può essere considerata “sicura” per due motivi:

1. sha256 è deterministica, il che vuol dire che a stessi input corrispondono stessi output. Questo non è un problema in quanto l’input della funzione hash per la nostra applicazione sarà sempre diverso, dato che viene utilizzato un numero progressivo (il contatore delle richieste), e l’address utente univoco.
2. sha256 produce un output di 256 bit: prima di trovare una collisione si devono esaurire tutte le possibilità, che sono  $2^{256} \approx 1,16 \cdot 10^{77}$ . Questo numero è talmente grande che in un’applicazione reale possiamo considerarlo inesauribile.

Dopo avere creato l’hash, il codice della funzione prosegue inserendo nel giusto mapping la richiesta appena arrivata, impostando il valore della risposta a “NotReady” e il valore del campo booleano “ready” a false. A questo punto viene effettuata una chiamata con i parametri necessari alla funzione *InterestDemonstrationProofOracolo* descritta precedentemente. Alla riga 134 è possibile notare come la funzione *InterestDemonstra-*

*tion* termini restituendo al chiamante il valore dell'ID della richiesta necessario per le operazioni future.

## Recupero dei risultati dall'Oracolo

```

149     function areResultsReady(bytes32 req_id) public returns (bool){
150
151     proof memory provaGenerata;
152     provaGenerata.valida=false;
153     uint256[9] memory vettoreProva = oracolo.retrieveResults(req_id); //funzione dell'oracolo che guarda se sono pronti i risultati e in caso li prende
154
155     //Ogni cella del vettore è una parte della prova (tranne l'ultima cella che contraddistingue se la prova è valida(0)/disponibile/NonValida
156
157     provaGenerata.a[0]=vettoreProva[0];
158     provaGenerata.a[1]=vettoreProva[1];
159     provaGenerata.b[0][0]=vettoreProva[2];
160     provaGenerata.b[0][1]=vettoreProva[3];
161     provaGenerata.b[1][0]=vettoreProva[4];
162     provaGenerata.b[1][1]=vettoreProva[5];
163     provaGenerata.c[0]=vettoreProva[6];
164     provaGenerata.c[1]=vettoreProva[7];
165
166     //L'ultimo elemento è:
167     //      0: proof valida - 1: proof non valida (formata da tutti zeri) - 2: proof non ancora disponibile
168
169     if(vettoreProva[8] == 2){
170         provaGenerata.valida=false; //la prova non è disponibile
171     }
172     else if(vettoreProva[8] == 1 || vettoreProva[8] == 0){
173         provaGenerata.valida=true;
174     }
175
176     if(provaGenerata.valida == false){ //l'oracolo non ha ancora una proof disponibile
177         return false;
178     }
179     else{ //la proof è disponibile (è stata generata correttamente o settata di default)
180         bool result;
181
182         if(vettoreProva[8] == 1){ //prova di default formata da tutti zeri
183             result=false; //si evita la chiamata al verificatore
184         }
185         else{
186             result = verifier.verifyTx(provaGenerata.a, provaGenerata.b, provaGenerata.c);
187         }
188         if(result == true){
189             mapp_richieste[req_id].result=RequestResult.Allow; //salvo il risultato della richiesta
190         }
191         else{
192             mapp_richieste[req_id].result=RequestResult.Deny; //salvo il risultato della richiesta
193         }
194         mapp_richieste[req_id].ready=true;
195     }
196 }

```

Figura 4.14: Funzione per il recupero dei risultati dall'Oracolo

Questa funzione rappresenta una dimostrazione di interesse emessa dal contratto ACS\_logic e inviata all'Oracolo, che esprime l'interesse nel recuperare i risultati (la proof) per una certa richiesta.

Inizialmente viene creato nella memoria *memory* un vettore di 9 interi, dove sarà salvato il risultato della chiamata alla funzione “retrieveResults” dell'Oracolo, visibile in figura 4.8. Una volta recuperata la prova essa viene copiata in una struct di tipo “proof”, e il campo booleano che rappresenta la validità viene impostato a false. L'ultimo elemento del vettore di 9 interi serve per distinguere due diversi casi:

1. Nel primo caso il valore dell'ultimo elemento è “2” che indica che la prova richiesta non è ancora stata generata. Il campo booleano “valida” rimane false.
2. Nel secondo caso, in cui il risultato è “1” o “0”, la prova è stata generata, e quindi il campo booleano “valida” viene impostato a true.

Dalla riga 176 inizia dunque il controllo sulla validità della proof: trovando il campo “valida” settato a false la funzione ritorna un valore “false”, evidenziando che la proof richiesta non è ancora stata generata. Alternativamente, dato che la proof è presente e può essere valutata, la funzione restituirà il valore “true”. Nel caso in cui l’ultimo intero del vettore avesse valore “1”, non si effettua la chiamata al verificatore, perché la prova generata è formata da tutti valori nulli (gli attributi utente non rispettavano le condizioni): viene salvato il risultato “Deny” nel mapping delle richieste. Trovando invece il valore “0” nell’ultima posizione del vettore, la proof è deve essere verificata tramite una chiamata di funzione del contratto “Verifier”, che restituirà true o false e a seconda della correttezza della prova.

### Risultato finale e distruzione del contratto

```

200     /// @notice Funzione che recupera il risultato finale della verifica se disponibile
201     /// @param req_id: identificativo univoco della richiesta per la quale vogliamo il risultato
202     /// @return Ritorna il risultato finale per la richiesta (0:Allow - 1:Deny - 2:NotReady)
203     function retrieveResult(bytes32 req_id) public view returns (RequestResult){
204         if(mapp_richieste[req_id].ready == true){
205             return mapp_richieste[req_id].result;
206         }
207         else{ //Risultato finale non ancora disponibile
208             return RequestResult.NotReady;
209         }
210     }
211
212
213     /// @notice Funzione che permette di distruggere il contratto e mandare i suoi rimanenti fondi al resource owner
214     function kill() public resourceOwner{
215         selfdestruct(msg.sender);
216     }
217 }
```

Figura 4.15: La prima funzione serve al recupero dell’esito finale della richiesta. La seconda invece è la funzione che permette la distruzione del contratto.

Le ultime due funzioni analizzate sono quelle relative al recupero dell’esito finale della richiesta e alla distruzione del contratto. La funzione “kill” è strutturata in modo standard e permette di distruggere il contratto trasferendone i fondi rimanenti al proprietario.

