

BingoEth

Relazione progetto P2P

Leonardo Manneschi

Leonardo Scoppitto

Indice

| | | |
|----------|---|-----------|
| 1 | Introduzione | 1 |
| 2 | Struttura del progetto | 1 |
| 3 | Funzionamento del progetto | 1 |
| 3.1 | Lato front-end | 1 |
| 3.1.1 | Creazione Game | 1 |
| 3.1.2 | Join delle stanze | 2 |
| 3.1.3 | Fase di gioco | 4 |
| 3.2 | Lato smart contract | 6 |
| 3.2.1 | Funzione <code>createGame</code> | 6 |
| 3.2.2 | Funzione <code>extractNumber</code> | 7 |
| 4 | Principali decisioni | 8 |
| 4.1 | Struttura dati | 8 |
| 4.2 | Vincoli | 8 |
| 4.3 | Uso di Merkle Tree | 8 |
| 4.3.1 | Generazione del Merkle Tree | 8 |
| 5 | Manuale utente | 12 |
| 5.1 | Utilizzo istanza pubblica | 12 |
| 5.2 | Esecuzione da sorgente | 12 |
| 6 | Valutazione del consumo di gas | 13 |
| 6.1 | Valutazione di esempio con un gioco con un creatore e 3 joiners | 13 |
| 6.1.1 | Calcolo del consumo di gas: | 13 |
| 6.1.2 | Totale del consumo di gas: | 14 |
| 6.1.3 | Calcolo: | 14 |
| 6.1.4 | Calcolo del consumo di gas per ciascun partecipante: | 14 |
| 6.1.5 | Conclusione: | 14 |
| 7 | Potenziiali vulnerabilità | 15 |
| 7.1 | Generazione di numeri casuali | 15 |

1 Introduzione

L'applicazione sviluppata come progetto finale del corso di Peer To Peer & Blockchains si chiama BingoEth e si tratta di un'implementazione del bingo (la versione più famosa del bingo, quella americana da 75 numeri) realizzata in React/javascript per la parte di front-end e in Solidity per la parte di smart contract. È possibile consultare il repository su GitHub.

2 Struttura del progetto

Il progetto è stato organizzato come segue:

```
├── client → Codice del client scritto in React
├── docs → Directory contenente la relazione del progetto
├── truffle → Root del progetto dello smart contract
│   ├── contracts
│   │   └── Bingo.sol → Sorgente del contratto
│   ├── migrations
│   │   └── 1_deploy_contract.js → Script di deploy del contratto
└── Dockerfile/docker-compose.yml → Container docker per il test e il deploy
```

3 Funzionamento del progetto

3.1 Lato front-end

Il front-end dell'applicazione BingoEth è sviluppato in React e consente agli utenti di interagire con il contratto smart su Ethereum per giocare a bingo. Le principali funzionalità del front-end includono:

- Creazione di una nuova partita scegliendo il numero massimo di giocatori e la cifra da scommettere.
- Partecipazione a una partita esistente facendo una join accettando o meno la cifra da scommettere.
- Visualizzazione dello stato del gioco in tempo reale.



Figure 1: Home page

Appena entrati nella nostra Dapp vedremo una schermata di home (Figura 1), vediamo subito che possiamo fare due cose principali:

- unirci ad una partita casuale o specifica.
- creare una nuova partita.

3.1.1 Creazione Game

Per creare la stanza (Figura 2) è necessario inserire il numero massimo di giocatori e l'importo della scommessa per partecipare. Questi due valori abbiamo deciso debbano essere (per semplicità) interi,

Figure 2: Creazione della stanza di gioco

Una volta inseriti i dati il pulsante **Scommetti** si abiliterà e, premendolo, verrà chiamata la funzione del contratto `creategame` di cui parleremo nel prossimo capitolo.

Solo dopo aver premuto il pulsante scommetti ci verrà mostrata la schermata di attesa (Figura 3), in cui attenderemo di aver raggiunto il numero totale di utenti che abbiamo scelto precedentemente.



Figure 3: Attesa dell'unione di altri player

3.1.2 Join delle stanze

Per eseguire i join delle stanze abbiamo due opzioni:

- scegliere una stanza random
- scegliere una stanza specifica, sapendo l'ID della stanza

Se cliccato il pulsante **Entra in una stanza** ci si aprirà una schermata (Figura 4) in cui dobbiamo obbligatoriamente inserire un ID per selezionare la camera.

Se l'ID selezionato risulta essere corretto e ci sono ancora posti disponibili allora apparirà la schermata contenente (Figura 5):

- ID della stanza scelta
- Gli Eth da scommettere per poter unirsi alla camera
- Il numero di posti disponibili

A questo punto se viene cliccato il pulsante **Entra nella stanza** L'ETH scommesso verrà prelevato e il numero dei giocatori verrà aggiornato attraverso la chiamata di funzione del contratto `JoinGame` di cui parleremo nel prossimo capitolo.

