Instituto Superior Técnico
Avenida Rovisco Pais
1049-001 Lisboa, PT
tecnico.ulisboa.pt

REPORT

Master in Electrical and Computer Engineering

Multiparty Battleship Game Project Report

4th Period 2024/2025

Authors:

- Ramiro Moldes (99313) — marquesmoldes@tecnico.ulisboa.pt
- João Gonçalves (99995) — jrazevedogoncalves@tecnico.ulisboa.pt
- Teresa Nogueira (100029) — maria.teresa.ramos.nogueira@tecnico.ulisboa.pt

Last revision: June 13, 2025

CONTENTS

LIST OF FIGURES

I. INTRODUCTION

1 Project Overview

This project implements a privacy-preserving, multiplayer variant of the Battleship game using zkSNARKs (Zero-Knowledge Succinct Non-interactive Arguments of Knowledge) [1] to enforce the rules without relying on a third party, built on the RISC Zero architecture [2], a zero-knowledge verifiable general computing platform. The goal is to ensure that all players follow the game's logic, such as placing valid ship configurations, reporting shot outcomes truthfully, and winning only when conditions are satisfied, while keeping sensitive information (i.e., fleet configuration) private throughout the game.

We extend the standard two-player Battleship game into a peer-to-peer, turn-based multiplayer model. Each player has a private 10×10 grid, as shown in Figure 1.

	A	B	C	D	E	F	G	H	I	J
0	0	1	2	3	4	5	6	7	8	9
1	10	11	12	13	14	15	16	17	18	19
2	20	21	22	23	24	25	26	27	28	29
3	30	31	32	33	34	35	36	37	38	39
4	40	41	42	43	44	45	46	47	48	49
5	50	51	52	53	54	55	56	57	58	59
6	60	61	62	63	64	65	66	67	68	69
7	70	71	72	73	74	75	76	77	78	79
8	80	81	82	83	84	85	86	87	88	89
9	90	91	92	93	94	95	96	97	98	99

Figure 1. Battleship 10×10 board. Each player places their fleet following the project constraints, in order to generate their private board. Cells are identified with a number from 0 to 99 (total 100).

Each board is populated with a fixed set of ships (inventory of 18 cells):

- $1 \times$ Carrier (length of 5 cells); i.e., ■ ■ ■ ■ ■
- $1 \times$ Battleship (length of 4 cells); i.e., ■ ■ ■ ■
- $1 \times$ Destroyer (length of 3 cells); i.e., ■ ■ ■
- $2 \times$ Cruisers (length of 2 cells); i.e., ■ ■
- $2 \times$ Submarines (length of 1 cell). i.e., ■

The game is driven by seven actions (all very self-explainatory): `create`, `join`, `fire`, `report`, `wave`, `win`, and `contest`. Each corresponds to a specific transition (or lack of) in the game's state, as detailed further on, registered on a blockchain "emulator" (akin to a bulletin board).

Our implementation[1] of the full game logic and proving system in Rust is available in the following public `git` repository: https://github.com/Kons-5/IST-SCom (Accessed: Jun. 13 2025).

2 Document Organization

The remainder of the report is organized as follows:

- Chapter II. describes the game's core implementation[2]. It covers the fleet commitment scheme, game actions based on zkSNARK proofs, and how state transitions are handled;
- Chapter III. introduces additional security mechanisms[2], including digital signatures and a token-based turn management scheme.

[1]Based on skeleton provided by Prof. Carlos Ribeiro: github.com/ribeirocn/comm-security (Accessed: Jun. 13 2025).
[2]Some gameplay behaviors and edge cases are not discussed in detail, as they fall outside the scope of the report.

II. CORE IMPLEMENTATION

The game is setup in three phases:

- *Commitment Phase*: Players commit their fleet layout without revealing it. This phase is intrinsic to the `join` and `create`[3] game actions.
- *Commitment Update Phase*: Players take turns firing at each other and updating their board commitments accordingly. This phase is driven by the `fire` and `report` game actions.
- *Announcement Phase*: A player declares victory, and others may contest it. This phase corresponds to the `win` and `contest` game actions.