La funzione “retrieveResults” prende in input l’identificatore di una richiesta e tramite esso controlla se fosse presente un risultato: in caso positivo lo restituisce, in caso negativo restituisce il risultato “NotReady”.

## 4.5 Esempio di utilizzo di zk-ABAC

Nella figura 4.16 è mostrato a scopo dimostrativo un esempio di utilizzo del sistema implementato nel capitolo 4.

Lo scenario vede due utenti che vogliono accedere a una risorsa protetta tramite zk-ABAC. Il primo utente disporrà degli attributi che soddisfano la politica di accesso, e gli sarà quindi garantito l’accesso alla risorsa, mentre il secondo utente si vedrà negato l’accesso.

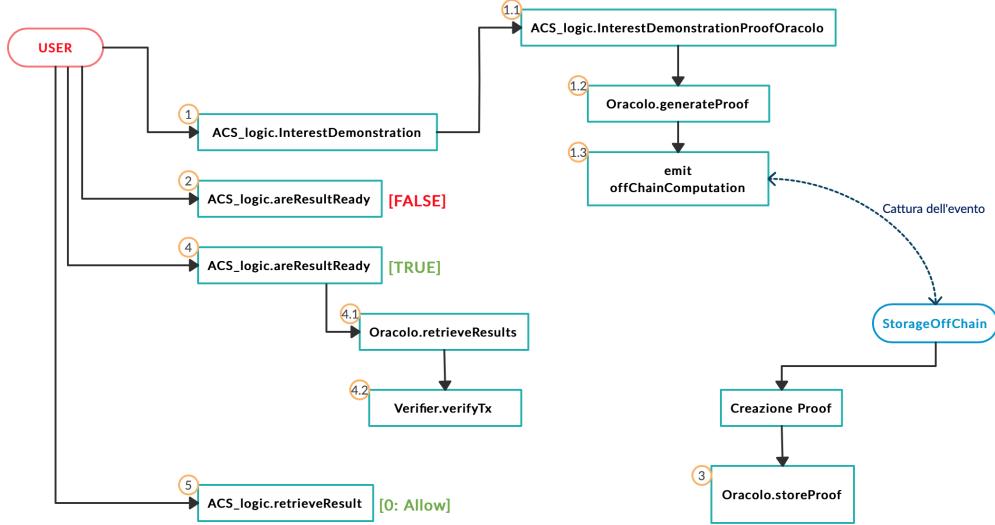


Figura 4.16: Operazioni effettuate per la richiesta di accesso dall’utente che riceve risposta positiva.

L’esempio qui riportato è stato sviluppato e testato su Remix. Si assume ovviamente che tutti i contratti necessari (oracolo, smart policy, verificatore) siano già sulla blockchain al momento delle richieste di accesso.

Supponiamo che il primo utente richieda l’accesso alla risorsa protetta:

1. L’utente dimostra l’interesse nell’accesso alla risorsa effettuando una chiamata alla funzione *InterestDemonstration* (Fig. 4.13) del contratto ACS\_logic. Questa funzione automaticamente richiama un’altra funzione del contratto Oracolo, “*generateProof*”, che a sua volta emetterà l’evento “trigger” per le computazioni off chain. L’utente ottiene come risultato l’ID della sua richiesta.
2. L’utente usa la funzione *areResultReady* del contratto ACS\_logic per verificare la disponibilità del risultato della sua richiesta. Se ottiene un risultato “false” capisce che il risultato non è disponibile e che deve reiterare l’invocazione della funzione.
3. Intanto lato off-chain è iniziata la creazione della prova, a seguito dell’emissione dell’evento “*offChainComputation*”. Una volta che il processo di creazione della prova termina, il nodo StorageOffChain usa la funzione *storeProof* del contratto Oracolo per salvare sulla chain la proof generata, associandola all’ID dell’utente.
4. L’utente verifica nuovamente se i risultati della richiesta sono pronti, utilizzando ancora la funzione *areResultReady* con input l’ID della richiesta. Ottiene un risultato “true” e capisce che può accedere al risultato della sua richiesta di accesso.

5. L'utente usa la funzione *retrieveResult* per recuperare l'esito della richiesta di accesso. Poiché abbiamo supposto che il primo utente possegga i corretti valori degli attributi privati, la proof generata per lui è corretta, e dunque ottiene come risultato “Allow”.

Il secondo utente richiede l'accesso alla risorsa protetta:

- I passaggi 1, 2, 3, 4 sono esattamente uguali a quelli effettuati dall'utente precedente.
- Quando però il secondo utente utilizza la funzione *retrieveResult*, esso si vedrà restituito il risultato “Deny”. Non può accedere alla risorsa perché non dispone dei giusti attributi.

Dalla console di Remix è possibile vedere che sulla blockchain sono state registrate le seguenti transazioni per l'accesso del primo utente (di cui saranno riportati solo il nome della funzione chiamata). Tra parentesi quadre sono inseriti i risultati delle chiamate di funzione se interessanti ai fini dell'esempio. Le operazioni indentate sono eseguite a seguito della chiamata di funzione principale.

1. ACS\_logic.InterestDemonstration [requestID]
  - (a) ACS\_logic.InterestDemonstrationProofOracolo
  - (b) Oracolo.generateProof
  - (c) emit offChainComputation
2. ACS\_logic.areResultReady [FALSE]
3. Oracolo.storeProof
4. ACS\_logic.areResultReady [TRUE]
  - (a) Oracolo.retrieveResults
  - (b) Verifier.verifyTx
5. ACS\_logic.retrieveResult [0]

# Capitolo 5

## Valutazioni

Obiettivo di questo capitolo è fornire una valutazione sperimentale del sistema proposto e tool utilizzati, al fine di evidenziare i punti di forza e criticità del progetto. Il capitolo sarà suddiviso in tre sezioni: nella prima verranno analizzati i costi di memorizzazione e computazionali, le performance e la scalabilità del tool ZoKrates; nella seconda verrà fornita un’analisi delle performance del sistema implementato nel capitolo 4 e nella terza sezione saranno mostrate considerazioni finali sul progetto.