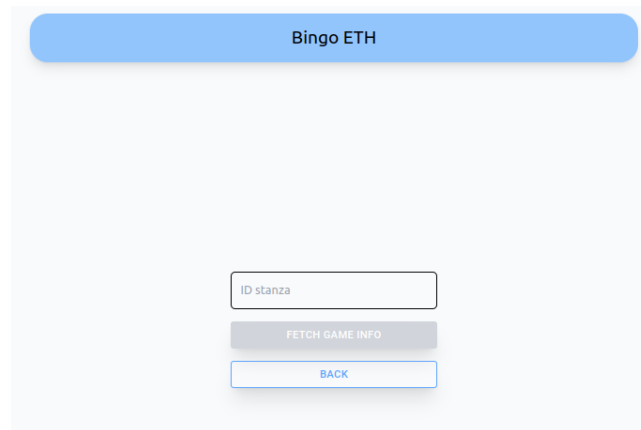


Figure 4: Pagina di raccolta informazioni sui game

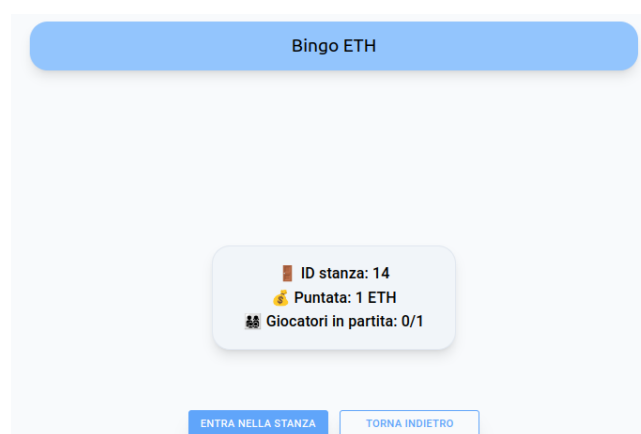


Figure 5: Pagina di scelta per la camera

Per il pulsante **Entra nella stanza randomica** invece la schermata rimane simile, viene solo rimosso il valore da inserire e viene chiesto solo di cliccare un pulsante per richiedere le informazioni per una stanza ancora disponibile casuale. La schermata di scelta per la camera rimane uguale invece (Figura 5).

3.1.3 Fase di gioco

Una volta che tutti i giocatori sono entrati si avvia la partita e tutti i giocatori creano la propria tabella (parleremo meglio di questo argomento nel capitolo sulla sicurezza).

3.1.3.1 Lato Creatore della stanza Come detto in precedenza solo il creatore delle stanze si prende carico della responsabilità di chiedere l'estrazione dei numeri al contratto. E come tale è in una posizione di comando rispetto agli altri joiners. Vediamo la schermata di gioco (Figura 6):

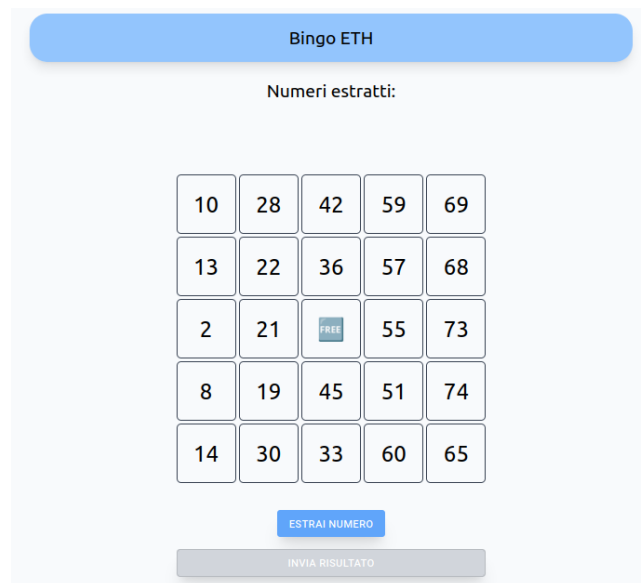


Figure 6: Schermata di gioco del creatore

Come si vede dall'immagine (Figura 6) abbiamo:

- La lista dei valori estratti da 1 a 75.
- la tabella contenente i numeri scelti CASUALMENTE lato client. abbiamo rimosso la possibilità di scelta dei numeri per evitare possibili problemi di imbrogli dai giocatori. Sarà cura dei giocatori selezionare la casella corretta, come nel vero gioco.
- Un pulsante per estrarre i numeri, chiamando la funzione `ExtractNumber` del contratto.
- Un pulsante per fare submit della cartella quando si ha fatto bingo. Il pulsante rimane disabilitato fino a che non si ha selezionato almeno 5 caselle in fila per fare bingo.

