



UNIVERSITÀ DEGLI STUDI DI BERGAMO

Dipartimento di Ingegneria Gestionale, dell'Informazione e della Produzione

Corso di laurea magistrale in Ingegneria Informatica

Classe n. LM-32 - Classe delle lauree magistrali in Ingegneria Informatica

Smart Contracts

Valutazione dei costi associati all'esecuzione e allo storage
di Ethereum Smart Contracts usando Solidity.

Sviluppo della DApp NotarizETH.

Relatore

Prof. Stefano Paraboschi

Tesi di Laurea Magistrale

Samuele Ferri

Matricola n. 1045975

ANNO ACCADEMICO 2020/2021

Indice

1	Introduzione	1
1.1	Blockchain	1
1.2	Ethereum	3
1.2.1	Ethereum Virtual Machine	4
1.2.2	Gas	7
1.2.3	Smart Contracts	14
1.2.4	EIP/ERC	15
1.2.5	Mainnet e Testnets	16
1.3	Solidity	17
1.3.1	Solidity Assembly	17
2	Obiettivo	19
2.1	Linea adottata	19
2.1.1	Analisi dei costi	19
2.1.2	Sviluppo della DApp NotarizETH	20
2.2	Tecnologie e Tools	21
2.2.1	Remix	21
2.2.2	Truffle	22
2.2.3	MetaMask	23
2.2.4	Infura	23
2.2.5	React	24
2.2.6	Tools di sviluppo	24
2.2.7	Tools di supporto	26
2.2.8	Siti di supporto	26
3	Analisi dei costi	27
3.1	Metodologia	27
3.2	Analisi relative al costo per il deployment del contratto	30
3.2.1	Contratti	30
3.2.2	Librerie	32

3.2.3	Dati on-chain e off-chain	33
3.3	Analisi relative al costo per l'esecuzione del contratto	34
3.3.1	Calcoli on-chain e off-chain	34
3.4	Chiarimenti riguardo ai tipi	36
3.4.1	Value Types	36
3.4.2	Reference Types	36
3.4.3	Arrays	37
3.4.4	Arrays speciali <code>bytes</code> e <code>string</code>	39
3.4.5	Structs	41
3.4.6	Mappings	41
3.5	Chiarimenti riguardo all'archiviazione	42
3.5.1	Layout dello storage	42
3.5.2	Layout della memoria	44
3.5.3	Locazione di default per l'archiviazione dei dati	45
3.5.4	Assegnamenti	45
3.6	Analisi dell'archiviazione nello storage	47
3.6.1	Archiviazione delle variabili di tipo elementare	47
3.6.2	Archiviazione degli arrays e delle structs	48
3.6.3	Compattare le variabili nello storage	49
3.6.4	Archiviazione nello storage (singolarmente, structs, encoding e decoding)	50
3.7	Analisi dei mappings nello storage	55
3.7.1	Analisi del funzionamento del Keccak256 hash con i mappings	55
3.7.2	Analisi dei mappings con valori di grande dimensione	58
3.7.3	Analisi dei mappings che non si compattano	59
3.7.4	Conclusioni sui mappings	60
3.8	Analisi degli arrays nello storage	62
3.8.1	Confronto dei costi tra i vari tipi di arrays	62
3.8.2	Analisi degli arrays dinamici	69
3.8.3	Analisi dei <code>bytes</code> e delle <code>string</code>	71
3.8.4	Unchecked	73
3.8.5	Conclusioni sugli arrays	74
3.9	Chiarimenti riguardo le funzioni	76
3.9.1	Visibilità di funzione	76
3.9.2	Modificatori di funzione	77
3.9.3	Chiamate di funzione	77
3.9.4	Overloading	78
3.9.5	Overriding	78
3.10	Analisi delle funzioni	79

3.10.1	Visibilità delle funzioni	79
3.10.2	Ordinamento delle funzioni (Method ID)	81
3.10.3	Compattare gli argomenti in input alle funzioni	83
3.10.4	Saturazione dello stack	83
3.11	Analisi degli statements	84
3.11.1	Cicli	84
3.11.2	Valutazione a corto circuito	87
3.12	Analisi degli eventi ed errori	89
3.12.1	Eventi	89
3.12.2	Errori	90
3.13	Inline Assembly	92
3.13.1	Manipolazione dello stack	93
3.13.2	Manipolazione della memoria	93
3.13.3	Manipolazione dello storage	95
3.13.4	Migliorare l'efficienza del compilatore	96
4	Sviluppo della DApp NotarizETH	99
4.1	Smart Contract	100
4.1.1	Struttura principale	102
4.1.2	Controllo dell'accesso	103
4.1.3	Analisi del costo del gas	104
4.2	Applicazione	106
4.2.1	React	108
4.2.2	MetaMask	111
4.2.3	Infura	111
4.3	Casi d'uso	112
4.3.1	Certificazione di un file	112
4.3.2	Verifica di un file hash	114
4.3.3	Reset di un file hash	115
5	Conclusioni	117
5.1	Sviluppi futuri	117
5.1.1	Analisi dei costi	117
5.1.2	Sviluppo della DApp NotarizETH	118
5.2	Conclusioni	119

Capitolo 1

Introduzione

Parte introduttiva sulla blockchain, sull'universo Ethereum e sulle problematiche relative al consumo del gas durante l'esecuzione degli smart contracts scritti in Solidity.

1.1 Blockchain

Una blockchain è una struttura dati condivisa e immutabile, può essere descritta come un database pubblico che viene aggiornato e condiviso tra molti nodi in una rete (*distributed ledger* o *libro mastro*).

Block si riferisce al fatto che i dati e lo stato del sistema vengono memorizzati in batch sequenziali.

Chain si riferisce al fatto che ogni blocco fa riferimento al suo genitore, la cui integrità è garantita dall'uso della crittografia. I dati di un blocco non possono essere modificati senza modificare tutti i blocchi successivi, il che richiederebbe il consenso della rete di nodi.

La dimensione della blockchain è destinata a crescere nel tempo, ogni nuovo blocco e la catena nel suo insieme devono essere concordati da ogni nodo della rete; in questo modo tutti conservano gli stessi dati. Perché funzioni, le blockchain necessitano di un meccanismo di consenso. Esistono vari meccanismi di consenso, i principali sono:

- **Proof of Work (PoW)**: i miners sono tenuti a risolvere dei problemi matematici estremamente complessi e computazionalmente difficili per poter aggiungere blocchi alla blockchain. Questo serve come misura economica (richiedendo grandi quantità di energia elettrica per effettuare la computazione) per scoraggiare attacchi denial of service e altri abusi di servizio come spam sulla rete. Un esempio è l'algoritmo Hashcash (SHA-256) usato alla base della criptovaluta Bitcoin¹.

¹Bitcoin (<https://bitcoin.org/>)

- **Proof of Stake (PoS):** si basa sul principio che a ogni utente venga richiesto di dimostrare il possesso di un certo ammontare di criptovaluta in modo tale da aumentare le possibilità di essere selezionati per convalidare un blocco. La criptovaluta messa in gioco è bloccata come deposito per garantire che il minatore convalidi il blocco secondo le regole stabilite; nel caso in cui il minatore violi le regole, il suo deposito sarà bruciato.
- **Proof of Authority (PoA):** alternativa al Proof of Stake, invece di depositare criptovaluta, si mette in gioco la propria identità. Le transizioni e i blocchi sono convalidati da account approvati, noti come validatori.

Le transizioni sono le richieste di calcolo proposte da un qualsiasi nodo della rete. Ogni transazione deve essere estratta e inclusa in un nuovo blocco. I nuovi blocchi vengono trasmessi ai nodi della rete, controllati e verificati (ogni nodo deve sempre verificare ogni blocco, mai fidarsi degli altri nodi), aggiornando lo stato condiviso per tutti.

Don't trust. Verify.

I meccanismi crittografici garantiscono che una volta che le transazioni siano state verificate come valide e aggiunte alla blockchain, non possano essere manomesse in seguito; inoltre, gli stessi meccanismi assicurano che tutte le transazioni siano firmate ed eseguite con autorizzazioni appropriate.

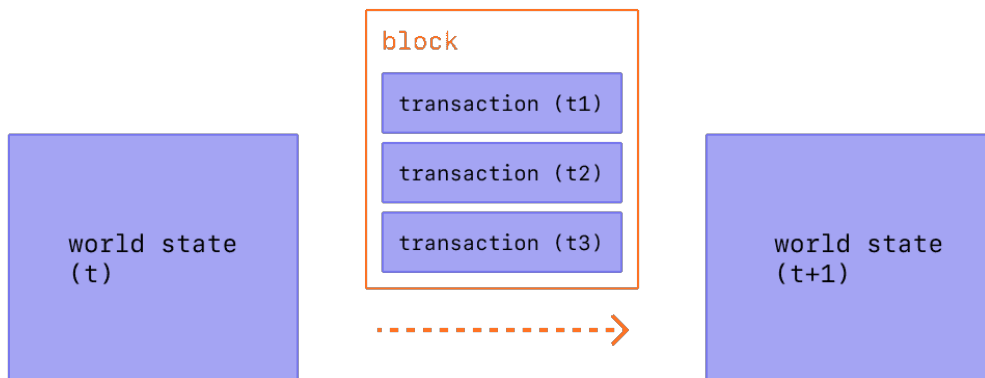


Figura 1.1: Transazioni contenute nei blocchi

1.2 Ethereum

Ethereum [1] [2] è una blockchain programmabile proposta nel 2013 da Vitalik Buterin e lanciata successivamente nel 2015. È una piattaforma decentralizzata del Web3 per la creazione e pubblicazione peer-to-peer di contratti intelligenti (smart contracts) creati in un linguaggio di programmazione Turing completo².



Figura 1.2: Logo di Ethereum

A differenza di una blockchain come Bitcoin e l'omonima criptovaluta che può essere vista come un *libro mastro distribuito* in cui alcune regole ben definite regolano la modifica del libro mastro, Ethereum ha anch'esso la sua criptovaluta nativa Ether (ETH) che segue determinate regole ma ha anche una funzione più potente: gli smart contracts (sottosezione 1.2.3). Serve quindi un'analogia più sofisticata: invece di un *libro mastro distribuito*, Ethereum è una *macchina a stati distribuita*. Lo stato di Ethereum è una grande struttura di dati che contiene non solo tutti gli accounts e i relativi saldi, ma uno stato macchina, che può cambiare da blocco a blocco secondo un insieme predefinito di regole e che può eseguire codice arbitrario. Le regole specifiche per cambiare stato da blocco a blocco sono definite dall'Ethereum Virtual Machine (sottosezione 1.2.1).

Attualmente adotta un meccanismo di consenso Proof of Work (PoW), basandosi quindi sul mining per risolvere un puzzle complicato per cui è necessaria molta potenza di calcolo. I miners procedono per tentativi ed errori, risolvere il puzzle dimostra che si sono spese molte risorse di calcolo per la creazione di un blocco. In caso di successo e convalida di un nuovo blocco si viene ricompensati da una determinata quantità di Ether, in parte generata e in parte derivante dalle fees delle transazioni; la criptovaluta nativa di Ethereum che permette l'esistenza del mercato per la computazione in cui gli utenti pagano Ether ad altri utenti per soddisfare le proprie richieste di esecuzione del codice.

²Un linguaggio è detto Turing completo se può eseguire qualunque programma che una macchina di Turing può eseguire dati sufficienti tempo e memoria.

Tuttavia, Ethereum si sta spostando verso un meccanismo di consenso Proof of Stake (PoS): con il termine ETH2 [3] si riferisce a una serie di aggiornamenti interconnessi (sono previste più fasi nella roadmap) che renderanno Ethereum più scalabile, più sicuro e più sostenibile. Questi aggiornamenti sono attualmente in sviluppo da più team di tutto l'ecosistema Ethereum.

1.2.1 Ethereum Virtual Machine

Nell'universo di Ethereum, esiste un'unica macchina virtuale deterministica chiamata Ethereum Virtual Machine (EVM) progettata per eseguire in modo sicuro codice non attendibile in una rete blockchain globale sul cui stato concordano tutti i nodi della rete Ethereum. L'EVM può essere considerata una macchina di Turing completa e perciò una Universal Turing Machine (UTM) permettendo, date risorse sufficienti (memoria e tempo), di eseguire un qualsiasi algoritmo concepibile.

Tutti coloro che partecipano alla rete Ethereum conservano una copia dello stato di questa macchina, compresi tutti gli account e gli smart contracts distribuiti. Qualsiasi partecipante può trasmettere una richiesta all'EVM per eseguire calcoli arbitrari. Ogni volta che tale richiesta viene trasmessa, altri partecipanti sulla rete verificano, convalidano ed eseguono il calcolo: ciò causa un cambiamento di stato nella EVM che viene propagato in tutta la rete.

Il record di tutte le transazioni effettuate così come lo stato attuale della EVM è archiviato nella blockchain, che a sua volta è archiviata e concordata da tutti i nodi connessi alla rete.

Ci sono due tipi di account che condividono lo stesso spazio d'indirizzi:

- *External Accounts*: controllati da coppie di chiavi pubblica-privata degli utenti.
- *Contract Accounts*: controllati dal codice dello smart contract.

L'indirizzo di un account esterno è determinato dalla chiave pubblica mentre l'indirizzo di un contratto è determinato al momento della creazione del contratto (è derivato dall'indirizzo del creatore e dal numero di transazioni inviate da quell'indirizzo, il cosiddetto *nonce*).

Indipendentemente dal fatto che l'account memorizzi o meno il codice, i due tipi vengono trattati allo stesso modo dall'EVM. Ogni account ha un saldo in Ether (in *wei* per l'esattezza) che può essere modificato inviando transazioni che includono Ether.

L'EVM adotta la *meccanica del gas* per limitare la portata delle esecuzioni degli smart contracts. Inoltre, ogni contratto viene eseguito in modalità sandbox e può modificare

soltanto il proprio stato e interagire con altri smart contracts trasmettendo singoli arrays di bytes di lunghezza arbitraria.

Struttura dell'Ethereum Virtual Machine

In Ethereum, lo stato è un'enorme struttura di dati chiamata Modified Merkle Patricia Trie [4] che mantiene tutti gli account collegati da un albero di hash e riducibili a un singolo hash di root archiviato sulla blockchain.

Le transazioni sono firmate crittografica mente dagli account; ne esistono di due tipi: quelle che danno luogo a chiamate di messaggi e quelle che danno luogo alla creazione di contratti. La creazione del contratto consiste nella creazione di un nuovo account contenente il bytecode dello smart contract. Ogniqualvolta un altro account effettua una chiamata al contratto, viene eseguito il suo bytecode.

I contratti, per archiviare dati tra varie esecuzioni del programma, contengono un Modified Merkle Patricia Trie associato all'account in questione e parte dello stato globale.

Il bytecode dello smart contract compilato viene eseguito come una serie di opcodes [5] che eseguono operazioni di stack standard come XOR, AND, ADD, SUB, operazioni in memoria come MSTORE, MLOAD e operazioni nello storage come SSTORE, SLOAD. L'EVM implementa anche una serie di operazioni specifiche della blockchain come ADDRESS, BALANCE, SHA3, BLOCKHASH.

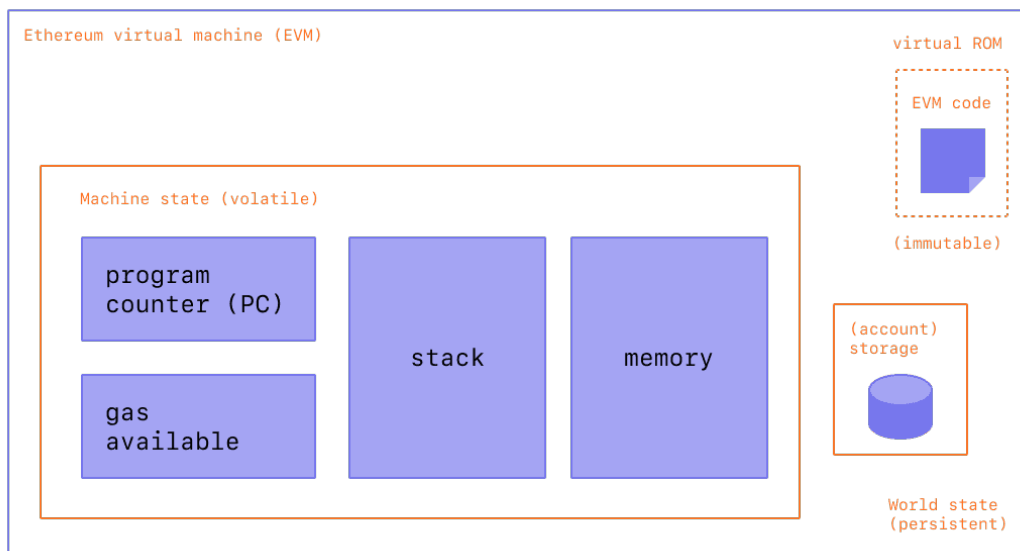


Figura 1.3: Diagramma della struttura dell'EVM

L'EVM, come si vede dalla Figura 1.3, ha tre aree in cui può memorizzare i dati:

- **Storage:** è un'area dati persistente tra le chiamate di funzione e le transazioni relative a ogni account. Lo storage è un archivio di key-value entrambi di 32 byte. Non è possibile enumerare lo storage dall'interno di un contratto, è relativamente costoso da leggere e ancor di più inizializzarlo e modificarlo. A causa dei costi elevati, è necessario ridurre al minimo ciò che si vuole archiviare nella memoria persistente; dati come calcoli derivati, caching e valori aggregati vanno salvati al di fuori dello storage. Inoltre, un contratto non può né leggere né scrivere in alcun storage diverso dal proprio.
- **Memoria:** è un byte-array che mantiene i dati fino alla chiusura di una determinata funzione, quindi non persiste tra varie transazioni. La memoria inizialmente ha dimensione nulla e può essere espansa in chunks di 32 byte. La memoria è lineare e può essere indirizzata a livello di byte; tuttavia, le letture sono limitate a una larghezza di 256 bit, mentre le scritture possono essere di 8 o 256 bit. Quando la memoria si espande, bisogna pagare del gas e questo costo cresce quadraticamente rispetto alla memoria usata.
- **Stack:** è usato per salvare piccole variabili locali, ha un costo simile alla memoria ma ha un limite al numero di valori che possono essere acceduti (solo i 16 in cima allo stack). Essendo l'EVM una macchina a stack e non a registri, tutti i calcoli locali vengono eseguiti nello stack. Ha una profondità di 1024 elementi e contiene words di 32 byte, grandezza scelta per la massima compatibilità con lo schema Keccak256 hash. L'accesso allo stack è limitato superiormente nel modo seguente: è possibile copiare uno dei 16 elementi posti più in alto in cima allo stack o scambiare l'elemento più in alto con uno dei 16 elementi sottostanti. Tutte le altre operazioni prendono i primi due (o uno, o più, a seconda dell'operazione) elementi dallo stack e inseriscono il risultato in cima allo stack. Ovviamente è possibile spostare gli elementi dello stack nello storage o nella memoria per ottenere un accesso più profondo allo stack (quindi oltre ai 16 valori in cima allo stack), ma non è possibile accedere solo a elementi arbitrari più in profondità nello stack senza aver prima rimosso la parte superiore dello stack.

Durante le analisi dei costi nel Capitolo 3 ci si concentrerà in particolar modo sullo storage, dato che è l'area che incide maggiormente sui costi totali delle transazioni effettuate dagli smart contracts.

1.2.2 Gas

Il gas [6] misura la quantità di sforzo computazionale richiesto per eseguire operazioni specifiche sulla rete Ethereum. Poiché ogni transazione richiede risorse computazionali per l'esecuzione, ogni transazione richiede una commissione. Il gas si riferisce alla commissione richiesta per condurre con successo una transazione su Ethereum che viene pagata in sottomultipli di Ether (ETH).

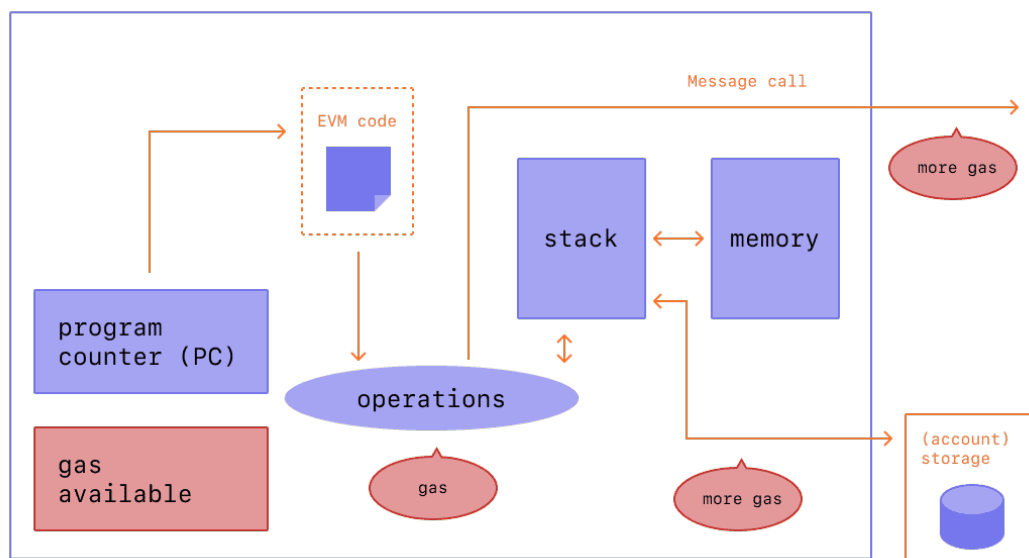


Figura 1.4: Diagramma che mostra dove è usato il gas nell'EVM

Le fees aiutano a mantenere sicura la rete Ethereum. Richiedendo una tassa per ogni calcolo eseguito sulla rete, vengono scoraggiate le transazioni computazionalmente costose da eseguire e si impedisce anche l'invio spam sulla rete. Al fine di prevenire cicli infiniti accidentali o altri sprechi computazionali nel codice, in ogni transazione è richiesto un limite al numero di passaggi computazionali che possono essere eseguiti dal bytecode.

Ogni operazione sulla EVM consuma una certa quantità di gas che va ad aggiungersi al costo della transizione base (solo trasferimento di Ether) che è pari a 21 000. Nel caso di chiamata a un contratto, bisogna tener conto anche del gas consumato da ogni singola istruzione di basso livello (opcodes).

L'elenco dettagliato dei costi di ogni singola istruzione di basso livello è presente nella Figura 1.5 ricavata dall'*Ethereum Yellow Paper* [7] originariamente scritto da Gavin Wood e attualmente mantenuto dalla comunità.

APPENDIX G. FEE SCHEDULE

The fee schedule G is a tuple of 31 scalar values corresponding to the relative costs, in gas, of a number of abstract operations that a transaction may effect.

Name	Value	Description*
G_{zero}	0	Nothing paid for operations of the set W_{zero} .
G_{base}	2	Amount of gas to pay for operations of the set W_{base} .
$G_{verylow}$	3	Amount of gas to pay for operations of the set $W_{verylow}$.
G_{low}	5	Amount of gas to pay for operations of the set W_{low} .
G_{mid}	8	Amount of gas to pay for operations of the set W_{mid} .
G_{high}	10	Amount of gas to pay for operations of the set W_{high} .
$G_{extcode}$	700	Amount of gas to pay for an EXTCODESIZE operation.
$G_{extcodehash}$	700	Amount of gas to pay for an EXTCODEHASH operation.
$G_{balance}$	700	Amount of gas to pay for a BALANCE operation.
G_{sload}	800	Paid for a SLOAD operation.
$G_{jumpdest}$	1	Paid for a JUMPDEST operation.
G_{sset}	20000	Paid for an SSTORE operation when the storage value is set to non-zero from zero.
G_{sreset}	5000	Paid for an SSTORE operation when the storage value's zeroness remains unchanged or is set to zero.
R_{sclear}	15000	Refund given (added into refund counter) when the storage value is set to zero from non-zero.
$R_{selfdestruct}$	24000	Refund given (added into refund counter) for self-destructing an account.
$G_{selfdestruct}$	5000	Amount of gas to pay for a SELFDESTRUCT operation.
G_{create}	32000	Paid for a CREATE operation.
$G_{codedeposit}$	200	Paid per byte for a CREATE operation to succeed in placing code into state.
G_{call}	700	Paid for a CALL operation.
$G_{callvalue}$	9000	Paid for a non-zero value transfer as part of the CALL operation.
$G_{callstipend}$	2300	A stipend for the called contract subtracted from $G_{callvalue}$ for a non-zero value transfer.
$G_{newaccount}$	25000	Paid for a CALL or SELFDESTRUCT operation which creates an account.
G_{exp}	10	Partial payment for an EXP operation.
$G_{expbyte}$	50	Partial payment when multiplied by $\lceil \log_{256}(exponent) \rceil$ for the EXP operation.
G_{memory}	3	Paid for every additional word when expanding memory.
$G_{txcreate}$	32000	Paid by all contract-creating transactions after the <i>Homestead</i> transition.
$G_{txdatazero}$	4	Paid for every zero byte of data or code for a transaction.
$G_{txdatanonzero}$	16	Paid for every non-zero byte of data or code for a transaction.
$G_{transaction}$	21000	Paid for every transaction.
G_{log}	375	Partial payment for a LOG operation.
$G_{logdata}$	8	Paid for each byte in a LOG operation's data.
$G_{logtopic}$	375	Paid for each topic of a LOG operation.
G_{sha3}	30	Paid for each SHA3 operation.
$G_{sha3word}$	6	Paid for each word (rounded up) for input data to a SHA3 operation.
G_{copy}	3	Partial payment for *COPY operations, multiplied by words copied, rounded up.
$G_{blockhash}$	20	Payment for BLOCKHASH operation.
$G_{quaddivisor}$	20	The quadratic coefficient of the input sizes of the exponentiation-over-modulo precompiled contract.

Figura 1.5: Ethereum Yellow Paper Gas Fees aggiornate alla Petersburg Version 41c1837 del giorno 2021-02-14

L'accesso alla memoria o la scrittura su disco hanno costi diversi durante l'esecuzione di un contratto. Salvare i dati nello stack attraverso l'operazione **PUSH** è quella più economia in termini di gas, seguita da salvare i dati nella memoria con l'operazione **MSTORE** e dal salvare i dati nello storage con l'operazione **SSTORE** che è la più costosa. In generale, le istruzioni che portano alla crescita della dimensione della blockchain, come la memorizzazione e la modifica delle variabili di stato, la creazione di contratti e l'emissione di eventi, sono molto costose. In sostanza, più complesso è il contratto e più operazioni esegue, più costoso è eseguirlo.

Definizioni di gas

Il mittente di una transazione specifica quanto è disposto a pagare per ogni unità di gas; questa quantità è nota come *gas price* e ha come unità di misura *Gwei/gas* dove un *Gwei* è un sottomultiplo di Ether pari a 0,000000001 ETH (10^{-9} ETH).

```
1 assert(1 wei == 1);
2 assert(1 gwei == 1e9);
3 assert(1 ether == 1e18);
```

Listing 1.1: Ether Units

Il costo di una transazione si può quindi calcolare facendo: *gas used * gas price*.

Tuttavia, per evitare consumi di gas elevati o addirittura infiniti tramite loop senza uscita, esiste un limite superiore di gas per ogni transazione. Nel caso si sforasse questo limite l'EVM interromperebbe l'esecuzione del contratto, ma addebiterebbe comunque alla transazione dannosa l'intera commissione; questo perché anche se la transazione ha fallito, i miners hanno dovuto convalidare ed eseguire la transazione, il che richiede potenza di calcolo.

Altra definizione è quella di *gas block limit*: è un limite che determina la quantità di calcoli che è possibile eseguire sulla rete Ethereum per blocco; è stato introdotto con il fine di tenere sotto controllo le dimensioni dello stato e l'*uncle rate*, ossia i premi che si danno ai miners nel caso di scoperta contemporanea di due blocchi che gareggiano attraverso la rete. Il vincitore, ossia il blocco che si propaga nella rete per primo, ottiene il blocco principale con una ricompensa di 2 ETH (alla data attuale, possibili halving futuri), mentre il perdente riceve un blocco con una ricompensa inferiore stabilita dall'indice *uncle rate* (circa al 4.50%³ alla data odierna) come premio al tentativo di combattere la centralizzazione e fornire blocchi alternativi.

Nella Tabella 1.1 è mostrato un riassunto delle definizioni di gas.

Termine	Definizione
Gas Used	Unità per il calcolo computazionale eseguito
Gas Price	Quanto il mittente è disposto a pagare per ogni unità di gas (<i>Gwei</i>)
Tx Cost	Gas Used * Gas Price
Gas limit	Max quantità di gas che il mittente è disposto a pagare per una Tx
Gas Block Limit	Max quantità di gas permessa in un blocco

Tabella 1.1: Definizioni di gas

³Ethereum Uncle Rate (https://ycharts.com/indicators/ethereum_uncle_rate)

Il *gas price* raccomandato e stimato per eseguire in tempi rapidi una transizione nella mainnet di Ethereum viene costantemente aggiornato su siti come ETH Gas Station⁴.

Nella Figura 1.6 è mostrato un grafico rappresentante il prezzo medio del gas in *Gwei*.

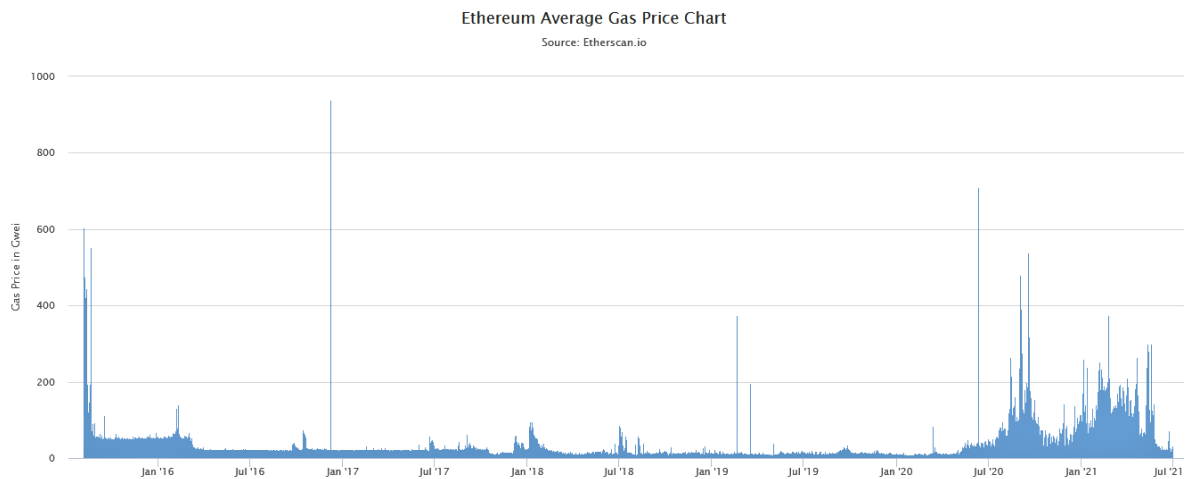


Figura 1.6: Prezzo medio del gas in Ethereum

Sebbene una transazione includa un limite, tutto il gas non usato in una transazione viene restituito all'utente come mostrato nella Figura 1.7.

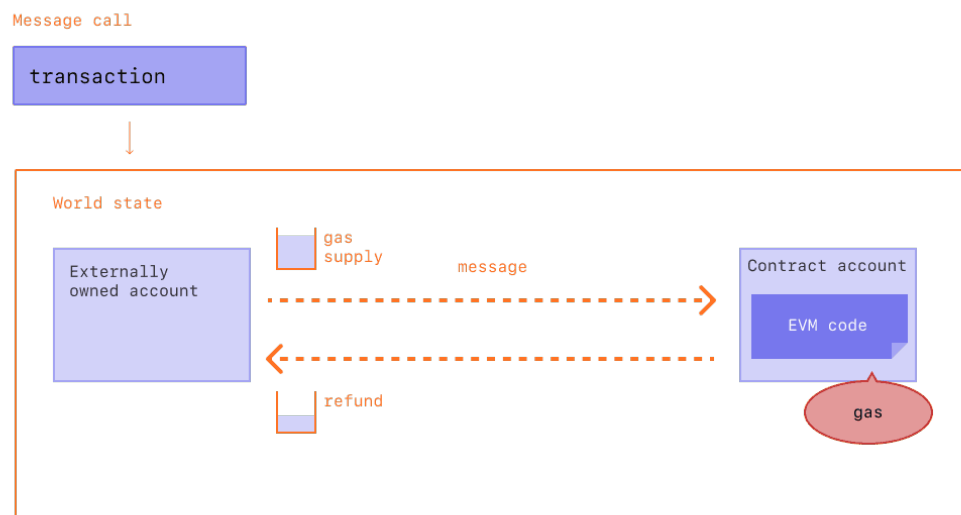


Figura 1.7: Diagramma che mostra il rimborso del gas non usato

⁴ETH Gas Station (<https://ethgasstation.info/>)

Consumo di gas per operazioni in memoria e nello storage

Il consumo di gas per la **memoria** è determinato dalla seguente formula definita nel Ethereum Yellow Paper [7]:

$$C_{mem}(a) \equiv G_{memory} \cdot a + \left\lfloor \frac{a^2}{512} \right\rfloor \quad (1.1)$$

In cui `G_memory` vale 3 (costo singolo), `a` è il numero di chunks da 32 bytes allocati.

Si può notare dalla formula che il costo della memoria cresce quadraticamente con lo spazio occupato. Per diminuire il costo del gas è possibile, come si vedrà nel Capitolo 3, compattare più variabili in un unico slot.