### 5.1 ZoKrates: costi computazionali e dimensionali

In questa sezione viene esaminato il reale funzionamento di ZoKrates in termini di tempo di esecuzione dei diversi moduli e di occupazione di memoria. L’obiettivo è di avere una prima valutazione delle performance del tool e delle possibilità di utilizzo su blockchain. Sono state specificate delle semplici politiche al fine di capire come variano i tempi computazionali e le dimensioni dei file in base agli input e ai “vincoli”. In questo contesto ci riferiamo ai vincoli degli schemi matematici utilizzati da ZoKrates: sono quindi un concetto differente dal termine “assert”, che indica invece una condizione imposta nel programma ZoKrates. Ad esempio, una politica con un solo assert (cioè una sola condizione sugli input), genera circa 916 vincoli matematici durante la compilazione.

#### 5.1.1 Analisi dell’occupazione di memoria

In tabella 5.1 sono riportate le dimensioni in byte dei vari file generati al variare del numero di argomenti del comando “assert”.

Esaminando la tabella 5.1 è possibile osservare come alcuni file abbiano dimensioni **costanti**: essi sono *proof.json* e *verification.key*. Si noti come la proof non dipenda, in termini di dimensioni, dagli input o dal programma, soddisfacendo la proprietà “succinctnes” di zk-SNARKs.

Tabella 5.1: La tabella mostra le dimensioni dei vari file in **byte** al variare del numero di input e assert. Tra parentesi quadre viene indicato di quanto le dimensioni aumentino relativamente al caso base (1 argomento, 1 assert). È possibile notare l'aumento lineare dettato dal numero di vincoli del programma.

	Numero vincoli	abi.json	out	out.ztf	file.zok	proof.json	proving.key	verification.key	witness
<b>1 argomento 1 assert</b>	916	114	146295	72909	56	1003	290648	1386	5536
<b>2 argomenti 2 assert</b>	1832 [x2]	191 [x1.67]	292534 [x2]	148966 [x2]	92	1003	580440 [x2]	1386	11616 [x2]
<b>3 argomenti 3 assert</b>	2748 [x3]	268 [x2.35]	438773 [x3]	225023 [x3]	128	1003	935768 [x3.21]	1386	17696 [x3.2]
<b>8 argomenti 8 assert</b>	7328 [x8]	653 [x5.7]	1169968 [x8]	609060 [x8.35]	308	1003	2319192 [x7.98]	1386	48855 [x8.82]
<b>1 argomento 2 assert</b>	1579 [x1.72]	114	258653 [x1.77]	124639 [x1.71]	76	1003	499736 [x1.72]	1386	9592 [x1.73]
<b>1 argomento 3 assert</b>	2242 [x2.44]	114	371011 [2.41]	176369 [x2.41]	95	1003	774360 [x2.66]	1386	13648 [x2.46]
<b>1 argomento 8 assert</b>	5557 [x6.06]	114	932801 [x6.38]	437503 [x6]	190	1003	1754264 [x6.06]	1386	34435 [x6.22]

Si fornisce ora una descrizione dei file usati in ZoKrates, indicandone la funzione in relazione al sistema di controllo degli accessi per spiegare quali potrebbero essere i limiti del sistema in base alle dimensioni di questi file.

- *file.zok*: è il programma scritto in linguaggio ZoKrates che codifica delle condizioni che faranno parte di una politica. È possibile memorizzare questo file esternamente alla chain in modo tale che un qualsiasi prover lo possa recuperare e successivamente. Il fatto che il programma sia salvato in chiaro non crea problemi, le politiche possono essere visualizzate da chiunque anche nell'ACS definito in [3]: esse non sono private.
- *abi.json*: è un file in formato JSON che riporta quali sono gli input del programma, specificandone la visibilità pubblica o privata.
- “*out*”: è il compilato del programma (flattened-code binario). Le sue dimensioni aumentano linearmente con il numero di vincoli, come è possibile notare dalla tabella. Non può essere salvato sulla chain di Ethereum perché, anche per una politica molto piccola, le sue dimensioni sono molto grandi se rapportate ai costi della blockchain: sarà necessario memorizzare questo file off-chain.
- *out.ztf*: è il compilato del programma in formato *human-readable-code*. Non è necessario ai fini del funzionamento del sistema proposto.
- *proof.json*: è la proof relativa a un certo programma. La sua struttura è molto semplice, come si vede dalla figura 5.1. Ha sempre la stessa dimensione e il solito formato per qualsiasi programma, come garantito dalla proprietà *succinctness* definita dagli schemi di zk-SNARKs. La dimensione è esigua, solamente 1003 byte. La memorizzazione sulla blockchain di questo file è importante per garantire l'auditability delle prove, e date le sue dimensioni non presenta un costo troppo ingente. Se si

```

1  {
2    "proof": {
3      "a": [
4        "0x1ce817f083eda6500f0a457a4873407f2ff302c4e8651abd32d395f0b64d973c",
5        "0x19da58ce29ef16418056ea3457334779f800ac7c9d6bfcc43652ac60c93c3a30",
6      ],
7      "b": [
8        [
9          [
10            "0x049863e79cf520890e8555640e82fe55730104eebf9bd6332414b03b72df010",
11            "0x0d62cd3d2c6d9a5749d447b9bbe3058d296af11ff57ec813030efaf091f1bda"
12          ],
13          [
14            "0x054ec44cdc4634cffd134d5dff0a982ab708be8cb7b8c9a215e498b71372ee31",
15            "0x1106a1d16a1c9da6d5f5bf6d1cb49aa0cd141ffeec5f7dd42b19c1a195c56f6"
16          ]
17        ],
18        [
19          [
20            "0x04f083ca26e05eb4d8e3e2018a290fa8c18a57145c4cb7fd2891b8170434a725",
21            "0x175f44044858cccd6fd9af61dd8cd8660fef599f1e9e4bc527545ce4721725a1"
22          ]
23        ],
24      ],
25      "inputs": [],
26      "raw": "9ce817f083eda6500f0a457a4873407f2ff302c4e8651abd32d395f0b64d973c049863e79cf520890e8555640e82fe55730104eebf9"
27    }
28  }

```

Figura 5.1: Struttura file “proof.json”. Tutti gli input del programma relativo a questa proof sono privati e infatti il campo “input” è vuoto.

analizza il file proof.json possiamo vedere che esso ha una dimensione di 1003 byte, ma, come detto precedentemente, i dati da salvare sulla blockchain sono solamente 256 byte.