3.1.3.2 Lato Joiner della stanza Dato che solo il creatore del gioco può estrarre i numeri, abbiamo dato ai joiner della stanza la possibilità di denunciarlo se cerca di bloccare il gioco per far allontanare i giocatori. Per questo motivo, è stato aggiunto un pulsante **Denuncia**. Se premuto, questo pulsante rimuove il creatore dalla partita dopo un certo numero di secondi, terminando il gioco.

Altre possibili implementazioni avrebbero potuto includere una logica che trasformava chi denunciava nel nuovo creatore. Tuttavia, abbiamo optato per una soluzione più semplice. Se la denuncia viene confermata, il creatore dovrà estrarre un numero entro un certo limite di tempo. Se lo fa, il gioco prosegue normalmente; altrimenti, tutti i giocatori riceveranno indietro il proprio denaro, insieme a una percentuale dei soldi scommessi dal creatore del gioco.

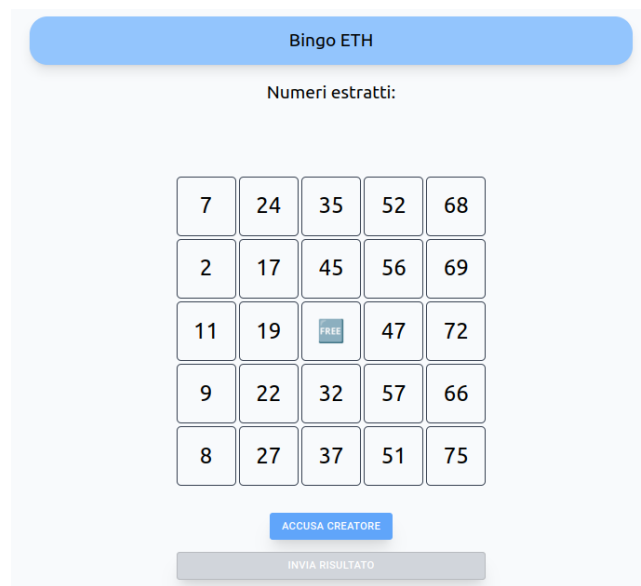


Figure 7: Schermata di gioco per i joiner

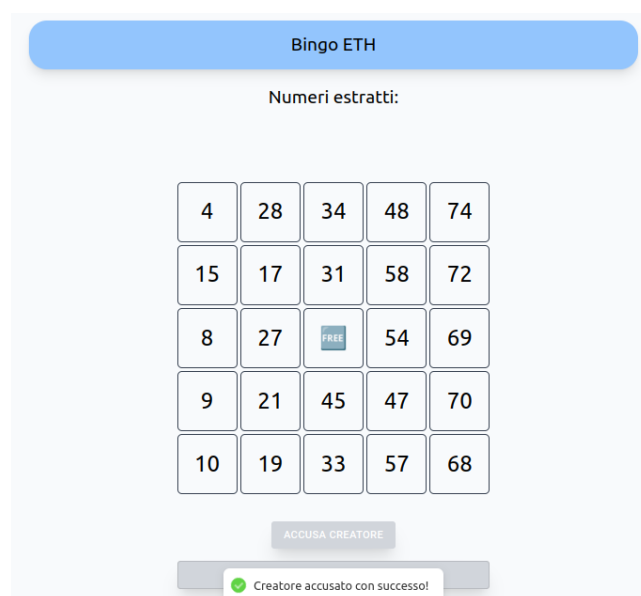


Figure 8: Schermata di denuncia per i joiner

3.2 Lato smart contract

Lo smart contract è scritto in Solidity e gestisce la logica del gioco, inclusa la gestione dei numeri estratti, la verifica delle cartelle dei giocatori e la distribuzione dei premi. Il contratto include le seguenti funzionalità:

- Creazione di una nuova partita e gestione dell'unione dei players alle partite.
- Gestione dei numeri estratti per ogni partita.
- Verifica delle vincite.
- Distribuzione dei premi.

Sotto vediamo i codici per queste funzionalita:

3.2.1 Funzione createGame

La funzione 'createGame viene chiamata per avviare una nuova partita. Verifica i parametri di input come il numero massimo di giocatori consentiti e l'ammontare della scommessa. Successivamente, genera un ID univoco per la partita e inizializza un nuovo oggetto di gioco nella lista. Vengono impostati vari parametri, tra cui il creatore della partita, l'ammontare della scommessa, il merkle root delle carte del creatore, eccetera. Infine, aggiunge la partita alla lista dei giochi disponibili e emette un evento per segnalare la creazione della partita.

```
function createGame(uint _maxJoiners, uint _betAmount, bytes32 _cardMerkleRoot)
public payable {
    // Requisiti di input
    require(_maxJoiners > 0, "Max joiners must be greater than 0");
    require(_betAmount > 0, "Bet amount must be greater than 0");
    require(msg.sender.balance/1 ether >= _betAmount,
        "Cannot bet more than you can afford!");
    require(msg.value == _betAmount*1 ether,
        "Please send exactly the amount you want to bet!");

    // Generazione dell'ID del gioco
    int256 gameId = gameId();

    // Creazione di una nuova istanza di gioco
    Info storage newGame = gameList[gameId];

    // Inizializzazione dei parametri del gioco
    newGame.creator = msg.sender;
    newGame.joiners = new address ;
    newGame.maxJoiners = _maxJoiners;
    newGame.totalJoiners = 0;
    newGame.ethBalance = 0;
    newGame.betAmount = _betAmount;
    newGame.creatorMerkleRoot = _cardMerkleRoot;
    newGame.accusationTime = 0;
    newGame.accuser = address(0);

    // Inizializzazione del mapping per il merkle root del creatore
    newGame.joinerMerkleRoots[msg.sender] = 0;

    // Aggiunta del gioco alla lista dei giochi disponibili
    elencoGiochiDisponibili.push(gameId);

    // Aggiunta del valore del betAmount all'ethBalance del gioco
```



```

newGame.ethBalance += _betAmount;

// Emit dell'evento GameCreated
emit GameCreated(
    gameId,
    newGame.maxJoiners,
    newGame.totalJoiners
);
}

```

3.2.2 Funzione extractNumber

La funzione `extractNumber` gestisce l'estrazione dei numeri nel gioco. Verifica se tutti i 75 numeri consentiti sono stati estratti. Genera un nuovo numero utilizzando una funzione per generare numeri casuali e controlla se è già stato estratto. Se il numero è un duplicato, ne genera uno nuovo fino a trovare un numero unico. Aggiunge quindi il numero estratto alla lista dei numeri estratti per il gioco corrispondente. Se viene specificato, resetta l'accusa e l'accusatore nel gioco. Infine, emette eventi per informare sul numero estratto o segnalare la fine del gioco se tutti i numeri sono stati estratti.

```

function extractNumber(int256 _gameId, bool accused) public {
    // Controlla se tutti i numeri sono stati estratti
    require(gameList[_gameId].numbersExtracted.length <= 75,
        "All numbers have been extracted!");

    // Genera un nuovo numero per il gioco
    uint8 newNumber = getNewNumber(_gameId);
    int8 i = 1;

    // Controlla se il numero è già stato estratto,
    // in tal caso genera un nuovo numero
    while (isExtracted(gameList[_gameId].numbersExtracted, newNumber)) {
        newNumber = getNewNumber(_gameId+i);
        i++;
    }

    // Aggiunge il nuovo numero alla lista dei numeri estratti per il gioco
    gameList[_gameId].numbersExtracted.push(newNumber);

    // Reset dell'accusa se il parametro accused è true
    if(accused){
        gameList[_gameId].accusationTime = 0;
        gameList[_gameId].accuser = address(0);
    }

    // Controlla se tutti i numeri sono stati estratti e emette eventi di conseguenza
    if(gameList[_gameId].numbersExtracted.length < 75){
        emit NumberExtracted(_gameId, newNumber, false);
    }else{
        emit GameEnded(_gameId, gameList[_gameId].creator,
            gameList[_gameId].ethBalance, WinningReasons.BINGO);
    }
}

```

4 Principali decisioni

4.1 Struttura dati

Per salvare tutte le informazioni sui dati riguardanti un game abbiamo scelto di optare per una struct, sotto il codice:

```
struct Info {  
    address creator;  
    address[] joiners;  
    uint maxJoiners;  
    uint totalJoiners;  
    uint ethBalance;  
    uint betAmount;  
    bytes32 creatorMerkleRoot;  
    mapping(address => bytes32) joinerMerkleRoots;  
    uint8[] numbersExtracted;  
    uint accusationTime;  
    address accuser;  
}
```

Qui possiamo vedere che ci salviamo l'address del creatore del game e di tutti i joiners, oltre che alle ovvie informazioni sul game come il tetto massimo di giocatori e il valore da scommettere in ethereum. Ci salviamo inoltre anche il MerkleRoot del creatore e di tutti i joiners attraverso un mapping. Le ultime variabili sono utilizzate rispettivamente per il controllo sui numeri estratti e sull'accusa dei giocatori per il creatore del game che ha la responsabilit  di estrarre i numeri del bingo.

4.2 Vincoli

I seguenti vincoli sono stati implementati per garantire il corretto funzionamento del gioco:

- **Numero delle caselle:** Il numero delle caselle   stato settato seguendo le regole base del bingo americano e quindi a 25 caselle totali.
- **La casella centrale:** La casella centrale   una casella jolly e quindi valida di base.
- **Il creatore:** Il creatore si assume il ruolo di chiedere al contratto di estrarre i numeri.
- **Limite scommesse:** I giocatori non possono scommettere piu di 1000 ETH.