Recentemente, con l'introduzione del EIP-2929 [8] (Ethereum Improvement Proposals, vedi sottosezione 1.2.4) avanzata da Vitalik Buterin e Martin Swende e resa effettiva sulla mainnet a partire dall'hard fork Berlin [9] avvenuto il 14 aprile 2021, sono cambiati i costi per gli opcodes che gestiscono i dati nello **storage** [10].

Prima di analizzarli, bisogna introdurre il nuovo concetto introdotto dall'EIP-2929 relativo agli *accessed addresses* e le *accessed storage keys*. Un address o una storage key è considerata *accessed* se è stata prima usata durante la transazione. Ad esempio, quando si chiama un altro contratto, l'indirizzo di tale contratto è considerato *accessed*. Similmente, quando si effettuano le operazioni di `SLOAD` o `SSTORE` su uno slot, quello slot verrà considerato *accessed* per tutto il resto della transazione. Non importa quale opcode lo fa: se un `SLOAD` legge uno slot, è considerato *accessed* per tutte le successive `SLOAD` e `SSTORE`. Ultima cosa, quando uno slot di storage è segnato come *accessed*, si sta dicendo che la coppia (address, storage_key) è stata segnata come *accessed*, quindi relativa solo a uno specifico account.

Il consumo di gas nello **storage** segue le seguenti regole:

L'operazione `SLOAD`, prima dell'hard fork Berlin, aveva un costo fisso di 800. Ora dipende dal fatto che lo slot di archiviazione sia stato già marcato come *accessed* o meno. Se non è stato eseguito l'accesso, il costo è 2100; se è marcato come *accessed* è 100. Quindi uno `SLOAD` costa 2000 in meno se lo slot è presente nell'elenco delle *accessed storage keys*. In questo modo, rispetto a prima dell'hard fork Berlin, il costo di tante operazioni `SLOAD` consecutive viene ridotto notevolmente.

L'operazione `SSTORE`, prima dell'hard fork Berlin, aveva un costo di 20 000 gas quando un valore veniva settato da zero a un valore non-zero e aveva un costo di 5 000 gas quando si modificava uno slot di storage già esistente (come mostrato dal non aggiornato Ethereum Yellow Paper in Figura 1.5).

Ora segue le seguenti regole:

- Se il valore dello slot cambia da un valore zero a un qualsiasi valore non-zero, il costo è:
 - 22 100 gas se la storage key non è marcata come accessed.
 - 20 000 gas se la storage key è marcata come accessed.
- Se il valore dello slot cambia da un valore non-zero a un qualsiasi altro valore non-zero, il costo è:
 - 5 000 gas se la storage key non è marcata come accessed.
 - 2 900 gas se la storage key è marcata come accessed.
- Se il valore dello slot cambia da un valore non-zero a un valore zero, il costo è identico al secondo caso, in aggiunta a un rimborso.
- Se il valore è già stato modificato durante la stessa transazione, tutte le successive operazioni `SSTORE` costano 100 gas.

Viene riassunto il tutto nella Tabella 1.2.

OPCODE	Before Berlin	After Berlin	
		Not Accessed	Accessed
SLOAD	800	2100	100
SSTORE from 0 to 1	20000	22100	20000
SSTORE from 1 to 2	5000	5000	2900
SLOAD + SSTORE*	5800	5000	3000
SSTORE* + SLOAD	5800	5100	3000
SSTORE of an already written slot	800	100	
*From a non-zero value to a different non-zero value, like in the third row			

Tabella 1.2: Riassunto sulle operazioni di storage dopo l'EIP-2929

Lo scopo dell'EIP-2929 è quello di fornire un'ulteriore protezione Denial of Service visto che ricerche precedenti [11] avevano evidenziato una vulnerabilità che consisteva nel creare alcuni blocchi che semplicemente facevano accesso a tanti accounts e ciò richiedeva fino a 80 secondi per processare il tutto. Ora le operazioni che richiedono più tempo (come l'accesso allo storage) costano più gas, perciò l'attacco DoS è stimato essere circa tre volte più debole.

Inoltre, vi sono delle conseguenze positive in termini di gas: solo i primi accessi allo storage costano molto, i successivi accessi sugli stessi slot costano molto meno andando a ridurre il costo totale delle operazioni `SLOAD` seguite da `SSTORE` (o il contrario) di ben 800

unità di gas (700 nel caso contrario). Grazie a ciò operazioni come l'invio di ERC-20 o il self-calling sono diventate più economiche. In aggiunta a ciò vi sono dei miglioramenti nei client dei nodi Ethereum che implementano le *on-disk storage caches*, riducendo il numero di accessi sullo storage.

Questo riprezzamento del costo del gas e i miglioramenti effettuati lato client rendono la blockchain Ethereum più sicura e permettono gas limits più alti di quelli attuali. Il motivo principale per cui, dopo l'EIP-2929, si evita l'aumento del gas limit non è relativo agli attacchi DoS ma alla crescita della dimensione dello stato della blockchain, aspetto prioritario che verrà trattato in future EIP.

Rimborsi di gas

Per quanto riguarda i rimborsi, essi vengono man mano aggiunti al *refund counter* durante la transizione e, sommati al gas non usato al termine della transizione, vengono restituiti al mittente. Tuttavia il rimborso accumulato non può eccedere la metà del gas usato nel contesto corrente: questo perché potrebbe i miners a pagare per l'esecuzione di un contratto. I seguenti opcodes li attivano:

- **SELFDESTRUCT**: operazione di autodistruzione del contratto che rimborsa 24 000 gas.
- **SSTORE[x] = 0**: operazione di cancellazione di uno slot dello storage azzerandone tutti i bytes che rimborsa 15 000 gas.

Tuttavia vi sono due proposte EIP-3298 [12] ed EIP-3403 [13] avanzate da Vitalik Buterin e Martin Swende, allo stato attuale di draft, che vogliono rimuovere del tutto i rimborsi (nel caso della prima proposta) oppure rimuoverli solo in parte (nel caso della seconda proposta). Nella seconda proposta EIP-3403, per quanto riguarda l'operazione di **SELFDESTRUCT** vengono rimossi tutti i rimborsi mentre per l'operazione di **SSTORE[x] = 0** l'operazione di rimborso viene ristretta solo al seguente caso: se il nuovo valore e il valore originale dello slot di storage sono entrambi uguali a 0 ma il valore corrente non lo è, viene effettuato il rimborso di 15 000 gas.

In aggiunta a queste due proposte, nell'aprile 2021 ne è stata avanzata un'altra, l'EIP-3529 [14], in cui si vuole rimuovere i rimborsi di gas per l'operazione di **SELFDESTRUCT** e ridurre i rimborsi di gas per l'operazione **SSTORE** a un livello più basso dove i rimborsi sono ancora sostanziosi, ma non sono più abbastanza alti da rendere praticabili gli attuali exploit del meccanismo di rimborso. Queste restrizioni sono state proposte per arginare alcuni exploit che sfruttano i periodi di basso costo del gas per stoccare contratti rilasciandoli nei periodi di maggior congestione della rete usufruendo del rimborso: un esempio è il GasToken [15].

1.2.3 Smart Contracts

Uno smart contract è un programma che gira sulla blockchain di Ethereum che può eseguire delle operazioni incluso il trasferimento di moneta o fare verifiche temporali sullo stato di oggetti o informazioni. È una raccolta di codice (le sue funzioni) e dati (il suo stato) che risiede in un indirizzo specifico sulla blockchain di Ethereum.

Gli smart contracts sono un tipo di account Ethereum: hanno un saldo e possono inviare transazioni sulla rete. Tuttavia, non sono controllati da un utente ma vengono invece distribuiti sulla rete ed eseguiti come sono stati programmati. Gli account utente possono quindi interagire con uno smart contract inviando transazioni che eseguono una funzione definita nel contratto. Inoltre, gli smart contracts possono definire regole e farle rispettare automaticamente tramite il codice.

Le potenzialità degli smart contracts sono enormi, sono in grado di permettere attività di *notarization* così come gestione di contratti di ogni tipo tra persone, enti o società, o regolare le condizioni per le quali devono avvenire passaggi di proprietà o di moneta/-token. Addirittura si pensa che, in ambito della Teoria dei giochi, possano trasformare un qualsiasi gioco non cooperativo in un gioco cooperativo (ad esempio il dilemma del prigioniero, la caccia al cervo, il gioco del pollo...) [16].

Permissionless

Chiunque può scrivere uno smart contract e distribuirlo sulla rete. Per farlo bisogna scrivere il contratto in un linguaggio specifico (ad esempio Solidity [17] (vedi sezione 1.3), Vyper [18] ...) e successivamente il codice deve essere compilato prima di poter essere distribuito in modo che l'EVM possa interpretare e archiviare il contratto. Inoltre, bisogna avere abbastanza Ether per effettuare il deploy del contratto: i costi del gas per l'implementazione del contratto sono molto più alti di una semplice transazione di trasferimento di Ether.

Modularità

Gli smart contracts sono pubblici su Ethereum e possono essere pensati come *open API*. Ciò significa che è possibile che un contratto chiami altri contratti estendendo notevolmente le cose che si possono fare. Inoltre, i contratti possono distribuire a loro volta altri contratti sulla rete.

Limitazioni

Gli smart contracts da soli non possono ottenere informazioni sugli eventi del mondo reale perché, ad esempio, non possono inviare richieste HTTP. Questa limitazione serve per impedire ai contratti di fare affidamento su informazioni esterne che potrebbero mettere a repentaglio il meccanismo del consenso. Tuttavia, stanno nascendo numerosi oracoli

che collegano Ethereum alle informazioni off-chain del mondo reale, in modo tale che gli smart contracts possano eseguire query su questi dati (ad esempio Chainlink ⁵).

Security

Oltre alle limitazioni imposte per garantire la sicurezza e la decentralizzazione della rete, uno sviluppatore di smart contracts deve effettuare ampi test completi sui propri contratti prima di distribuirli. Infatti una volta distribuiti non sarà possibile modificarli ed eventuali vulnerabilità potrebbero causare comportamenti indesiderati.

Autodistruzione e disattivazione

L'unico modo per rimuovere il codice dallo stato della blockchain è procedere con l'operazione di `SELFDESTRUCT` del contratto (il codice rimarrà tuttavia parte della storia della blockchain). Il saldo Ether rimanente memorizzato a quell'indirizzo viene inviato a una destinazione designata in precedenza e sia la memoria che il codice vengono rimossi dallo stato. Rimuovere il contratto in teoria suona come una buona idea, ma è potenzialmente pericoloso nel caso in cui qualcuno mandasse Ether a contratti rimossi, in quanto la somma verrebbe perduta per sempre. Se si volesse disattivare i propri contratti, bisognerebbe invece disabilitarli modificando uno stato interno che causa il revert di tutte le funzioni; ciò renderebbe impossibile l'uso del contratto, poiché restituirebbe gli Ether ai mittenti immediatamente.

1.2.4 EIP/ERC

Le Ethereum Improvement Proposals (EIPs) [19] sono proposte o standard a tutti gli effetti di nuove funzionalità o processi per la piattaforma Ethereum. Alcuni di questi standard critici relativi al livello applicativo sono elencati nell'Ethereum Requests for Comment (ERC) [20].

Uno degli standards per smart contracts più significativi su Ethereum è noto come ERC-20 [21], usato in tutti i contratti sulla blockchain di Ethereum riguardanti implementazioni di token fungibili, cioè token scambiabili fra loro “senza che nessuno se ne accorga” (un ERC-20 vale quanto un altro ERC-20 come ad esempio i Tether (USDT)⁶). Lo standard ERC-20 definisce un elenco comune di regole a cui dovrebbero aderire tutti i token fungibili su Ethereum. Questo standard permette a tutti gli sviluppatori di prevedere con precisione come funzioneranno i nuovi token all'interno del sistema Ethereum. Ciò semplifica e facilita la vita degli sviluppatori perché possono procedere con il loro lavoro implementando questa interfaccia, sapendo che ogni progetto non dovrà essere rifatto ogni volta che viene rilasciato un nuovo token purché il token adotti lo standard.

⁵Chainlink (<https://chain.link/>)

⁶Tether (<https://tether.to/>)

I token non fungibili (NFT) invece sono descritti nel ERC-721 [22]; questi token sono tutti unici e diversi l'uno dall'altro, ognuno è caratterizzato da varie proprietà e rarità.

1.2.5 Mainnet e Testnets

Oltre alla mainnet di Ethereum, esistono varie testnets (sia pubbliche che locali) in cui gli sviluppatori possono testare le applicazioni per Ethereum e i relativi smart contracts prima del deploy finale sulla mainnet.

Alcune tra le testnets più usate sono:

- **Ropsten:** la test network ufficiale, creata dall'Ethereum Foundation; le sue funzionalità sono simili alla mainnet, serve per testare gli hard fork e le EIP che verranno implementate prima del definitivo rilascio sulla rete principale.
- **Kovan:** testnet che adotta il Proof of Authority, originariamente iniziata dal Parity team.
- **Rinkeby:** testnet che adotta il Proof of Authority, originariamente iniziata dal Geth team.

La testnet Ropsten verrà usata anche nello sviluppo dell'applicazione decentralizzata NotarizETH [23] (vedi Capitolo 4).

Possono essere create anche testnets locali a partire da un *genesis block* su cui effettuare tutti gli sviluppi e test possibili.

1.3 Solidity

Solidity [17] [24] è un linguaggio di alto livello object-oriented tipizzato staticamente per l'implementazione di smart contracts proposto da Gavin Wood nel 2014 e sviluppato da molti core contributors di Ethereum. Solidity è stato influenzato dal C++, da Python e da JavaScript ed è progettato per la Ethereum Virtual Machine (EVM).



Figura 1.8: Logo di Solidity

Solidity supporta l'ereditarietà, le librerie e molto altro. È possibile salvare le variabili nello stack, nella memoria o nello storage (con alcune restrizioni ed elevati costi per il salvataggio). Inoltre, sono presenti numerose funzionalità riguardanti le specifiche dei contratti e le proprietà delle transazioni.

1.3.1 Solidity Assembly

Nonostante vi sia già un ottimizzatore integrato in Solidity che in compilazione riduce il numero d'istruzioni macchina che comporranno il bytecode del contratto, non sempre è efficiente come si vedrà nelle casistiche successivamente analizzate nel Capitolo 3.

Solidity permette di scrivere il codice in linguaggio Assembly, grazie a ciò è possibile interagire direttamente con la EVM a basso livello utilizzando gli opcodes. L'Assembly fornisce un maggiore controllo su alcune logiche che non possono essere possibili solo con Solidity, come ad esempio il puntamento a un blocco di memoria specifico. Questo controllo ai minimi dettagli può risultare molto utile nello scrivere librerie che verranno usati in moltissimi casi dagli sviluppatori, minimizzando il consumo di gas in tutte le future transizioni.

Solidity ha due modi per implementare il linguaggio Assembly:

- **Standalone Assembly:** può essere usato senza Solidity; tuttavia non è più presente nella documentazione di Solidity e viene supportato solo per compatibilità con le versioni precedenti.

- **Inline Assembly:** può essere usato all'interno di un codice scritto in Solidity come mostrato nel Listing 1.2.

```
1 // SPDX-License-Identifier: MIT
2 pragma solidity >=0.8.0 <0.9.0;
3
4 contract InlineAssemblyTest {
5     function addition(uint256 x, uint256 y) public pure returns (uint256) {
6         assembly {
7             // Assembly language statements
8             let result := add(x, y) // x + y
9             mstore(0x0, result) // store result in memory
10            return(0x0, 32) // return 32 bytes from memory
11        }
12    }
13 }
```

Listing 1.2: Inline Assembly

Uno dei principali vantaggi è la riduzione costo del gas speso per implementare il contratto grazie a un controllo più fine sugli opcodes usati nel contratto dall'EVM. Il rovescio della medaglia è che accedere all'EVM a basso livello bypassa diverse importanti funzioni di sicurezza e numerosi controlli di Solidity, aumentando il rischio d'introduzione di vulnerabilità nel contratto (ad esempio l'assenza di check nelle operazioni sugli arrays).

Capitolo 2

Obiettivo

Parte riguardante l'esposizione dell'obiettivo della tesi, dei vari approcci adottati, delle tecnologie e dei tools usati.

2.1 Linea adottata

Il lavoro di tesi è composto da due parti: una prima parte riguardante l'analisi dei costi del gas di Ethereum Smart Contracts usando Solidity e una seconda parte riguardante lo sviluppo della DApp NotarizETH [23].

2.1.1 Analisi dei costi

La parte più cospicua del lavoro è la prima parte riguardante l'analisi dei costi associati all'esecuzione/storage degli smart contracts in ambito Ethereum. Si è scelta la piattaforma Ethereum perché è molto diffusa, destinata a crescere in futuro e ha una community di sviluppatori molto ampia.

Attualmente Ethereum ha due grossi problemi: la scalabilità e il costo del gas. Riguardo alla scalabilità, vi sono limiti per la quantità di transizioni che possono essere convalidate e inserite in un blocco, così come vi è anche un limite temporale per la convalida di nuovi blocchi. Una soluzione non definitiva è il prossimo passaggio a ETH2 [3] basato sul Proof of Stake (PoS); altri progetti mirano ad aumentare la velocità delle transazioni, migrando la maggior parte delle transizioni *off-chain*. L'aspetto legato al costo del gas verrà trattato in questa tesi.

Gli smart contracts svolgono un ruolo molto importante nell'economia di Ethereum. L'efficienza è la priorità numero uno, a pari livello della sicurezza, la quale non si limita a una suite di test da effettuare a lavoro finito ma inizia anche con processi di design e di sviluppo dei contratti adeguati. È bene scrivere contratti semplici, evitando istruzioni o

strutture computazionalmente costose da eseguire, evitando anche di effettuare numerose chiamate ad altri contratti.

L'obiettivo di questa prima parte del lavoro di tesi è quello di analizzare le performance degli smart contracts in Ethereum in varie casistiche e proporre delle *best-practices* che permettano di ottimizzare i contratti riducendo il carico computazionale e quindi risparmiando gas [25]. In particolare, si andranno ad analizzare i costi legati al deployment e all'esecuzione dello smart contract, così come i costi di archiviazione nello storage. Successivamente si cercherà di applicare queste best-practices in un caso reale.

Vi sono numerose piattaforme di sviluppo degli smart contracts, in questa tesi ci si concentrerà sul linguaggio di programmazione Solidity e si andranno ad analizzare numerose casistiche in cui è possibile ottimizzare il dispendio di gas scrivendo codice direttamente in Assembly, riducendo notevolmente il numero d'istruzioni macchina che la EVM dovrà eseguire.

In un tipico linguaggio di programmazione non è così indispensabile conoscere come i dati sono rappresentati a basso livello. In Solidity (o qualsiasi altro linguaggio per l'EVM), questa conoscenza è necessaria perché l'accesso allo storage è molto costoso e in particolare le due operazioni `SSTORE` e `SLOAD` dominano il costo d'esecuzione degli smart contracts.

Grazie alla newsletter settimanale *Week in Ethereum News* [26] che contiene notizie del mondo Ethereum sono venute a conoscenza di numerose EIPs introdotte con il Berlin hard fork [9] che hanno rivoluzionato i costi per le operazioni nello storage come già descritto nella sezione 1.2.2.

2.1.2 Sviluppo della DApp NotarizETH

Nella seconda parte della tesi verrà trattato lo sviluppo della DApp NotarizETH [23].

Dopo tutta la prima parte riguardante l'analisi dei costi nell'EVM con il linguaggio Solidity, volevo concludere la tesi sviluppando un'applicazione decentralizzata da zero, a partire dal deploy del contratto sulla blockchain (sul Ropsten Network [27], una testnet di Ethereum) fino al frontend. In questo modo sono entrato in contatto con molti tools e tecnologie interessanti tra le quali: React [28], Infura [29], Metamask [30], Amazon S3 [31], GitHub Actions [32] e molte altre relative a frameworks come useDApp [33] o standard di smart contracts per la gestione dei permessi forniti da OpenZeppelin [34].

L'applicazione vuole fornire un servizio di certificazione di documenti digitali sulla blockchain salvando, grazie a uno smart contract, alcune informazioni tra cui l'hash del file in modo da garantire l'integrità e la *Proof of Existence* del documento in un dato momento, permettendo a chiunque la verifica tramite un'interfaccia Web.

2.2 Tecnologie e Tools

In questa sezione verranno descritte le principali tecnologie e i vari tools usati sia per la prima parte di tesi sull'analisi dei costi che per la seconda parte riguardante lo sviluppo della DApp NotarizETH.

2.2.1 Remix

Remix [35] è un IDE open source per gli smart contracts che ne facilita la distribuzione, il testing e il debugging.



Figura 2.1: Logo di Remix

Si basa su un'architettura a plugins, è scritto in JavaScript ed è disponibile sia come applicazione web direttamente nel browser che per desktop. Molto semplice e intuitivo per piccole applicazioni ma presenta svantaggi rispetto a Truffle per applicazioni più grandi in cui è necessario effettuare una batteria di test sul contratto. Tuttavia è adatto per analizzare i costi delle transizioni che vengono effettuate grazie anche a numerosi plugins tra cui il *gas profiler*. Inoltre, dato che Solidity non lancia nessun warning di compilazione, ci sono alcuni plugins utili che analizzano staticamente il codice.

Remix è stato usato per l'analisi di piccoli contratti usando la JavaScript VM integrata in locale, oppure attraverso a MetaMask è possibile connettersi alla mainnet o alle testnets di Ethereum tramite l'Injected Web3 come effettuato nello sviluppo di NotarizETH.

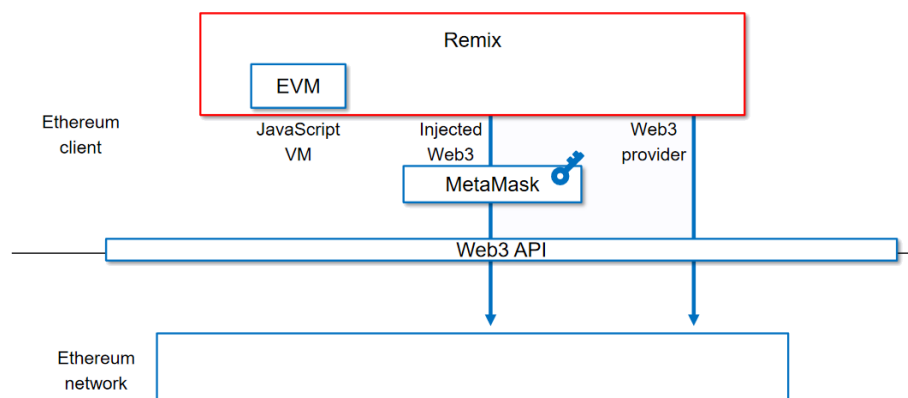


Figura 2.2: Schema di Remix

2.2.2 Truffle

Truffle [36] è un ambiente di sviluppo, un framework di test e una serie di risorse per lo sviluppo di applicazioni su reti blockchain basate sulla EVM.



Figura 2.3: Logo di Truffle

Le principali features di Truffle sono:

- Compilatore integrato per smart contracts che permette di effettuare il linking, il deployment e gestire i binari prodotti.
- Test automatizzati sui contratti basato su Mocha¹ (JavaScript, Solidity).
- Framework per la distribuzione e la migrazione degli script.
- Network management per la distribuzione su reti pubbliche e private.
- Console interattiva per interagire con i contratti distribuiti (debug).

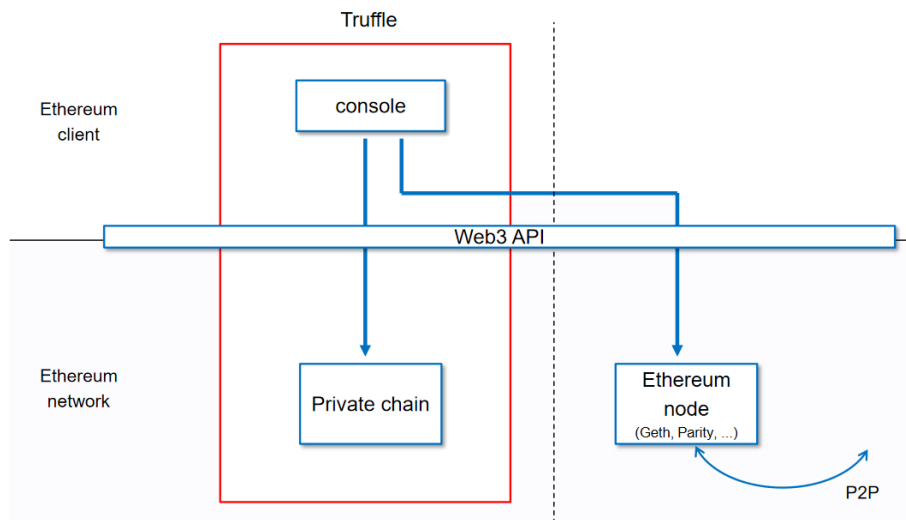


Figura 2.4: Schema di Truffle

Come mostrato nella Figura 2.4 è possibile usare delle blockchain private generate da tool come Ganache [37] da usare durante lo sviluppo e il testing delle proprie applicazioni.

¹Mocha (<https://mochajs.org/>)

2.2.3 MetaMask

MetaMask [30] è un portafoglio di criptovalute usato per interagire con varie blockchain tra cui quella Ethereum.

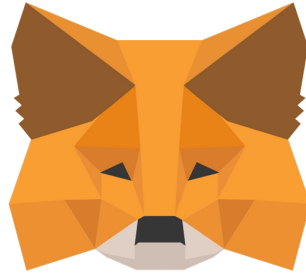


Figura 2.5: Logo di MetaMask

Consente agli utenti di archiviare e gestire chiavi dell'account, trasmettere transazioni, inviare e ricevere criptovalute e token basati su Ethereum e connettersi in modo sicuro ad applicazioni decentralizzate tramite un browser Web compatibile. Si basa sull'infrastruttura Infura anch'essa sviluppata da ConsenSys².

MetaMask è stato usato nell'applicazione NotarizETH per interagire attraverso la funzione Injected Web3 di Remix con il contratto deployato sulla testnet Ropsten Network di Ethereum.

2.2.4 Infura

Infura [29] è una suite di sviluppo per blockchain che fa da gateway con la blockchain per sviluppatori di applicazioni decentralizzate.



Figura 2.6: Logo di Infura

Permette l'interazione con la blockchain di Ethereum comprese le sue testnets grazie a un'infrastruttura formata da moltissimi nodi Ethereum che gestiscono e distribuiscono le richieste degli utenti.

²ConsenSys (<https://consensys.net/>)

Le API fornite da Infura sono state usate nell'applicazione NotarizETH per gestire le richieste in lettura dello smart contract deployato sulla testnet Ropsten Network di Ethereum.

2.2.5 React

React [28] è una libreria JavaScript open source usata nel frontend per la creazione d'interfacce utente o di componenti UI.

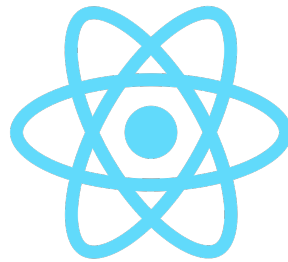


Figura 2.7: Logo di React

Inizialmente sviluppata da Facebook, attualmente è supportata da una grande community di programmatori. React consente di sviluppare applicazioni dinamiche che non necessitano di ricaricare la pagina per visualizzare i dati modificati. Inoltre, nelle applicazioni React le modifiche effettuate sul codice si possono visualizzare in tempo reale, permettendo uno sviluppo rapido, efficiente e flessibile delle applicazioni web.

React è stato usato per lo sviluppo frontend dell'applicazione NotarizETH.

2.2.6 Tools di sviluppo

Sono presentati dei tools di sviluppo dell'applicazione e dei plugins usati durante l'analisi dei contratti.

Framework useDApp

Il framework useDApp [33] è stato usato come base per un rapido sviluppo di un'applicazione decentralizzata con React.

Questo framework include librerie come *ethers.js* [38] e *web3-react* [39] oltre ai numerosi React hooks che rendono più semplice l'interazione con la blockchain.

Dracula UI

Dracula UI [40] è una raccolta di pattern e componenti a tema scuro usata per modellare l'interfaccia utente dell'applicazione.

Amazon S3

Amazon Simple Storage Service (Amazon S3) [31] è uno storage di oggetti cloud ed è stato usato come hosting di sito dinamico.

Python

Il linguaggio di programmazione Python [41] è stato usato in alcuni script di supporto all'analisi dei contratti.

Solc-JS

Solc-JS [42] è un tool che fornisce un collegamento JavaScript per il compilatore di Solidity eseguibile da linea di comando.

Online Solidity Decompiler

Online Solidity Decompiler [43] è un tool online che permette di decompilare un contratto scritto in Solidity mostrandone il suo bytecode. Il tool è stato utile per analizzare alcuni opcodes presenti nei vari contratti successivamente trattati.

Package eth-gas-reporter

L'eth-gas-reporter [44] è un Mocha reporter per Truffle. Permette di:

- Analizzare l'uso del gas per ogni unità di test.
- Visualizzare alcune metriche per le chiamate ai metodi e i deployments.
- Convertire il costo del gas per l'implementazione del contratto in una valuta corrente.

L'eth-gas-reporter non è perfetto nel rilevare costi del gas in caso di transizioni a costo elevato oppure quando una transizione si interrompe, il che potrebbe essere interessante analizzare il caso in cui venga lanciata un'eccezione per esaurimento del gas. Inoltre, è limitato ai dati raccolti dai casi di test.

Plugin Solidity

Il plugin Solidity per Visual Studio Code [45] aggiunge l'highlighting della sintassi di Solidity, gli snippets, la compilazione e il linting all'interno dell'ambiente di sviluppo.

Il plugin prettier-plugin-solidity [46] è usato per formattare i files di Solidity.

2.2.7 Tools di supporto

Sono presentati dei tools di supporto al lavoro di tesi.

GitHub

Il servizio GitHub [32] è stato usato come hosting per gli smart contracts dell'analisi dei costi e tutto quanto riguardante l'applicazione NotarizETH (Smart Contract, React App).

Grazie alle GitHub Actions [47] è stato possibile realizzare un workflow che permette il deploy dell'applicazione dal repository GitHub direttamente su Amazon S3.

Disponibile il repository *Master's Thesis* [48] contenente documentazione, codice degli smart contracts usati nell'analisi dei costi e lo smart contract di NotarizETH in aggiunta a tutto al codice dell'applicazione sviluppata in React.

Overleaf

L'editor LaTeX online Overleaf [49] è stato usato per la scrittura della tesi.

Slack

Lo strumento di collaborazione Slack [50] è stato usato per la comunicazione con i ricercatori del UniBG Seclab³ che mi hanno seguito nel lavoro di tesi dandomi ampie libertà d'indagine.

Trello

La piattaforma Trello [51] è stata usata per l'organizzazione del lavoro e della scrittura della tesi.

Visual Studio Code

L'ambiente di sviluppo Visual Studio Code [52] è stato usato sia per la prima parte riguardante l'analisi dei costi che per lo sviluppo dell'applicazione in React.

2.2.8 Siti di supporto

Menzione speciale per i due siti Ethereum Stack Exchange [53] e Stack Overflow [54] e la loro comunità di utenti che mi ha aiutato a risolvere alcuni problemi riscontrati sia per quanto riguarda la parte dell'analisi dei costi che quella riguardante lo sviluppo dell'applicazione decentralizzata.

³UniBG Seclab (<https://seclab.unibg.it/>)

Capitolo 3

Analisi dei costi

In questo capitolo si analizzeranno tutte le problematiche relative al costo del gas per il deployment, l'esecuzione e lo storage degli smart contracts e si proporranno soluzioni per l'ottimizzazione delle performance.

Dopo una breve presentazione della metodologia usata e di alcune definizioni, saranno analizzati vari aspetti degli smart contracts corredati da alcuni chiarimenti teorici necessari per conoscere la sintassi di Solidity e il funzionamento dell'EVM in particolari situazioni.

Tutto il materiale, compresa la documentazione e gli smart contracts analizzati, sono disponibili nel repository di GitHub [48].

3.1 Metodologia

In tutti i test verrà usata la versione di Solidity:

```
solc 0.8.5+commit.a4f2e591
```

con l'ottimizzatore abilitato con il parametro `runs` pari a 200¹; verrà specificato l'uso di una versione diversa se necessario. In alcuni casi l'ottimizzatore è disabilitato appositamente per analizzare alcuni salvataggi nella memoria o nello storage. In questo modo, adottando per tutti i test la stessa versione e gli stessi parametri, si potranno fare confronti più precisi.

¹Il parametro `runs` corrisponde a una stima sul numero di volte che le funzioni interne allo smart contract verranno eseguite. Se si usa lo smart contract per il vesting o il locking di tokens, si può settare il valore a 1 in modo tale che il compilatore produca un bytecode più piccolo possibile (ma poco efficiente). Al contrario, se si distribuisce un contratto che verrà usato spesso, bisogna settare il parametro a un numero alto in modo tale che le chiamate alle funzioni siano molto più efficienti.

Nei listati di codice che si troveranno nel corso delle analisi saranno specificate la licenza e la versione di Solidity come mostrato nel Listing 3.1: si specifica la licenza del codice MIT² e ne si limita l'esecuzione a versioni di Solidity minori di 0.9.0, in modo di evitare che future modifiche sostanziali introdotte in Solidity provochino comportamenti anomali nel codice.

```
1 // SPDX-License-Identifier: MIT
2 pragma solidity >=0.8.0 <0.9.0;
3 pragma experimental ABIEncoderV2;
```

Listing 3.1: Licenza e versione di Solidity

Per le analisi su piccoli smart contracts si userà il tool Remix con il gas profiler attivato; i contratti verranno deployati sulla JavaScript VM integrata o sulla testnet Ropsten con l'Injected Web3 usando il tool MetaMask. Per i contratti più grandi da analizzare approfonditamente si userà Truffle.