Tramite un rapido calcolo viene fornita una stima, in termini sia di gas, che di Ether, che di dollari, del salvataggio della prova on-chain:

$$1 \text{ gas} = 64.547 \text{ Gwei} @ 28/10/2020 [8]$$

$$GasCostProof = \lceil \frac{256}{32} \rceil \cdot 20000 \approx 160000 \text{ Gas}$$

$$GweiCostProof \approx 160000 \cdot 64.547 \approx 10327520 \text{ Gwei}$$

$$1 \text{ Gwei} = 0.000000001 \text{ Ether}$$

$$EtherCostProof = 10327520 \cdot 0.000000001 \approx 0.010327 \text{ Ether}$$

$$1 \text{ Ether} \approx 387 \$ @ 28/10/2020 [8]$$

$$DollarsCostProof = 387 \cdot 0.010327 = 3.9965 \approx 4\$$$

Queste considerazioni consentono di affermare che l’auditability delle proof ha un costo non trascurabile ma neanche improponibile.

Sorge chiaramente un problematica riguardo al salvataggio di una proof sulla chain: chi deve pagare il costo dell’operazione, l’utente o il sistema? Questo è sicuramente un punto interessante da discutere in scenari di applicazioni reali: una possibilità potrebbe essere quella di dividere il costo tra sistema e utente in parti uguali.

- *proving.key*: questo file rappresenta la chiave di prova, ovvero quei dati indispensabili a un qualsiasi prover per poter creare una proof. Le dimensioni di questo file aumentano linearmente con il numero dei vincoli, e questo obbliga a memorizzarlo off-chain: “*proving.key*” e “*out*” dovranno essere memorizzati in coppia in modo tale da essere recuperabili insieme.
- *verification.key*: è la chiave di verifica che servirà solamente *una-tantum* per la creazione del verificatore in linguaggio Solidity. Le sue dimensioni sono costanti e non dipendono dal numero di vincoli. Dopo la creazione del relativo verificatore questo file non è più necessario e può essere eliminato per liberare spazio.
- *witness*: è il testimone, cioè il file che attesta la conoscenza di una computazione corretta per il programma. Le sue dimensioni aumentano linearmente con il numero di vincoli, ma questo file non comporta nessun tipo di problema in quanto può essere eliminato subito dopo la creazione della proof, e quindi non occupa memoria se non per un breve periodo di tempo. Non deve essere memorizzato sulla chain, quindi la sua occupazione di memoria è irrilevante considerando anche che viene eliminato subito dopo la creazione della relativa proof.
- *verifier.sol*: è il programma in linguaggio Solidity che rappresenta il verificatore per il programma ZoKrates. La sua struttura è standard, ed è sempre uguale anche per programmi diversi. Le sue dimensioni sono costanti e indipendenti dal programma per cui esso viene generato: tali dimensioni sono di 24 KB.

### 5.1.2 Analisi dei Tempi

In questa sezione vengono esaminati i tempi di esecuzione delle diverse fasi di ZoKrates. Queste operazioni sono quelle che verranno effettuate off-chain dal nodo StorageOffChain per poter creare le proof di una politica: esse determinano quindi un certo ritardo non trascurabile e non eliminabile. Se questi tempi fossero troppo lunghi il sistema risulterebbe inutilizzabile.

La misura dei tempi è stata fatta utilizzando il comando “*time*” di UNIX e riportando il “*real time*”, cioè il tempo impiegato dall'inizio dell'esecuzione del comando fino al suo termine. I test sono stati eseguiti sul seguente hardware:

1. **CPU**: Intel i5-6267U dual-core (4 thread) @ 2,90 GHz (max 3,30 GHz), 4MB cache
2. **RAM**: 2 x 4 GB LPDDR3 @2133 MHz (tot: 8 GB)
3. **SSD**: APPLE SSD AP0256J

Analizzando la tabella 5.2 si nota come esistano alcune operazioni eseguite in tempo costante e indipendenti dal numero di vincoli, mentre altre vengono portate a termine in un tempo variabile e dipendente dal numero di vincoli. Sull'ultima riga sono riportati i tempi relativi al programma ZoKrates per la risoluzione del “Sudoku 2x2”, problema interessante dato che la sua compilazione genera molti più vincoli rispetto ai programmi

Tabella 5.2: Tempo impiegato dei vari comandi per la creazione del verificatore e delle prove. I tempi riportati sono espressi in millisecondi

	Numero vincoli	compile	setup	export-verifier	compute-witness	generate-proof
<b>1 argomento 1 assert</b>	916	174 ms	566 ms	10 ms	59 ms	70 ms
<b>2 argomenti 2 assert</b>	1832 [x2]	185 ms	1089 ms	10 ms	111 ms	110 ms
<b>3 argomenti 3 assert</b>	2748 [x3]	264 ms	1721 ms	10 ms	163 ms	166 ms
<b>8 argomenti 8 assert</b>	7328 [x8]	713 ms	4098 ms	10 ms	404 ms	342 ms
<b>1 argomento 2 assert</b>	1579 [x1.72]	179 ms	912 ms	10 ms	96 ms	106 ms
<b>1 argomento 3 assert</b>	2242 [x2.44]	250 ms	1364 ms	10 ms	128 ms	150 ms
<b>1 argomento 8 assert</b>	5557 [x6.06]	644 ms	3066 ms	10 ms	285 ms	301 ms
<b>Sudoku 2x2</b>	25426	2780 ms	14842 ms	10 ms	1285 ms	1272 ms

ZoKrates che codificano politiche riportati in tabella (circa 5 volte i vincoli della politica più grande): è un buon esempio per capire come il tool scali anche su problemi più complessi. Dall’analisi di “Sudoku 2x2” si deduce che le fasi più onerose in termini di tempo sono quella della compilazione e del setup: quest’ultima impiega circa 15 secondi, un tempo sicuramente non trascurabile. Queste due fasi impiegano tanto più tempo quanto più sono complessi i programmi. Rapportando le misure effettuate all’ACS con garanzie di privacy, questo non comporta alcuna problematica: sia il setup che la compilazione sono operazioni da effettuare una volta soltanto, nella fase di creazione della politica. Una volta ottenuti i file da mettere online e il verificatore da mettere sulla chain questi due passaggi non saranno più necessari. Il problema del tempo impiegato da queste due fasi può quindi essere considerato trascurabile, dato che non impatta nelle operazioni d’accesso dell’utente.