4.3 Uso di Merkle Tree

Abbiamo deciso di utilizzare un Merkle Tree per garantire l'integrit  dei dati dei giocatori senza rivelare le informazioni sensibili durante la verifica delle vincite. Questo approccio consente una verifica efficiente e sicura delle cartelle di bingo.

4.3.1 Generazione del Merkle Tree

Il codice per la generazione del Merkle Tree si trova nella funzione `generateMerkleTree`.

Funzionalit : - La funzione prende in input una tabella (`table`), rappresentata come un array. - Ogni elemento della tabella viene convertito in un hash utilizzando `utils.soliditySha3` della libreria `web3`. - Gli hash risultanti costituiscono il livello delle foglie (`leaves`) del Merkle Tree. - Viene quindi calcolato il Merkle Tree iterativamente, concatenando e hashando le coppie di nodi fino a raggiungere la radice del Merkle Tree.

```
export function generateMerkleTree(table) {  
    console.log(table);  
    let merkleTree = [];  
  
    let tmp = [];
```

```

for (const element of table) {
    tmp.push(utis.soliditySha3(element.toString()));
}
merkleTree.push(tmp);

while (tmp.length > 1) {
    const nextLevel = [];
    for (let j = 0; j < tmp.length; j += 2) {
        if (tmp[j + 1]) {
            nextLevel.push(utis.soliditySha3((tmp[j] + tmp[j + 1].slice(2))));
        } else {
            nextLevel.push(utis.soliditySha3((tmp[j] + tmp[j].slice(2))));
        }
    }
    tmp = nextLevel;
    merkleTree.push(nextLevel);
}
return merkleTree;
}

```

Una particolarità dell calcolo del merkle tree in questo contesto è che il numero di elementi nella cartella del bingo non è una potenza del 2, infatti viene considerato un array di 24 elementi (non includiamo la casella centrale essendo sempre valida) rappresentante la cartella del giocatore:

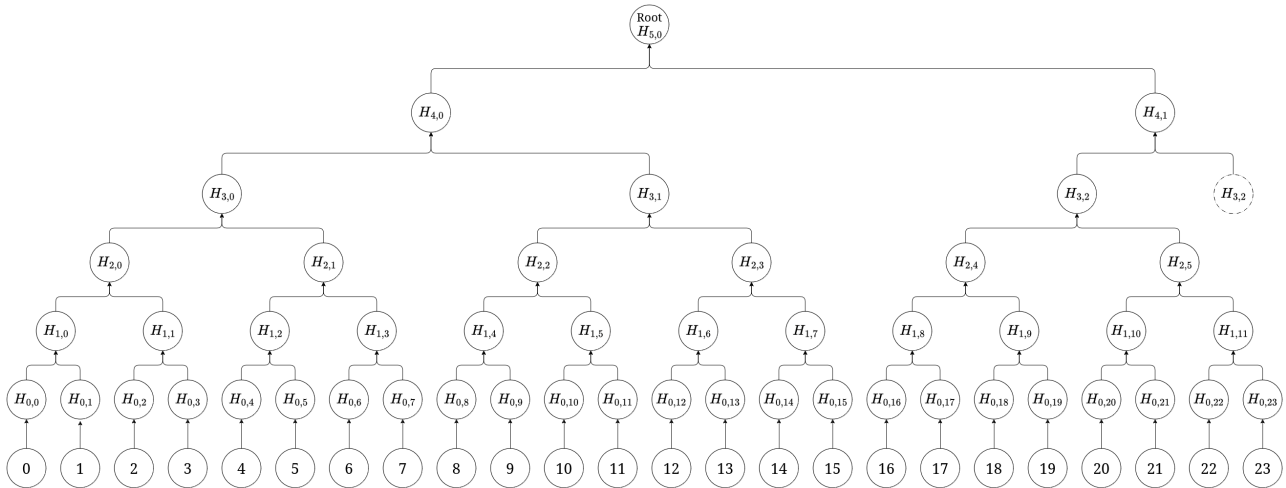


Figure 9: MerkleTree

Come possiamo vedere dal diagramma al livello 3, l'elemento con indice 2 viene duplicato per permettere il calcolo dell'elemento (4, 1). Questa soluzione è di semplice implementazione, infatti, durante il calcolo del merkle tree, è sufficiente controllare all'interno del ciclo for se si ha un elemento successivo a quello corrente con cui eseguire l'hash, altrimenti si esegue l'hash "raddoppiando" il nodo corrente:

```

for (let j = 0; j < tmp.length; j += 2) {
    if (tmp[j + 1]) {
        nextLevel.push(utis.soliditySha3((tmp[j] + tmp[j + 1].slice(2))));
    } else {
        nextLevel.push(utis.soliditySha3((tmp[j] + tmp[j].slice(2))));
    }
}