Il debugger di Remix se usato in locale con la JavaScript VM può avere i costi dei singoli opcodes non aggiornati alle ultime modifiche introdotte con le EIPs e gli hard fork, motivo per cui in alcuni casi si è preferito usare con la testnet Ropsten Network [27] attraverso l'Injected Web3.

L'Application Binary Interface (ABI) [55] è l'interfaccia standard per interagire con i contratti dell'ecosistema Ethereum, sia dall'esterno della blockchain che per l'interazione da contratto a contratto. L'ABI Coder di default, a partire dalla versione 0.8.0 di Solidity, è l'ABI Coder v2 e visto che non più sperimentale, non è nemmeno più necessario specificarlo a inizio contratto.

In Solidity gli opcodes `SHA3` e `KECCAK256` sono alias e il loro output è identico. Nel bytecode si potrà trovare indipendentemente l'opcode `SHA3` oppure l'opcode `KECCAK256` definiti come riportato nella Figura 3.1. Questi opcodes non sono da confondere con la funzione di Solidity `sha3` che era un alias della funzione `keccak256` ed è stata rimossa dalla versione di Solidity 0.5.0 mantenendo soltanto `keccak256`.

20s: SHA3				
Value	Mnemonic	δ	α	Description
0x20	SHA3	2	1	Compute Keccak-256 hash. $\mu'_s[0] \equiv \text{KEC}(\mu_m[\mu_s[0] \dots (\mu_s[0] + \mu_s[1] - 1)])$ $\mu'_i \equiv M(\mu_i, \mu_s[0], \mu_s[1])$

Figura 3.1: Opcode `SHA3` alias di `KECCAK256`

²MIT License (<https://mit-license.org/>)

Molto spesso nell'analisi si potranno trovare le seguenti due voci di costo:

- *Transaction Cost*: costo per la quantità di dati (deployment) che deve essere inviata sulla blockchain. Si può scomporre in quattro componenti (costi riferiti alla Figura 1.5 vista in precedenza):
 - Il costo base della transazione (21 000 gas).
 - Il costo per il deployment del contratto usando l'opcode `CREATE` (32 000 gas).
 - Il costo per ogni *zero byte* di dati o codice per una transazione.
 - Il costo per ogni *non-zero byte* di dati o codice per una transazione.
- *Execution Cost*: costo computazionale per eseguire il codice, quest'ultimo molto interessante per valutare due diverse esecuzioni.

Il costo del gas, se non fosse specificata l'unità di misura, è espresso in *wei*.

L'EVM usa due formati per l'endianness in base al tipo di variabile:

- *Big Endian Format*: per i tipi speciali `string` e `bytes`.
- *Little Endian Format*: per tutti gli altri tipi.

Come esempio, nella Figura 3.2 è mostrato in che modo una variabile con valore esadecimale `0x01234567` viene salvata nei due formati.

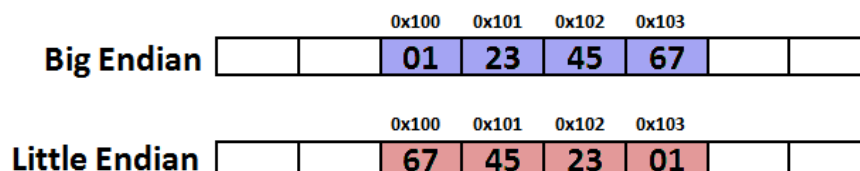


Figura 3.2: Endianness

Infine, si ricordi che non è possibile ottimizzare in toto lo smart contract ma soltanto una parte specifica perché solitamente quando si cerca di ottimizzare una funzione specifica, un'altra peggiora; bisogna porsi degli obiettivi e gestire questo trade-off: ad esempio si potrebbero ottimizzare soltanto le funzioni principali che verranno teoricamente più usate.

Molto utile come base di partenza per le analisi, oltre la documentazione di Solidity [24], è stata la serie di articoli di Jean Cvllr riguardanti molti aspetti di Solidity [56].

3.2 Analisi relative al costo per il deployment del contratto

In questa sezione verranno effettuate delle analisi sul costo relativo al deployment dello smart contract sulla blockchain. In particolare faremo riferimento al *transaction cost*, costo una tantum proporzionale alla dimensione del bytecode.

Come già visto in parte a inizio di questo capitolo con il transaction cost, il costo per il deployment di uno smart contract può essere scomposto in:

- Il costo base della transazione (21 000 gas).
- Il costo per il deployment del contratto usando l'opcode `CREATE` (32 000 gas).
- Il costo per il bytecode del contratto compilato: per ogni byte vi è un costo di 200 gas. Bisogna notare che anche i contatti ereditati sono da includere nel bytecode.
- Il costo per il transaction data: un costo di 16 per ogni *non-zero byte* e 4 per ogni *zero byte*.
- Il costo per il codice che viene eseguito prima della creazione del contratto, ad esempio del costruttore del contratto.
- Il gas price indicato per la transazione.

3.2.1 Contratti

Nel 2016 con l'hard fork Spurious Dragon è stato introdotto l'EIP-170 [57] che ha messo un limite superiore alla dimensione di uno smart contract, precisamente a 24 576 byte. Questo limite è stato introdotto per prevenire attacchi DoS (Denial of Service), visto che il solo limite naturale dovuto al block gas limit non era immutabile e quindi bisognava introdurre un limite alla dimensione dei contratti (nel 2016 il gas limit era intorno ai 4.7 milioni e nel 2021 è cresciuto fino a 12.5 milioni)³.

Ora verranno proposti dei metodi per ridurre la dimensione dei contratti, evitando di superare questo limite per smart contracts di grande dimensione e risparmiando anche gas per il deployment sulla blockchain [58].

Dividere un contratto

Dividere un contratto di dimensioni elevate in più contratti piccoli è una buona prassi di design dell'architettura dei contratti, magari raggruppando le funzioni per contesto oppure raggruppando quelle che richiedono o meno la lettura dello stato del contratto.

³Ethereum Average Gas Limit Chart (<https://etherscan.io/chart/gaslimit>)

Proxy

I contratti proxy sono contratti che riescono ad avere una dimensione ridotta prendendo in prestito funzionalità da altri contratti usando l'opcode `DELEGATECALL`. Questa operazione si comporta come una normale chiamata con la differenza che viene eseguito il codice del contratto del destinatario specificato ma nell'ambiente del chiamante, preservando quindi lo storage, gli indirizzi con i relativi saldi e i parametri della chiamata originale (`msg.sender` e `msg.value`).

I contratti proxy sono usati anche nel rendere aggiornabili gli smart contracts distribuiti su una blockchain separando i dati dalla logica, quest'ultima modificabile rilasciando un'altra versione del contratto come descritto nella documentazione di Smart Contract Upgradability [59].

Clonare i contratti

Usando l'EIP-1167 [60] è possibile clonare i contratti, ad esempio nel caso in cui si abbia la necessità di distribuire lo stesso contratto più volte. Usando un sistema di proxy si effettua il deploy del codice del contratto solo la prima volta e i futuri contratti punteranno al codice iniziale: gli smart contracts condivideranno lo stesso codice ma avranno dati e storage propri e distinti tra loro.

Bytecode

Qualsiasi cosa non necessaria per l'esecuzione dello smart contract è rimossa durante la compilazione e l'ottimizzazione. Questo include, tra le altre cose, i commenti, i nomi delle variabili usati e i nomi dei tipi. Rimane solo il bytecode (ottimizzato) composto da una serie di opcodes in successione.

Dead code e predicato opaco

Il dead code è quella porzione di codice che non viene mai eseguito o le cui condizioni di esecuzione non verranno mai soddisfatte. Nell'esempio presente nel Listing 3.2, nella funzione `deadCode` vi sono delle condizioni auto-contraddittorie che non verranno mai soddisfatte, la cui esecuzione non ha nessun effetto se non quello di sprecare gas [61].

Allo stesso modo, in alcuni casi le condizioni possono essere semplificate in un'unica riga, come mostrato nella funzione `opaquePredicate` dove mostra un predicato opaco [61].

```
1 // SPDX-License-Identifier: MIT
2 pragma solidity >=0.8.0 <0.9.0;
3
4 contract DeadCodeOpaquePredicate {
5     function deadCode(uint256 x) public pure returns (uint256 out) {
```

```

6   if (x < 1) {
7       if (x > 2) {
8           return x;
9       }
10  }
11  }
12
13  function opaquePredicate(uint256 x) public pure returns (uint256 out) {
14      if (x < 1) {
15          if (x < 0) {
16              return x;
17          }
18      }
19  }
20 }

```

Listing 3.2: Dead code e predicato opaco

3.2.2 Librerie

Le librerie si possono usare per evitare di riscrivere funzioni comuni e condivise tra più contratti, risparmiando sui costi di deployment dato che il loro codice viene distribuito solamente una sola volta sulla blockchain e viene riutilizzato usando la `DELEGATECALL`. Quando viene effettuata la chiamata a una libreria, il suo codice viene eseguito nel contesto del contratto chiamante. Inoltre, le librerie sono stateless, non possono avere variabili di stato, non possono ricevere Ether, non possono essere distrutte e non possono né ereditare e né essere ereditate.

Esistono già numerose librerie distribuite nella blockchain a cui è possibile riferirsi nei propri contratti, oppure se si hanno diversi smart contracts che condividono alcune funzionalità è meglio estrarre le funzionalità comuni e inserirle all'interno di una libreria. Nel caso si voglia creare una libreria, non bisogna dichiarare le funzioni con il modificatore `internal` perché queste funzioni verrebbero aggiunte direttamente al contratto chiamante durante la compilazione; bisogna usare il modificatore `public`, in questo modo le funzioni rimarranno nel contratto della libreria.

Tuttavia bisogna evitare l'uso eccessivo di librerie: nel caso in cui si usino solo poche funzioni appartenenti a librerie molto grandi, la restante parte del codice è inutile per il proprio smart contract. Implementando in modo sicuro ed efficiente la funzionalità richiesta nel proprio contratto senza dover chiamare la libreria porterebbe un'ottimizzazione del gas usato. Un esempio è il seguente contratto in cui viene implementata una funzione che fa la somma in modo sicuro senza dover usare la libreria *SafeMath*⁴ (risparmiando

⁴SafeMath (<https://docs.openzeppelin.com/contracts/2.x/api/math#SafeMath>)

tutto il codice non usato relativo alle altre funzioni della libreria).

```
1 // SPDX-License-Identifier: MIT
2 pragma solidity >=0.8.0 <0.9.0;
3
4 import "./SafeMath.sol";
5
6 contract SafeAddition {
7     function safeAdd(uint256 a, uint256 b) public pure returns (uint256) {
8         return SafeMath.add(a, b);
9     }
10
11     function safeAddOwn(uint256 a, uint256 b) public pure returns (uint256) {
12         uint256 c = a + b;
13         require(c >= a, "Addition overflow");
14         return c;
15     }
16 }
```

Listing 3.3: Uso della libreria *SafeMath*

La funzione `safeAdd` che usa la libreria consuma 62 gas in più rispetto alla funzione `safeAddOwn` che implementa nel proprio codice l'addizione.

3.2.3 Dati on-chain e off-chain

Quando si progetta un contratto bisogna decidere quale parte dei dati va messa *on-chain* e quale *off-chain*.

Minimizzare i dati on-chain

Non tutta la logica del contratto va salvata nello storage sulla blockchain, soltanto i dati più critici (ad esempio dati relativi all'economia del contratto o dati che devono essere immutabili) mentre qualsiasi altro dato non critico (ad esempio i metadata dei files) può essere tenuto su un server centralizzato.

Un approccio più estremo per mitigare la crescita di dimensione dello stato è archiviare solo i 32 bytes del Merkle Root sulla blockchain. Il mittente di una transazione è responsabile di fornire valori e garanzie (*proofs*) appropriate per tutti i dati che la transazione deve utilizzare durante la sua esecuzione. I contratti possono verificare che la prova sia corretta, ma non è necessario memorizzare nessun'altra informazione in modo persistente sulla blockchain: sono necessari solo i 32 bytes del Merkle Root che vengono conservati e aggiornati.

3.3 Analisi relative al costo per l'esecuzione del contratto

In questa sezione verranno effettuate delle analisi sul costo relativo all'esecuzione delle funzioni di uno smart contract distribuito sulla blockchain. In particolare faremo riferimento all'*execution cost*, costo variabile d'esecuzione delle funzioni che dipende da numerosi fattori tra i quali lo storage usato.

Inoltre, bisogna ricordare che una volta effettuato il deploy del contratto tutte le sue istanze lavoreranno sulle stesse strutture statiche o dinamiche presenti nello storage.

3.3.1 Calcoli on-chain e off-chain

Quando si progetta un contratto bisogna decidere quale parte dei calcoli va fatta *on-chain* e quale *off-chain*.

Minimizzare i calcoli on-chain

Bisogna evitare di fare svolgere allo smart contract qualsiasi calcolo inutile che consuma gas quando si può benissimo calcolare al di fuori della blockchain e riportare soltanto una *proof* (ad esempio l'hash crittografico) sulla blockchain. Nel caso in cui si conosca già il valore di un dato al tempo di compilazione, scrivere direttamente il literal invece di ricalcolarlo ogni volta evita di spendere calcolo computazionale on-chain. Un esempio è quello mostrato nel Listing 3.4 in cui è più efficiente scrivere direttamente il valore hash già conosciuto nel codice (con una stringa) piuttosto che calcolarlo on-chain con una funzione hash spendendo gas inutile.

```
1 // Good: Salvare direttamente il literal conosciuto
2 bytes32 constant hash = 'b5b41c34b0ba6a0...';
3
4 // Bad: Salvare il valore calcolato spreca gas sulla blockchain
5 bytes32 constant hash = keccak256(abi.encodePacked('MyKey'));
```

Listing 3.4: Salvare direttamente il literal invece che il valore calcolato se già conosciuto

Le implementazioni naive di strutture dati comuni come la lista ordinata richiedono un'iterazione attraverso l'intera raccolta per trovare la posizione corretta quando si aggiunge un elemento affinché si assicuri che la lista sia ancora ordinata. Un approccio più efficiente (vedi figura Figura 3.3) sarebbe che il contratto richieda una computazione off-chain per fornirgli la posizione esatta dell'elemento da aggiungere [62]. La computazione on-chain deve solo verificare che tutte le invarianti siano preservate (cioè che il valore aggiunto si collochi tra i suoi elementi adiacenti). Tutto questo impedisce al costo del gas di crescere linearmente con la dimensione totale della struttura dei dati.

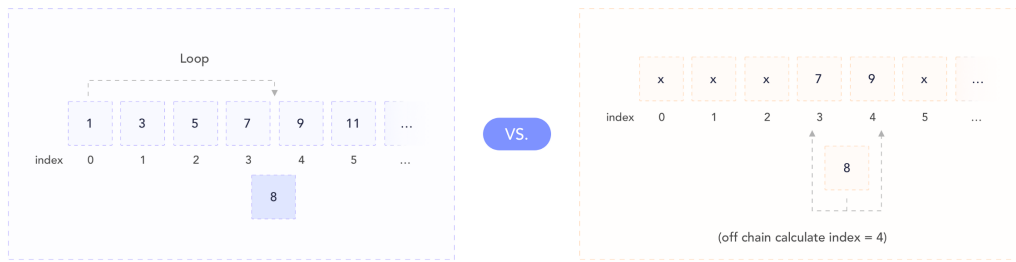


Figura 3.3: A sinistra: scorrere un elenco in catena costa $O(n)$ gas, il costo cresce linearmente man mano che la lista cresce. A destra: calcolare la posizione off-chain e verificare il valore nella catena costa una quantità costante di gas indipendentemente dalle dimensioni della lista.

Pattern per la distribuzione

Un altro esempio è fornito dal pattern per la distribuzione di token [62], ad esempio, come mostrato nella Figura 3.4. Invece di eseguire il looping ed distribuire a ogni singolo indirizzo i tokens in un'unica transazione e magari andando a sfiorare il gas limit, uno smart contract può mantenere un mapping indicante se un determinato utente ha eseguito o meno il claim dei tokens. Ogni utente è responsabile dell'invio di una transazione per avviare il claim, mentre lo smart contract verifica solo che non vengano eseguite azioni duplicate dallo stesso utente; con questo schema, il costo di ogni transazione rimane costante. Ciò elimina la possibilità di andare oltre il gas limit con un'unica transazione; tuttavia, è importante notare che il numero totale complessivo del costo del gas nel caso in cui tutti gli utenti richiedessero il claim sarebbe maggiore che fare tutto in una transazione a causa dei costi delle transazioni multiple.

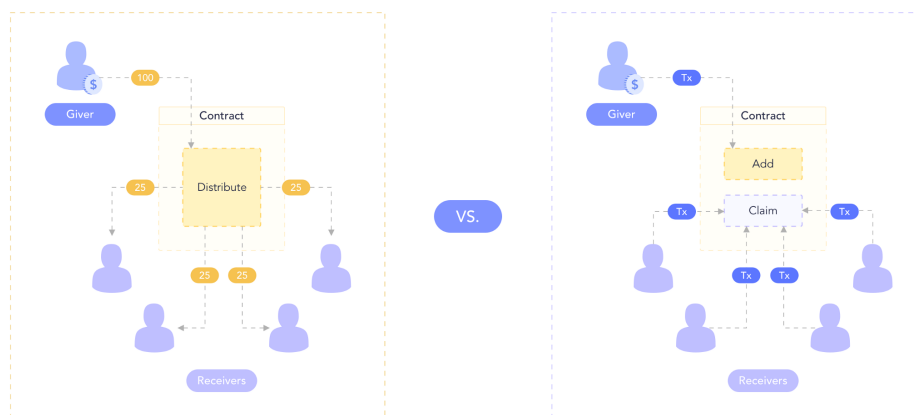


Figura 3.4: A sinistra: invocare un'azione **Distribute** ha un costo in gas proporzionale al numero di ricevitori in una transazione; inoltre, la transazione può fallire al superamento del gas limit. A destra: tutte le transazioni (1 **Add** e 4 **Claim**) hanno costi costanti.

3.4 Chiarimenti riguardo ai tipi

Alcuni chiarimenti riguardo ai tipi dalla documentazione di Solidity [24].

Si farà riferimento agli ABI Types definiti nella specifica dell'interfaccia ABI (Application Binary Interface) [55].

3.4.1 Value Types

I value types sono tipi di variabili che vengono sempre passate per valore, ossia viene effettuata sempre una copia del valore quando vengono usate come argomenti di funzione o negli assegnamenti. Tra i value types rientrano:

- Booleans (`bool`).
- Integers (`int`, `uint`) e gli interi di varie dimensioni (da `int8` a `int256` con step di 8 bits e da `uint` a `uint256` con step di 8 bit).
- Fixed Point Numbers (`fixed`, `ufixed`).
- Enums (`enum`) che non sono altro che `uint8`.
- Addresses (`address`) dove rientrano:
 - Address Payable (`address payable`).
 - Contratti (`contract`) che è un address (20 bytes).
 - Funzioni (`function`) che è un address (20 bytes) seguito da un selettore di funzione (4 bytes).
- Literals degli address, dei numeri interi e razionali e delle stringhe.
- Byte Arrays a dimensione fissa (da `bytes1` a `bytes32`) e gli arrays di bytes (`byte[]`)⁵.

3.4.2 Reference Types

I reference types sono tipi di variabili che vengono passate per riferimento, il cui valore può essere modificato da più variabili avente nome diverso ma lo stesso riferimento. Tra i reference types rientrano:

- Arrays (vedi sottosezione 3.4.3).
- Byte arrays dinamici (`bytes`, `string`) (vedi sottosezione 3.4.4).
- Structs (vedi sottosezione 3.4.5).
- Mappings (vedi sottosezione 3.4.6).

⁵Arrays di bytes (`byte[]`) rimossi dalla versione 0.8.0 di Solidity, erano un alias di `byte1`.

Bisogna sempre specificare esplicitamente l'area di dati in cui è memorizzato il tipo: `storage`, `memory` o `calldata`.

Un assegnamento o una conversione di tipo che cambia l'area di dati forza la copia automatica del valore, mentre gli assegnamenti all'interno della stessa area dati avvengono passando solo il riferimento (tranne alcuni casi per lo storage).

3.4.3 Arrays

Gli arrays sono un gruppo di elementi dello stesso tipo di dato in cui ogni elemento ha una particolare locazione chiamata indice. In Solidity sono *zero based* e possono contenere qualsiasi tipo, compresi i mappings e le structs.

Esistono due suddivisioni che si possono fare sugli arrays:

- Arrays a dimensione fissa e dinamica.
- Arrays monodimensionali e multidimensionali.

Arrays a dimensione fissa e dinamica

Gli arrays a dimensione fissa `T[k]` necessitano di un parametro tra le parentesi quadre che ne indichi la dimensione a differenza degli arrays a dimensione dinamica `T[]`.

Ecco i campi e le funzioni che si possono usare sugli arrays in Solidity:

- `.length`: può essere usata per ritornare il numero di elementi contenuti nell'array oppure per ridimensionare un array di dimensione dinamiche (solo se salvato nello storage). Quindi solo gli arrays dinamici che fanno riferimento allo storage possono essere ridimensionati, gli arrays in memoria devono avere obbligatoriamente una dimensione fissa⁶.
- `.push(new_element)`: può essere usata per aggiungere un nuovo elemento alla fine dell'array ritornando la lunghezza del nuovo array. Questa funzione è disponibile solo per gli arrays dinamici e nei `bytes`, non è possibile aggiungere un elemento a `string`. Per gli arrays a dimensione fissa non si può usare il metodo `.push(new_element)` ma bisogna specificare l'indice tra parentesi quadre.
- `.pop()`: può essere usata per rimuovere un nuovo elemento dalla testa di un array dinamico.

Gli arrays a dimensione fissa devono essere inizializzati in compilazione e non a runtime come si evince dal Listing 3.5 in cui viene lanciato un errore di compilazione nella prima

⁶A partire dalla versione 0.6.0 di Solidity, l'accesso al membro `length` è read-only anche per gli arrays dinamici nello storage.

funzione del Listing 3.5 usando il parametro `x`. Gli arrays a dimensione dinamica sono inizializzati automaticamente a lunghezza zero, in memoria è necessario l'operatore `new` per specificarne la dimensione.

```

1 // SPDX-License-Identifier: MIT
2 pragma solidity >=0.8.0 <0.9.0;
3
4 contract ArrayFissaDinamica {
5     // Array a dimensione fissa T[k]
6     function bar1(uint256 x) public view {
7         uint256[7] memory foo1;
8         uint256[7] storage foo2;
9         //uint[x] memory foo3; // Compile-time Error
10        //uint[x] storage foo4; // Compile-time Error
11    }
12
13    // Array a dimensione dinamica T[]
14    function bar2(uint256 x) public view {
15        uint256[] memory foo1;
16        uint256[] storage foo2;
17    }
18 }

```

Listing 3.5: Arrays a dimensione fissa e dinamica

Non è possibile assegnare in memoria un array a dimensione fissa a uno con dimensione dinamica e inoltre Solidity effettua il bound checking sugli indici degli arrays per verificarne il rispetto dei limiti.

Arrays monodimensionali e multidimensionali

Gli arrays monodimensionali sono quelli classici:

<code>T[k]</code>	One dimensional, Fixed-size
<code>T[]</code>	One dimensional, Dynamic-size

Gli arrays multidimensionali sono essenzialmente arrays annidati:

<code>T[k][k]</code>	Two-dimensional, Fixed-size
<code>T[][]</code>	Two-dimensional, Dynamic-size
<code>T[][k] o T[k][]</code>	Two-dimensional, Mixed-size
<code>T[2][2][2]</code>	Three-dimensional, Fixed-size (all k are the same)
<code>T[2][8][4][12]</code>	Four-dimensional, Fixed-sizes (k's are of different values)

Tuttavia, non si possono avere tipi diversi all'interno degli arrays annidati e inoltre si potrebbe ottenere l'errore *"Stack Too Deep"*.

La notazione degli arrays multidimensionali in Solidity è invertita rispetto agli altri linguaggi di programmazione. Ad esempio, `string[2][] names` è un array dinamico di dimensione non definita i cui elementi sono arrays di dimensione fissa 2; invece `string[][6] names` è un array di dimensione fissa i cui 6 elementi sono arrays dinamici.

3.4.4 Arrays speciali `bytes` e `string`

Si è visto che i value types `bytes1`, `bytes2`, ... `bytes32` contengono una sequenza di byte di dimensione fissa da 1 a 32 e possono essere usati negli argomenti di funzione per passare dei dati oppure anche come return nei contratti.

Un array di bytes è `byte[]`⁷ in cui ogni elemento occupa 32 byte.

Vengono ora descritti i due tipi speciali `bytes` e `string` che usano la notazione *Big Endian* a differenza di tutti gli altri tipi visti.

Il tipo speciale `bytes` è usato per dati di singoli byte; questo tipo speciale è trattato da Solidity come se fosse un array (può avere lunghezza zero). Rispetto all'ormai deprecato `byte[]` è più compatto visto che non spreca 31 bytes di padding per ogni singolo elemento `byte1` (tranne per lo storage).

Inoltre, può essere usato negli argomenti di funzione ma solo per un uso interno allo stesso contratto, perché l'interfaccia ABI Coder v1 non supporta il passaggio di arrays multidimensionali dinamici come input per chiamate esterne. Invece ABI Coder v2⁸ permette l'uso di arrays multidimensionali anche nelle chiamate a funzioni con modificatore `external` come mostrato nel Listing 3.6.

```
1 // SPDX-License-Identifier: MIT
2 pragma solidity >=0.8.0 <0.9.0; // A partire dalla versione 0.8.0 di
   Solidity, ABI Coder v2 è di default e si può omettere
3
4 // pragma experimental ABIEncoderV2;
5
6 contract ABICoderV2 {
7     bytes[] array;
8
9     function push(bytes calldata value) external {
10         array.push(value);
11     }
12 }
```

⁷Arrays di bytes (`byte[]`) rimossi dalla versione 0.8.0 di Solidity, erano un alias di `byte1`.

⁸A partire dalla versione 0.8.0 di Solidity, ABI Coder v2 è di default.

```

13 function get() external view returns (bytes[] memory) {
14     return array;
15 }
16 }

```

Listing 3.6: ABI Coder v2 con array multidimensionale di singoli byte

Il tipo speciale `string` è usato per stringhe di lunghezza arbitraria UTF-8 ed è denotato con virgolette doppie o singole (`"foo"` o `'bar'`). Fondamentalmente è identico al tipo speciale `bytes`, la differenza è che considera i bytes sottoforma di UTF-8 Encoding di una vera stringa e ciò è più dispendioso visto che l'encoding di qualche carattere può occupare più di un singolo byte.

Le `string` non possono essere passate tra contratti (chiamate esterne) perché non sono a dimensione fissa, men che meno gli arrays di stringhe `string[]` essendo un array bi-dimensionale dato che già `string` è un array. Invece, come riportato sopra, ABI Coder v2 permette l'uso di arrays di arrays anche nelle chiamate a funzioni con modificatore `external`.

Nel momento in cui le stringhe sono lunghe meno di 32 bytes (ad esempio gli username) conviene convertire le `string` nella dimensione esatta in bytes (da `byte1` a `byte32`) visto che è molto più economico rispetto a questi tipi speciali. Come si vede nel Listing 3.7, il costo per chiamare la funzione `useString()` è di 479 gas, mentre per chiamare la funzione `useByte()` che fa la conversione a `byte32` è di soli 150 gas, un notevole risparmio.

```

1 // SPDX-License-Identifier: MIT
2 pragma solidity >=0.8.0 <0.9.0;
3
4 contract Strings {
5     function useString() public pure returns (string memory a) {
6         return "Hello World!";
7     }
8
9     function useByte() public pure returns (bytes32 a) {
10         return bytes32("Hello World!");
11     }
12 }

```

Listing 3.7: Conversione di `strings` in `bytes32`

Per la manipolazione delle stringhe Solidity non offre nulla ma ci sono alcune librerie terze che lo permettono come le Solidity Standard Utilities⁹.

⁹Solidity Standard Utilities (<https://github.com/willitscale/solidity-util>)

3.4.5 Structs

La struct è una struttura dati che permette di raggruppare diverse variabili di diversi tipi (sia value types che reference types) in un unico posto.

Possono essere usate all'interno dei mappings, degli arrays e possono contenere loro stesse mappings o arrays. Non è possibile per una struct contenere un membro dello stesso tipo, sebbene sia possibile che la struct contenga un membro mapping il cui `ValueType` sia la struct stessa; allo stesso modo è possibile che la struct contenga un array dinamico del suo stesso tipo. Queste restrizioni sono necessarie perché la dimensione della struct deve essere finita.

In riferimento agli ABI Types, la struct è vista come una `tuple`.

3.4.6 Mappings

Il mapping è una struttura dati che permette di trovare molto efficientemente una area di dati corrispondente a una chiave data come input. Possono essere visti come delle hash tables, in cui tutti i possibili valori sono virtualmente inizializzati a zero. Tuttavia non viene mai salvata la chiave ma solo il Keccak256 hash usato per cercare il valore.

La struttura del mapping è la seguente:

```
mapping(KeyType => ValueType) VariableName
```

dove il `KeyType` è un value type, un `bytes`, una `string` oppure un qualsiasi contratto o tipo enumerativo; altri tipi complessi definiti dallo sviluppatore tra cui i mappings, le structs o gli arrays non sono possibili. Invece, il `ValueType` può essere di qualunque tipo, pure un mapping, una struct o un array.

I mappings possono risiedere solo nello storage, non nella memoria, e quindi è consentito il loro uso per le variabili di stato, per i tipi di riferimento allo storage nelle funzioni o come parametri per le funzioni delle librerie. Non possono essere però usati come parametri di funzione o parametri di return delle funzioni pubbliche dei contratti. Queste restrizioni sono anche valide per gli arrays e le structs che contengono mappings.

3.5 Chiarimenti riguardo all'archiviazione

Alcuni chiarimenti riguardo all'archiviazione dalla documentazione di Solidity [24].

3.5.1 Layout dello storage

Una dichiarazione di variabile nello storage non costa nulla, visto che non è presente l'inizializzazione della variabile. Solidity riserva spazio per una variabile ma si paga solamente quando si salva effettivamente un valore.

Le variabili di dimensioni statiche (qualsiasi variabile a eccezione dei mappings e degli arrays dinamici) sono disposte in modo contiguo nello storage a partire dalla posizione `0x0`. Gli elementi delle structs e degli arrays sono salvati in posizioni contigue così come sono presenti nella struttura dati. Elementi multipli e contigui che richiedono meno di 32 bytes vengono impacchettati in un unico slot di storage, se possibile, in base alle seguenti regole:

- Il primo elemento in uno storage slot è salvato allineandolo nella parte bassa, ossia nei bytes di ordine inferiore.
- I tipi elementari usano tanti bytes quanti ne necessitano.
- Se un tipo elementare non riesce a inserirsi nella parte rimanente di uno slot di storage, tutto il tipo è salvato nello slot successivo.
- Le structs e gli arrays iniziano sempre in un nuovo slot e occupano l'intero slot (tuttavia gli elementi interni alle structs e agli arrays sono compattati seguendo queste regole).

A causa della loro dimensione imprevedibile, i mappings e gli arrays di dimensioni dinamiche non possono essere memorizzati in modo continuo ma vengono memorizzati a partire da uno slot di archiviazione diverso che viene calcolato utilizzando un Keccak256 hash.

Si supponga che la posizione di archiviazione del mapping o dell'array finisca per essere uno slot `p` dopo aver applicato le regole sopra indicate. Per gli arrays dinamici, questo main slot memorizza solamente la *length*, ossia il numero di elementi nell'array (gli arrays di `bytes` e `strings` sono un'eccezione come si vedrà successivamente). Per i mappings, lo slot rimane vuoto, ma è comunque necessario per garantire che, se anche ci fossero due mappings uno accanto all'altro, il loro contenuto vada a finire in posizioni di archiviazione diverse.

I dati effettivi dell'array sono presenti a partire dall'indirizzo `keccak256(p)` e sono salvati nello stesso modo dei dati di un array a dimensione fissa: un elemento dopo l'altro e, nel

caso in cui gli elementi non siano lunghi più di 16 bytes, più elementi possono condividere lo stesso slot. In caso di arrays dinamici multidimensionali questa regola è applicata ricorsivamente.

Il valore corrispondente a una key di mapping `k` si trova all'indirizzo `keccak256(h(k).p)` dove `.` è l'operazione di concatenazione e `h()` è la funzione che è applicata alla chiave (dipende dal tipo della chiave):

- Per i value types, la funzione `h()` si limita a considerare il valore esteso a 32 bytes (quindi non vengono usate le funzioni hash).
- Per le `string` e i `bytes`, la funzione `h()` computa il Keccak256 hash dei dati senza padding.

Se il mapping value non è un value type, lo slot calcolato indica l'inizio dei dati. Ad esempio, se il mapping value è una struct, bisogna aggiungere un offset corrispondente al membro della struct che si vuole raggiungere.