Il tempo impiegato per la creazione del verificatore in linguaggio Solidity invece è costante e indipendente dal programma. Inoltre anche questa operazione comunque sarà da effettuare una volta soltanto, e solamente in fase iniziale di setup del sistema. La creazione del verificatore comunque, come è possibile vedere in tabella 5.2, impiega solamente 10 millisecondi.

Infine l’analisi delle operazioni di creazione della prova, cioè la generazione del witness e la generazione della proof, evidenzia tempi minori: nel caso del Sudoku sono impiegati meno di due secondi per creare il witness e meno di due secondi per generare la proof. Queste due operazioni sono quelle che dovranno essere effettuate ad ogni richiesta di accesso di un utente; il delay introdotto è accettabile, soprattutto se comparato alle tempistiche della blockchain Ethereum, dove la creazione di nuovi blocchi avviene in media ogni 13 secondi [8].

## 5.2 Valutazioni del sistema su blockchain

In questa sezione vengono analizzate le prestazioni del sistema in termini di gas e tempi di attesa, in relazione alla blockchain Ethereum. Per valutare le prestazioni del nuovo sistema implementato è stato utilizzato il tool Remix, che permette di simulare e recuperare il costo delle varie operazioni effettuate sui contratti.

Presentiamo prima i costi delle operazioni e, successivamente, forniamo un'analisi sulle diverse casistiche del sistema. Le operazioni nel sistema implementato nel capitolo 4 sono riportate, con i relativi costi in gas, in tabella 5.3.

Tabella 5.3: Costi in gas. Alcune operazioni hanno un costo diverso in base al loro esito, true [T] o false [F]; il costo delle operazioni può anche variare a seconda che la proof considerata sia formata da tutti zeri [default], o meno [valida].

	<b>ACS_logic Interest Demonstration</b>	<b>ACS_logic areResultReady</b>	<b>ACS_logic retrieveResults</b>	<b>Oracolo isArrived</b>	<b>Oracolo storeProof</b>
<b>GAS</b>	135745	36035 [F] 54061 [T, default] 258689 [T, valida]	26062	25000	60014 [default] 229934 [valida]

Lo scenario di un utente che vuole accedere prevede le seguenti operazioni, raggruppate in transazioni:

- 1<sup>a</sup> transazione -  $T_a$ : ACS\_logic.InterestDemonstration
- Generazione proof off-chain
- 2<sup>a</sup> transazione -  $T_b$ : Oracolo.storeProof
- 3<sup>a</sup> transazione -  $T_c$ : recupero dei risultati e valutazione
  - ACS\_logic.areResultReady
  - ACS\_logic.retrieveResults

Le statistiche aggiornate su Ethereum sono ottenute da [8]. I dati della blockchain sono altamente dinamici, quindi abbiamo deciso di riportare l'andamento temporale delle informazioni considerate per assicurarci che i dati attuali non siano frutto di un periodo anomalo.

Il “Gas Limit” è il gas massimo disponibile all'interno di un certo blocco. Da Figura 5.2 è possibile notare come dal mese di Luglio 2020 al mese di Novembre 2020 questo dato

sia pressoché stabile, attestandosi attorno al valore di 12470000 gas. È quindi possibile considerare tale valore come buon rappresentante del gas limit attuale.

Il “Tempo di blocco” rappresenta il tempo medio che deve trascorrere prima che un certo blocco sia “minato” (cioè validato) ed inserito nella blockchain Ethereum. È utile per fare stime sull’attesa prevista per l’esecuzione delle transazioni. Dal grafico 5.3 è possibile notare come questo dato subisca poche variazioni, e come nel periodo da Gennaio 2020 a Novembre 2020 il suo valore sia rimasto pressoché stabile ed attestandosi a 13 secondi circa.

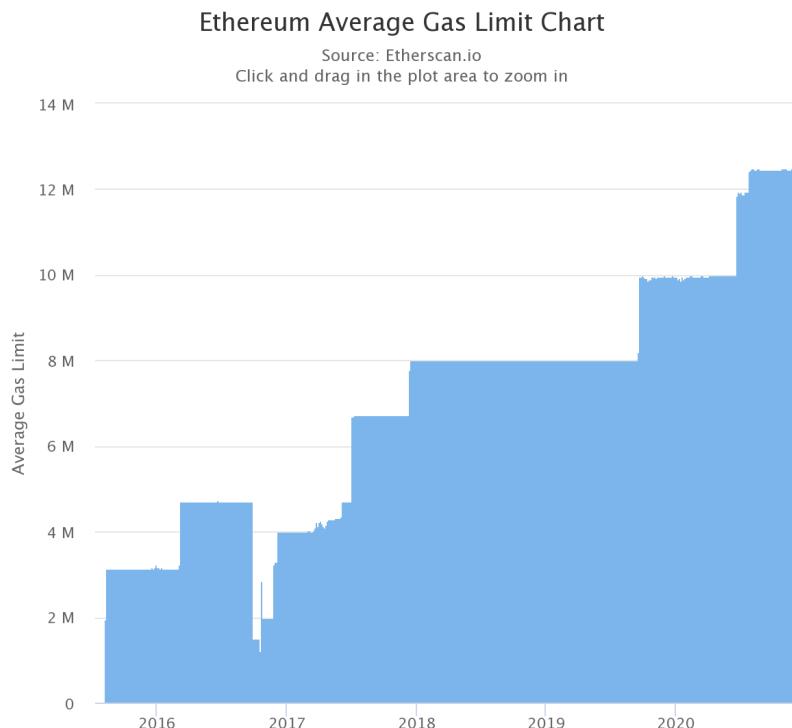


Figura 5.2: Andamento del “Gas Limit” negli ultimi cinque anni.

Dati gli andamenti dei grafici 5.2 e 5.3, ad oggi, 20 Novembre 2020, è ragionevole considerare i seguenti dati:

- Limite di gas per singolo blocco:  $\approx 12470000$  gas
- Tempo di blocco:  $\approx 13$  secondi

Ottenuti questi dati è possibile effettuare alcune analisi sul funzionamento del sistema per ottenere stime sulla sua efficienza.