```

Così come nella generazione dell'albero, si deve prestare attenzione anche alla generazione delle proof. In caso di bingo, a seconda se la combinazione contiene o no la casella centrale, la proof consiste in un

array contenente tanti array quanti sono i numeri estratti da verificare:

```
const exampleProof = [
  [firstElement, elementIndex, H1, H2, H3, H4, H5],
  ...,
  [lastElement, elementIndex, H1, H2, H3, H4, H5]
]
```

La funzione generateMerkleProof prende in input la cartella card di 24 elementi e l'array result sempre di 24 elementi booleani, dove l'elemento i è true se l'elemento i della cartella fa parte di una combinazione vincente.

```
export const generateMerkleProof = (card, result) => {
  const proofs = [];
  const merkleTree = generateMerkleTree(card);
  const leaves = merkleTree[0];
  for (let i = 0; i < result.length; i++) {
    if (!result[i]) {
      continue;
    }
    const elementHash = utils.soliditySha3(card[i].toString());
    const index = leaves.indexOf(elementHash);

    let proof = [];
    let currentIndex = index;

    proof.push(stringToBytes32(card[i].toString()));
    proof.push(stringToBytes32(i.toString()));

    for (let level = 0; level < merkleTree.length - 1; level++) {
      const currentLevel = merkleTree[level];
      const isRightNode = currentIndex % 2 === 1;
      const siblingIndex = isRightNode ? currentIndex - 1 : currentIndex + 1;

      if (siblingIndex < currentLevel.length) {
        proof.push(`${currentLevel[siblingIndex]}`);
      }

      currentIndex = Math.floor(currentIndex / 2);
    }
    if (index > 15) {
      let last = proof.pop();
      proof.push(
        `${merkleTree[merkleTree.length - 3][merkleTree[merkleTree.length - 3].length - 1]}`
      );
      proof.push(last);
    }
    proofs.push(proof);
  }
  return proofs;
};
```

Come possiamo vedere, dopo il ciclo for che va a calcolare gli hash da inserire nella proof, si va a controllare se l'indice dell'elemento di cui si sta calcolando la proof è maggiore di 15. Se è maggiore di 15, infatti si dovrà aggiungere l'hash $H_{3,2}$ in penultima posizione così che durante la verifica si tenga

conto del *raddoppio* eseguito durante la generazione dell'albero.

Questa soluzione permette di verificare in modo classico una proof, la cui implementazione si trova all'interno del contratto:

```
function verifyMerkleProof(
    bytes32 _root,
    string memory _leaf,
    bytes32[] memory _proof,
    uint256 _index
) internal pure returns (bool) {
    bytes32 _hash = keccak256(abi.encodePacked(_leaf));
    // Starting from 2 to avoid resizing the proof array
    for (uint256 i = 2; i < _proof.length; i++) {
        if (_index % 2 == 0) {
            _hash = keccak256(abi.encodePacked(_hash, _proof[i]));
        } else {
            _hash = keccak256(abi.encodePacked(_proof[i], _hash));
        }
        _index /= 2;
    }
    return _hash == _root;
}
```

4.3.1.1 Esempio 1 Se si deve verificare un elemento con indice minore o uguale a 15, ad esempio 7 la sua merkle proof sarà:

$$[element, 7, H_{0,6}, H_{1,2}, H_{2,0}, H_{3,1}, H_{4,1}]$$

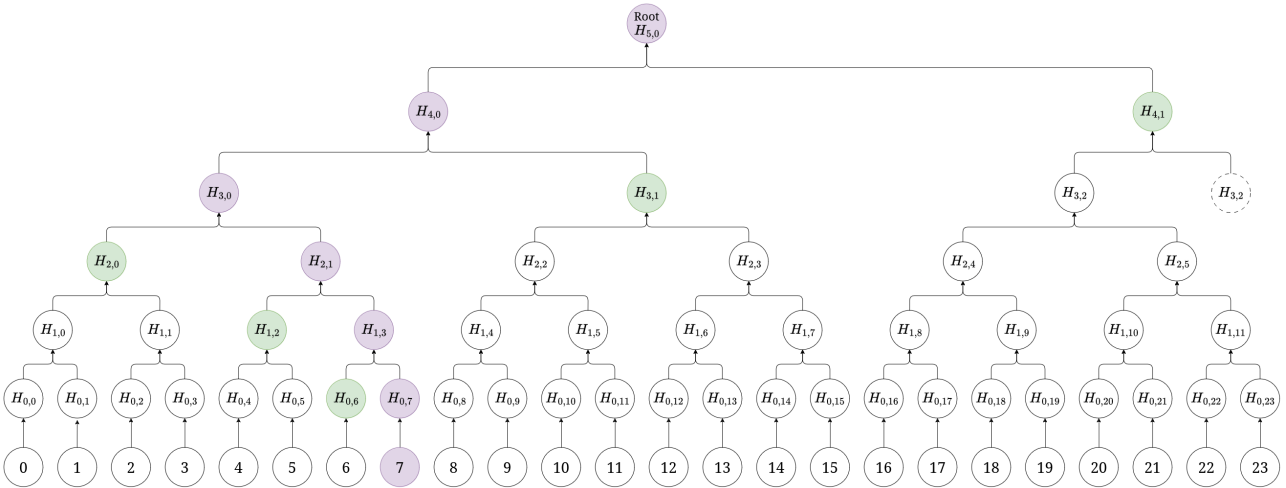


Figure 10: Merkle Proof dell'indice 7 - In verde gli hash forniti dalla proof, in viola quelli calcolati