Viene mostrato ora un esempio esplicativo prendendo come riferimento il Listing 3.8: si vuole trovare la posizione nello storage del valore `data[4][9].c`. La posizione del mapping di per sé è 1 perché la variabile `x` lo precede nella posizione 0 dello storage. Questo significa che `data[4]` è salvato a `keccak256(uint256(4).uint256(1))`. Il tipo di `data[4]` è a sua volta un mapping e il valore di `data[4][9]` inizia a `keccak256(uint256(9).keccak256(uint256(4).uint256(1)))`. L'offset del membro `c` all'interno della struct è 1 perché i membri `a` e `b` sono compattati in un unico slot. Ciò significa che `data[4][9].c` è presente nello storage a `keccak256(uint256(9).keccak256(uint256(4).uint256(1))) + 1` e il suo tipo di valore è `uint256`, quindi usa un singolo slot.

```
1 // SPDX-License-Identifier: MIT
2 pragma solidity >=0.8.0 <0.9.0;
3
4 contract MappingsAnnidati {
5     struct S {
6         uint128 a;
7         uint128 b;
8         uint256 c;
9     }
10
11     uint256 x;
12
13     mapping(uint256 => mapping(uint256 => S)) data;
14 }
```

Listing 3.8: Mappings annidati

Per quanto riguarda i `bytes` e le `string`, essi sono codificati in modo identico.

- Per gli arrays di `bytes` brevi, i dati vengono memorizzati nello stesso slot in cui viene memorizzata la lunghezza. In particolare: se i dati sono lunghi al massimo 31 byte, i dati vengono memorizzati nei bytes di ordine superiore (allineati a sinistra) e nel byte di ordine inferiore viene memorizzata la lunghezza moltiplicata per due ($\text{length} * 2$).
- Per gli arrays di `bytes` che memorizzano dati lunghi 32 o più byte, lo slot principale `p` memorizza la lunghezza dell'array moltiplicata per due in aggiunta a questo primo slot ($\text{length} * 2 + 1$) e i dati vengono memorizzati come al solito alla posizione `keccak256(p)`.

Grazie a ciò è possibile distinguere un array corto da uno lungo controllando il bit più basso: array corto minore di 31 bytes (bit non settato dato che la length è pari) e lungo maggiore di 31 bytes (bit settato dato che la length è dispari).

3.5.2 Layout della memoria

Solidity riserva in memoria i seguenti quattro slot da 32 bytes in intervalli di bytes specifici (inclusi gli endpoints) usati come segue:

- 0x00 - 0x3f (64 bytes): Scratch Space per i metodi di hashing.
- 0x40 - 0x5f (32 bytes): Dimensione della memoria attualmente allocata (Free Memory Pointer).
- 0x60 - 0x7f (32 bytes): Zero Slot.

Lo scratch space può essere usato durante l'hashing e dalle istruzioni scritte in Inline Assembly. Lo zero slot viene usato come valore iniziale per arrays di memoria dinamici, inizialmente punta a 0x80 e non dovrebbe mai essere modificato dallo sviluppatore nel codice. Solidity pone sempre i nuovi oggetti creati all'indirizzo presente nel free memory pointer e in generale la memoria non viene mai liberata.

Gli elementi negli arrays di memoria occupano sempre multipli di 32 bytes (questo vale anche per `byte[]`, ma non per `bytes` e `string`). Gli arrays di memoria multidimensionali sono puntatori agli arrays di memoria. La lunghezza di un array dinamico viene memorizzata nel primo slot e seguita dagli elementi.

Ci sono alcune operazioni in Solidity che richiedono un'area di memoria temporanea maggiore di 64 bytes e quindi non è sufficiente lo scratch space. Queste operazioni verranno effettuate dove punta il free memory pointer, ma data la loro breve durata, il puntatore non verrà aggiornato. Per questo motivo l'area puntata dal free memory pointer non è necessariamente inizializzata a zero.

3.5.3 Locazione di default per l'archiviazione dei dati

Per ogni tipo di variabile esiste una locazione obbligatoria o di default per l'archiviazione dei dati:

- Variabili di stato relative al contratto sono sempre nello storage.
- Parametri delle funzioni sono sempre nella memoria (a esclusione del return e dei parametri delle funzioni con visibilità `external` che sono in `calldata`).
- Variabili locali dei tipi referenziali come structs, arrays o mappings sono salvate nello storage di default (obbligatorio per i mapping) ma possono essere salvate anche in memoria (la locazione deve essere esplicitata quando si dichiara la variabile).
- Variabili locali dei tipi elementari, come `uint`, sono salvate nello stack.

Per le chiamate di funzione è vantaggioso passare le variabili salvate in memoria (default), magari sotto forma di structs, in modo che i dati non vengano copiati ma si riesca a effettuare l'accesso direttamente in memoria. Nel caso in cui la variabile passata come parametro sia nello storage, viene effettuata una copia in memoria e qualsiasi modifica successiva non persisterà.

3.5.4 Assegnamenti

Alcuni chiarimenti riguardo agli assegnamenti:

- Lo storage è preallocato durante la costruzione del contratto e non può essere allocato all'interno di una funzione. Qualsiasi modifica persiste anche dopo la chiamata di funzione.
- La memoria non può essere allocata durante la costruzione del contratto ma solo creata durante l'esecuzione della funzione. Si possono istanziare sia tipi complessi come arrays o structs oppure copiare una variabile di storage. Essa viene resettata tra due chiamate di funzione (esterne).
- Gli assegnamenti tra storage e memoria (o da `calldata`) creano sempre una copia indipendente. Quando si assegna un dato di memoria a un dato di storage, si copia il valore dalla memoria allo storage (preallocato in precedenza). Non viene creato alcun nuovo slot di storage. Quando si assegna un dato di storage a un dato di memoria, si copia il valore dallo storage alla memoria. Viene allocata nuova memoria.
- Gli assegnamenti dalla memoria alla memoria creano solo riferimenti. Ciò significa che le modifiche a una variabile di memoria sono visibili anche in tutte le altre variabili di memoria che fanno riferimento allo stesso dato.

- Gli assegnamenti dallo storage allo storage locale creano solo riferimenti. Non viene creato alcun nuovo spazio di storage.
- Tutti gli altri assegnamenti allo storage creano sempre nuove copie indipendenti. Un esempio sono gli assegnamenti a variabili di stato o a membri di structs locali salvate nello storage, anche nel caso in cui la variabile locale è solo un riferimento.

Ogni volta che viene eseguito il cast da un dato presente nello storage a un dato in memoria viene eseguita una copia e ulteriori modifiche sull'oggetto non si propagano allo stato del contratto come si può vedere nel Listing 3.9. Il dato in memoria può essere assegnato al dato nello storage solo se i dati della memoria possono essere copiati in una variabile di stato preallocata.

```

1 // SPDX-License-Identifier: MIT
2 pragma solidity >=0.8.0 <0.9.0;
3
4 contract MemoryStorage {
5     mapping(uint256 => Account) accounts;
6
7     struct Account {
8         uint256 id;
9         uint256 balance;
10    }
11
12    function addAccount(uint256 id, uint256 balance) public {
13        accounts[id] = Account(id, balance);
14    }
15
16    function updateBalance(uint256 id, uint256 balance) public {
17        Account storage account = accounts[id]; // Accede via reference
18        //Account memory account = accounts[id]; // Crea una copia in memoria
19        account.balance = balance;
20    }
21
22    function getBalance(uint256 id) public view returns (uint256) {
23        return accounts[id].balance;
24    }
25 }

```

Listing 3.9: Aggiornare un valore nello storage

3.6 Analisi dell'archiviazione nello storage

In questa sezione verranno fatte delle analisi relative all'archiviazione delle variabili e delle strutture di dati nello storage.

3.6.1 Archiviazione delle variabili di tipo elementare

Verranno ora analizzati alcuni aspetti relativi alle variabili di tipo elementare.

Inizializzazione delle variabili

Nel caso delle variabili di tipo elementare non è necessario inizializzarle dato che hanno già il loro valore di default al momento della creazione (varia in base al tipo: 0, false, 0x0...); quindi è uno spreco di gas inizializzarle come nel Listing 3.10.

```
1 uint256 var = 0; // Consumo di gas inutile
2 uint256 var;
```

Listing 3.10: Inizializzazione delle variabili

Al contrario, se si fa riferimento a zone di archiviazione, la memoria può o non può essere azzerata. Per questo motivo, non ci si dovrebbe aspettare che la memoria libera punti a memoria azzerata. Con l'operazione `delete` è possibile assegnare il valore di default a un qualsiasi tipo: ad esempio, per gli interi è equivalente a settare a 0 la variabile, ma può essere utilizzato anche su array, dove viene assegnato un array dinamico di lunghezza zero o un array statico della stessa lunghezza con tutti gli elementi impostati al loro valore iniziale. Per le structs, viene assegnata una struct con tutti i membri resettati.

Scambio di variabili

In Solidity è possibile scambiare i valori di due variabili in una singola riga senza dover appoggiarsi a variabili temporanee di supporto o ad altre funzioni (risparmiando quindi gas) come mostrato dal Listing 3.11.

```
1 (var1, var2) = (var2, var1)
```

Listing 3.11: Scambio di variabili

Variabili booleane

Le variabili booleane in Solidity occupano 8 bits quando basterebbe solamente 1 bit. Infatti, dietro la logica di Solidity, i `bool` sono visti come `uint8` e quindi occupano 8 bits di storage. Tuttavia è possibile salvare ben 256 booleani, invece di 32, all'interno

di una variabile `uint256` che occupa un solo slot nello storage come si può vedere dal Listing 3.12.

```
1 // SPDX-License-Identifier: MIT
2 pragma solidity >=0.8.0 <0.9.0;
3
4 contract Bool {
5     uint256 packedBools;
6
7     function test(uint256 pos, bool value) public returns (bool) {
8         setBoolean(pos, value);
9         return getBoolean(pos);
10    }
11
12    function getBoolean(uint256 pos) public view returns (bool) {
13        uint256 flag = (packedBools >> pos) & uint256(1);
14        return (flag == 1 ? true : false);
15    }
16
17    function setBoolean(uint256 pos, bool value) public {
18        if (value) packedBools = packedBools | (uint256(1) << pos);
19        else packedBools = packedBools & ~(uint256(1) << pos);
20    }
21 }
```

Listing 3.12: Valori booleani

3.6.2 Archiviazione degli arrays e delle structs

È vantaggioso usare elementi di dimensioni ridotte solo se si sta lavorando con valori di storage perché il compilatore comprimerà più elementi in uno slot di archiviazione e quindi combinerà più letture o scritture in un'unica operazione. Quando si ha a che fare con argomenti di funzioni o valori di memoria, non vi è alcun vantaggio intrinseco perché il compilatore non compatta questi valori, anzi, in questi casi il consumo di gas dello smart contract potrebbe essere maggiore. Questo perché l'EVM opera sempre su 32 bytes alla volta in memoria; pertanto, se l'elemento è più piccolo di 32 byte, l'EVM deve utilizzare più operazioni (inutili) per ridurre la dimensione dell'elemento da 32 bytes alla dimensione desiderata risultando in uno spreco di gas. Lo stack invece è praticamente a costo nullo e serve per tenere piccole variabili locali ma l'accesso è anche limitato a circa 16 variabili.

Il seguente array nel Listing 3.13 occupa 32 bytes (1 slot) nello storage, ma 128 bytes (4 elementi da 32 bytes ciascuno) nella memoria.

```
1 uint64[4] a;
```

Listing 3.13: Archiviazione degli arrays

La seguente struct nel Listing 3.14 occupa 96 bytes (3 slot di 32 byte) nello storage, ma 128 bytes (4 elementi da 32 bytes ciascuno) nella memoria.

```
1 struct S {  
2     uint256 a;  
3     uint256 b;  
4     uint128 c;  
5     uint128 d;  
6 }
```

Listing 3.14: Archiviazione delle structs

3.6.3 Compattare le variabili nello storage

Al fine di consentire all'EVM di ottimizzare il salvataggio negli slots di storage, si assicuri di ordinare le variabili e i membri della struct in modo che possano essere compattati nel miglior modo possibile. Ad esempio, dichiarando le variabili di archiviazione nell'ordine di `uint128`, `uint128`, `uint256` invece di `uint128`, `uint256`, `uint128`, poiché nel primo caso si occuperà solo due slot di archiviazione (consumo di 44 649 gas alla prima esecuzione) mentre nel secondo caso se ne occuperà tre (consumo di 66 786 gas alla prima esecuzione).

```
1 // SPDX-License-Identifier: MIT  
2 pragma solidity >=0.8.0 <0.9.0;  
3  
4 contract PackVariables {  
5     uint128 aa;  
6     uint128 bb;  
7     uint256 cc;  
8  
9     function saveStorage(  
10         uint128 a,  
11         uint128 b,  
12         uint256 c  
13     ) external {  
14         aa = a;  
15         bb = b;  
16         cc = c;  
17     }  
18 }  
19  
20 contract PackVariablesWrong {
```

```

21  uint128 aa;
22  uint256 bb;
23  uint128 cc;
24
25  function saveStorageWrong(
26      uint128 a,
27      uint256 b,
28      uint128 c
29  ) external {
30      aa = a;
31      bb = b;
32      cc = c;
33  }
34 }

```

Listing 3.15: Compattare le variabili nello storage

3.6.4 Archiviazione nello storage (singolarmente, structs, encoding e decoding)

In questi esempi verranno mostrati tre metodi di salvataggio nello storage più o meno efficienti.

Archiviare singolarmente

Il primo metodo, assolutamente non efficiente, è quello di salvare singolarmente nello storage ciascuna variabile.

```

1  // SPDX-License-Identifier: MIT
2  pragma solidity >=0.8.0 <0.9.0;
3
4  contract PackSingle {
5      mapping(uint256 => address) owners;
6      mapping(uint256 => uint256) dnas;
7      mapping(uint256 => uint16) heights;
8      mapping(uint256 => uint16) weights;
9      mapping(uint256 => uint64) creationTimes;
10
11     function setPerson(
12         uint256 _id,
13         address _owner,
14         uint256 _dna,
15         uint256 _height,
16         uint256 _weight,
17         uint256 _creationTime
18     ) external {

```



```

19     owners[_id] = _owner;
20     dnas[_id] = _dna;
21     heights[_id] = uint16(_height);
22     weights[_id] = uint16(_weight);
23     creationTimes[_id] = uint64(_creationTime);
24 }
25
26 function getPerson(uint256 _id)
27     external
28     view
29     returns (
30         address _owner,
31         uint256 _dna,
32         uint256 _height,
33         uint256 _weight,
34         uint256 _creationTime
35     )
36 {
37     _owner = owners[_id];
38     _dna = dnas[_id];
39     _height = heights[_id];
40     _weight = weights[_id];
41     _creationTime = creationTimes[_id];
42 }
43 }

```

Listing 3.16: Archiviare singolarmente le variabili nello storage

Eseguendo lo smart contract, alla prima chiamata della funzione `setPerson` si consuma ben 111 363 gas, dovuti principalmente alla creazione di ben 5 slot nello storage, uno per ogni variabile salvata.

Archiviare con structs

Un secondo metodo più efficiente è quello di archiviare i valori in una struct compattando il più possibile le variabili.

```

1 // SPDX-License-Identifier: MIT
2 pragma solidity >=0.8.0 <0.9.0;
3
4 contract PackStructs {
5     struct Person {
6         address owner; // Address 160 bit
7         uint64 creationTime;
8         uint16 height;
9         uint16 weight;
10        uint256 dna;

```

```

11 }
12
13 mapping(uint256 => Person) persons;
14
15 function setPerson(
16     uint256 _id,
17     address _owner,
18     uint256 _creationTime,
19     uint256 _height,
20     uint256 _weight,
21     uint256 _dna
22 ) external {
23     persons[_id] = Person(_owner, uint64(_creationTime), uint16(_height),
24         uint16(_weight), _dna);
25 }
26
27 function getPerson(uint256 _id) external view returns (Person memory
28     _person) {
29     _person = persons[_id];
30 }

```

Listing 3.17: Archiviare le structs nello storage

Eseguendo lo smart contract, alla prima chiamata della funzione `setPerson` si consuma 45 051 gas, un notevole risparmio dovuti all'occupazione di soli 2 slot nello storage (contro i 5 di prima). Si è potuto compattare nel primo slot l'address (il quale ha dimensione di 160 bit), e le tre variabili successive di dimensione 64, 16 e 16 bit. La variabile `dna` occupa un intero slot.

Archiviare con encoding e decoding

Il terzo modo proposto è fare l'encoding dei primi 4 campi in una singola variabile `uint256`.

Si usano i seguenti operatori:

- Operatore bitwise od operatore di assegnamento (`|=`): è usato per combinare due valori binari, il risultato viene settato a 1 se almeno uno dei due bit dei membri è 1.
- Operatore bit-shift (left) (`<<`): è usato per spostare i bits di una posizione a sinistra.
- Operatore bit-shift (right) (`>>`): usato per spostare i bits di una posizione a destra.

Come si può vedere nel Listing 3.18, grazie all'operatore bitwise ci si assicura che i bits superiori al 160esimo (dimensione dell'address) siano tutti zero. Grazie all'operatore bit-shift (left) si è potuto spostare i bits nella parte alta dei 256 bits disponibili in modo da

riempire i rimanenti con gli altri valori passati alla funzione. Grazie a questi due operatori si è potuto fare l'encoding; per il decoding si è usato l'operatore bit-shift (right).

```
1 // SPDX-License-Identifier: MIT
2 pragma solidity >=0.8.0 <0.9.0;
3
4 contract PackEncoding {
5     mapping(uint256 => uint256) persons;
6     mapping(uint256 => uint256) dnas;
7
8     function setCharacter(
9         uint256 _id,
10        address _owner,
11        uint256 _creationTime,
12        uint256 _height,
13        uint256 _weight,
14        uint256 _dna
15    ) external returns (bytes32 s) {
16        uint256 person = uint256(uint160(address(_owner)));
17        person |= _creationTime << 160;
18        person |= _height << 224;
19        person |= _weight << 240;
20        persons[_id] = person;
21        dnas[_id] = _dna;
22        return bytes32(person);
23    }
24
25    function getCharacter(uint256 _id)
26        external view
27        returns (
28            address _owner,
29            uint256 _creationTime,
30            uint256 _height,
31            uint256 _weight,
32            uint256 _dna
33        )
34    {
35        uint256 person = persons[_id];
36        _owner = address(uint160(person));
37        _creationTime = uint256(uint64(person >> 160));
38        _height = uint256(uint16(person >> 224));
39        _weight = uint256(uint16(person >> 240));
40        _dna = dnas[_id];
41    }
42 }
```

Listing 3.18: Archiviare con encoding e decoding nello storage

Nella Figura 3.5 è mostrato come sono compattate le varie variabili in un singolo slot da 256 bit.

Slot1: 0x0046 000f 000000005fee57f0 1234574773fc4f943711e57a7542f5af2c83210f
weight height creationTime address
Slot2: 0x1b3daf00ec7779af26574c5707becaed4c4846604c341676d4facdda1a5033bd
dna

Figura 3.5: Compattare più variabili in uno slot usando l'encoding

Eseguendo lo smart contract, alla prima chiamata della funzione `setPerson` si consuma 44 820 gas, un piccolo risparmio dovuto all'uso di operazioni binarie poco costose da eseguire per l'EVM e alle sole due operazioni di storage.

Bisogna notare che queste funzioni non fanno nessun tipo di error checking, introducendo possibili rischi per la sicurezza ma riducendo il consumo di gas. L'archiviazione tramite structs compattate risulta la miglior scelta derivante dal trade-off tra risparmio di gas e sicurezza.

3.7 Analisi dei mappings nello storage

In questa sezione verranno fatte delle analisi relative ai mappings e al loro funzionamento nello storage.

Nel corso delle analisi saranno di supporto le seguenti funzioni Python per calcolare il Keccak256 hash:

```
1 import binascii
2
3 from _pysha3 import keccak_256
4
5
6 def bytes32(i):
7     return binascii.unhexlify('%064x' % i)
8
9
10 def keccak256(x):
11     return keccak_256(x).hexdigest()
```

Listing 3.19: Funzioni Python per calcolare il Keccak256 hash

In Solidity è possibile trovare l'opcode `SHA3` oppure `KECCAK256`, essi sono alias l'uno dell'altro.

3.7.1 Analisi del funzionamento del Keccak256 hash con i mappings

Prendendo come riferimento il primo contratto presente nel Listing 3.20, si possono notare che i valori dei due mappings vengono salvati alla rispettive chiavi non direttamente ma facendo il Keccak256 hash della loro chiave.

```
1 // SPDX-License-Identifier: MIT
2 pragma solidity >=0.8.0 <0.9.0;
3
4 contract Mappings1 {
5     mapping(uint256 => uint256) itemsA;
6     mapping(uint256 => uint256) itemsB;
7
8     function map1() public {
9         itemsA[0xAAAA] = 0xAAAA;
10        itemsB[0xB BBB] = 0xB BBB;
11    }
12 }
```

Listing 3.20: Contratto Mapping1

La formula per calcolare l'indirizzo di storage per una chiave è quella mostrata nel Listing 3.21 in cui sono mostrate anche a che indirizzi sono salvati i due mappings. Si può notare come i due slot principali risiedono nella posizione 0 e 1 dello storage.

```

1 # Formula
2 keccak256(bytes32(key) + bytes32(position))
3
4 # Mapping A
5 >>> keccak256(bytes32(0xAAAA) + bytes32(0))
6 '839613f731613c3a2f728362760f939c8004b5d9066154aab51d6dadf74733f3'
7
8 # Mapping B
9 >>> keccak256(bytes32(0xB BBB) + bytes32(1))
10 '34cb23340a4263c995af18b23d9f53b67ff379ccaa3a91b75007b010c489d395'

```

Listing 3.21: Calcoli del Keccak256 hash per i due mappings

Nel bytecode del contratto presente nel Listing 3.22, se si va ad analizzare il debug della chiamata a funzione, i valori dei due hash vengono precalcolati e inseriti come literals nel bytecode risparmiando operazioni di calcolo on-chain. Infatti, il compilatore, grazie all'ottimizzazione attiva, è stato in grado di precalcolare l'address data una chiave perché i valori erano costanti. Se la chiave usata fosse stata una variabile, l'hash obbligatoriamente deve essere calcolato durante l'esecuzione del bytecode.

```

1 # CODICE OMESSO
2 .data
3 0:
4 # CODICE OMESSO
5 tag 3 function map1() public {\n ...
6 JUMPDEST
7 PUSH [tag] 4
8 PUSH AAAA 0xAAAA
9 PUSH 839613F731613C3A2F728362760F939C8004B5D9066154AAB51D6DADF74733F3
   itemsA[0xAAAA]
10 SSTORE itemsA[0xAAAA] = 0xAAAA
11 PUSH BBBB 0xB BBB
12 PUSH 0 itemsA
13 DUP2 itemsB[0xB BBB]
14 SWAP1 itemsB[0xB BBB]
15 MSTORE itemsB[0xB BBB]
16 PUSH 1 itemsB
17 PUSH 20 itemsA[0xAAAA]
18 MSTORE itemsB[0xB BBB]
19 PUSH 34CB23340A4263C995AF18B23D9F53B67FF379CCAA3A91B75007B010C489D395
   itemsB[0xB BBB]
20 SSTORE itemsB[0xB BBB] = 0xB BBB

```

```

21      JUMP          function map1() public {\n      ...
22      # CODICE OMESSO

```

Listing 3.22: Bytecode del contratto Mappings1

Nel Listing 3.23 viene mostrato un esempio in cui il Keccak256 hash viene effettuato on-chain nel bytecode anche quando l'ottimizzatore è attivo, visto che la chiave è salvata in una variabile `i` e l'ottimizzatore non è in grado di precalcolarla.

```

1 // SPDX-License-Identifier: MIT
2 pragma solidity >=0.8.0 <0.9.0;
3
4 contract Mappings2 {
5     mapping(uint256 => uint256) itemsC;
6
7     uint256 i = 0xC000;
8
9     function map2() public {
10         itemsC[i] = 0x7777;
11     }
12 }

```

Listing 3.23: Contratto Mapping2

Si può notare eseguendo il debug del contratto che, a differenza del bytecode del precedente contratto, nel bytecode mostrato nel Listing 3.24 è presente l'opcode `KECCAK256` nel che computa il Keccak256 hash on-chain dopo aver caricato il valore della variabile `i` precedentemente salvata nello storage per avere l'address in cui salvare il valore `0x7777`.

```

1 .code
2     # CODICE OMESSO
3     PUSH CCCC          0xC000
4     PUSH 1             uint256 i = 0xC000
5     SSTORE             uint256 i = 0xC000
6     # CODICE OMESSO
7 .data
8     0:
9     # CODICE OMESSO
10    tag 3              function map2() public {\n      ...
11        JUMPDEST
12        PUSH [tag] 4
13        PUSH 1         i
14        SLOAD          i
15        PUSH 0         itemsC
16        SWAP1          itemsC[i]
17        DUP2           itemsC[i]

```

```

18     MSTORE      itemsC[i]
19     PUSH 20     itemsC[i]
20     DUP2       itemsC[i]
21     SWAP1      itemsC[i]
22     MSTORE      itemsC[i]
23     PUSH 40     itemsC[i]
24     SWAP1      itemsC[i]
25     KECCAK256   itemsC[i]    # ALIAS DI SHA3
26     PUSH 7777   0x7777
27     SWAP1      itemsC[i] = 0x7777
28     SSTORE     itemsC[i] = 0x7777
29     JUMP
30     # CODICE OMESSO

```

Listing 3.24: Bytecode del contratto Mappings2

La prima parte del bytecode va a concatenare la key e la position salvando i valori in due chunks di memoria contigui (nel dettaglio agli indirizzi `0x00` e `0x20` di memoria). Successivamente viene effettuato il Keccak256 hash su 64 bytes (`0x40` sono 64 bit) a partire dalla posizione `0x00`. In pratica viene effettuato l'hash sulla concatenazione dei due valori salvati in precedenza.

Il costo per ogni operazioni di Keccak256 hash dipende da quanti dati devono essere trattati:

- Per ogni operazione di `SHA3` (alias di `KECCAK256`) si consuma 30 unità di gas.
- Per ogni word da 32 bytes (arrotondata per eccesso) come input per l'operazione di `SHA3` si consuma 6 unità di gas. Solitamente due word, una relativa alla key e una alla position.

Quindi, per una chiave di tipo `uint256` il costo del gas è 42 dato da $(30 + 6 * 2)$.

3.7.2 Analisi dei mappings con valori di grande dimensione

Nel caso in cui il value di un mapping sia di dimensione superiore a 32 bytes, esso viene accodato in posizioni successive al primo Keccak256 hash calcolato.

Un esempio è mostrato nel Listing 3.25 in cui come value si ha una struct con tre campi ciascuno di dimensione 32 bytes.

```

1 // SPDX-License-Identifier: MIT
2 pragma solidity >=0.8.0 <0.9.0;
3
4 contract MappingStruct {
5     mapping(uint256 => Tuple) tuples;
6

```



```

7  struct Tuple {
8      uint256 a;
9      uint256 b;
10     uint256 c;
11 }
12
13 function mapStruct() public {
14     tuples[0x1].a = 0x1A;
15     tuples[0x1].b = 0x1B;
16     tuples[0x1].c = 0x1C;
17 }
18 }

```

Listing 3.25: Contratto MappingStruct

Andando a visualizzare lo spazio di storage finale dopo aver eseguito il debug della chiamata di funzione `mapStruct()` mostrato nel Listing 3.26, si può notare che i tre campi della struct risiedono in tre slot contigui che partono dal primo Keccak256 hash calcolato (`0xada...7d`, `0xada...7e`, `0xada...7f`).

```

1 # Storage
2 key: '0xada5013122d395ba3c54772283fb069b10426056ef8ca54750cb9bb552a59e7d '
3 val: '0x0000000000000000000000000000000000000000000000000000000000000001a '
4
5 key: '0xada5013122d395ba3c54772283fb069b10426056ef8ca54750cb9bb552a59e7e '
6 val: '0x0000000000000000000000000000000000000000000000000000000000000001b '
7
8 key: '0xada5013122d395ba3c54772283fb069b10426056ef8ca54750cb9bb552a59e7f '
9 val: '0x0000000000000000000000000000000000000000000000000000000000000001c '

```

Listing 3.26: Debug dello stato dopo l'esecuzione della funzione `mapStruct()`

3.7.3 Analisi dei mappings che non si compattano

Nel caso in cui il value è minore di 16 bytes, a differenza delle variabili nelle structs, i mappings non compattano più valori in un singolo slot ma ogni singolo valore necessiterà di un proprio slot come si può verificare eseguendo il codice presente nel Listing 3.27.

```

1 // SPDX-License-Identifier: MIT
2 pragma solidity >=0.8.0 <0.9.0;
3
4 contract MappingNotPack {
5     mapping(uint256 => uint8) items;
6
7     function mapNotPack() public {
8         items[0xA] = 0xAA;
9     }
10 }

```

```

9     items[0xB] = 0xBB;
10 }
11 }

```

Listing 3.27: Contratto MappingNotPack

Andando a visualizzare lo spazio di storage finale dopo aver eseguito il debug della chiamata di funzione `mapNotPack()` mostrato nel Listing 3.28, si può notare che i due valori risiedono in due differenti slot distinti.

```

1 # Storage
2 key: '0x3e9abaca0aad9ede81f4474766c846d8539f70688e1c8f521bbe1597874e3dc4 '
3 val: '0x00000000000000000000000000000000000000000000000000000000000000aa '
4
5 key: '0x9115655cbcdb654012cf1b2f7e5dbf11c9ef14e152a19d5f8ea75a329092d5a6 '
6 val: '0x000000000000000000000000000000000000000000000000000000000000bb '

```

Listing 3.28: Debug dello stato dopo l'esecuzione della funzione `mapNotPack()`

3.7.4 Conclusioni sui mappings

Come esempio finale per trarre le conclusioni sui mapping viene preso come riferimento il contratto presente nel Listing 3.29.

```

1 // SPDX-License-Identifier: MIT
2 pragma solidity >=0.8.0 <0.9.0;
3
4 contract MappingEnd {
5     mapping (uint256 => uint128) itemsA;
6     mapping (uint256 => uint128) itemsB;
7
8     function map(uint256 x) public {
9         itemsA[72] = 11 ;
10        itemsA[73] = 12 ;
11        itemsB[x] = 22 ;
12    }
13 }

```

Listing 3.29: Contratto MappingEnd

Appositamente si sono messi i value types come `uint128` per verificare se i mappings compattassero le variabili in un unico slot di storage; dopo le verifiche del caso, si può dire che non vi è compattazione in un unico slot, ogni elemento parte da uno slot di storage vuoto.

Vengono mostrati graficamente nella Figura 3.6 gli slot di storage usati, il funzionamento dei mapping e il fatto che i mappings non compattano i valori nello stesso slot ma usano sempre slot separati per ogni elemento.

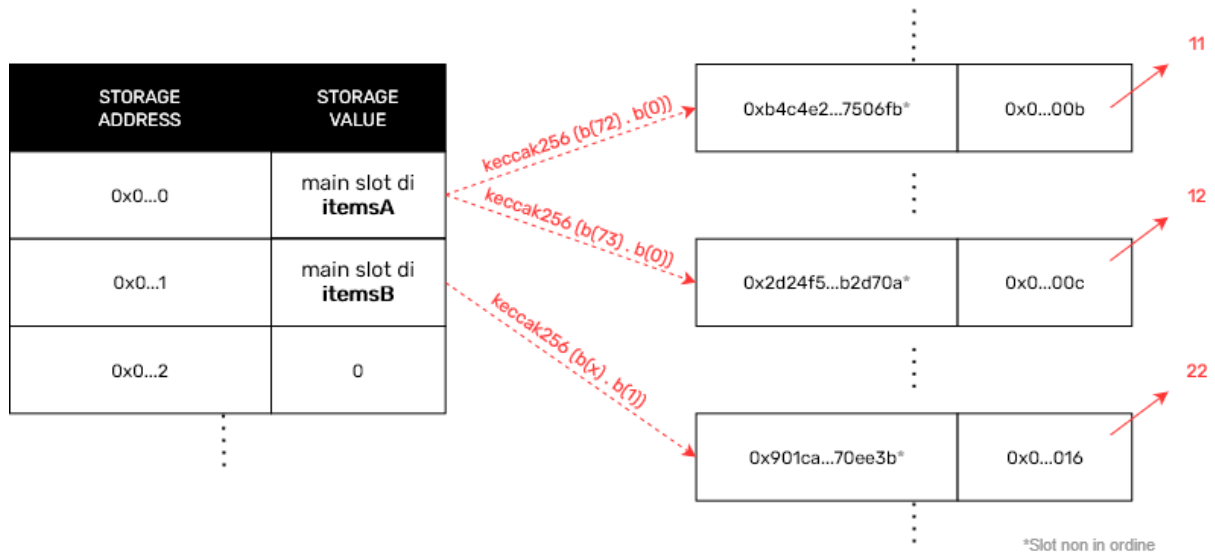


Figura 3.6: Figura esplicativa dei mappings in riferimento al contratto MappingEnd

3.8 Analisi degli arrays nello storage

In questa sezione verranno fatte delle analisi relative agli arrays e al loro funzionamento nello storage.

3.8.1 Confronto dei costi tra i vari tipi di arrays

In questa sottosezione si andranno a mostrare con due contratti che verranno presi come riferimento per mostrare tutti i possibili tipi e gli usi degli arrays.