Considerando il caso nel quale un utente voglia accedere a una risorsa protetta, si evidenziano 3 diverse transazioni: una per dimostrare l’interesse nell’accesso alla risorsa

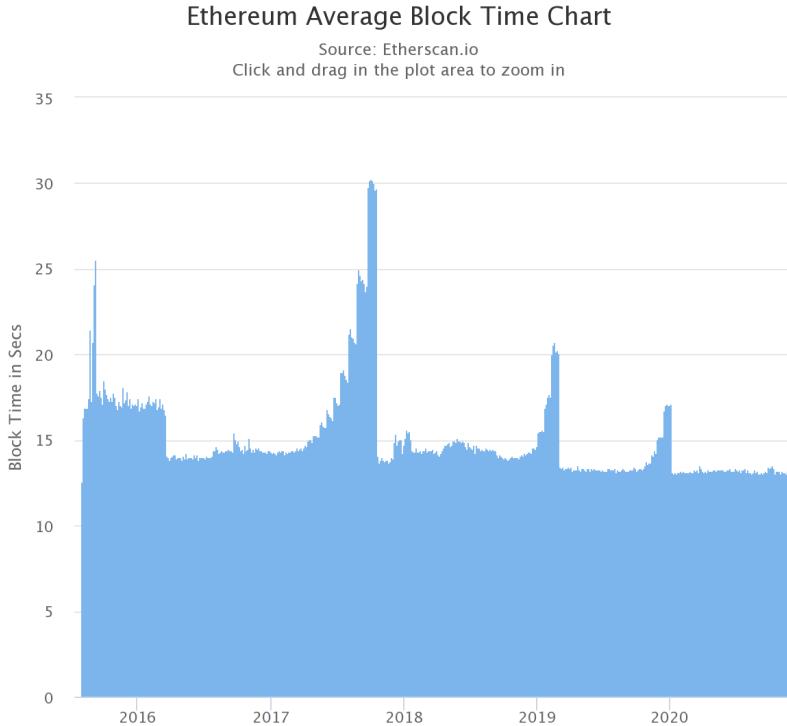


Figura 5.3: Andamento del “Tempo di blocco” negli ultimi cinque anni.

protetta ( $T_a$ ), una per effettuare la memorizzazione su Ethereum della prova generata off-chain ( $T_b$ ), e una per il recupero e la valutazione dei risultati ( $T_c$ ). Nella seguente sezione vengono analizzate le performance del sistema in diverse situazioni.

### 5.2.1 Il caso ottimo non deterministico

Il caso in cui il sistema funziona nel modo più efficiente possibile è quello nel quale la transazione  $T_a$  viene inserita in un blocco, e le altre due transazioni  $T_b$  e  $T_c$  vengono inserite nel blocco immediatamente successivo ed eseguite nell'ordine corretto. Questo è un caso “*non deterministico*” in quanto le transazioni  $T_b$  e  $T_c$  soffrono del problema di concorrenza, essendo inserite nel solito blocco. Se venisse eseguita per prima  $T_c$ , il suo risultato sarebbe negativo e l’utente dovrebbe inviare un’ulteriore transazione per il recupero dei risultati. Nel caso preso in esame si suppone invece che la transazione  $T_b$  sia “minata” prima della transazione  $T_c$ .

*In questo caso l’utente attende 2 blocchi prima di poter ottenere il risultato della sua richiesta. Secondo i dati mostrati nella sezione precedente, l’attesa è di circa 13 secondi per blocco, per un totale di 26 secondi.*

Gli accessi gestibili in questo caso, considerando come accesso tutto il processo dalla richiesta al recupero dei risultati, si possono calcolare considerando le transazioni nello stesso blocco che hanno costo maggiore, cioè  $T_b$  e  $T_c$ . Si considera il costo delle operazioni al caso pessimo (il caso in cui il costo è il massimo possibile):

$$\text{RichiestePerBlocco: } \left\lfloor \frac{\text{GasBlocco}}{\text{Costo}(T_b+T_c)} \right\rfloor = \left\lfloor \frac{12470000}{514685} \right\rfloor = \lfloor 24.2284 \rfloor = 24$$

In questo caso quindi sono gestibili al massimo 24 richieste di accesso simultanee in ogni blocco, nell'ipotesi che tutto il gas disponibile del blocco venga utilizzato per il sistema.

$$\text{Richieste al secondo: } \left\lfloor \frac{24}{13} \right\rfloor = 1$$

Si precisa che la blockchain di Ethereum è una blockchain permissionless condivisa dai nodi che la utilizzano, quindi è improbabile che tutto il gas di un blocco venga utilizzato esclusivamente per zk-ABAC.

Il caso appena esaminato risulterebbe perfetto: rappresenta la situazione nella quale il tempo di attesa per recuperare i risultati di una richiesta è minimo, ma la situazione non è deterministica e quindi non controllabile. È possibile tentare di incentivare i nodi “miner” a elaborare prima le transazioni che rappresentano le richieste di interesse ( $T_a$ ), e solo dopo le altre ( $T_b$ ,  $T_c$ ), utilizzando diversi *GasPrice* per le transazioni. Come specificato nella sezione 2.1.6 ogni transazione può avere un prezzo diverso per unità di gas, in base al quale il nodo miner avrà un guadagno maggiore o minore. Alzando il costo del gas per le transazioni che devono essere eseguite prima si aumentano le probabilità che esse vengano eseguite prima di altre transazioni con costi minori nel solito blocco. Da notare che da [5] è possibile avere una stima dei costi consigliati per unità di gas al fine di far eseguire una transazione più o meno velocemente.

### 5.2.2 Caso di funzionamento standard

Nel caso standard le tre transazioni sono inserite in tre blocchi diversi, e non si verificano problemi di concorrenza. Si supponga che nel primo blocco sia inserita la transazione  $T_a$ , nel secondo  $T_b$ , nel terzo  $T_c$ .

Tramite le seguenti formule è possibile avere una stima delle operazioni gestibili in ogni blocco e conseguentemente delle operazioni gestibili al secondo.

- Operazioni per blocco:  $\left\lfloor \frac{\text{GasBlocco}}{\text{CostoTransazione}} \right\rfloor$
- Operazioni al secondo:  $\left\lfloor \frac{\text{OperazioniPerBlocco}}{\text{TempoBlocco}} \right\rfloor$

In tabella 5.4 sono mostrate le transazioni al secondo e le transazioni per blocco gestibili nel caso di funzionamento standard. Per tutte le transazioni è stato considerato il costo in gas al caso pessimo. In questo caso un utente, per recuperare i risultati della sua richiesta, attende un tempo pari a tre blocchi, che corrisponde a circa  $13 \cdot 3 = 39$  secondi [12].