4.3.1.2 Esempio 2 Se si deve verificare un elemento con indice maggiore di 15, ad esempio 20 la sua merkle proof sarà:

$$[element, 20, H_{0,21}, H_{1,11}, H_{2,3}, H_{3,2}, H_{4,0}]$$

Steps:

1. Clonare il repository da https://github.com/leomanne/P2P_Project o da <https://github.com/ski-by7/BingoEth> (il secondo repository è il mirror del primo) o scompattare l'archivio BingoEth.zip
2. Aprire il terminale/powershell e spostarsi dentro la directory del progetto
3. Eseguire il comando `docker compose up`

Eseguiti i passi indicati, inizierà il processo di build del container docker. Dopodiché si avvierà il container che compilerà il contratto, per poi avviare l'applicazione **BingoEth**. Una volta terminato il processo di build, sarà possibile accedere all'applicazione tramite browser all'url <http://localhost:80>.

Gli account verranno configurati su un'istanza di ganache locale (127.0.0.1:7545).

6 Valutazione del consumo di gas

- Il deploy del progetto si consuma una quantità di gas pari a 3653200 Gas
- La funzione CreateGame si consuma 179890 Gas
- La funzione JoinGame consuma 131088 Gas
- La sola funzione accuse consuma 67810 Gas
- La sola funzione di CheckAccuse consuma $\sim 1300 + (2500 \times n)$ Gas dove n = numero di joiners
- L'estrazione di un numero ~ 70705 Gas * un massimo di 75 numeri = massimo 5302875 Gas
- L'estrazione delle informazioni su un game ~ 36536 Gas
- Stima per la sottomissione di una board con SubmitBoard ~ 40181 Gas

6.1 Valutazione di esempio con un gioco con un creatore e 3 joiners

Ipotizziamo che ci siano stati 37 numeri per aver raggiunto il termine del gioco e che ci siano state 6 accuse (ovviamente non andate a buon fine). Calcoliamo il costo complessivo:

Per calcolare il costo complessivo del consumo di gas considerando le operazioni specificate, ipotizziamo i seguenti dati:

- **Creatore del gioco:** 1
- **Joiners:** 3
- **Numeri estratti:** 37
- **Accuse:** 6

6.1.1 Calcolo del consumo di gas:

1. **Deploy del progetto:** 3653200 gas
2. **Creazione di un gioco:**
 - 179890 gas
3. **JoinGame** per 3 joiners:
 - $131088 * 3 = 393264$ gas
4. **Accuse** per 6 accuse:
 - $67810 * 6 = 406860$ gas
5. **CheckAccuse** per 6 accuse e 3 joiners:
 - Ogni chiamata a **CheckAccuse** con 3 joiners:
 - $1300 + (2500 * 3) = 1300 + 7500 = 8800$ gas
 - $8800 * 6 = 52800$ gas
6. **Estrazione di numeri:**
 - Numero massimo di estrazioni: 37

- $70705 * 37 = 2616085$ gas

7. Estrazione delle informazioni su un gioco:

- 36536 gas (assumiamo 1 chiamata)

8. Sottomissione della board con SubmitBoard:

- 40181 gas (al caso pessimo)

6.1.2 Totale del consumo di gas:

Gas totale = $3653200 + 179890 + 393264 + 406860 + 52800 + 2616085 + 36536 + 40181$

6.1.3 Calcolo:

Gas totale = $7949016 + 52800 + 2616085 + 36536 + 40181$ Gas totale = $7949016 + 2716902 + 40181$
 Gas totale = $10656099 + 40181$ Gas totale = 10696280 gas

Quindi, il costo complessivo del consumo di gas per le operazioni descritte, considerando un creatore del gioco e 3 joiners, con 37 numeri estratti e 6 accuse, è di circa 10,696,280 gas.