Arrays (Parte 1)

Nel Listing 3.30 è presente un contratto che mostra molti dei possibili tipi di arrays presenti in Solidity.

```
1 // SPDX-License-Identifier: MIT
2 pragma solidity >=0.8.0 <0.9.0;
3
4 contract Arrays1 {
5     bytes16[300] b_one;
6
7     bytes b_two;
8     string s_one;
9
10    uint64[5] numbers_one;
11    uint64[4] numbers_two;
12
13    uint64[] numbers_three;
14
15    // Execution cost: 109781 gas
16    function fixedByteArray() public returns (uint256) {
17        b_one[0] = bytes16("0x1111");
18        b_one[1] = bytes16("0x1111");
19        b_one[2] = bytes16("0x2222");
20        b_one[3] = bytes16("0x2222");
21        b_one[4] = bytes16("0x3333");
22        b_one[200] = bytes16("0x4444");
23
24        return b_one.length;
25    }
26
27    // Execution cost: 31 - 73459 gas / 32 - 96612 gas
28    function dynamicByteArray() public returns (uint256) {
29        for (int256 i = 0; i < 32; i++) {
30            // Cambiare 31/32
31            b_two.push("a");
```

```

32     }
33
34     // b_two = '0xaaaa'; // Alternativa
35     return b_two.length;
36 }
37
38 // Execution cost: 43846 gas
39 function dynamicStringArray() public {
40     s_one = "Hello World!"; // UTF-8 Encoding
41
42     // s_one.push('Test'); // Non si può usare il .push
43
44     // return s_one.length; // Non ha salvato la sua length
45 }
46
47 // Execution cost: 87650 gas
48 function fixedArray() public returns (uint256) {
49     numbers_one[0] = 0x1111111111111111;
50     numbers_one[2] = 0x2222222222222222;
51     numbers_one[4] = 0x5555555555555555;
52
53     numbers_two[0] = 0x1111111111111111;
54
55     return numbers_one.length;
56 }
57
58 // Execution cost: 90313 gas
59 function dynamicArray() public returns (uint256) {
60     // numbers_three[0] = 0x9999999999999999; // Errore, prima deve esserci
        un dato a quell'indice
61
62     numbers_three.push(0x1111111111111111);
63     numbers_three.push(0x2222222222222222);
64     numbers_three.push(0x3333333333333333);
65     numbers_three.push(0x4444444444444444);
66     numbers_three.push(0x5555555555555555);
67
68     numbers_three[0] = 0x9999999999999999; // Ora è possibile
69
70     // numbers_three.length = 10; // A partire dalla versione 0.6.0 di
        Solidity, la length è read-only anche per gli arrays dinamici nello
        storage
71
72     return numbers_three.length;
73 }
74
75 // Execution cost: 21557 gas

```

```

76 function dynamicMemoryArray() public returns (uint256) {
77     uint32[] memory a_mem = new uint32[](5); // Operatore new
78
79     // a_mem.push(1); // Non è possibile per gli arrays dinamici in memoria
    , bisogna usare l'indice
80
81     a_mem[0] = 1111;
82     a_mem[4] = 5555;
83     // a_mem[100] = 1; // Errore "Out of Bound"
84
85     return a_mem.length;
86 }
87 }

```

Listing 3.30: Contratto di riferimento sugli arrays (Parte 1)

Nella funzione `fixedByteArray()` del Listing 3.30 si analizzano gli arrays di dimensione fissa di tipo `byte16`. La prima coppia di operazioni va a compattare due valori da 16 bytes in un unico slot, stessa cosa per la seconda coppia di operazioni. L'operazione che scrive in `b_one[4]` va a creare un nuovo slot di storage di seguito ai primi due. L'ultima operazione che scrive in `b_one[200]` non va a salvare il valore nello spazio rimanente nel terzo slot di storage ma bensì ne va a creare uno nuovo alla posizione `0x0...064`; questo perché gli indici non sono contigui.

In totale il consumo di gas dovuto all'esecuzione della funzione è di 109 781 gas: le maggiori fonti di costo, in aggiunta al costo della transazione di 21 000 gas, sono i quattro accessi a slot di storage che consumano 2 100 gas ciascuno (si può trascurare il costo di 100 gas per ogni `SSTORE` su slot di storage già acceduti).

```

1 # Storage
2 key: '0x0000000000000000000000000000000000000000000000000000000000000000'
3 val: '0x3078313131310000000000000000000000307831313131000000000000000000'
4
5 key: '0x0000000000000000000000000000000000000000000000000000000000000001'
6 val: '0x3078323232320000000000000000000000307832323232000000000000000000'
7
8 key: '0x0000000000000000000000000000000000000000000000000000000000000002'
9 val: '0x0000000000000000000000000000000000307833333333000000000000000000'
10
11 key: '0x0000000000000000000000000000000000000000000000000000000000000064'
12 val: '0x0000000000000000000000000000000000307834343434000000000000000000'

```

Listing 3.31: Debug dello stato dopo l'esecuzione della funzione `fixedByteArray()`

Nella funzione `dynamicByteArray()` del Listing 3.30 si analizza il tipo speciale `bytes`. Grazie a un ciclo `for`, si vanno a salvare 31 singoli bytes nello storage in un caso e 32

In totale il consumo di gas dovuto all'esecuzione della funzione è di 73 459 gas (nel caso di 31 bytes) e di 96 612 gas (nel caso di 32 bytes). La differenza di costo tra i due casi è dovuta all'accesso di un ulteriore slot di storage nel caso di 32 bytes.

Listing 3.32: Debug dello stato dopo l'esecuzione della funzione `dynamicByteArray()` nel caso di 31 byte

Listing 3.33: Debug dello stato dopo l'esecuzione della funzione `dynamicByteArray()` nel caso di 32 byte

In totale il consumo di gas dovuto all'esecuzione della funzione è di 43 846 gas.

Listing 3.34: Debug dello stato dopo l'esecuzione della funzione `dynamicStringArray()`

65

Notare che non esiste lo slot per la length visto che è a dimensione fissa.

In totale il consumo di gas dovuto all'esecuzione della funzione è di 87 650 gas.

```
1 # Storage
2 key: '0x0000000000000000000000000000000000000000000000000000000000000098 '
3 val: '0x0000000000000000222222222222220000000000000001111111111111111 '
4
5 key: '0x0000000000000000000000000000000000000000000000000000000000000099 '
6 val: '0x0000000000000000000000000000000000000000000005555555555555555 '
7
8 key: '0x000000000000000000000000000000000000000000000000000000000000009a '
9 val: '0x0000000000000000000000000000000000000000000001111111111111111 '
```

Listing 3.35: Debug dello stato dopo l'esecuzione della funzione `fixedArray()`

Nella funzione `dynamicArray()` del Listing 3.30 si analizzano gli arrays di dimensione dinamica. Si effettua il push di cinque elementi ciascuno di dimensione 8 bytes. Come si può vedere dal Listing 3.36, nel primo slot di storage è salvata solamente la `length`, nei seguenti i dati compattati a quartetti in un unico slot di storage.

Notare che non è possibile accedere per indice se in quella posizione non si è già effettuato l'accesso. Inoltre, a partire dalla versione 0.6.0 di Solidity, `length` è read-only anche per gli arrays dinamici nello storage.

In totale il consumo di gas dovuto all'esecuzione della funzione è di 90 313 gas.

```
1 # Storage
2 key: '0x000000000000000000000000000000000000000000000000000000000000009b '
3 val: '0x0000000000000000000000000000000000000000000000000000000000000005 '
4
5 key: '0xbba9db4cdbea0a37c207bbb83e20f828cd4441c49891101dc94fd20dc8efc349 '
6 val: '0x44444444444444443333333333333333222222222222229999999999999999 '
7
8 key: '0xbba9db4cdbea0a37c207bbb83e20f828cd4441c49891101dc94fd20dc8efc34a '
9 val: '0x0000000000000000000000000000000000000000000005555555555555555 '
```

Listing 3.36: Debug dello stato dopo l'esecuzione della funzione `dynamicArray()`

Nella funzione `dynamicMemoryArray()` del Listing 3.30 si analizzano gli arrays di dimensione dinamica salvati nella memoria (a differenza di tutti quelli già visti che erano nello storage). Attraverso l'operatore `new` si crea l'array in memoria al quale deve essere specificata però la lunghezza massima.

Notare che l'operazione di `push` non è possibile su arrays di memoria dinamici. Inoltre, vi è anche un controllo sui limiti degli arrays.

In totale il consumo di gas dovuto all'esecuzione della funzione è di 21 557 gas.

Arrays (Parte 2)

Nel Listing 3.37 è presente un contratto che mostra l'uso degli arrays multidimensionali, della conversione tra array, degli arrays passati come parametri e di quelli ritornati dalle funzioni.

```
1 // SPDX-License-Identifier: MIT
2 pragma solidity >=0.8.0 <0.9.0;
3
4 contract Arrays2 {
5     uint256[][3] fixedMultiArray;
6     uint256[2][] dynamicMultiArray;
7
8     uint256[] storageDynamicArray; // Tipo uint[] storage ref (all'interno di
9                                     una funzione)
10
11     function multidimensional() public {
12         uint256[4] memory memArray = [uint256(6), 7, 8, 9];
13
14         fixedMultiArray[0] = memArray;
15         fixedMultiArray[1] = new uint256[](4);
16         fixedMultiArray[2] = [1, 2, 3, 4, 5];
17
18         //dynamicMultiArray = memArray; // Errore
19         dynamicMultiArray = new uint256[2][](3); // Istanziare un array
20         dinamico nello storage con new (Memoria allo storage)
21         dynamicMultiArray[0] = [1, 2];
22         dynamicMultiArray[1] = [3, 4];
23         dynamicMultiArray[2] = [5, 6];
24
25         // dynamicMultiArray[3] = [7,8]; // Errore, non possibile da estendere
26     }
27
28     function conv() public {
29         uint256[3] memory memArray = [uint256(6), 7, 8]; // Tipo uint[] memory
30         uint256[] storage localStorageDynamicArray; // Tipo uint[] storage
31         pointer
32
33         storageDynamicArray = new uint256[](3);
34         storageDynamicArray = memArray;
35
36         // localStorageDynamicArray = new uint[](3); // Errore
37         // localStorageDynamicArray = memArray; // Errore
38         localStorageDynamicArray = storageDynamicArray;
39     }
40
41     // PARAMS
```

```

39  function params(uint256[] memory paramsArray) public {
40      // Anche con memory/calldata
41      storageDynamicArray = paramsArray; // Copia l'intero memory/calldata
      array nello storage
42  }
43
44  function mutliParams(uint256[2][2] memory paramsArray) public {
45      // Anche con memory/calldata
46  }
47
48  function mutliParams(uint256[][2] memory paramsArray) public {
49      // Anche con memory/calldata (Illegale senza ABICoderV2)
50  }
51
52  function mutliParams(uint256[2][] memory paramsArray) public {
53      // Anche con memory/calldata
54  }
55
56  function mutliParams(uint256[][] memory paramsArray) public {
57      // Anche con memory/calldata (Illegale senza ABICoderV2)
58  }
59
60  // RETURNS BYTES/STRING ARRAY (Sono bidimensionali)
61  function retBytes(bytes[] memory s) public returns (bytes[] memory) {}
62
63  function retString(string[] memory s) public returns (string[] memory) {}
64  }

```

Listing 3.37: Contratto di riferimento sugli arrays (Parte 2)

Gli arrays `fixedMultiArray` e `dynamicMultiArray` sono dichiarati come variabili di stato del contratto e per tale motivo sono riferimenti allo storage. In particolare, l'array `fixedMultiArray` è di lunghezza fissa 3 i cui elementi sono arrays dinamici d'interi senza segno, mentre `dynamicMultiArray` è un array dinamico i cui elementi sono coppie d'interi senza segno.

Nella funzione `multidimensional()`, i tre elementi del `fixedMultiArray` sono istanziati in tre modi diversi: copiando i valori da un array di memoria precedentemente creato, attraverso l'operatore `new` oppure passandogli direttamente l'array di valori che viene copiato nello storage. Successivamente, l'array dinamico `dynamicMultiArray` viene istanziato specificandogli di avere tre elementi, ognuno dei quali contiene una coppia d'interi senza segno; inoltre, non è possibile estendere l'array successivamente.

Nella funzione `conv()` viene mostrato il fatto che i tipi del `storageDynamicArray` e del `localStorageDynamicArray` sono diversi: il primo è una variabile di stato del contratto e quando è usata all'interno di una funzione il suo tipo è `uint[] storage ref`, mentre

il secondo array è del tipo `uint[] storage pointer`. Nel primo caso quindi si ha un riferimento a una locazione nello storage, un assegnamento copia l'array dalla memoria allo storage. Nel secondo caso si ha solo un puntatore, nell'esempio si tentava erroneamente di cambiare il puntatore facendolo puntare a una variabile locale in memoria.

Nella funzione `params()` e nelle successive funzioni `mutliParams()` (in cui viene usato il polimorfismo per scrivere funzioni con stesso nome ma differenti segnature) vengono esplorate tutte le combinazioni di arrays bidimensionali dinamici e a dimensione fissa. Due di queste funzioni, senza l'ABI Coder v2 abilitato o di default, sono illegali.

Nelle funzioni `retBytes()` e `retString()` viene mostrato come ritornare tipi bidimensionali quali `bytes[]` e `string[]` (ricordando che normalmente questi tipi speciali sono già da considerare come se fossero arrays).

3.8.2 Analisi degli arrays dinamici

In un tipico linguaggio di programmazione, un array è una lista di elementi che risiedono insieme in memoria. Ad esempio, un array di 100 elementi ciascuno da 1 byte occuperebbe 100 bytes in memoria e in questo schema è relativamente economico caricare tutto l'array in blocco nella cache della CPU e iterare tra gli elementi. Per molti linguaggi, gli arrays richiedono meno calcolo computazionale dei mappings.

Tuttavia, in Solidity, un array può essere visto come una versione molto più costosa di un mapping. Gli elementi di un array sono disposti in modo contiguo nello storage ma bisogna tenere a mente che ogni accesso a questi slots di storage è in realtà un key-value lookup. Accedere a un elemento di un array non è differente da accedere a un elemento di un mapping.

Si consideri il tipo `uint256[]`, è essenzialmente identico a un `mapping(uint256=>uint256)` con in aggiunta le seguenti features che lo rendono *array-like*:

- Un campo `length` per indicare quanti elementi ci sono.
- Bound checking, verifica sui limiti dell'array che lancia un errore se si legge o si scrive al di fuori dei limiti dettati dalla lunghezza.
- Uno storage packing più sofisticato rispetto ai mappings. Infatti, a differenza dei mappings, è possibile compattare più valori in un singolo slot di storage.
- Azzeramento automatico degli slots di storage quando un array viene ridotto o cancellato del tutto.
- Ottimizzazioni speciali per `bytes` e `string` per creare array piccoli (minori di 31 bytes) che sono molto più efficienti per quanto riguarda lo storage.

Si prenda come riferimento la funzione `ad1()` nel Listing 3.38.

```
1 // SPDX-License-Identifier: MIT
2 pragma solidity >=0.8.0 <0.9.0;
3
4 contract ArraysDynamic {
5     uint256[] array1;
6     uint64[] array2;
7
8     function ad1() public {
9         array1.push(0xAA);
10        array1.push(0xBB);
11        array1.push(0xCC);
12    }
13
14    function ad2() public {
15        array2.push(0xAA);
16        array2.push(0xBB);
17        array2.push(0xCC);
18        array2.push(0xDD);
19    }
20 }
```

Listing 3.38: Contratto ArraysDynamic

Si può notare, come mostrato nel Listing 3.39, che viene usato uno slot di storage per salvare la lunghezza dell'array e altri tre slot di storage contigui che contengono gli elementi (il cui indirizzo parte dal Keccak256 hash).

```
1 # Storage
2 key: '0x0000000000000000000000000000000000000000000000000000000000000000 '
3 val: '0x0000000000000000000000000000000000000000000000000000000000000003 '
4
5 key: '0x290decd9548b62a8d60345a988386fc84ba6bc95484008f6362f93160ef3e563 '
6 val: '0x00000000000000000000000000000000000000000000000000000000000000aa '
7
8 key: '0x290decd9548b62a8d60345a988386fc84ba6bc95484008f6362f93160ef3e564 '
9 val: '0x00000000000000000000000000000000000000000000000000000000000000bb '
10
11 key: '0x290decd9548b62a8d60345a988386fc84ba6bc95484008f6362f93160ef3e565 '
12 val: '0x00000000000000000000000000000000000000000000000000000000000000cc '
```

Listing 3.39: Debug dello stato dopo l'esecuzione della funzione `ad1()`

Tuttavia, nel caso gli elementi sono di dimensione minore di 16 bytes, è possibile compattare più valori dell'array in un unico slot come si può vedere nella funzione `ad2()` sempre presente nel Listing 3.38. Infatti, andando a guardare il debug mostrato nel Listing 3.40,

si può notare che, oltre allo slot relativo alla lunghezza dell'array, viene occupato un unico slot di storage che contiene i quattro elementi compattati.

```
1 # Storage
2 key: '0x0000000000000000000000000000000000000000000000000000000000000001 '
3 val: '0x0000000000000000000000000000000000000000000000000000000000000004 '
4
5 key: '0xb10e2d527612073b26eecd7d717e6a320cf44b4afac2b0732d9fcbe2b7fa0cf6 '
6 val: '0x0000000000000000dd00000000000000cc00000000000000bb00000000000000aa '
```

Listing 3.40: Debug dello stato dopo l'esecuzione della funzione `ad2()`

Tuttavia vi è una nota negativa: il bytecode non è ben ottimizzato dato che, invece di compattare le variabili in memoria e accedere solamente una volta nello storage, vengono usate quattro operazioni `SSTORE` sullo stesso slot di storage. Questo perché il bound checking e altre features degli arrays impediscono l'ottimizzazione; anche usando l'attributo `unchecked` (vedi sottosezione 3.8.4) non si risolve il problema.

3.8.3 Analisi dei `bytes` e delle `string`

Come visto, le `string` sono simili ai `bytes`, perciò ci si concentrerà solo sui `bytes`. Entrambi sono ottimizzati per il tipo di dato che contengono e soprattutto vi è un'elevata efficienza per arrays lunghi al massimo 31 bytes in quanto un solo slot di storage è usato per tutto. Nel caso di lunghezza maggiore, i `bytes` sono rappresentati nello stesso modo dei normali arrays. Si analizza ora il contratto mostrato nel Listing 3.41.

```
1 // SPDX-License-Identifier: MIT
2 pragma solidity >=0.8.0 <0.9.0;
3
4 contract ArraysBytes {
5     bytes b1;
6     bytes b2;
7
8     function by1() public {
9         b1.push(0xAA);
10        b1.push(0xBB);
11        b1.push(0xCC);
12    }
13
14    function by2() public {
15        b2 = "ABCDEFGHJKLMNOPQRSTUVWXYZABCDEFGHJKLMNOPQRSTUVWXYZ";
16    }
17 }
```

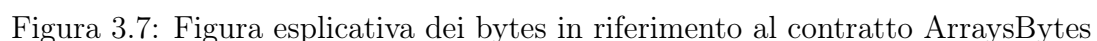
Listing 3.41: Contratto `ArraysBytes`

```
# Storage
key: '0x0000000000000000000000000000000000000000000000000000000000000000'
val: '0xaabbcc000000000000000000000000000000000000000000000000000000006'
```

I dati vengono salvati man mano da sinistra a destra, gli zeri sono i dati mancanti mentre l'ultimo byte `0x06` rappresenta l'encoded length dell'array: la formula è `length = encodedLength / 2`, nel nostro caso quindi la lunghezza è di tre bytes.

```
1 # Storage
2 key: '0x0000000000000000000000000000000000000000000000000000000000000000'
3 val: '0x00000000000000000000000000000000000000000000000000000000000059'
4
5 key: '0xb10e2d527612073b26eecdfd717e6a320cf44b4afac2b0732d9fcbe2b7fa0cf6'
6 val: '0x41424344454647484a4b4c4d4e4f505152535455565a141424344454647484a4b'
7
8 key: '0xb10e2d527612073b26eecdfd717e6a320cf44b4afac2b0732d9fcbe2b7fa0cf7'
9 val: '0x4c4d4e4f505152535455565a00000000000000000000000000000000000000'
```

In Figura 3.7 viene mostrata la disposizione degli slot di storage.



3.8.4 Unchecked

Solidity attualmente esegue i bound checks sull'accesso all'array, assicurandosi che l'indice specificato è inferiore alla lunghezza dell'array e che non sia negativo. In alcuni scenari, questo controllo può essere saltato dallo sviluppatore per risparmiare gas usando l'attributo `unchecked`, basandosi però su altre garanzie per il range degli indici come può essere un `require` (vedi sottosezione 3.12.2).

Come si può vedere nel Listing 3.44, la funzione `check()` che effettua i controlli consuma 22 116 gas per l'esecuzione, mentre la funzione `unchecked()` che non effettua i controlli ma usa il `require` consuma 22 051 gas per l'esecuzione.

```
1 // SPDX-License-Identifier: MIT
2 pragma solidity >=0.8.0 <0.9.0;
3
4 contract Unchecked {
5     function check() public {
6         uint8[8] memory array = [8, 8, 8, 8, 8, 8, 8, 8];
7         uint256 start = 2;
8         uint256 size = 3;
9
10        for (uint256 i = 0; i < size; ++i) {
11            foo(array[start + i]);
12        }
13    }
14
15    function unchecked() public {
16        uint8[8] memory array = [8, 8, 8, 8, 8, 8, 8, 8];
17        uint256 start = 2;
18        uint256 size = 3;
19
20        require(start + size <= array.length);
21
22        for (uint256 i = 0; i < size; ++i) {
23            unchecked {
24                foo(array[start + i]);
25            }
26        }
27    }
28 }
```

Listing 3.44: Unchecked

3.8.5 Conclusioni sugli arrays

C'è una piccola differenza nel costo del gas nel bounds checking per arrays statici e dinamici: quelli dinamici sono un po' più costosi perché richiedono sempre uno slot in più contenente la `length`, ovvero il main slot a partire dal quale viene calcolato il Keccak256 hash per gli altri slot di dati.

Prendendo come riferimento il contratto mostrato nel Listing 3.45, si sono svolti numerosi test variando il numero di elementi inseriti negli arrays: si è partiti da un solo elemento fino ad arrivare a cinque ricordando che ogni volta veniva istanziato un nuovo contratto a ogni run in modo da avere uniformità di giudizio e slot si storage azzerati.

```
1 // SPDX-License-Identifier: MIT
2 pragma solidity >=0.8.0 <0.9.0;
3
4 contract ArraysDiff {
5     uint64[5] staticArr;
6     uint64[] dynamicArr;
7
8     function fixedArray() public {
9         staticArr[0] = 0x0000000000000000;
10        staticArr[4] = 0x4444444444444444; // Non in ordine
11        staticArr[1] = 0x1111111111111111;
12        staticArr[2] = 0x2222222222222222;
13        staticArr[3] = 0x3333333333333333;
14    }
15
16    function dynamicArray() public {
17        dynamicArr.push(0x0000000000000000);
18        dynamicArr.push(0x1111111111111111);
19        dynamicArr.push(0x2222222222222222);
20        dynamicArr.push(0x3333333333333333);
21        dynamicArr.push(0x4444444444444444);
22    }
23 }
```

Listing 3.45: Differenze di costo tra arrays a dimensione fissa e dinamici

Viene confermato anche dalla figura Figura 3.8 il fatto che già dal primo elemento, l'array dinamico necessita di uno slot di storage aggiuntivo per la `length`, quindi comporta un costo di circa 20 000 in più fin dall'inizio.

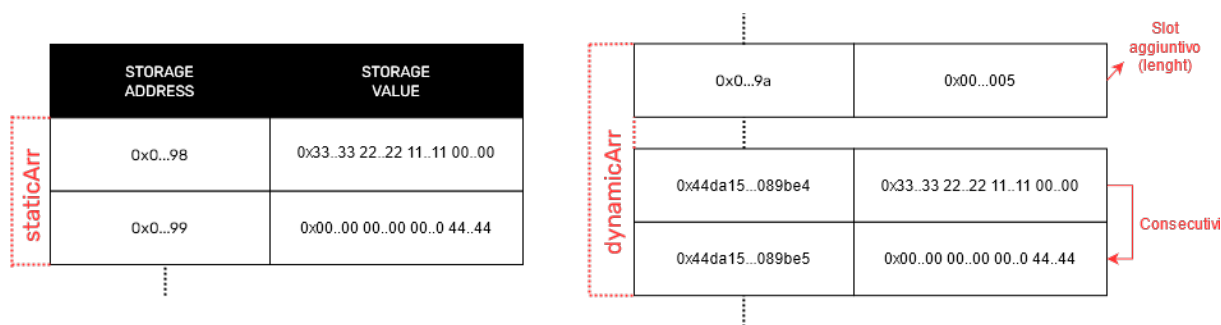


Figura 3.8: Figura esplicativa degli arrays a dimensione fissa e dinamica in riferimento al contratto ArraysDiff

Come si può vedere in Figura 3.9, si sono registrati i costi d'esecuzione nei vari casi in base al numero di elementi che man mano si andavano ad aggiungere agli arrays e si è calcolato anche il delta dal caso precedente con un elemento in meno. Si può vedere nei delta che l'aggiunta di ogni elemento in un array a dimensione fissa è molto basso, circa 30 unità di gas, mentre per l'aggiunta in un array dinamico è di un ordine di grandezza maggiore, circa 564 gas, principalmente dovuto ai controlli aggiuntivi sui limiti dell'array.

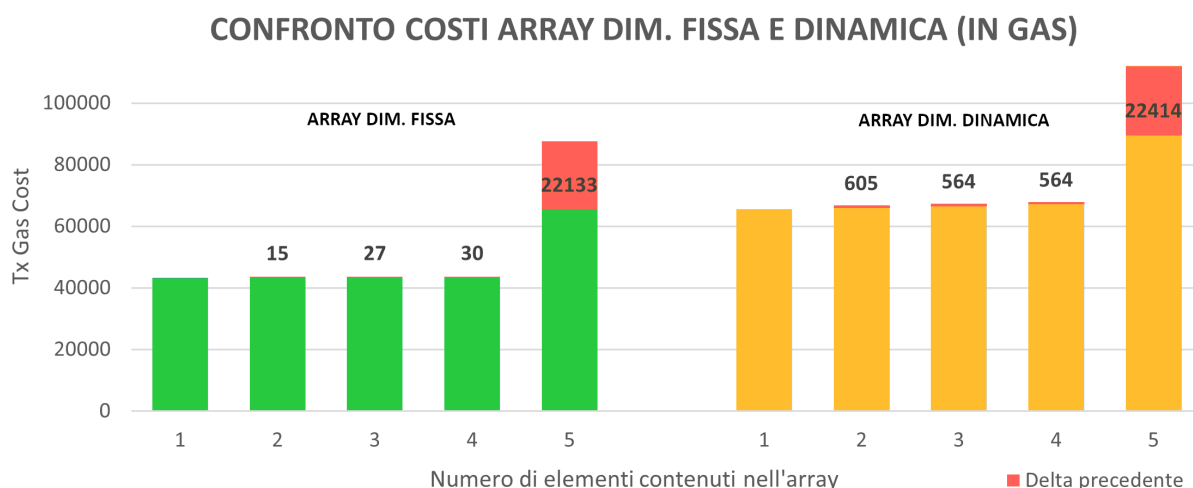


Figura 3.9: Confronto dei costi per arrays a dimensione fissa e dinamica

Quindi, se si volesse adottare una politica di minimizzazione dei costi aggressiva, il dimensionamento statico è la strada da percorrere. Tuttavia, è raccomandato il dimensionamento dinamico per maggiore flessibilità, chiarezza nella programmazione e minor rischi per la sicurezza dovuti a casi limite.

3.9 Chiarimenti riguardo le funzioni

Alcuni chiarimenti riguardo alle funzioni dalla documentazione di Solidity [24].

Le funzioni possono essere dei seguenti tipi:

- Tipo **view**: funzioni che non alterano lo stato dello storage, ossia che non possono:
 - Modificare le variabili di stato.
 - Attivare gli eventi.
 - Creare altri contratti.
 - Attivare l'autodistruzione del contratto.
 - Inviare Ether attraverso le chiamate.
 - Chiamare qualsiasi altra funzione che non abbia il modificatore **view** o **pure**.
 - Usare le chiamate di basso livello.
 - Usare l'Inline Assembly contenente alcuni opcodes speciali.
- Tipo **pure**: funzioni che non leggono nemmeno lo stato dello storage (più restrittivo), ossia che non possono:
 - Leggere dalle variabili di stato.
 - Accedere al saldo di un address.
 - Accedere a ogni dettaglio del blocco, della transazione, o del **msg** (a eccezione del **msg.sig** e del **msg.data**).
 - Chiamare qualsiasi altra funzione che non abbia il modificatore **pure**.
 - Usare l'Inline Assembly contenente alcuni opcodes speciali.
- Tipo **payable**: funzioni che accettano Ether, nel caso in cui non ci fosse l'attributo, la funzione rifiuterebbe l'Ether inviato a essa.
- Tipo non specificato: funzioni che permettono anche di alterare lo stato dello storage.

3.9.1 Visibilità di funzione

Esistono quattro tipi di visibilità possibili per le funzioni:

- La visibilità **private** permette solo chiamate interne dallo specifico contratto, non dai contratti derivati da quello.

- La visibilità `internal` abilita la funzione a chiamate dallo stesso contratto o da contratti derivati da quello in cui è presente. Questo tipo di chiamate sono le più economiche, sono gestite molto efficientemente dall'EVM visto che vengono tradotte in semplici jumps all'interno dell'EVM senza il bisogno di resettare la memoria corrente che può essere usata per passare alla funzione chiamata dei riferimenti a dati già presenti in memoria. Anche le chiamate ricorsive sono chiamate interne, tuttavia bisogna evitare l'eccessiva ricorsione dato che per ogni volta viene occupato uno slot nello stack.
- La visibilità `public` permette a chiunque di effettuare la chiamata alla funzione.
- La visibilità `external` permette alla funzione di essere chiamata esternamente da altri contratti ma mai da funzioni interne allo stesso contratto in cui risiede.

3.9.2 Modificatori di funzione

Esistono anche i modificatori di funzione utili per cambiare il comportamento delle funzioni in modo dichiarativo. Ad esempio, è possibile utilizzare un modificatore per verificare automaticamente una condizione prima di eseguire la funzione, come nel caso in cui solo alcuni addresses con determinati ruoli possano o meno eseguire quella determinata funzione.

Nella definizione di un modificatore, il corpo della funzione principale viene inserito dove compare il simbolo speciale `_`; e inoltre questo simbolo può essere inserito in qualsiasi posizione del modificatore.

```

1 modifier onlyOwner() {
2     require(msg.sender == owner, "Only owner!");
3     _;
4 }
```

Listing 3.46: Modificatore di funzione

3.9.3 Chiamate di funzione

I dati d'input per una chiamata di funzione devono essere nel formato definito dall'interfaccia ABI. Inoltre, la specifica ABI richiede che gli argomenti di funzione siano riempiti a multipli di 32 byte. Le chiamate di funzione interne utilizzano una convenzione diversa.

A partire dalla versione 0.5.0 di Solidity bisogna necessariamente specificare se i parametri di una funzione sono di tipo `storage`, `memory` o `calldata` (area non modificabile, non persistente che si comporta come la memoria e che salva gli argomenti di funzio-

ne). Per le chiamate a funzioni con modificatore `external`, i parametri devono essere obbligatoriamente in `calldata`.

3.9.4 Overloading

Un contratto può avere più funzioni con lo stesso nome ma con differenti tipi di parametri. Le funzioni sono selezionate facendo il matching dei parametri richiesti dalla chiamata di funzione e se tutti gli argomenti possono essere implicitamente convertiti nei tipi richiesti. Se non si trova esattamente un candidato, l'operazione fallisce.

3.9.5 Overriding

Si può eseguire l'override di funzioni marcate con `virtual` ed ereditate da un contratto nel caso in cui si voglia modificarne il comportamento. La funzione nel contratto figlio deve avere la parola chiave `override` nell'header della funzione. La visibilità della funzione può cambiare solo da `external` a `public`. Il tipo di funzione può essere cambiato in uno più stringente: `nonpayable` può essere ristretta a `view` e `pure`, `view` può essere ristretta a `pure`, `payable` è un'eccezione e non può essere modificata.

Per quanto riguarda le interfacce, tutte le funzioni sono automaticamente considerate virtuali e, per le funzioni senza implementazione al di fuori delle interfacce, è richiesto di specificare la parola chiave `override`. Altra cosa, le funzioni con visibilità `private` non possono essere marcate come virtuali.

Inoltre, come mostrato nel Listing 3.47 nel caso di ereditarietà multipla, i contratti devono essere esplicitamente specificati dopo la parola chiave `override`.

```
1 // SPDX-License-Identifier: MIT
2 pragma solidity >=0.8.0 <0.9.0;
3
4 contract Base1 {
5     function foo() public virtual {}
6 }
7
8 contract Base2 {
9     function foo() public virtual {}
10 }
11
12 contract Inherited is Base1, Base2 {
13     // Derives from multiple bases defining foo(), so we must explicitly
14     // override it
15     function foo() public override(Base1, Base2) {}
16 }
```

Listing 3.47: Multiple Inheritance Overriding

3.10 Analisi delle funzioni

In questa sezione verranno fatte delle analisi relative alle funzioni.