We note that the `wave` game action falls outside these defined phases, as it allows a player with the turn to forgo their move.

Each of the following sections approaches a phase of the game from a perspective on how to counter a dishonest player.

1 Fleet Commitment Scheme

When a player initiates the `join` game action (and the same can be said for the create game action), they must prove they have committed to a valid fleet layout without revealing it, this is done through a commitment scheme.

A commitment scheme allows one party to bind themselves to a particular set of data without revealing that data. The commitment can then be used to generate proofs which will verify if the input data matches the original commitment.

In the context of the game, the commitment is implemented as a hash of the player's fleet layout combined with a secret nonce. The player runs a zkVM (Zero-Knowledge Virtual Machine) program which enforces the fleet layout to be valid, according to a set of rules, as illustrated in Figure 2. If the layout is invalid, the guest execution halts and no receipt is produced.

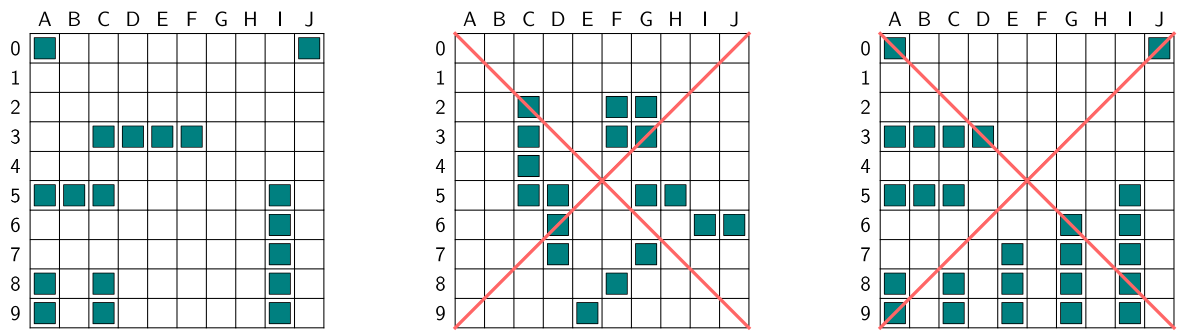


Figure 2. A set of three battlefield grids. The first shows a valid fleet layout based on specifications defined in Chapter I. The second and third illustrate violations: (1) ships placed diagonally or touching sides (invalid shapes), and (2) exceeding the allowed number of ships.

The RISC Zero zkVM then outputs a receipt, which proves that a specific program (identified by its image ID) executed correctly and committed the expected hash.

The verifier (e.g., the blockchain) checks the receipt against the agreed guest program to confirm that the commitment was generated honestly, without ever revealing the fleet itself. Once verified, the commitment is stored on-chain.

[3]Although the game begins with a `create` action and other players join later via `join`, both actions share the same implementation logic in the Rust code implementation of the host-side.

1.1 Exhaustive Search

Computing the number of possible Battleship fleet configurations is generally a difficult problem however, we may establish an uppen bound assuming all 18 pieces of ship behave independently. We can estimate the entropy of a 10×10 battleship grid with 18 one-square battleships.

The Shannon entropy is given by

$$H(X) = -\sum_{i=1}^{N} p_i \log p_i,$$

where p_i is the probability of the i^{th} fleet layout. Assuming all layouts are equally likely ($p_i = 1/N$, for all $i = 1, \ldots, N$), we get

$$H(X) = -\sum_{i=1}^{N} \frac{1}{N} \log N = \log N.$$

The number of ways of putting 18 cells in 100 squares (non-overlapping, but without further restrictions) is given by:

$$N = \binom{100}{18} = \frac{100!}{18!(100-18)!} \approx 3 \times 10^{19}.$$

Thus,

$$H(X) \approx \log_2(3 \times 10^{19}) \approx 65 \text{ bits}.$$

However, this is a theoretical upper bound. In practice, the fleet must conform to strict game rules (e.g., ship sizes, no diagonal placement, no adjacent ships), meaning the actual number of valid configurations is much smaller.