Per calcolare quanto spende un joiner e quanto spende il creatore del gioco in base al consumo di gas specificato, consideriamo che i joiner possono eseguire tutte le operazioni elencate:

Dati forniti: - Creatore del gioco: 1 - Joiners: 3 - Numeri estratti: 37 - Accuse: 6

Consumo totale di gas: 10,696,280 gas

6.1.4 Calcolo del consumo di gas per ciascun partecipante:

6.1.4.1 Creatore del gioco:

1. **Deploy del progetto:** 3,653,200 gas
2. **Creazione di un gioco:** 179,890 gas
3. **JoinGame** per 3 joiners: 393,264 gas (si presume che il creatore del gioco non esegua JoinGame, quindi consideriamo solo il costo iniziale per la creazione)
4. **Accuse:** 406,860 gas
5. **CheckAccuse:** 52,800 gas (calcolato come 8800 gas per chiamata * 6 accuse)
6. **Estrazione di numeri:** 2,616,085 gas (37 estrazioni)
7. **Estrazione delle informazioni su un gioco:** 36,536 gas (1 chiamata)
8. **Sottomissione della board con SubmitBoard:** 40,181 gas (al caso pessimo)

Totale per il creatore: [$3653200 + 179890 + 393264 + 406860 + 52800 + 2616085 + 36536 + 40181$
 = 6763376 gas]

6.1.4.2 Joiner:

1. **JoinGame:** 131,088 gas (per ciascuno dei 3 joiners)
2. **Accuse:** 67,810 gas (per ciascuna delle 6 accuse)
3. **CheckAccuse:** 52,800 gas (stesso costo totale del creatore, poiché coinvolto solo come partecipante)
4. **Estrazione delle informazioni su un gioco:** 36,536 gas (1 chiamata)
5. **Sottomissione della board con SubmitBoard:** 40,181 gas (al caso pessimo)

Totale per un joiner: [$131088 + 67810 + 52800 + 36536 + 40181 = 329815$ gas]

6.1.5 Conclusione:

- **Creatore del gioco:** Spenderebbe circa **6,763,376 gas** per tutte le operazioni descritte.

- **Joiner:** Ogni joiner spenderebbe circa **329,815 gas** per le operazioni di JoinGame, Accuse, Estrazione delle informazioni su un gioco e Sottomissione della board con SubmitBoard.

7 Potenziali vulnerabilità

La funzione utilizzata per generare numeri casuali in uno smart contract utilizza l'hashing di dati come il timestamp del blocco, la difficoltà del blocco, l'indirizzo del mittente e un seed fornito. Tuttavia, la sicurezza di questa implementazione merita attenzione. L'hashing di parametri come il timestamp e la difficoltà del blocco non è completamente imprevedibile e potrebbe essere influenzato da attacchi che manipolano questi valori, compromettendo la casualità dei numeri generati. Inoltre, la funzione itera per trovare un numero non ancora estratto, il che può aumentare il consumo di gas e causare ritardi se il numero di tentativi è elevato.

7.1 Generazione di numeri casuali

```
function extractNumber(int256 _gameId, bool accused) public {
    uint startGas = gasleft();
    require(gameList[_gameId].numbersExtracted.length <= 75,
        "All numbers have been extracted!");
    uint8 newNumber = getNewNumber(_gameId);
    int8 i = 1;
    while (isExtracted(gameList[_gameId].numbersExtracted, newNumber)) {
        newNumber = getNewNumber(_gameId+i);
        i++;
    }
    gameList[_gameId].numbersExtracted.push(newNumber);
    if(accused){
        gameList[_gameId].accusationTime = 0;
        gameList[_gameId].accuser = address(0);
        emit ConfirmRemovedAccuse(_gameId);
    }
    if(gameList[_gameId].numbersExtracted.length < 75){
        emit NumberExtracted(_gameId, newNumber,false);
    }else{
        emit GameEnded(_gameId, msg.sender,
            gameList[_gameId].ethBalance * 1 ether, 0, true, WinningReasons.BINGO);
        payable(msg.sender).transfer(gameList[_gameId].ethBalance * 1 ether);
    }
    gameList[_gameId].weiUsed += (startGas - gasleft()) * tx.gasprice;
}

function getNewNumber(int256 seed) internal view returns(uint8) {
    uint256 randomHash = uint256(keccak256(abi.encodePacked
        (block.timestamp, block.difficulty, msg.sender, seed)));
    uint256 randomNumber = (randomHash % 75) + 1;
    return uint8(randomNumber);
}
```

La funzione `extractNumber` gestisce l'estrazione di numeri casuali per un gioco all'interno di un contratto intelligente. Inizia registrando il gas disponibile all'inizio dell'esecuzione. Verifica se il numero di numeri estratti per il gioco specificato è inferiore o uguale a 75. Se tutti i numeri sono stati estratti, interrompe l'esecuzione con un errore. Genera un nuovo numero casuale utilizzando la funzione

getNewNumber. Verifica l'unicità del numero attraverso un ciclo while, incrementando un contatore in caso di collisione e ottenendo un nuovo numero fino a trovare uno unico. Aggiunge il numero estratto all'array del gioco. Se l'argomento `accused` è vero, reimposta l'accusa nel gioco. Emette eventi per segnalare l'estrazione di un numero o la fine del gioco. Infine, calcola e aggiorna i costi di gas utilizzati durante l'esecuzione della funzione.