3.10.1 Visibilità delle funzioni

Nel seguente esempio sono mostrate tutte le possibili visibilità delle funzioni e le loro possibili chiamate di funzione:

```
1 // SPDX-License-Identifier: MIT
2 pragma solidity >=0.8.0 <0.9.0;
3
4 // Test Array: [5,5,5,5,5,5,5,5,5,5,5,5,5,5,5,5,5,5,5,5]
5
6 contract FunctionVisibility {
7     function fPublic(uint256[20] memory a) public returns (uint256) {
8         // Ho messo memory
9         return a[1] * 2;
10    }
11
12    function fExternal(uint256[20] calldata a) external returns (uint256) {
13        // Obbligatoriamente calldata
14        return a[1] * 2;
15    }
16
17    function fInternal(uint256[20] calldata a) internal returns (uint256) {
18        return a[1] * 2;
19    }
20
21    function fPrivate(uint256[20] calldata a) private returns (uint256) {
22        return a[1] * 2;
23    }
24
25    function call(uint256[20] calldata x) public virtual returns (uint256) {
26        fPublic(x);
27        this.fExternal(x); // Solo chiamata esternamente
28        fInternal(x);
29        fPrivate(x);
30    }
31 }
32
33 contract Contract1 {
34     FunctionVisibility c = new FunctionVisibility();
35
36     function call(uint256[20] calldata x) public returns (uint256) {
37         c.fPublic(x);
```

```

38     c.fExternal(x);
39     // c.fInternal(x); // Solo chiamata internamente dal contratto padre o
    da un contratto figlio
40     // c.fPrivate(x); // Solo chiamata internamente dal contratto padre
41 }
42
43 function callPub(uint256[20] calldata x) public returns (uint256) {
44     return c.fPublic(x);
45 }
46
47 function callExt(uint256[20] calldata x) public returns (uint256) {
48     return c.fExternal(x);
49 }
50 }
51
52 contract Contract2Child is FunctionVisibility {
53     // FunctionVisibility c = new FunctionVisibility();
54
55     function call(uint256[20] calldata x) public override returns (uint256) {
56         fPublic(x);
57         // fExternal(x); // Solo chiamate esternamente
58         fInternal(x);
59         // c.fPrivate(x); // Solo chiamata internamente dal contratto padre
60     }
61 }

```

Listing 3.48: Visibilità e chiamate di funzione

Rispetto alla visibilità **public**, le funzioni **external** sono più efficienti se si passano grandi arrays di dati come input dato che con modificatore **external** è necessario (obbligatoriamente) che i parametri siano di tipo `calldata`. Grazie a ciò Solidity può leggere direttamente le variabili da `calldata`, mentre nelle funzioni pubbliche con parametri in memoria Solidity deve obbligatoriamente copiare l'array in memoria consumando gas. La ragione per cui le funzioni pubbliche devono salvarsi tutti i parametri nella memoria è che le funzioni pubbliche possono essere chiamate internamente e che, per il loro funzionamento, sfruttano jumps all'interno della memoria per funzionare. Per questo motivo gli argomenti devono essere presenti in memoria nelle funzioni pubbliche mentre in quelle esterne, dato che non abilitano chiamate interne, non è necessario copiare gli argomenti in memoria ma possono essere letti direttamente da `calldata`. Attualmente anche le funzioni **public** possono avere parametri in `calldata` (non obbligatoriamente a differenza delle funzioni **external**).

Non ha senso usare il pattern `this.fExternal(x)` quando la funzione risiede nel proprio contratto perché richiede l'esecuzione di un'intera chiamata il che è molto dispendioso in termini di gas. L'unico motivo per cui si hanno benefici usando il modificatore **external**

è quando, da un contratto esterno, si chiama la funzione passandogli arrays di dimensione elevata come input: nell'esempio presente nel Listing 3.48 la chiamata dal contratto `Contract1` della funzione `callPub()` costa 31741 gas, mentre la chiamata dal contratto esterno `callExt()` costa 29927 gas a dimostrare quanto detto. Tuttavia, nella chiamata pubblica si è tenuto il parametro in memoria appositamente, se fosse stato salvato in calldata i due costi della chiamata pubblica ed esterna sarebbero stati simili.

Come best practice, bisogna usare il modificatore `external` se ci si aspetta che la funzione sia chiamata solo esternamente, mentre bisogna usare il modificatore `public` se può essere chiamata pure internamente.

Altra cosa, nel contratto `Contract1` non era possibile eseguire la funzione `fInternal(x)` perché il contratto `FunctionVisibility` non era stato derivato ma solo creato con l'operatore `new`; invece, nel contratto `Contract2Child` che ha ereditato dal padre è possibile eseguire la funzione `fInternal(x)` ma non è possibile eseguire la funzione `fExternal(x)` visto che si è ereditato il codice del padre.

3.10.2 Ordinamento delle funzioni (Method ID)

I nomi delle funzioni hanno un impatto sul consumo di gas [63]. Questo perché negli smart contracts c'è una differenza nell'ordine delle funzioni basato sul Method ID e non sull'ordine in cui vengono scritte.

Il Method ID è un valore che viene calcolato prendendo i primi 4 bytes dell'hash crittografico `keccak256(function_signature)`. La firma della funzione contiene sia il nome della funzione che il tipo dei parametri se presenti ed è in formato ASCII. Per ogni posizione successiva nell'ordine determinato dal Method ID si consumano 22 unità di gas in più. Ciò è ben visibile nel Listing 3.49: il primo contratto, contenente solo la funzione `b` consuma 98 gas, mentre nel secondo contratto mostrato nel Listing 3.50 consuma ben 142 unità di gas essendo in terza posizione il suo Method ID; notare anche che nel secondo contratto il Method ID non dipende dall'ordine alfabetico dei nomi delle funzioni.

```
1 // SPDX-License-Identifier: MIT
2 pragma solidity >=0.8.0 <0.9.0;
3
4 // Method ID:
5 // b: 0x4df7e3d0
6
7 contract FunctionsNames {
8     function b() public {}
9 }
```

Listing 3.49: Nomi delle funzioni 1

```

1 // SPDX-License-Identifier: MIT
2 pragma solidity >=0.8.0 <0.9.0;
3
4 // Method ID:
5 // a: 0x0dbe671f
6 // f: 0x26121ff0
7 // b: 0x4df7e3d0
8
9 contract FunctionsNames {
10     function a() public {}
11
12     function b() public {}
13
14     function f() public {}
15 }

```

Listing 3.50: Nomi delle funzioni 2

Inoltre, anche tutte le variabili pubbliche esterne dichiarate sono incluse in questo ordine, comprese le variabili e le funzioni ereditate da altri contratti. Per le variabili, il loro Method ID viene generato allo stesso modo trattandole come se fossero funzioni senza parametri, considerando quindi solo il loro nome e ignorando il loro tipo. Per quanto riguarda le variabili mapping, la chiave viene trattata come se fosse il primo parametro come mostrato nel Listing 3.51:

```

1 // Variabili
2 uint256 public a;
3 address public b;
4 mapping(address => uint256) public c;
5
6 // La loro function signature
7 a()
8 b()
9 c(address)
10
11 // Viene calcolato il loro Keccak-256 hash e presi solo i primi 4 byte
12 keccak256(function_signature)

```

Listing 3.51: Generazione del Method ID

Ovviamente, in caso di contratti piccoli, questo ordine non incide molto sui costi, a differenza di contratti con molte funzioni. In generale, per un'ottimizzazione spinta, è meglio avere le funzioni chiamate più spesso in alto nell'ordine del Method ID.

3.10.3 Compattare gli argomenti in input alle funzioni

Per quanto riguarda gli argomenti di funzione, se si hanno molti valori da passare, l'alternativa è usare le structs o l'encoding come fatto per lo storage.

3.10.4 Saturazione dello stack

Bisogna ricordare che esiste un limite agli argomenti di una funzione, nello stack è possibile accedere rapidamente ai soli 16 elementi in cima aventi dimensione di 32 byte. Inoltre, l'ultima variabile passata nella funzione sarà quella più in alto nello stack e se si volessero evitare errori del compilatore del tipo *Stack Too Deep* bisogna far sì che le ultime variabili passate alla funzione siano le prime usate nella logica e rimosse in seguito dalla cima dello stack per poter successivamente accedere alle altre.

Un modo per evitare di saturare lo stack è mostrato nel seguente esempio: si supponga di dover effettuare un calcolo con molti membri, se fatto tutto in un'istruzione si saturerebbe lo stack; l'alternativa è usare il *block scoping* come mostrato nel Listing 3.52 in cui l'operazione viene spezzata in più blocchi che generano risultati intermedi liberando la loro parte di stack.

```
1 // SPDX-License-Identifier: MIT
2 pragma solidity >=0.8.0 <0.9.0;
3
4 contract BlockScoping {
5     function functionStack(uint256 X1, uint256 X2, uint256 X3, uint256 X4,
6         uint256 X5, uint256 X6, uint256 X7, uint256 X8, uint256 X9
7     ) external pure returns (uint256 result) {
8         uint256 result = 0;
9
10        // result = X1+X2+X3+X4+X5+X6+X7+X8+X9; // CompilerError: Stack Too
11        Deep
12
13        // Block Scoping
14        {
15            result = X1 + X2 + X3 + X4 + X5;
16        }
17        {
18            result = result + X6 + X7 + X8 + X9;
19        }
20        return result;
21    }
```

Listing 3.52: Block scoping delle variabili

3.11 Analisi degli statements

In questa sezione verranno fatte delle analisi relative agli statements, in particolare ai cicli.

3.11.1 Cicli

L'EVM è una macchina di Turing completa, permette l'implementazione di cicli ma se possibile è meglio evitarli, soprattutto quelli illimitati o il cui limite superiore non è conosciuto. Infatti se non si conoscesse l'upper bound di un ciclo, non si saprebbe quante iterazioni sono richieste e quindi non si potrebbe stimare il costo d'esecuzione di una determinata funzione. Addirittura potrebbe finire tutto il gas a disposizione portando al fallimento della transazione.

Il prezzo del gas in Ether è tempo variabile, dipende anche dalla congestione della rete e da molti altri fattori, quindi se una determinata funzione ha un ciclo che fa un numero d'iterazioni con un costo ritenuto adeguato allo stato attuale, in futuro con l'aumento del costo del gas renderebbe il contratto non conveniente e lo renderebbe obsoleto.

Molto dipende dal numero e dalla complessità delle operazioni da eseguire a ogni iterazione, se invece di semplici operazioni ci fossero operazioni complesse dell'ordine $O(n^2)$, ogni iterazione sarebbe da pesare in modo molto maggiore.

Inoltre, bisognerebbe evitare d'iterare su un parametro variabile, come la lunghezza di un array dato che potrebbe crescere nel tempo portando il costo del gas alle stelle rendendo inusabile il contratto. Un esempio è fornito dal contratto presente nel Listing 3.53 in cui si cerca un determinato valore in un array e se non presente lo si aggiunge all'array.

```
1 // SPDX-License-Identifier: MIT
2 pragma solidity >=0.8.0 <0.9.0;
3
4 contract LoopFind {
5     uint256[] array = new uint256[](0); // Storage
6
7     function find(uint256 x) public returns (bool) {
8         bool find = false;
9
10        for (uint256 i = 0; i < array.length; i++) {
11            if (x == array[i]) return true;
12        }
13
14        array.push(x);
15
16        return false;
```

```

17 }
18 }

```

Listing 3.53: Ciclo for con limite superiore variabile

Eseguendo più volte il contratto, man mano che la dimensione dell'array cresce, il costo cresce altrettanto per le maggiori iterazioni richieste dal ciclo for rendendo inusabile il contratto. Nella Tabella 3.1 sono mostrati i costi di ricerca dell'ultimo valore dell'array man mano che la sua dimensione cresce (costo d'esecuzione della transazione tolto 21 204 gas dovuto alla transazione base).

Dimensione dell'array	Costo in gas
1	4 707 gas
2	7 230 gas
3	9 753 gas
4	12 276 gas
5	14 799 gas

Tabella 3.1: Costo per la ricerca dell'ultimo valore nell'array (dimensione crescente)

Vi è un incremento lineare costante di 2 523 unità di gas per ogni ulteriore elemento presente nell'array.

Un altro esempio è quello mostrato nel Listing 3.54 in cui sono mostrati due modi per fare la somma, la prima funzione `expensiveLoop` meno efficiente va a scrivere nello storage la variabile cumulativa (consumo di 24 230 gas per 5 iterazioni), l'altra funzione `lessExpensiveLoop` più efficiente usa una variabile in memoria temporanea (consumo di 6 202 gas per 5 iterazioni) visto che la scrittura in memoria è molto più economica della scrittura nello storage [61].

```

1 // SPDX-License-Identifier: MIT
2 pragma solidity >=0.8.0 <0.9.0;
3
4 contract Loop {
5     uint256 num = 0; // Storage
6
7     function expensiveLoop(uint256 x) public {
8         for (uint256 i = 0; i < x; i++) {
9             num += 1;
10        }
11    }
12
13    function lessExpensiveLoop(uint256 x) public {
14        uint256 temp = num;
15        for (uint256 i = 0; i < x; i++) {

```

```

16     temp += 1;
17 }
18 num = temp;
19 }
20 }

```

Listing 3.54: Cicli for con l'uso di una variabile in memoria temporanea

Un altro esempio è quello mostrato nel Listing 3.55 in cui si combinano più cicli for in un unico, risparmiando gas e calcolando nello stesso ciclo più operazioni. La prima funzione `dividedLoops` consuma 46 597 gas con 100 come parametro passato, l'altra funzione `combinedLoop` più efficiente consuma 34 843 gas con 100 come parametro passato riuscendo a combinare in un unico ciclo più operazioni [61].

```

1 // SPDX-License-Identifier: MIT
2 pragma solidity >=0.8.0 <0.9.0;
3
4 contract LoopCombined {
5     function dividedLoops(uint256 x) public {
6         uint256 m = 0;
7         int256 v = 0;
8
9         for (uint256 i = 0; i < x; i++) m += i;
10
11        for (uint256 j = 0; j < x; j++) v -= int256(j);
12    }
13
14    function combinedLoop(uint256 x) public {
15        uint256 m = 0;
16        int256 v = 0;
17
18        for (uint256 i = 0; i < x; i++) {
19            m += i;
20            v -= int256(i);
21        }
22    }
23 }

```

Listing 3.55: Cicli for combinati

Concludendo, se si dovessero usare per forza dei cicli, bisognerebbe farli iterare su parametri non variabili (il cui limite superiore è prevedibile) e le cui iterazioni siano ampiamente considerate non troppo costose al tempo di stesura del codice.

3.11.2 Valutazione a corto circuito

Agli operatori logici `||` e `&&` in Solidity viene applicata la valutazione a corto circuito, ciò significa che il secondo operando viene valutato unicamente se il valore del primo operando non è sufficiente da solo a determinare il risultato dell'espressione.

```
1 // SPDX-License-Identifier: MIT
2 pragma solidity >=0.8.0 <0.9.0;
3
4 contract ShortCircuit {
5     function f1(bool b) public pure returns (bool) {
6         // Computazione leggera
7         return b;
8     }
9
10    function f2() public view returns (bool) {
11        // Computazione pesante
12        uint256 nonce = 0;
13
14        uint256 randomnumber = uint256(keccak256(abi.encodePacked(block.
15            timestamp, msg.sender, nonce))) % 6;
16        nonce++;
17
18        if (randomnumber >= 3) return true;
19        else return false;
20    }
21
22    function compare() public view returns (uint256) {
23        uint256 result = 0;
24
25        // Corto circuito sempre (144 gas)
26        if (f1(true) || f2()) result = 1;
27
28        // Corto circuito se f2()=true (665 gas), entrambe (671 gas)
29        if (f2() || f1(true)) result = 2;
30
31        // Corto circuito sempre (144 gas)
32        if (f1(false) && f2()) result = 3;
33
34        // Corto circuito se f2()=true (664 gas), entrambe (668 gas)
35        if (f2() && f1(false)) result = 4;
36
37        return result;
38    }
39 }
```

Listing 3.56: Valutazione a corto circuito

Nel Listing 3.56 sono usate due funzioni di appoggio e una funzione `compare` che mostra, con i relativi costi di gas commentati, l'esecuzione di uno statement `if` su due operandi usando gli operatori logici.

Si può notare che Solidity adotta la valutazione a corto circuito, motivo per cui è più efficiente mettere al primo operando la funzione meno dispendiosa, in modo tale da evitare la valutazione del secondo operando se non necessario.

Nel caso in cui le due funzioni hanno un costo di esecuzione simile, si può procedere nel seguente modo:

- Se `f1()` ha una probabilità considerevolmente maggiore di ritornare falso rispetto alla funzione `f2()`, mettere `f1() && f2()` potrebbe consumare meno gas grazie alla valutazione a corto circuito.
- Se `f1()` ha una probabilità considerevolmente maggiore di ritornare vero rispetto alla funzione `f2()`, mettere `f1() || f2()` potrebbe consumare meno gas grazie alla valutazione a corto circuito.

3.12 Analisi degli eventi ed errori

In questa sezione verranno fatte delle analisi relative agli eventi e alla gestione degli errori.

3.12.1 Eventi

Gli eventi forniscono un'astrazione per il logging dell'EVM, permettendo l'iscrizione e l'ascolto di questi eventi tramite l'interfaccia RPC di un client Ethereum. Quando l'evento viene attivato, gli argomenti vengono archiviati (attualmente per sempre) nel *transaction's log*, una speciale struttura di dati all'interno del blocco attuale della blockchain ma inaccessibili dal codice.

È possibile aggiungere l'attributo `indexed` fino a un massimo di tre parametri in modo tale da aggiungere questi parametri a una struttura dati speciale nota come *topics* invece della parte dati del log. Grazie a ciò è possibile cercare eventi per topics, ad esempio quando si filtra una sequenza di blocchi per determinati eventi oppure anche per filtrare gli eventi in base all'indirizzo del contratto che ha emesso l'evento.

Gli eventi con attributo `anonymous` sono più economici da distribuire e chiamare dato che l'hash della firma dell'evento non viene salvata nel blocco. Infatti non sarà nemmeno possibile filtrare eventi anonimi specifici in base al nome ma solo in base all'indirizzo del contratto.

Usare gli eventi per salvare dati che non sono richiesti on-chain è un exploit che è possibile fare, ricordando però che non sarà possibile accedervi dal codice. Se si hanno dati che devono essere creati una sola volta e non necessiteranno di essere aggiornati o letti dallo smart contract è meglio usare gli eventi che consumano meno gas rispetto alle normali variabili di Solidity.

Si scoraggia invece l'uso eccessivo di eventi per il logging in quanto gli opcodes per il logging consuma comunque gas e il loro costo aumenta all'aumentare dei loro parametri. Si può verificare con il seguente contratto:

```
1 // SPDX-License-Identifier: MIT
2 pragma solidity >=0.8.0 <0.9.0;
3
4 contract EventsLogging {
5     event Result(uint256 c);
6
7     function noEvent(uint256 a, uint256 b) public pure returns (uint256 c) {
8         c = a + b;
9         return c;
10    }
```

```

11
12 function hasEvent(uint256 a, uint256 b) public returns (uint256 c) {
13     c = a + b;
14     emit Result(c);
15     return c;
16 }
17 }

```

Listing 3.57: Eventi per il logging

Per avere un'idea, una chiamata alla funzione `noEvent(2,2)` costa 352 gas per esecuzione, mentre la funzione con eventi `hasEvent(2,2)` con gli stessi parametri costa 1 443 gas, differenza di 1 091 da attribuire all'emissione dell'evento.

3.12.2 Errori

Attualmente Solidity supporta due tipi di errore: `Error(string)` che è usato per le condizioni di errore di regolare utilizzo mentre `Panic(uint256)` è usato per gli errori che non dovrebbero essere presenti in un codice *bug-free*.

Per gestire gli errori in Solidity si usano le seguenti due funzioni che lanciano delle eccezioni che annullano tutte le modifiche apportate allo stato nella chiamata corrente e nelle sue sottochiamate:

- Panic via `assert`: usata solo per verificare la presenza di errori interni e controllare le invarianti. Un codice non dovrebbe mai fallire un assert (valutarli nella fase di testing), ciò indica che vi è un bug nel contratto.
- Error via `require`: usata per assicurarsi che le condizioni che non sono verificabili prima dell'esecuzione siano valide; viene usato sugli input, sulle variabili di stato del contratto o per convalidare i valori restituiti dalle chiamate a contratti esterni.

Le eccezioni lanciate con l'assert consumano tutto il gas disponibile in una chiamata, mentre le eccezioni lanciate dal require, a partire dall'hard fork Metropolis di Ethereum, non consumano nessuna unità di gas. Questo si può vedere dal seguente codice:

```

1 // SPDX-License-Identifier: MIT
2 pragma solidity >=0.8.0 <0.9.0;
3
4 contract AssertRequire {
5     function isAssert(uint256 a, uint256 b) public pure returns (uint256 c) {
6         assert(a + b != 0);
7         c = a + b;
8     }
9 }

```



```

10 function isRequire(uint256 a, uint256 b) public pure returns (uint256 c)
    {
11     require(a + b != 0, "I due numeri sommati fanno zero");
12     c = a + b;
13 }
14 }

```

Listing 3.58: Eccezioni con assert e require

Tuttavia, se possibile e senza sacrificare la chiarezza del messaggio d'errore, bisognerebbe cercare di ridurre la stringa del messaggio dato che occupa spazio nel bytecode a multipli di 32 byte. Notare anche che è possibile fornire una stringa come messaggi per `require` ma non per `assert`.

Per fare il reverting della chiamata corrente, è possibile utilizzare la funzione `revert` di Solidity per generare eccezioni per la visualizzazione dell'errore. Questa funzione creerà un'eccezione `Error(string)` che opzionalmente accetta un messaggio di stringa contenente i dettagli sull'errore.

Tutte le eccezioni di questi tre tipi lanciate durante la creazione del contratto e le chiamate esterne possono essere catturate dal `try/catch` statement e gestite nel modo migliore.

3.13 Inline Assembly

L'EVM è una macchina a stack. Lo stack è una struttura dati dove è possibile solo aggiungere (**PUSH**) e rimuovere (**POP**) valori dalla cima. Vigè una politica *Last In, First Out* (LIFO), l'ultimo elemento aggiunto allo stack è anche il primo che verrà rimosso. Tutti gli operandi dei vari opcodes vengono salvati nello stack e le varie istruzioni solitamente rimuovono uno o più valori dallo stack, fanno una determinata computazione e pushano il risultato nello stack: questo ordine è chiamato *Reverse Polish Notation* come mostrato nel Listing 3.59.

```
1 a + b      // Standard Notation (Infix)
2 a b add    // Reverse Polish Notation
```

Listing 3.59: Notazione standard e polacca inversa

Nel caso si voglia gestire nei minimi dettagli le operazioni che verranno svolte dall'EVM a basso livello, è possibile usare l'Inline Assembly introdotto precedentemente nella sotto-sezione 1.3.1, riducendo ancor di più il consumo di gas dovuto all'esecuzione del contratto molto utile nello sviluppo di librerie che verranno eseguite molte volte. Nel Listing 3.60 è mostrato come inserire un blocco di codice Inline Assembly in un contratto Solidity.

```
1 contract Assembler {
2     function test() public {
3         // Solidity code
4         uint256 a;
5
6         assembly {
7             // Assembly code
8         }
9     }
10 }
```

Listing 3.60: Blocco di Inline Assembly

Il codice all'interno di un blocco Inline Assembly è scritto nel linguaggio Yul [64] definito nella documentazione di Solidity.

Blocchi di Inline Assembly differenti non condividono lo stesso namespace, quindi non è possibile chiamare una funzione Yul o accedere a una variabile Yul definita in un altro blocco.

Esistono due stili per scrivere l'Assembly:

- **Functional Style Assembly**: stile funzionale che rende facile vedere quale operando è usato per un determinato opcode. In Solidity viene usato questo stile.

```

1 // Functional Style Assembly
2 mstore(0x80, add(mload(0x80), 3))

```

Listing 3.61: Functional Style Assembly

- Non-Functional Style Assembly: stile che offre una maggiore visione dello stack, permette di vedere dove i valori finiscono nello stack. Fornisce un quadro completo rispetto all'ordine di esecuzione eseguito dall'EVM.

```

1 // Non-Functional Style Assembly
2 3 0x80 mload add 0x80 mstore

```

Listing 3.62: Non-Functional Style Assembly

La versione funzionale di un codice Assembly, come mostrato dall'equivalenza dei due listati sopra che semplicemente aggiungono il valore 3 al contenuto presente all'indirizzo di memoria `0x80`, non è altro che la notazione inversa della versione non funzionale.

```

1 // Layout dello stack dopo ogni istruzione
2
3 empty    PUSH 3    PUSH 0x80    MLOAD      ADD      PUSH 0x80    MSTORE
4
5          |_0x80| > |__5__|          |_0x80|
6 |_____| > |__3__| > |__3__| > |__3__| > |__8__| > |__8__| > |_____|

```

Listing 3.63: Layout dello stack dopo ogni istruzione

Inoltre, per efficienza, l'Assembly tratta ogni valore come se fosse un numero da 256 bit, ignorando completamente il fatto che possano esistere tipi di lunghezza inferiore. I bits di ordine superiore sono azzerati solo quando è necessario, ad esempio prima di effettuare un confronto o scrivere nella memoria, per il resto, se si vuole accedere a una determinata variabile all'interno dell'Inline Assembly, si potrebbe dover azzerare i bits di ordine superiore manualmente.

3.13.1 Manipolazione dello stack

La manipolazione dello stack può essere effettuata soltanto dall'Inline Assembly. Tuttavia, visto il basso costo per le operazioni nello stack, non conviene operare a un così basso livello per manipolare lo stack con il fine di ridurre i costi.

3.13.2 Manipolazione della memoria

La manipolazione della memoria può essere usata ad esempio per accedere a blocchi di memoria specifici non accessibili normalmente, ad esempio nello scratch space (`0x00 - 0x3f`) usato durante l'hashing. Nel Listing 3.64 viene mostrato un esempio dell'Inline

Assembly che calcola il Keccak256 hash in memoria per poi salvare il valore 3 sullo storage all'indirizzo calcolato.

```
1 // SPDX-License-Identifier: MIT
2 pragma solidity >=0.8.0 <0.9.0;
3
4 contract AssemblyScratchSpace {
5     function getStorageValue(uint256 key, uint256 pslot) public returns (
6         uint256 result) {
7         assembly {
8             // Salvare la key nello scratch space
9             mstore(0, key)
10            // Salvare il pslot nello scratch space dopo la key
11            mstore(32, pslot)
12            // Creare l'hash dei due valori precedenti
13            let hash := keccak256(0, 64)
14
15            // Salvare un value all'indirizzo hash del mapping
16            sstore(hash, 3)
17            // Caricare un mapping value usando l'hash calcolato
18            result := sload(hash)
19        }
20    }
21 }
```

Listing 3.64: Accedere allo scratch space grazie all'Inline Assembly

Inoltre, grazie all'Inline Assembly è possibile leggere intere words da 256 bits da tipi di dato come `bytes` e `string` in una singola operazione. La libreria *solidity-stringutils*¹⁰ sfrutta questa funzionalità per fare confronti veloci tra stringhe facendo la sottrazione su chunks di 32 bytes (invece che di singoli byte senza l'Assembly) delle due stringhe da comparare. Nel Listing 3.65 è riportato uno snippet.

```
1 // Slice struct
2 struct slice {
3     uint256 _len;
4     uint256 _ptr;
5 }
6
7 /*
8  * @dev Returns a positive number if `other` comes lexicographically after
9  *       `self`, a negative number if it comes before, or zero if the
10  *       contents of the two slices are equal. Comparison is done per-rune,
11  *       on unicode codepoints.
12  * @param self The first slice to compare.
```

¹⁰Libreria *solidity-stringutils* (<https://github.com/Arachnid/solidity-stringutils>)

```

13 * @param other The second slice to compare.
14 * @return      The result of the comparison.
15 */
16 function compare(slice memory self, slice memory other) internal pure
    returns (int256) {
17     uint256 shortest = self._len;
18     if (other._len < self._len) shortest = other._len;
19
20     uint256 selfptr = self._ptr;
21     uint256 otherptr = other._ptr;
22     for (uint256 idx = 0; idx < shortest; idx += 32) {
23         uint256 a;
24         uint256 b;
25         assembly {
26             a := mload(selfptr)
27             b := mload(otherptr)
28         }
29         if (a != b) {
30             // Mask out irrelevant bytes and check again
31             uint256 mask = uint256(-1); // 0xffff...
32             if (shortest < 32) {
33                 mask = ~(2**(8 * (32 - shortest + idx)) - 1);
34             }
35             uint256 diff = (a & mask) - (b & mask);
36             if (diff != 0) return int256(diff);
37         }
38         selfptr += 32;
39         otherptr += 32;
40     }
41     return int256(self._len) - int256(other._len);
42 }

```

Listing 3.65: Snippet relativo al confronto tra stringhe effettuato dalla libreria *solidity-stringutils*

3.13.3 Manipolazione dello storage

La manipolazione dello storage può essere usata per accedere a una qualsiasi posizione nello storage senza il bisogno di doverlo espandere. Come mostrato nel Listing 3.66, è possibile scrivere in una posizione arbitraria dello storage oppure usando il suffisso `.slot` per accedere all'area di storage puntata da un puntatore.

```

1 // SPDX-License-Identifier: MIT
2 pragma solidity >=0.8.0 <0.9.0;
3
4 contract AssemblySlot {

```

```

5  uint256 uintStorage = 1337;
6
7  function getStorageByPointer() external view returns (uint256) {
8      uint256 _foundStorage;
9
10     assembly {
11         // Usare '.slot' per accedere a ogni variabile del contratto
12         _foundStorage := sload(uintStorage.slot)
13     }
14
15     return _foundStorage;
16 }
17
18 function setStorageByPointer(uint256 _newNumber) external {
19     assembly {
20         // Usare '.slot' per salvare nello storage nella posizione puntata
21         // dal puntatore
22         sstore(uintStorage.slot, _newNumber)
23
24         // Scrivere in una posizione arbitraria
25         sstore(0xc0fefe, 0x42)
26     }
27 }

```

Listing 3.66: Accedere a una posizione arbitraria nello storage con l'Inline Assembly

3.13.4 Migliorare l'efficienza del compilatore

L'Inline Assembly può essere utile in alcuni casi dove l'ottimizzatore di Solidity fallisce a produrre una versione del codice ottimizzata. Un esempio è mostrato nel Listing 3.67 in cui viene sviluppata una libreria che ha lo scopo di fare la somma, attraverso un ciclo for, di tutti gli elementi di un vettore passato come parametro.

```

1  // SPDX-License-Identifier: MIT
2  pragma solidity >=0.8.0 <0.9.0;
3
4  // Test Array: [1,2,3,4,5]
5
6  library AssemblyVectorSum {
7      // Funzione meno efficiente perché l'ottimizzatore fallisce a rimuovere i
7      // bounds checks sull'accesso all'array
8      function sumSolidity(uint256[] memory _data) public pure returns (uint256
8      sum) {
9          for (uint256 i = 0; i < _data.length; ++i) sum += _data[i];
10     }

```

```

11
12 // Si è certi che si rimanga nei bounds dell'array, quindi si possono
    evitare i check
13 // Il valore 0x20 deve essere aggiunto all'array perché il primo slot
    contiene la length
14 function sumAssembly(uint256[] memory _data) public pure returns (uint256
    sum) {
15     for (uint256 i = 0; i < _data.length; ++i) {
16         assembly {
17             sum := add(sum, mload(add(add(_data, 0x20), mul(i, 0x20))))
18         }
19     }
20 }
21
22 // Tutto il codice scritto in Inline Assembly
23 function sumPureAssembly(uint256[] memory _data) public pure returns (
    uint256 sum) {
24     assembly {
25         let len := mload(_data)
26
27         let data := add(_data, 0x20)
28
29         // Iterare fino a che non si incontra il bound finale
30         for {
31             let end := add(data, mul(len, 0x20))
32             } lt(data, end) {
33                 data := add(data, 0x20)
34             } {
35                 sum := add(sum, mload(data))
36             }
37         }
38     }
39 }

```

Listing 3.67: Somma degli elementi di un vettore con l'Inline Assembly

Verifichiamo il costo delle tre funzioni passandogli come parametro il vettore `[1,2,3,4,5]`. La funzione `sumSolidity()` scritta in Solidity costa 2254 gas. La funzione `sumAssembly()`, con parte del codice in Assembly, costa 1826 gas. La funzione `sumPureAssembly()`, con l'intero codice in Assembly, costa 1443 gas. Come si può notare, il risparmio è notevole.

Capitolo 4

Sviluppo della DApp NotarizETH

In questa seconda parte del progetto di tesi verrà trattato lo sviluppo dell'applicazione decentralizzata NotarizETH [23] disponibile all'indirizzo <https://notarizeth.xyz>.



Figura 4.1: Logo di NotarizETH

Dopo tutta la prima parte riguardante l'analisi dei costi nell'EVM con il linguaggio Solidity, volevo concludere la tesi sviluppando un'applicazione decentralizzata da zero, a partire dal deploy del contratto sulla blockchain (sul Ropsten Network [27], una testnet di Ethereum) fino al frontend. In questo modo sono entrato in contatto con molti tools e tecnologie interessanti tra le quali: React [28], Infura [29], Metamask [30], Amazon S3 [31], GitHub Actions [32] e molte altre relative a frameworks come useDApp [33] o standard di smart contracts per la gestione dei permessi forniti da OpenZeppelin [34].

L'applicazione vuole fornire un servizio di certificazione di documenti digitali sulla blockchain salvando, grazie a uno smart contract, alcune informazioni tra cui l'hash del file in modo da garantire l'integrità e la *Proof of Existence* del documento in un dato momento, permettendo a chiunque la verifica tramite un'interfaccia Web.

Tutto il materiale, compreso lo smart contract usato e il codice dell'applicazione sviluppata in React, sono disponibili nel repository di GitHub [48].

4.1 Smart Contract

Lo smart contract di NotarizETH è stato scritto in Solidity e in seguito deployato sul Ropsten Network¹.