As pointed out in this Arstechnica study article , a modern attacker with high-end GPUs can perform hundreds of billions of guesses per second ($\approx 10^{12}$). Thus, without a nonce, the hash of the fleet layout would be vulnerable to brute-force inversion. An adversary could simply iterate over all valid fleet configurations, hash them, and compare the result with the commitment.

To address this, a secret nonce is included in the hash, as seen in Figure 3 making brute force infeasible.

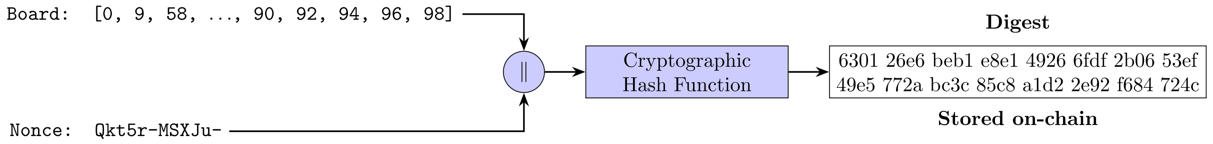

Board: [0, 9, 58, ..., 90, 92, 94, 96, 98]
Nonce: Qkt5r-MSXJu-
||
Cryptographic Hash Function
Digest
6301 26e6 beb1 e8e1 4926 6fdf 2b06 53ef
49e5 772a bc3c 85c8 a1d2 2e92 f684 724c
Stored on-chain

Figure 3. Generation of the hash of the fleet position together with some random nonce.

This way, the commitment scheme elegantly ensures players can't cheat or reveal their strategy.

2 Fleet Commitment Update

2.1 Fire Game Action

Once all players have joined the game, the fire game action can be taken. During this phase, a player targets an opponent's grid. The fire game action require proof that the fleet is not sunk. From the *host's perspective*, we assume that the player has not altered the board committed at the start of the game, so the the zkVM guest program makes a simple check to see if there still are ships placed on the board. Figure 4 below illustrates this check.

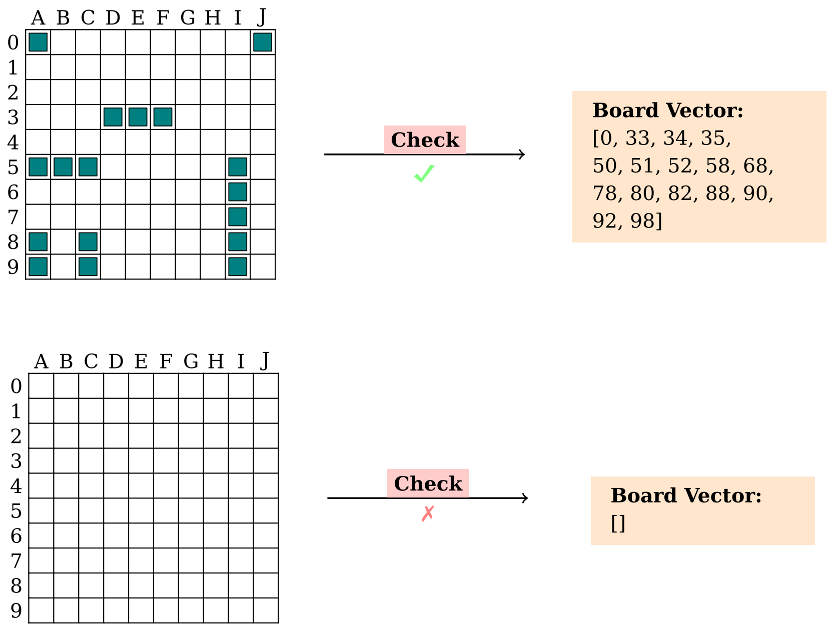


Figure 4. Host-side validation of board integrity. The top example shows a valid board with it's matching vector, allowing the program to proceed with the proof. In the bottom case, the empty board indicates a fully sunk fleet, preventing further action.