Tabella 5.4: Operazioni gestibili al secondo e per blocco ipotizzando un uso esclusivo dello spazio nei blocchi.

	<b>Dimostrazione interesse</b>	<b>Memorizzazione proof</b>	<b>Recupero risultati</b>
<b>Per blocco</b>	91	54	43
<b>Al secondo</b>	7	4	3

### 5.2.3 Utilizzo di una blockchain dedicata per garantire buone prestazioni

Ipotizzando di avere a disposizione una blockchain dove poter impostare i valori per il limite di gas nei blocchi a piacimento, è possibile capire, tramite alcuni calcoli, quali sarebbero le risorse necessarie per garantire prestazioni buone per il sistema implementato.

Il costo di un accesso “completo” è la somma dei costi delle transazioni  $T_a$ ,  $T_b$ ,  $T_c$ , che equivale a 650430 gas.

Si supponga di volere gestire 1000 utenti al secondo che vogliono accedere a delle risorse protette.

$$T_a + T_b + T_c = 650430 \text{ gas}$$

$$\text{Tempo di Blocco} = 13 \text{ secondi}$$

In ogni blocco è necessario gestire 13000 tentativi di accesso diversi. Il gas limit di un blocco necessario per gestire queste operazioni è:

$$\text{Gas Limit Blocco: } 13000 \cdot 650430 = 8455590000 \text{ gas}$$

Il tool ZoKrates e le sue tempistiche potrebbero influenzare le richieste. A scopo di esempio, per generare una prova relativa di una politica con 3 input e 3 assert, come evidenziato nella tabella 5.2, è necessario calcolare un witness (in circa 163 ms) e generare poi l’effettiva proof (in circa 166 ms), per un totale di 0.329 secondi a prova. Disponendo di un elevato numero di macchine per la generazione delle prove, questi tempi non risultano un problema nel funzionamento del sistema.

## 5.3 Considerazioni finali

Dalle valutazioni effettuate sul sistema implementato si sono evidenziati alcuni punti critici. Il problema dell’attesa dell’utente è sicuramente il più importante. Un utente che voglia accedere ad una risorsa protetta sarebbe costretto ad aspettare il mining di al massimo a 3 blocchi, che si traduce in più di mezzo minuto di attesa. Questi tempi sono molto lunghi e possono limitare l’applicabilità del sistema. Si consideri, a scopo di esempio, un utente che voglia accedere ad una certa ala di un edificio tramite badge, e con tale ala protetta tramite zk-ABAC. L’utente, dopo aver strisciato il badge, dovrebbe attendere in media 40 secondi prima di poter accedere: questo è ovviamente un tempo eccessivo.

I costi del sistema, in termini di gas, sono un altro problema. La sola operazione di memorizzazione di una prova, al fine di garantire l’auditability, ha un costo di circa 4 dollari, come mostrato nella sezione 5.1.1. Se si volesse utilizzare il sistema “normalmente” in una situazione con un elevato numero di accessi giornalieri alle risorse, questa sarebbe una spesa considerevole. Una soluzione a tale problema potrebbe essere quella di suddividere il costo in parti uguali, o in proporzione, tra sistema e utente, in modo da ammortizzare le spese.

Relativo al problema del gas è presente anche il problema di scalabilità del sistema: come discusso nella sezione 5.1.1, per gestire una quantità attesa di utenti, il sistema richiede molto gas, ben oltre il limite attuale di Ethereum.

Questi problemi sono ereditati dalla tecnologia di base sottostante al sistema di controllo degli accessi. Ethereum è una blockchain *permissionless* e *pubblica*, dove ogni nodo è considerato *trustless*: nessuno dei nodi “può fidarsi” degli altri, e dunque tutti devono elaborare ogni transazione. Ciò garantisce un alto livello di sicurezza, ma allo stesso tempo limita notevolmente l’efficienza generale, al punto che l’intera blockchain non può elaborare più transazioni di un singolo nodo [16].

L’utilizzo di Ethereum per il progetto ha garantito, a scapito della velocità e delle prestazioni, la proprietà di auditability del sistema. Concentrando l’attenzione sulle performance, è ragionevole pensare di adottare una blockchain *permissioned*. In questo tipo di chain, a scapito della sicurezza, solo alcuni nodi considerati *trusted* possono validare i blocchi di transazioni. Con blockchain *permissioned* è possibile aumentare l’efficienza generale del sistema di controllo degli accessi, sia diminuendo i tempi di attesa per l’utente, sia aumentando il numero di richieste gestibili per unità di tempo.

# Capitolo 6

## Conclusioni

Lo scopo del progetto di tirocinio era quello di introdurre meccanismi di privacy in sistemi di controllo ABAC implementati su blockchain (basati sullo standard XACML). Nella prima parte di questo lavoro ci siamo focalizzati sullo studio dello stato dell'arte, in particolare della blockchain Ethereum e di che cosa sono e come funzionano le tecnologie zero-knowledge. È stato condotto un attento studio del tool ZoKrates, per capire se fosse possibile utilizzarlo allo scopo del progetto, sia in termini di semplicità che di efficienza e efficacia. Sono stati quindi proposti modelli per l'integrazione del tool ZoKrates con i sistemi di controllo precedentemente proposti per garantire la privacy nell'uso degli attributi privati. Sono stati mostrati modelli diversi, e infine è stata fornita un'implementazione di un possibile metodo di funzionamento, per dimostrare la fattibilità del progetto. Sono stati quindi effettuati alcuni test, per mostrare il funzionamento in casistiche e scenari realistici, raccogliendo poi dati nella fase di valutazione che hanno mostrato come sia necessario scendere a compromessi per garantire auditability o scalabilità: queste due proprietà non potranno essere garantite contemporaneamente, per lo meno in una blockchain permissionless come Ethereum. Nel progetto è stata rivolta particolare attenzione e concentrazione sulle garanzie di auditability per il sistema, ma nelle valutazioni sono stati evidenziati anche altri possibili approcci che permetterebbero il miglioramento delle performance del sistema di controllo, cioè le blockchain *permissioned*.