Si andrà ora ad analizzare il corpo centrale del contratto, descrivendo brevemente in seguito la gestione dei permessi ottenuta grazie al supporto di contratti forniti da OpenZeppelin [34] come `AccessControl.sol`.

```
1 // SPDX-License-Identifier: MIT
2 pragma solidity >=0.8.0 <0.9.0;
3
4 import "../AccessControl.sol";
5
6 contract NotarizETH is AccessControl {
7     // Structured data that describes a file certificate
8     struct Certificate {
9         bool exist;
10        address author;
11        uint80 timestamp;
12        bytes32 filehash;
13    }
14
15    // Event that will be fired when a file is certified
16    event FileCertified(address author, uint256 timestamp, bytes32 filehash);
17
18    // Object that will store the file certificates by hash
19    mapping(bytes32 => Certificate) records;
20
21    // Constructor
22    constructor() {
23        // Who creates the contract is admin
24        _setupRole(DEFAULT_ADMIN_ROLE, msg.sender);
25    }
26
27    // Function that allows users to verify if a file has been certified
28    // before
29    function verifyFile(bytes32 filehash)
30        public
31        view
32        returns (
33            bool,
34            address,
```

¹Contract (<https://ropsten.etherscan.io/address/0x39B1704688cBBE66A9bdd88A06eaf3A496d132F6>)

```

35     bytes32
36 )
37 {
38     Certificate memory record = records[filehash];
39
40     if (record.exist == true) {
41         return (true, record.author, record.timestamp, record.filehash);
42     } else {
43         return (false, address(0x0), uint80(0), bytes32(0));
44     }
45 }
46
47 // Function that allows users to certify a file
48 function certifyFile(bytes32 filehash) public {
49     // Control that the file hash not exist yet
50     require(records[filehash].exist != true, "Certificate with given file
51     hash already exists");
52
53     records[filehash] = Certificate(true, msg.sender, uint80(block.
54     timestamp), filehash);
55
56     // Permissions to modify the file data to the actual sender
57     grantRoleHash(filehash, msg.sender);
58
59     emit FileCertified(msg.sender, uint80(block.timestamp), filehash);
60 }
61
62 // Function that reset a file certificate from records
63 function resetFile(bytes32 filehash) public onlyRole(filehash) {
64     revokeRoleHash(filehash, records[filehash].author);
65
66     delete records[filehash];
67 }
68
69 // Return true if the account belongs to the admin role
70 function isAdmin(address account) public view virtual returns (bool) {
71     return hasRole(DEFAULT_ADMIN_ROLE, account);
72 }
73
74 // Grant the permissions to modify the file data to the actual sender
75 function grantRoleHash(bytes32 role, address account) private {
76     require(account == _msgSender(), "AccessControl: can only grant file
77     hash role for self");
78     _grantRole(role, account);
79 }
80
81 // Revoke the permissions to modify the file data to the actual sender

```

```

79 function revokeRoleHash(bytes32 role, address account) private {
80     require(account == _msgSender(), "AccessControl: can only revoke file
      hash role for self");
81     _revokeRole(role, account);
82 }
83
84 // 'Removes' the contract code from the blockchain and invalidates all
      signatures
85 function removeContract() public onlyRole(DEFAULT_ADMIN_ROLE) {
86     selfdestruct(payable(msg.sender));
87 }
88 }

```

Listing 4.1: Smart Contract dell'applicazione NotarizETH

4.1.1 Struttura principale

Si è scelto di usare il mapping `records` per salvare, per ogni file hash, un certificato contenente informazioni riguardanti il file. Il mapping `records` ha questa struttura:

```

1 // Object that will store the file certificates by hash
2 mapping(bytes32 => Certificate) records;

```

Nella struct `Certificate` vengono salvati un booleano che indica se quel determinato file hash del mapping è stato inserito o meno, l'address del mittente della transazione ottenuto con l'istruzione `msg.sender`, l'hash del file salvato in `bytes32` in modo tale da risparmiare spazio e il timestamp dell'inserimento del file hash nella blockchain ottenuto con l'istruzione `block.timestamp`.

```

1 // Structured data that describes a file certificate
2 struct Certificate {
3     bool exist;
4     address author;
5     uint80 timestamp;
6     bytes32 filehash;
7 }

```

In aggiunta è stato inserito anche un evento `FileCertified` che viene emesso a ogni inserimento di un file hash.

```

1 // Event that will be fired when a file is certified
2 event FileCertified(address author, uint256 timestamp, bytes32 filehash);

```

La funzione `verifyFile` permette di verificare se un dato file hash è presente o meno all'interno del mapping dello smart contract. In caso affermativo verranno restituite come output tutte le informazioni contenute nel certificato.

La funzione `certifyFile` permette l'inserimento di un file hash nel mapping. Viene eseguito un controllo con `require` sul fatto che non esista già quel determinato file hash, in caso sia già presente quell'hash viene restituito un errore. Inoltre, viene emesso l'evento `FileCertified`, utile alle applicazioni che si iscrivono e sono in ascolto sugli eventi.

La funzione `resetFile` permette di rimuovere informazioni relative ai propri file hash precedentemente inseriti nello smart contract.

```
1 // Function that reset a file certificate from records (Modifier onlyRole)
2 function resetFile(bytes32 filehash) public onlyRole(filehash) {
3     revokeRoleHash(filehash, records[filehash].author);
4
5     delete records[filehash];
6 }
```

4.1.2 Controllo dell'accesso

A partire dal contratto `AccessControl.sol` di OpenZeppelin [34] è stato implementato un controllo dell'accesso *role-based*. Lo schema scelto è piatto, quando un utente inserisce un contratto attraverso la funzione `certifyFile` gli viene attribuito al proprio address pubblico un nuovo ruolo corrispondente all'hash del documento inserito (grazie alla funzione `grantRoleHash`).

```
1 // Grant the permissions to modify the file data to the actual sender
2 function grantRoleHash(bytes32 role, address account) private {
3     require(account == _msgSender(), "AccessControl: can only grant file hash
4         role for self");
5     _grantRole(role, account);
6 }
```

In questo modo esisteranno tanti ruoli quanti file hash sono stati inseriti nel mapping. Ogni utente avrà il permesso di chiamare la funzione `resetFile` per rimuovere un proprio file hash dal mapping, resettando tutte le informazioni del certificato e perdendo il ruolo associato a quell'hash con la funzione `revokeRoleHash`.

```
1 // Revoke the permissions to modify the file data to the actual sender
2 function revokeRoleHash(bytes32 role, address account) private {
3     require(account == _msgSender(), "AccessControl: can only revoke file
4         hash role for self");
5     _revokeRole(role, account);
6 }
```

}

Sulla funzione `resetFile` viene esercitato un modificatore `onlyRole(file_hash)` che restringe l'operazione solo sui propri file hash precedentemente inseriti.

```
modifier onlyRole(bytes32 role) {
    _checkRole(role, _msgSender());
    -;
}
```

Un unico ruolo speciale è quello `DEFAULT_ADMIN_ROLE` dato al creatore del contratto, colui che l'ha deployato sulla blockchain, e che gli permette di autodistruggere lo smart contract attraverso la funzione `removeContract` annullando tutti i valori salvati nello stato del contratto. Tuttavia, il codice del contratto continuerà a essere parte della storia passata della blockchain.

```
// 'Removes' the contract code from the blockchain and invalidates all
// signatures
function removeContract() public onlyRole(DEFAULT_ADMIN_ROLE) {
    selfdestruct(payable(msg.sender));
}
```

4.1.3 Analisi del costo del gas

Alla luce di quanto visto nelle analisi del Capitolo 3, si è cercato di ridurre al minimo il costo del gas per l'esecuzione dello smart contract.

Nella struct `Certificate` si è cercato di compattare il più possibile le variabili in essa contenute come si può vedere dal Listing 4.2.

```
key: '0x478ab6444ff92f28c1a6cefa2e7caf3aad10ce95c45c8276321141ba18abc570 '  
val: '0x000000000000060cdb4c55b38da6a701c568545dcfcb03fcb875f56beddc401 '  
  
key: '0x478ab6444ff92f28c1a6cefa2e7caf3aad10ce95c45c8276321141ba18abc571 '  
val: '0x9aaaae0aa0255710c9d4f5e7c74401bb44907bde7e26847e5b19a48f9ce86af98 '
```

Listing 4.2: Variabili della struct `Certificate` nello storage

Ogni certificato occupa due slot interi di storage grazie alla compattazione delle variabili che sono anche disposte nel giusto ordine. Sapendo che per ogni valore esadecimale corrispondono quattro bits in binario, nel primo slot di storage vengono salvati:

- `bool exist` (in verde): occupa 8 bits essendo il tipo `bool` equivalente, in Solidity, a un tipo `uint8`.
- `address author` (in giallo): occupa 160 bit.

- `uint80 timestamp` (in blu): occupa 80 bit, la variabile `block.timestamp` inizialmente è di tipo `uint256`, tuttavia è possibile convertirla a `uint80` senza perdere informazione relativa al timestamp.

Nel secondo slot viene salvato soltanto:

- `bytes32 filehash` (in marrone): occupa tutti i 256 bits dello slot essendo il Keccak256 hash del file; l'array, essendo sempre di 32 bytes, è di dimensione fissa e inoltre non necessita spazio aggiuntivo per salvarne la lunghezza.

Il mapping `records` è una soluzione ottima sia in termini di gas che per velocità nella ricerca di valori come mostrato nel sezione 3.7.

Gli eventi come `FileCertified` sono molto economici, motivo per cui si è scelto di tenerli fornendo un'alternativa alle applicazioni che vorranno iscriversi e rimanere all'ascolto sugli eventi emessi dallo smart contract a ogni certificazione.

Inoltre, sempre in ottica di ottimizzazione, all'interno di `AccessControl.sol` sono state usate anche alcune librerie tra le quali `String.sol` per la gestione delle stringhe.

Ultima cosa, nel cercare di ridurre al minimo la computazione on-chain, il file hash arriva direttamente dal frontend dell'applicazione (viene calcolato in locale nel client) e non calcolato on-chain come mostrato di seguito:

```
1 bytes32 constant hash = keccak256(abi.encodePacked('FileBytecode'));
```

Ethereum Block Validation Algorithm Check

Per quanto riguarda il `block.timestamp`, la documentazione dice che il timestamp del blocco corrente deve essere rigorosamente maggiore del timestamp dell'ultimo blocco, ma l'unica garanzia è che sarà compreso tra i timestamp di due blocchi consecutivi nella blockchain. In aggiunta a ciò, in Ethereum vi è il *basic block validation algorithm* che esegue il check sul fatto che il timestamp del blocco deve essere maggiore del blocco precedente e minore di 15 minuti nel futuro.

```
Is parent.timestamp <= block.timestamp <= now + 900 sec?
```

4.2 Applicazione

L'applicazione presenta tre sezioni principali:

- Homepage
- Transactions Page
- Info Page

In homepage è presente una descrizione del servizio, un link al contratto deployato sul Ropsten Network su Etherscan² e altri link per il repository GitHub [48] e la documentazione.

Inoltre, in alto a destra è presente un tasto *Connect* che permette di connettere l'applicazione con il wallet MetaMask dell'utente per interagire con il Web3 (MetaMask può essere installato tramite estensione del browser).

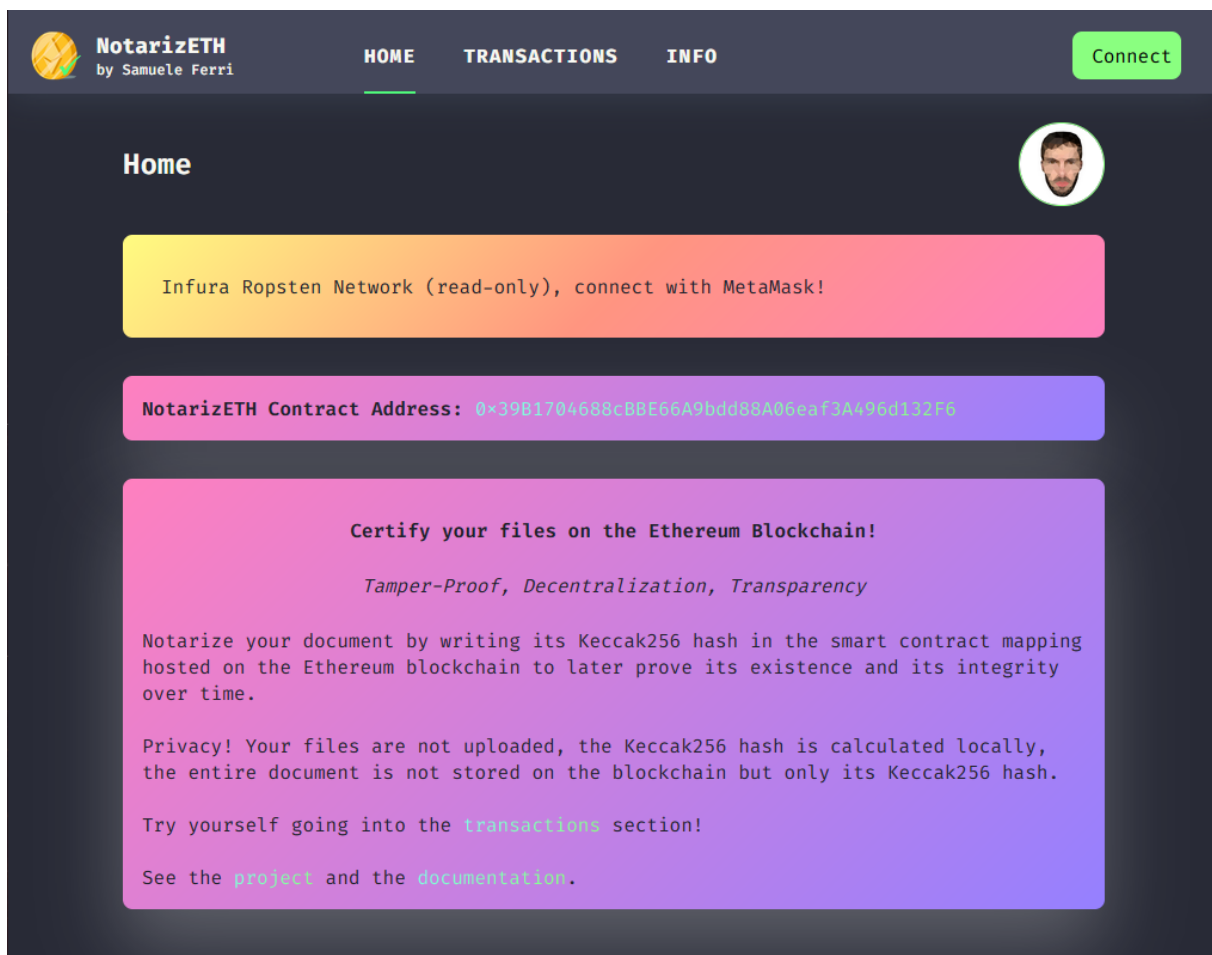


Figura 4.2: Homepage di NotarizETH

²Contract (<https://ropsten.etherscan.io/address/0x39B1704688cBBE66A9bdd88A06eaf3A496d132F6>)

La transactions page è la parte principale dell'applicazione. Oltre a due box contenenti lo storico delle transazioni eseguite dall'account utente MetaMask collegato sul Ropsten Network e lo storico delle notifiche con link rimandanti alla transazione su Etherscan, vi sono le tre funzioni che rispecchiano quelle definite nello smart contract: `verifyFile`, `certifyFile`, `resetFile` (analizzate in seguito nella sezione 4.3).

Nella figura Figura 4.3, appena sotto il titolo, vi è un box che indica con quale provider si è connessi (Infura di default solo in modalità lettura oppure MetaMask per interagire con il Web3 e inviare le transazioni) e su quale network si è connessi (l'unico network accettato è il Ropsten Network, sono emessi warnings nel caso si fosse connessi su altre reti). Inoltre, sono rappresentati i due box per lo storico delle transazioni e delle notifiche e sempre più sotto ci sono i tre box per le tre funzioni.

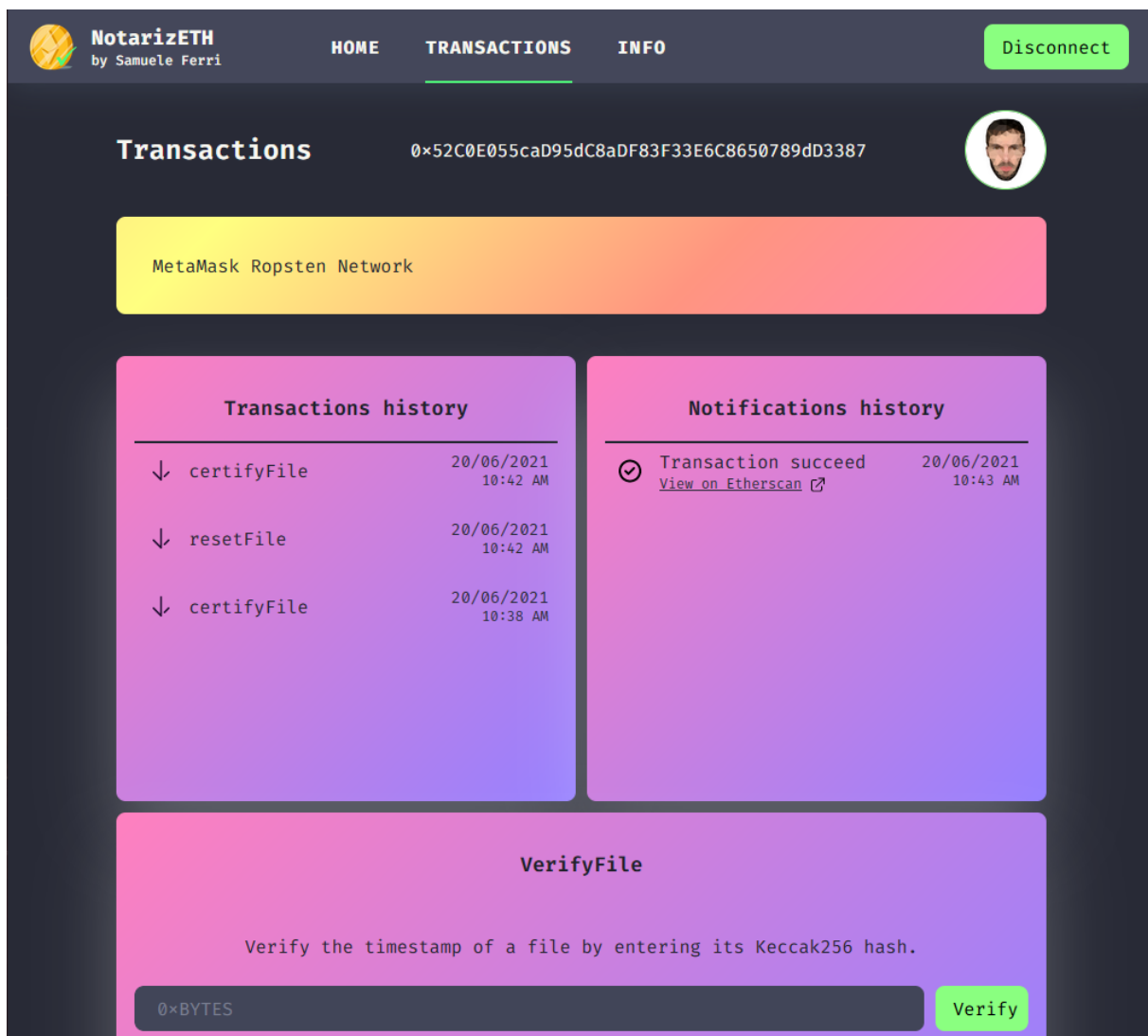


Figura 4.3: Transactions page di NotarizETH

L'info page contiene semplicemente informazioni relative all'account utente MetaMask collegato (address, ether balance) e informazioni relative al Ropsten Network (informazioni relative alla blockchain e all'ultimo blocco creato).

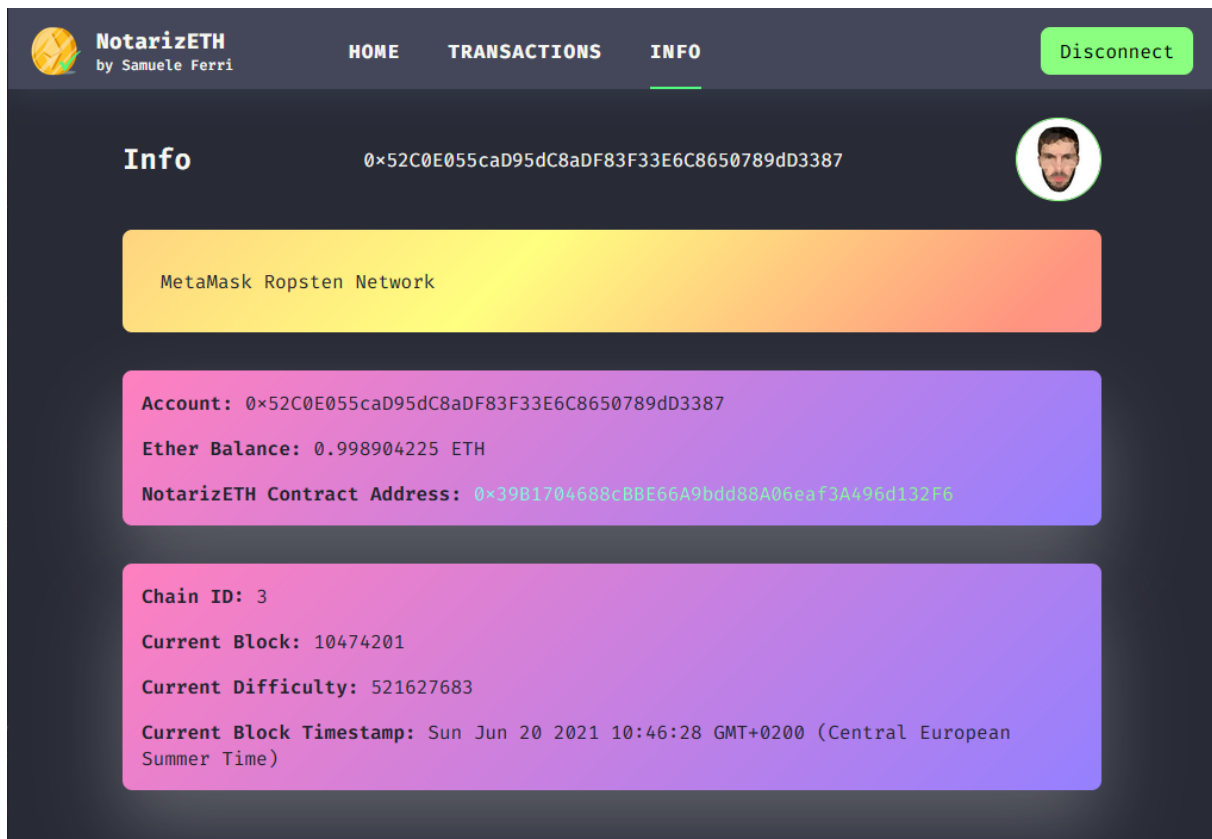


Figura 4.4: Info page di NotarizETH

4.2.1 React

L'applicazione è stata sviluppata in React [28] basandosi sul framework useDApp [33]. Questo framework mette a disposizione molti hooks che svolgono varie funzioni quali le chiamate ai contratti presenti sulla blockchain senza programmare direttamente con librerie quali *ethers.js* [38] e *web3-react* [39].

Tuttavia, essendo il framework recente e ancora nelle prime fasi di sviluppo, è stato necessario dialogare con lo smart contract deployato sulla blockchain direttamente con *ethers.js* come nel caso della gestione dell'info box di output mostrato nel Listing 4.3.

```
1 const contractEther = new ethers.Contract(NOTARIZETH_ADDRESS ,  
    NOTARIZETH_ABI_INTERFACE , library)  
2  
3 contractEther.verifyFile(valuehash).then(function (res: boolean[]) {  
4     console.log('VerifyFile', res),  
5     (document.getElementById('idHolderVerify')!.innerHTML =
```

```

6      res[0] == false
7      ? 'File hash ' + valuehash + ' not exist in the smart contract
mapping!'
8      : 'File hash ' +
9      valuehash +
10     ' exist in the smart contract mapping:' +
11     '<br>' +
12     'Sender: <a href="https://ropsten.etherscan.io/address/' +
13     res[1] +
14     '" target="_blank" class="drac-anchor drac-text drac-text-cyan
-green drac-text-yellow-pink--hover drac-mb-sm">' +
15     res[1] +
16     '</a>' +
17     '<br>' +
18     'Timestamp: ' +
19     new Date(Number(res[2]) * 1000)),
20     (document.getElementById('idHolderVerify')!.className =
21     res[0] == false
22     ? 'drac-text drac-line-height drac-text-yellow'
23     : 'drac-text drac-line-height drac-text-green')
24 })

```

Listing 4.3: Gestione dell'info box di output in React usando *ethers.js*

Non vengono riportati tutti packages e le funzioni di appoggio usate, si può trovare tutto il materiale e il codice sul repository di GitHub [48] nella sezione **react**.

Sempre su GitHub, è stato configurato un workflow con GitHub Actions [32] che permette di deployare automaticamente (a ogni push) l'applicazione su un bucket di Amazon S3 [31] dopo aver fatto la build ed eseguito dei controlli come mostrato nel Listing 4.4.

```

1 name: Deploy React App to AWS S3
2
3 on:
4   push:
5     branches:
6       - master
7
8 jobs:
9   deploy:
10    runs-on: ubuntu-latest
11
12    strategy:
13      matrix:
14        node-version: [12.x]
15
16    steps:

```

```

17   - uses: actions/checkout@master
18
19   - name: Use Node.js ${ matrix.node-version }
20     uses: actions/setup-node@v1
21     with:
22       node-version: ${ matrix.node-version }
23
24   - name: Create .npmrc
25     uses: bduff9/use-npmrc@v1.1
26     with:
27       dot-npmrc: ${ secrets.NPMRC_DRACULA_UI } # Contains all three
line of the .npmrc file, token included
28       working-directory: ./react/
29
30   - name: Yarn Install
31     run: |
32       yarn install
33     working-directory: ./react/
34
35   - name: Yarn Production Build # CI='' Not treating warnings as errors
36     run: |
37       CI='' yarn build
38     working-directory: ./react/
39
40   - name: Deploy to S3 Bucket (S3 Sync)
41     uses: jakejarvis/s3-sync-action@master
42     with:
43       args: --acl public-read --follow-symlinks --delete
44     env:
45       AWS_S3_BUCKET: ${ secrets.AWS_S3_BUCKET }
46       AWS_ACCESS_KEY_ID: ${ secrets.AWS_ACCESS_KEY_ID }
47       AWS_SECRET_ACCESS_KEY: ${ secrets.AWS_SECRET_ACCESS_KEY }
48       AWS_REGION: 'eu-south-1'
49       SOURCE_DIR: './react/dist'
50       # DEST_DIR: ''

```

Listing 4.4: GitHub Actions Workflow (Deploy React App to AWS S3)

4.2.2 MetaMask

MetaMask [30] è un'estensione (*injected wallet*) necessaria per dialogare con l'applicazione. Al primo accesso cliccando il tasto *Connect* viene chiesto di connettersi a NotarizETH come mostrato in Figura 4.5.

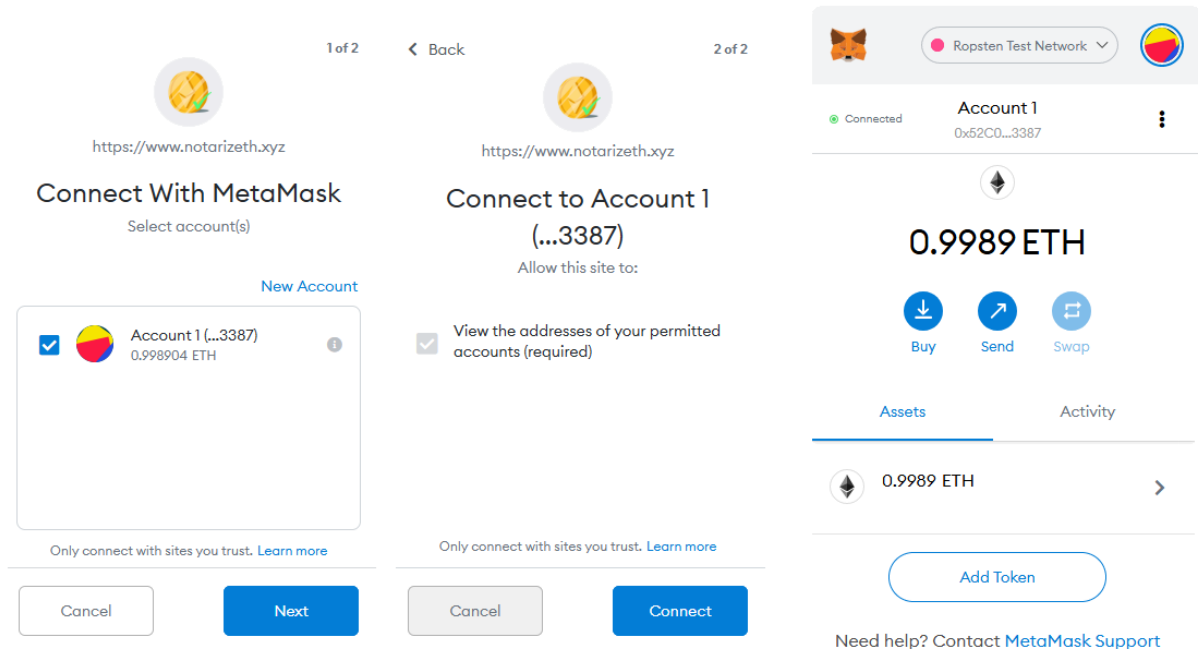


Figura 4.5: Connessione con MetaMask

In seguito è necessario spostarsi sul Ropsten Network (vi sono dei controlli sul chain ID) e avere a disposizione degli Ether necessari per effettuare le transazioni. Esistono, per le testnets, faucet di Ether come Ropsten Ethereum Faucet³ in cui è possibile richiedere gratuitamente 1 ETH inserendo il proprio indirizzo pubblico del wallet. Ora si è pronti per poter interagire con l'applicazione e lo smart contract deployato sulla blockchain.

4.2.3 Infura

Per la lettura dello stato della blockchain si può evitare di essere connessi MetaMask; grazie al provider Infura [29] sono possibili letture dello stato dello smart contract deployato su blockchain (anche sul Ropsten Network). Il tutto è stato configurato sul sito di Infura che ha rilasciato dei tokens d'autenticazione a loro Web3 Provider.

³Ropsten Ethereum Faucet (<https://faucet.ropsten.be/>)

4.3 Casi d'uso

In questa sezione verranno analizzati alcuni casi d'uso dell'applicazione relativi alle funzioni offerte nella transactions page.

4.3.1 Certificazione di un file

Per poter certificare un file, bisogna innanzitutto portarsi nella sezione `CertifyFile` e selezionare un file di qualsiasi tipo trascinandolo nel box o cliccandoci sopra e selezionandolo dal proprio disco. Lato client viene precalcolato il Keccak256 hash grazie alla libreria CryptoJS⁴



The screenshot shows a web interface titled "CertifyFile" with a purple-to-pink gradient background. The instructions state: "Certify the existence of your file on the blockchain by saving its Keccak256 hash as proof." Below this is a dashed box containing the text "Drag 'n' drop one file here, or click to select file". Underneath, it shows "File accepted: 'Piano degli studi (Magistrale) (2020-2021).pdf' - 41.1 KB" and "File rejected:". A dark grey box displays the Keccak256 hash: "0x95557e45a60b37140d707537d011a281c2ccbb5e2815c7f67a410edd7fd9d2e8", with a green "Copy" button to its right. A green "Certify" button is centered below the hash. Under the button is a QR code. At the bottom, a dark grey box contains the green text: "Successfull! Your file hash is stored in the smart contract mapping!".

Figura 4.6: Certificazione di un file

⁴CryptoJS (<https://cryptojs.gitbook.io/docs/>). Piccola precisazione, la funzione denominata SHA-3 all'interno di CryptoJS è stata, per stessa ammissione degli sviluppatori, erroneamente associata al Keccak hash. Lo standard SHA-3 che il NIST ha scelto nella competizione per lo SHA-3 non è altro che una versione derivata dal Keccak.

Dopodiché è possibile inviare la transazione sulla blockchain grazie a MetaMask specificando il gas price e attendere l'avvenuta conferma fino a ottenere, come mostrato nella Figura 4.6, l'hash da salvare per eventuali verifiche future sull'integrità del file (anche in formato QR Code).

In caso di errori, come nel caso in cui un dato file hash fosse già presente nel mapping dello smart contract, nel box di output verranno riportate informazioni utili relative all'errore come mostrato nella Figura 4.7.

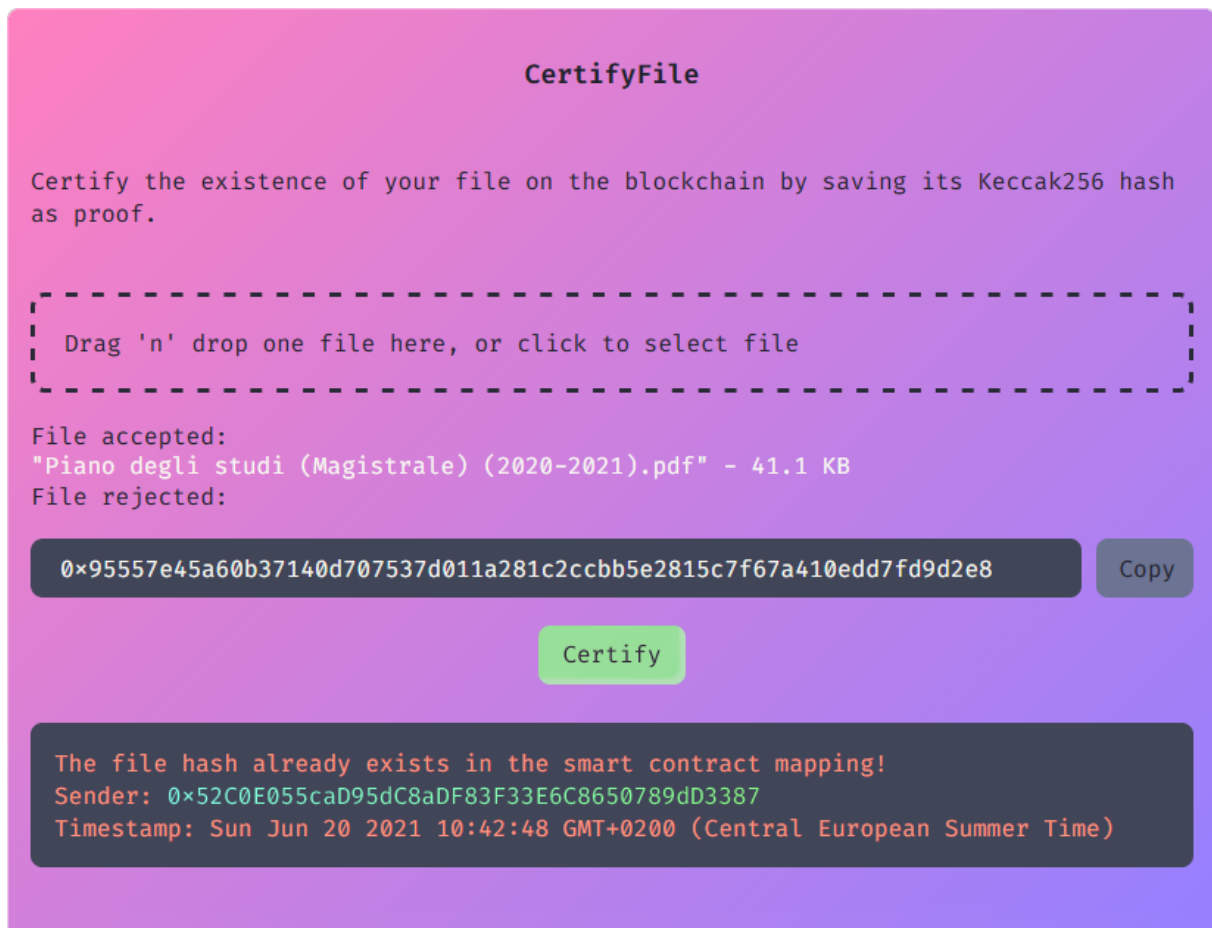


Figura 4.7: Errore nella certificazione di un file

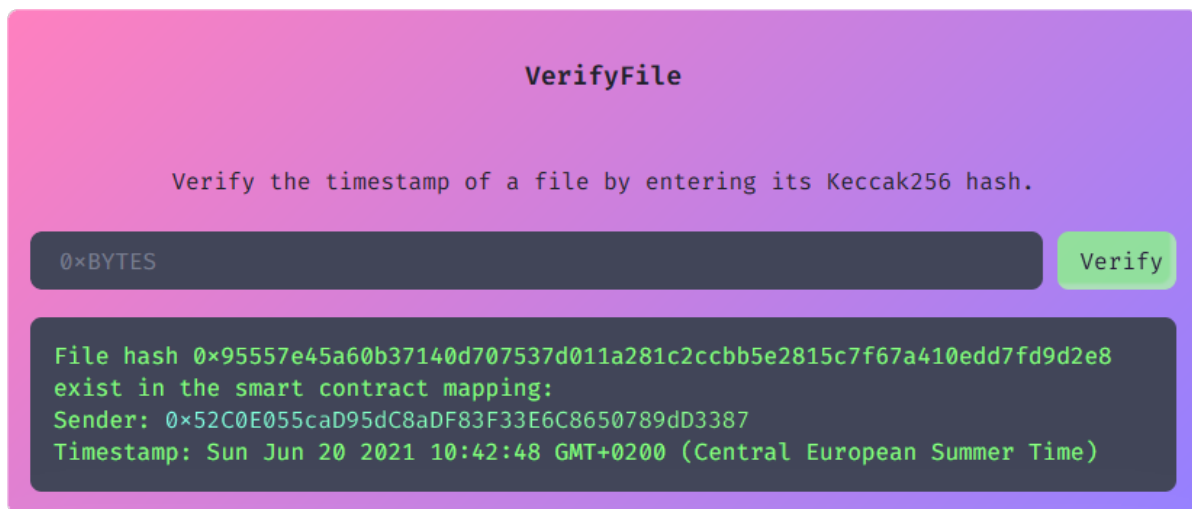
La confidenzialità delle informazioni contenute nei documenti è garantita grazie al fatto che l'hash del file viene calcolato in locale lato client, non vi è un caricamento dei file ma solo del loro hash.

In questo modo è possibile dimostrare la *Proof of Existence* del documento in un dato momento, ossia che un determinato documento (o meglio il suo hash) è stato caricato sulla blockchain e quindi esisteva già in quel preciso istante.

4.3.2 Verifica di un file hash

Viene offerta anche la possibilità di verificare un file hash dato come input nella sezione **VerifyFile**. Non è necessario connettersi a MetaMask, non c'è bisogno di effettuare una transazione, viene usato il provider Infura [29] per le letture dello stato dello smart contract.

Grazie a questa verifica è possibile verificare l'esistenza di un file hash in un determinato istante passato grazie al timestamp e visualizzare anche l'indirizzo del mittente della transazione dell'inserimento nella blockchain come mostrato nella Figura 4.8.



The screenshot shows a web interface titled "VerifyFile" with a purple-to-pink gradient background. Below the title is the instruction "Verify the timestamp of a file by entering its Keccak256 hash." There is a dark input field containing the placeholder text "0xBYTES" and a green "Verify" button. Below the input field, a dark box displays the following green text: "File hash 0x95557e45a60b37140d707537d011a281c2ccbb5e2815c7f67a410edd7fd9d2e8 exist in the smart contract mapping:", "Sender: 0x52C0E055caD95dC8aDF83F33E6C8650789dD3387", and "Timestamp: Sun Jun 20 2021 10:42:48 GMT+0200 (Central European Summer Time)".

Figura 4.8: Verifica di un file hash

In caso di errori, come nel caso in cui un dato file hash non sia presente nel mapping dello smart contract, nel box di output vengono riportate informazioni come mostrato nella Figura 4.9.

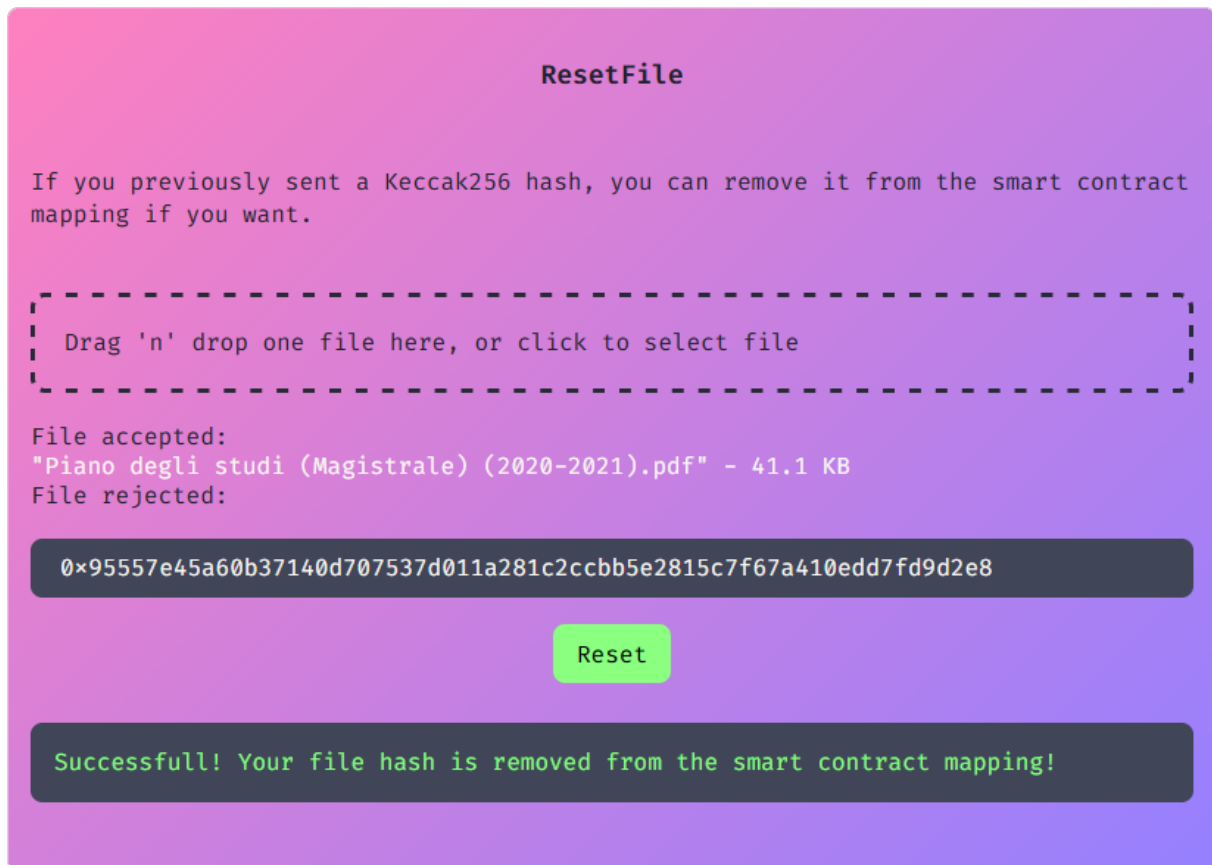


The screenshot shows the same "VerifyFile" interface as Figure 4.8. The input field contains "0xBYTES" and the "Verify" button is green. Below the input field, a dark box displays the following yellow text: "File hash 0x23557e45a60b37140d707537d011a281c2ccbb5e2815c7f67a410edd7fd9d265 not exist in the smart contract mapping!".

Figura 4.9: Errore nella verifica di un file hash

4.3.3 Reset di un file hash

Nella sezione `ResetFile` è possibile rimuovere dal mapping dello smart contract un proprio file hash precedentemente inserito e tutte le informazioni aggiuntive relative come mostrato nel Figura 4.10.



ResetFile

If you previously sent a Keccak256 hash, you can remove it from the smart contract mapping if you want.

Drag 'n' drop one file here, or click to select file

File accepted:
"Piano degli studi (Magistrale) (2020-2021).pdf" - 41.1 KB
File rejected:

0x95557e45a60b37140d707537d011a281c2ccbb5e2815c7f67a410edd7fd9d2e8

Reset

Successfull! Your file hash is removed from the smart contract mapping!

Figura 4.10: Reset di un file hash

In caso di errori, come nel caso in cui si voglia rimuovere un file hash inserito dal wallet di qualcun altro e non il proprio (per le regole di controllo dell'accesso precedentemente illustrate), nel box di output vengono riportate informazioni come mostrato nella Figura 4.11.

ResetFile

If you previously sent a Keccak256 hash, you can remove it from the smart contract mapping if you want.

Drag 'n' drop one file here, or click to select file

File accepted:
"Piano degli studi (Magistrale) (2020-2021).pdf" - 41.1 KB
File rejected:

0x95557e45a60b37140d707537d011a281c2ccbb5e2815c7f67a410edd7fd9d2e8

Reset

You are not the sender of the file hash!
Sender: 0x52C0E055caD95dC8aDF83F33E6C8650789dD3387
Timestamp: Sun Jun 20 2021 17:30:04 GMT+0200 (Central European Summer Time)

Figura 4.11: Errore nel reset di un file hash

116

Capitolo 5

Conclusioni

Parte finale che descrive i possibili sviluppi futuri sia dell'analisi dei costi che dell'applicazione e che contiene le conclusioni del lavoro di tesi.

5.1 Sviluppi futuri

Vengono presentati gli sviluppi futuri su entrambe le parti di tesi.

5.1.1 Analisi dei costi

La blockchain di Ethereum è costantemente in aggiornamento con le Ethereum Improvement Proposals [19] e con i nuovi standard Ethereum Requests for Comment [20]; il rischio è che nel tempo le informazioni contenute in questo lavoro di tesi diventino deprecate e inesatte.

Già durante la scrittura della tesi, a causa del rilascio del Berlin hard fork sulla mainnet avvenuto il 14 aprile 2021, sono cambiati i costi per gli opcodes che gestiscono i dati nello storage. Grazie alla newsletter Week in Ethereum News [26] sono stato informato sulle nuove modifiche introdotte e in questo modo ho potuto aggiornare il mio lavoro di tesi alla luce di questi cambiamenti riguardanti lo storage.

Mantenere aggiornato costantemente l'analisi dei costi è sicuramente un'operazione non facile data la mole di EIPs introdotte periodicamente: l'Ethereum Yellow Paper [7] sarebbe un'ottima fonte d'informazioni sul funzionamento di Ethereum se solo venisse aggiornato costantemente e non a tratti anche da fonti fidate.

Nel futuro sarà interessante analizzare tutti i cambiamenti che verranno introdotti dopo il definitivo passaggio a ETH2 adottando il Proof of Stake (PoS); sicuramente con lo sharding verrà drasticamente aumentata la scalabilità, la network bandwidth e verrà ridotto il consumo di gas per interagire con gli smart contracts.

Un altro possibile sviluppo futuro interessante riguarda alcuni standard tra cui ERC-20 [21] ed ERC-721 [22] precedentemente descritti nella sottosezione 1.2.4; sarebbe interessante analizzare i costi della creazione e del trasferimento di questi token.

Un confronto tra Solidity e altri linguaggi di programmazione per smart contracts sarebbe molto interessante per analizzarne differenze ed eventuali scenari in cui preferire un linguaggio rispetto all'altro.

Vyper [18] è un linguaggio di programmazione contract-oriented, derivato da Python e che ha come target l'Ethereum Virtual Machine (EVM). È un linguaggio semplice orientato alla sicurezza in modo tale da scrivere contratti sicuri con costrutti semplici utile anche per chi non ha molta esperienza nella programmazione. Non contiene molti dei costrutti familiari alla maggior parte dei programmatori: ereditarietà di classi, overloading di funzioni, overloading di operatori e ricorsione; nessuno di questi è tecnicamente necessario per un linguaggio Turing completo e inoltre alcuni di questi costrutti rappresentano rischi per la sicurezza andando ad aumentare la complessità dello smart contract. Al contrario, Vyper fornisce dei controlli sui limiti degli arrays e sull'aritmetica (overflow); inoltre, garantisce la decidibilità, ossia è possibile calcolare con precisione un limite superiore al consumo di gas di una qualsiasi chiamata di funzione.

In breve, Vyper non si sforza di sostituire al 100% tutto ciò che può essere fatto in Solidity; vieta deliberatamente alcune pratiche o le rende più difficili da implementare con l'obiettivo di aumentare la sicurezza e ridurre le vulnerabilità presenti nello smart contract. Confrontare Solidity e Vyper e i rispettivi scenari di utilizzo migliori può essere un ulteriore sviluppo futuro del lavoro di tesi.

5.1.2 Sviluppo della DApp NotarizETH

L'applicazione decentralizzata NotarizETH [23] certifica solo la *Proof of Existence* e l'integrità dei documenti, ossia certifica che un determinato file è esistito in una determinata data e ora e permette la successiva verifica dell'integrità e dell'assenza di manipolazioni (*tamper proof*) a chiunque abbia una copia del file originale.

Inoltre, l'applicazione certifica solamente che un determinato utente è entrato in possesso dell'hash relativo a un determinato file. Questo deriva dal fatto che l'applicazione riceve (e scrive su blockchain) direttamente la sequenza di hash dal client.

Per risolvere il problema è sufficiente utilizzare una tecnica (o uno schema) di firma digitale. Ci sono diverse alternative:

- Firmare il file tramite un certificato X509 che viene allegato al documento da certificare (procedendo successivamente con il calcolo del digest).
- Utilizzare una tecnica HMAC (Keyed-Hash Message Authentication Code).

- Utilizzare i servizi offerti da una CA (Certification Authority).

Nonostante l'applicazione sia stata sviluppata in modo responsive e adattabile a vari tipi di schermi, la versione mobile non è così usufruibile anche per il fatto che è necessario un wallet per il Web3 tipo MetaMask. Sicuramente ci saranno soluzioni anche su questo aspetto in futuro.

Altra possibile aggiunta, per piccoli file non confidenziali si potrebbe offrire la possibilità di salvarli interamente sulla blockchain grazie a IPFS¹ ad esempio.

5.2 Conclusioni

L'ambito blockchain era per me tutto nuovo, ho avuto l'occasione di approfondire l'argomento grazie a questo lavoro di tesi che ha nutrito in me sempre di più l'interesse sugli aspetti legati agli smart contracts e le loro applicazioni. Focalizzandomi su Ethereum ho potuto indagare il funzionamento dell'EVM e apprendere il linguaggio Solidity per la scrittura dei contratti. Ho usato anche molti tools e tecnologie sia nella prima parte che nella seconda relativa a NotarizETH, la quale mi ha dato una visione completa del processo di sviluppo di un'applicazione decentralizzata.

L'ambito blockchain è in una spirale di costante innovazione, il mio personale obiettivo è rimanere aggiornato su tali tecnologie e sull'evoluzione della programmazione di smart contracts.

Ringrazio i ricercatori del UniBG Seclab² che mi hanno seguito nel lavoro di tesi dandomi ampie libertà d'indagine sugli argomenti che ritenevo più interessanti e per tutto il supporto tecnico fornito.

¹IPFS (<https://ipfs.io/>)

²UniBG Seclab (<https://seclab.unibg.it/>)

Bibliografia & Sitografia

- [1] *Ethereum*. URL: <https://ethereum.org/> (cit. a p. 3).
- [2] *Ethereum Docs (Risorse per sviluppatori)*. URL: <https://ethereum.org/it/developers/> (cit. a p. 3).
- [3] *Ethereum ETH2*. URL: <https://ethereum.org/en/eth2/> (cit. alle pp. 4, 19).
- [4] *Modified Merkle Patricia Trie Specification (also Merkle Patricia Tree)*. URL: <https://eth.wiki/en/fundamentals/patricia-tree> (cit. a p. 5).
- [5] *Ethereum Virtual Machine Opcodes*. URL: <https://www.ethervm.io/> (cit. a p. 5).
- [6] Eric Conner. «Understanding Ethereum Gas, Blocks and the Fee Market» (2019). URL: <https://medium.com/@eric.conner/understanding-ethereum-gas-blocks-and-the-fee-market-d5e268bf0a0e> (cit. a p. 7).
- [7] Gavin Wood. «Ethereum Yellow Paper: A secure decentralised generalised transaction ledger» (2014,2021). URL: <https://ethereum.github.io/yellowpaper/paper.pdf> (cit. alle pp. 7, 11, 117).
- [8] *EIP-2929: Gas cost increases for state access opcodes*. URL: <https://eips.ethereum.org/EIPS/eip-2929> (cit. a p. 11).
- [9] Tim Beiko. «Ethereum Berlin Upgrade Announcement» (2021). URL: <https://blog.ethereum.org/2021/03/08/ethereum-berlin-upgrade-announcement/> (cit. alle pp. 11, 20).
- [10] Franco Victorio. «Understanding gas costs after Berlin» (2021). URL: <https://hackmd.io/@fvictorio/gas-costs-after-berlin> (cit. a p. 11).
- [11] Daniel Perez e Benjamin Livshits. «Broken Metre: Attacking Resource Metering in EVM» (2020). URL: <https://arxiv.org/pdf/1909.07220.pdf> (cit. a p. 12).
- [12] *EIP-3298: Removal of refunds*. URL: <https://eips.ethereum.org/EIPS/eip-3298> (cit. a p. 13).
- [13] *EIP-3403: Partial removal of refunds*. URL: <https://eips.ethereum.org/EIPS/eip-3403> (cit. a p. 13).
- [14] *EIP-3529: Reduction in refunds*. URL: <https://eips.ethereum.org/EIPS/eip-3529> (cit. a p. 13).

- [15] Aodhgan Gleeson. «GasToken: or how I learned to stop worrying and love gas price surges» (2020). URL: <https://medium.com/coinmonks/gastoken-or-how-i-learned-to-stop-worrying-and-love-gas-price-surges-6aaee9fb0ba3> (cit. a p. 13).
- [16] Virgil Griffith. «Ethereum is game-changing technology, literally» (2019). URL: <https://medium.com/@virgilgr/ethereum-is-game-changing-technology-literally-d67e01a01cf8> (cit. a p. 14).
- [17] *Solidity*. URL: <https://soliditylang.org/> (cit. alle pp. 14, 17).
- [18] *Vyper*. URL: <https://vyper.readthedocs.io/en/stable/> (cit. alle pp. 14, 118).
- [19] *Ethereum Improvement Proposals (EIPs)*. URL: <https://eips.ethereum.org/> (cit. alle pp. 15, 117).
- [20] *Ethereum Requests for Comment (ERC)*. URL: <https://eips.ethereum.org/erc> (cit. alle pp. 15, 117).
- [21] *ERC-20 Token Standard*. URL: <https://eips.ethereum.org/EIPS/eip-20> (cit. alle pp. 15, 118).
- [22] *ERC-721 Non Fungible Token Standard*. URL: <https://eips.ethereum.org/EIPS/eip-721> (cit. alle pp. 16, 118).
- [23] *NotarizETH*. URL: <http://notarizeth.xyz/> (cit. alle pp. 16, 19, 20, 99, 118).
- [24] *Solidity Docs*. URL: <https://docs.soliditylang.org/en/v0.8.1/> (cit. alle pp. 17, 29, 36, 42, 76).
- [25] Gerardo Canfora, Andrea Di Sorbo, Sonia Laudanna, Anna Vacca e Corrado Aaron Visaggio. «Profiling Gas Leaks in the Deployment of Solidity Smart Contracts» (2020). URL: <https://arxiv.org/pdf/2008.05449.pdf> (cit. a p. 20).
- [26] *Week in Ethereum News*. URL: <https://weekinethereumnews.com/> (cit. alle pp. 20, 117).
- [27] *Ropsten Test Network*. URL: <https://ropsten.etherscan.io/> (cit. alle pp. 20, 28, 99).
- [28] *React*. URL: <https://reactjs.org/> (cit. alle pp. 20, 24, 99, 108).
- [29] *Infura*. URL: <https://infura.io/> (cit. alle pp. 20, 23, 99, 111, 114).
- [30] *MetaMask*. URL: <https://metamask.io/> (cit. alle pp. 20, 23, 99, 111).
- [31] *Amazon S3*. URL: <https://aws.amazon.com/it/s3/> (cit. alle pp. 20, 25, 99, 109).
- [32] *GitHub*. URL: <https://github.com/> (cit. alle pp. 20, 26, 99, 109).
- [33] *Framework useDApp*. URL: <https://usedapp.io/> (cit. alle pp. 20, 24, 99, 108).
- [34] *OpenZeppelin*. URL: <https://openzeppelin.com/> (cit. alle pp. 20, 99, 100, 103).
- [35] *Remix*. URL: <http://remix.ethereum.org/> (cit. a p. 21).
- [36] *Truffle*. URL: <https://www.trufflesuite.com/truffle> (cit. a p. 22).
- [37] *Ganache*. URL: <https://www.trufflesuite.com/ganache> (cit. a p. 22).
- [38] *Libreria ethers.js*. URL: <https://github.com/ethers-io/ethers.js> (cit. alle pp. 24, 108).

- [39] *Libreria web3-react*. URL: <https://github.com/NoahZinsmeister/web3-react> (cit. alle pp. 24, 108).
- [40] *Dracula UI*. URL: <https://draculatheme.com/ui> (cit. a p. 25).
- [41] *Python*. URL: <https://www.python.org/> (cit. a p. 25).
- [42] *Solc-JS*. URL: <https://github.com/ethereum/solc-js> (cit. a p. 25).
- [43] *Online Solidity Decompiler*. URL: <https://ethervm.io/decompile> (cit. a p. 25).
- [44] *Tool eth-gas-reporter*. URL: <https://github.com/cgewecke/eth-gas-reporter> (cit. a p. 25).
- [45] *Plugin Solidity per Visual Studio Code*. URL: <https://marketplace.visualstudio.com/items?itemName=JuanBlanco.solidity> (cit. a p. 25).
- [46] *Plugin Prettier per Solidity*. URL: <https://github.com/prettier-solidity/prettier-plugin-solidity> (cit. a p. 25).
- [47] *GitHub Actions*. URL: <https://github.com/features/actions> (cit. a p. 26).
- [48] *Repository GitHub Master's thesis*. URL: <https://github.com/samuelexferri/masterthesis> (cit. alle pp. 26, 27, 99, 106, 109).
- [49] *Overleaf*. URL: <https://overleaf.com/> (cit. a p. 26).
- [50] *Slack*. URL: <https://slack.com/> (cit. a p. 26).
- [51] *Trello*. URL: <https://trello.com> (cit. a p. 26).
- [52] *Visual Studio Code*. URL: <https://code.visualstudio.com/> (cit. a p. 26).
- [53] *Ethereum Stack Exchange*. URL: <https://ethereum.stackexchange.com/> (cit. a p. 26).
- [54] *Stack Overflow*. URL: <https://stackoverflow.com/> (cit. a p. 26).
- [55] *Contract ABI Specification*. URL: <https://docs.soliditylang.org/en/develop/abi-spec.html#abi> (cit. alle pp. 28, 36).
- [56] Jean Cvllr. «All About Solidity — Article Series» (2019). URL: <https://jeancvllr.medium.com/all-about-solidity-article-series-f57be7bf6746> (cit. a p. 29).
- [57] *EIP-170: Contract code size limit*. URL: <https://eips.ethereum.org/EIPS/eip-170> (cit. a p. 30).
- [58] Markus Waas. «Downsizing contracts to fight the contract size limit» (2020). URL: <https://ethereum.org/it/developers/tutorials/downsizing-contracts-to-fight-the-contract-size-limit/> (cit. a p. 30).
- [59] *Smart Contract Upgradability*. URL: <https://docs.upgradablecontracts.com/> (cit. a p. 31).
- [60] *EIP-1167: Minimal Proxy Contract*. URL: <https://eips.ethereum.org/EIPS/eip-1167> (cit. a p. 31).
- [61] Ting Chen, Xiaoqi Li, Xiapu Luo e Xiaosong Zhang. «Under-optimized smart contracts devour your money» (2017). URL: <https://arxiv.org/pdf/1703.03994.pdf> (cit. alle pp. 31, 85, 86).

- [62] Sorawit Suriyakarn. «Keeping Gas Cost under Control» (2019). URL: <https://medium.com/bandprotocol/solidity-102-1-keeping-gas-cost-under-control-ae95b835807f> (cit. alle pp. 34, 35).
- [63] Yi-Cyuan Chen. «How does function name affect gas consumption in smart contract» (2018). URL: <https://medium.com/joyso/solidity-how-does-function-name-affect-gas-consumption-in-smart-contract-47d270d8ac92> (cit. a p. 81).
- [64] *Yul*. URL: <https://docs.soliditylang.org/en/latest/yul.html> (cit. a p. 92).

Elenco delle figure

1.1	Transazioni contenute nei blocchi	2
1.2	Logo di Ethereum	3
1.3	Diagramma della struttura dell'EVM	5
1.4	Diagramma che mostra dove è usato il gas nell'EVM	7
1.5	Ethereum Yellow Paper Gas Fees aggiornate alla Petersburg Version 41c1837 del giorno 2021-02-14	8
1.6	Prezzo medio del gas in Ethereum	10
1.7	Diagramma che mostra il rimborso del gas non usato	10
1.8	Logo di Solidity	17
2.1	Logo di Remix	21
2.2	Schema di Remix	21
2.3	Logo di Truffle	22
2.4	Schema di Truffle	22
2.5	Logo di MetaMask	23
2.6	Logo di Infura	23
2.7	Logo di React	24
3.1	Opcode SHA3 alias di KECCAK256	28
3.2	Endianness	29
3.3	A sinistra: scorrere un elenco in catena costa $O(n)$ gas, il costo cresce linearmente man mano che la lista cresce. A destra: calcolare la posizione off-chain e verificare il valore nella catena costa una quantità costante di gas indipendentemente dalle dimensioni della lista.	35
3.4	A sinistra: invocare un'azione Distribute ha un costo in gas proporzionale al numero di ricevitori in una transazione; inoltre, la transazione può fallire al superamento del gas limit. A destra: tutte le transazioni (1 Add e 4 Claim) hanno costi costanti.	35
3.5	Compattare più variabili in uno slot usando l'encoding	54
3.6	Figura esplicativa dei mappings in riferimento al contratto MappingEnd . .	61
3.7	Figura esplicativa dei bytes in riferimento al contratto ArraysBytes	72

3.8	Figura esplicativa degli arrays a dimensione fissa e dinamica in riferimento al contratto ArraysDiff	75
3.9	Confronto dei costi per arrays a dimensione fissa e dinamica	75
4.1	Logo di NotarizETH	99
4.2	Homepage di NotarizETH	106
4.3	Transactions page di NotarizETH	107
4.4	Info page di NotarizETH	108
4.5	Connessione con MetaMask	111
4.6	Certificazione di un file	112
4.7	Errore nella certificazione di un file	113
4.8	Verifica di un file hash	114
4.9	Errore nella verifica di un file hash	114
4.10	Reset di un file hash	115
4.11	Errore nel reset di un file hash	116

Elenco dei listati

1.1	Ether Units	9
1.2	Inline Assembly	18
3.1	Licenza e versione di Solidity	28
3.2	Dead code e predicato opaco	31
3.3	Uso della libreria <i>SafeMath</i>	33
3.4	Salvare direttamente il literal invece che il valore calcolato se già conosciuto	34
3.5	Arrays a dimensione fissa e dinamica	38
3.6	ABI Coder v2 con array multidimensionale di singoli byte	39
3.7	Conversione di <code>strings</code> in <code>bytes32</code>	40
3.8	Mappings annidati	43
3.9	Aggiornare un valore nello storage	46
3.10	Inizializzazione delle variabili	47
3.11	Scambio di variabili	47
3.12	Valori booleani	48
3.13	Archiviazione degli arrays	48
3.14	Archiviazione delle structs	49
3.15	Compattare le variabili nello storage	49
3.16	Archiviare singolarmente le variabili nello storage	50
3.17	Archiviare le structs nello storage	51
3.18	Archiviare con encoding e decoding nello storage	53
3.19	Funzioni Python per calcolare il Keccak256 hash	55
3.20	Contratto Mapping1	55
3.21	Calcoli del Keccak256 hash per i due mappings	56
3.22	Bytecode del contratto Mappings1	56
3.23	Contratto Mapping2	57
3.24	Bytecode del contratto Mappings2	57
3.25	Contratto MappingStruct	58
3.26	Debug dello stato dopo l'esecuzione della funzione <code>mapStruct()</code>	59
3.27	Contratto MappingNotPack	59
3.28	Debug dello stato dopo l'esecuzione della funzione <code>mapNotPack()</code>	60

3.29	Contratto MappingEnd	60
3.30	Contratto di riferimento sugli arrays (Parte 1)	62
3.31	Debug dello stato dopo l'esecuzione della funzione <code>fixedByteArray()</code> . . .	64
3.32	Debug dello stato dopo l'esecuzione della funzione <code>dynamicByteArray()</code> nel caso di 31 byte	65
3.33	Debug dello stato dopo l'esecuzione della funzione <code>dynamicByteArray()</code> nel caso di 32 byte	65
3.34	Debug dello stato dopo l'esecuzione della funzione <code>dynamicStringArray()</code>	65
3.35	Debug dello stato dopo l'esecuzione della funzione <code>fixedArray()</code>	66
3.36	Debug dello stato dopo l'esecuzione della funzione <code>dynamicArray()</code>	66
3.37	Contratto di riferimento sugli arrays (Parte 2)	67
3.38	Contratto ArraysDynamic	70
3.39	Debug dello stato dopo l'esecuzione della funzione <code>ad1()</code>	70
3.40	Debug dello stato dopo l'esecuzione della funzione <code>ad2()</code>	71
3.41	Contratto ArraysBytes	71
3.42	Debug dello stato dopo l'esecuzione della funzione <code>by1()</code>	72
3.43	Debug dello stato dopo l'esecuzione della funzione <code>by2()</code>	72
3.44	Unchecked	73
3.45	Differenze di costo tra arrays a dimensione fissa e dinamici	74
3.46	Modificatore di funzione	77
3.47	Multiple Inheritance Overriding	78
3.48	Visibilità e chiamate di funzione	79
3.49	Nomi delle funzioni 1	81
3.50	Nomi delle funzioni 2	82
3.51	Generazione del Method ID	82
3.52	Block scoping delle variabili	83
3.53	Ciclo for con limite superiore variabile	84
3.54	Cicli for con l'uso di una variabile in memoria temporanea	85
3.55	Cicli for cominati	86
3.56	Valutazione a corto circuito	87
3.57	Eventi per il logging	89
3.58	Eccezioni con assert e require	90
3.59	Notazione standard e polacca inversa	92
3.60	Blocco di Inline Assembly	92
3.61	Functional Style Assembly	93
3.62	Non-Functional Style Assembly	93
3.63	Layout dello stack dopo ogni istruzione	93
3.64	Accedere allo scratch space grazie all'Inline Assembly	94

3.65	Snippet relativo al confronto tra stringhe effettuato dalla libreria <i>solidity-stringutils</i>	94
3.66	Accedere a una posizione arbitraria nello storage con l'Inline Assembly . . .	95
3.67	Somma degli elementi di un vettore con l'Inline Assembly	96
4.1	Smart Contract dell'applicazione NotarizETH	100
4.2	Variabili della struct Certificate nello storage	104
4.3	Gestione dell'info box di output in React usando <i>ethers.js</i>	108
4.4	GitHub Actions Workflow (Deploy React App to AWS S3)	109