### 6.1 Lavori futuri

I vantaggi del sistema con l'integrazione del tool ZoKrates sono evidenti: è possibile per il sistema di controllo degli accessi fare uso di informazioni private sulla blockchain Ethereum, senza che queste debbano essere rese pubbliche e diffuse. Nel progetto sono stati evidenziati sia gli aspetti positivi, come le garanzie di privacy e l'auditability del sistema, che gli aspetti negativi, derivanti dalla natura stessa della blockchain utilizzata, quali le performance, la scalabilità del sistema, e la poca potenza espressiva di ZoKrates. Proprio su questo ultimo punto viene proposta una soluzione da sviluppare in futuro: l'uso

di un parser per ZoKrates. Sarebbe infatti interessante sviluppare un parser che presa una politica XACML la traduca in linguaggio ZoKrates automaticamente, utilizzando i costrutti base del linguaggio del tool.

Dalle valutazioni sono poi sorti problemi riguardanti l'efficienza del sistema e il suo costo: esso infatti nel caso di funzionamento “standard” arriva a poter accettare solamente 91 richieste di accesso a blocco, che si traducono in sole 7 richieste al secondo, un dato basso se si pensa anche al fatto che è stato considerato l’uso per intero del gas disponibile in un blocco. Anche i tempi di attesa per un utente sono problematici e troppo lunghi per applicazioni realistiche. Tutto ciò a causa delle tecnologie utilizzate alla base del sistema, cioè la blockchain *permissionless* e pubblica di Ethereum, che garantisce molti vantaggi al prezzo di un’efficienza ridotta. L’idea proposta è quella di continuare lo sviluppo del sistema di controllo degli accessi focalizzandosi sul modello modulare discusso in 3.2.2; inoltre si propone di sviluppare l’utilizzo di blockchain *permissioned* per aumentare le performance generali del sistema. Infine sarebbe interessante anche sviluppare un nuovo modello di interazione tra le varie componenti del sistema, non più basato sulle dimostrazioni di interesse, ma piuttosto su un protocollo di tipo *publish-subscribe* con *callback*.

# Bibliografia

- [1] Alberto Ballesteros Rodriguez. «zk-SNARKs Analysis and Implementation on Ethereum». Master's thesis. 2020.
- [2] Cornell Blockchain. *A Brief Dive Into zk-SNARKs and the ZoKrates Toolbox on the Ethereum Blockchain*. 2019. URL: <https://medium.com/cornellblockchain/a-brief-dive-into-zk-snarks-and-the-zokrates-toolbox-on-the-ethereum-blockchain-cb7bd7f00fdc>.
- [3] Damiano Di Francesco Maesa, Paolo Mori e Laura Ricci. «A blockchain based approach for the definition of auditable Access Control systems». In: *Computers & Security* 84 (2019), pp. 93–119. ISSN: 0167-4048. DOI: <https://doi.org/10.1016/j.cose.2019.03.016>. URL: <http://www.sciencedirect.com/science/article/pii/S0167404818309398>.
- [4] Jacob Eberhardt e Stefan Tai. «Zokrates-scalable privacy-preserving off-chain computations». In: *2018 IEEE International Conference on Internet of Things (iThings) and IEEE Green Computing and Communications (GreenCom) and IEEE Cyber, Physical and Social Computing (CPSCom) and IEEE Smart Data (SmartData)*. IEEE. 2018, pp. 1084–1091.
- [5] ETH GAS STATION. Consultato il 17/11/2020. URL: <https://ethgasstation.info/>.
- [6] Ethereum. *Ethereum Foundation*. URL: <https://ethereum.org/en/>.
- [7] Ethereum. *Ethereum Foundation*. URL: <https://ethereum.org/en/whitepaper/#messages-and-transactions/>.
- [8] Etherscan. URL: <https://etherscan.io/>.
- [9] Battaglia Ilenia. «Le curve ellittiche in crittografia». 2006. URL: <http://www1.unipa.it/~giovanni.falcone/tesilenia.pdf>.
- [10] Thomas Kerber. *Verifiable Computation in Smart Contracts*. 2017.
- [11] Loi Luu et al. «Making smart contracts smarter». In: *Proceedings of the 2016 ACM SIGSAC conference on computer and communications security*. 2016, pp. 254–269.
- [12] *Minimizing Data Storage Cost on the Ethereum Network*. 2019. URL: <http://proderivatives.com/blog/2019/5/10/minimizing-data-storage-cost-on-the-ethereum-network>.

- [13] OASIS. *eXtensible Access Control Markup Language (XACML) Version 3.0 Plus Errata 01*. 12-06-2017. URL: <http://docs.oasis-open.org/xacml/3.0/errata01/os/xacml-3.0-core-spec-errata01-os-complete.html>.
- [14] Alber Palau. *Storing on Ethereum. Analyzing the costs*. 17-06-2018. URL: <https://medium.com/coinmonks/storing-on-ethereum-analyzing-the-costs-922d41d6b316>.
- [15] *Remix IDE*. URL: <https://remix.ethereum.org>.
- [16] Mattias Scherer. *Performance and scalability of blockchain networks and smart contracts*. 2017.
- [17] William Stallings et al. *Computer security: principles and practice*. Pearson Education Upper Saddle River, NJ, USA, 2012.
- [18] *Types - Solidity 0.4.21 documentation*. [Consultato il 06-11-2020, versione Solidity 0.4.21]. URL: <https://solidity.readthedocs.io/en/v0.4.21/types.html>.
- [19] Wikipedia. *Blockchain — Wikipedia, L'enciclopedia libera*. [Consultata in data 26-10-2020]. 2020. URL: <http://it.wikipedia.org/w/index.php?title=Blockchain&oldid=116125247>.
- [20] Wikipedia. *Crittografia omomorfica — Wikipedia, L'enciclopedia libera*. [Online; in data 28-ottobre-2020]. 2020. URL: [http://it.wikipedia.org/w/index.php?title=Crittografia\\_omomorfica&oldid=114983089](http://it.wikipedia.org/w/index.php?title=Crittografia_omomorfica&oldid=114983089).
- [21] Wikipedia contributors. *Bitcoin — Wikipedia, The Free Encyclopedia*. <https://en.wikipedia.org/w/index.php?title=Bitcoin&oldid=986730123>. [Online; accessed 7-November-2020]. 2020.
- [22] Gavin Wood. *Ethereum: a secure decentralized generalised transaction ledger, pete-sburg version 3e2c089 – 2020-09-05*.
- [23] *ZoKrates*. URL: <https://zokrates.github.io>.
- [24] *ZoKrates*. URL: <https://zokrates.github.io/sha256example.html>.