Note that the board is represented internally as `pub board: Vec<u8>`, where each value corresponds to a specific cell in the grid. Therefore the board is a flat list of cell indices (0–99) occupied by ships. Therefore, it is sufficient to do:

Check used to see if fleet is not sunk

```
assert!(
    !input.board.is_empty(),
    "Your fleet is fully sunk. Cannot fire!");
)
```

Subsequentially, from the *blockchain's perspective*, since the board may have been altered by the player, we compare the committed board digest stored on-chain with the one submitted in the journal during the fire game action.

2.2 Report Game action

After the fire game action is taken, the opponent must report whether the shot was a hit or a miss. The report proof must:

- Confirm truthfulness of the reported outcome of the shot based on their committed fleet layout.

- Update the fleet commitment to reflect the new state after the shot, if applicable.

Furthermore, the journal used to advertise the report method includes two fields related to the board state: one contains the hash of the board before the shot was applied, and the other contains the updated hash after the shot. The latter is intended to replace the previously stored commitment on the blockchain:

Struct used to specify the output journal for report method

```
pub struct ReportJournal {
    pub gameid: String,
    pub fleet: String,
    pub report: String,
    pub pos: u8,
    pub board: Digest,
    pub next_board: Digest,
    ...
```

To complete the report action, the following logic in Section 2.2.1 and Section 2.2.2 is applied.

2.2.1 Host perspective

From the host's perspective, we assume that the shot details reported by the player match those initiated by the opponent, and that the player has not altered the board committed at the start of the game.

1. The zkVM guest program receives as input the claimed result of the shot (either "Hit" or "Miss"), the current board layout (after the shot has been applied), the random nonce, and the position of the shot.
2. The logic now splits depending on whether the player claimed "Hit" or "Miss":

If "Miss" is claimed: The program verifies that the reported shot position is not present in the board vector. If the position is found, it indicates one of two possible forms of dishonesty:

1. The player falsely claimed a "Miss" when the position actually corresponds to a ship segment, meaning the shot was a "Hit".
2. The player altered the committed board state after being shot at.

If the claim is true, the board has not changed (the shot hit water). The program hashes the board and nonce, and outputs that as both the previous and current commitment hashes. Figure 5 and Figure 6 illustrate a honest and a dishonest miss report, respectively:

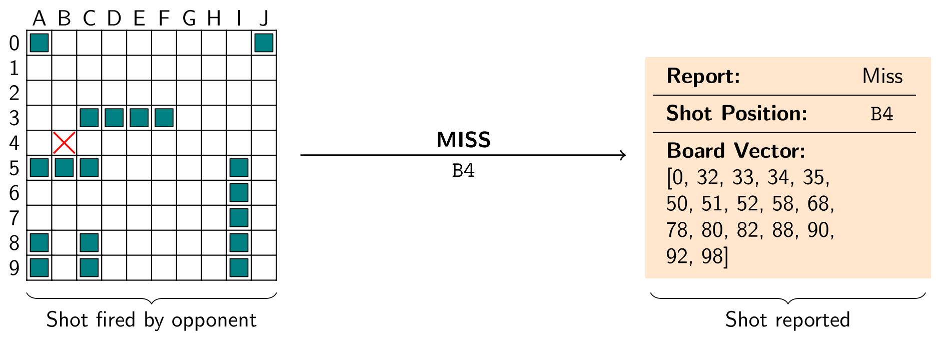


Figure 5. Example of a correct "Miss" report. The opponent fires at position B4, which does not correspond to any ship segment, this is confirmed by the absence of B4 in the board vector. As a result, the proof proceeds, and the program outputs the same board hash before and after the shot.

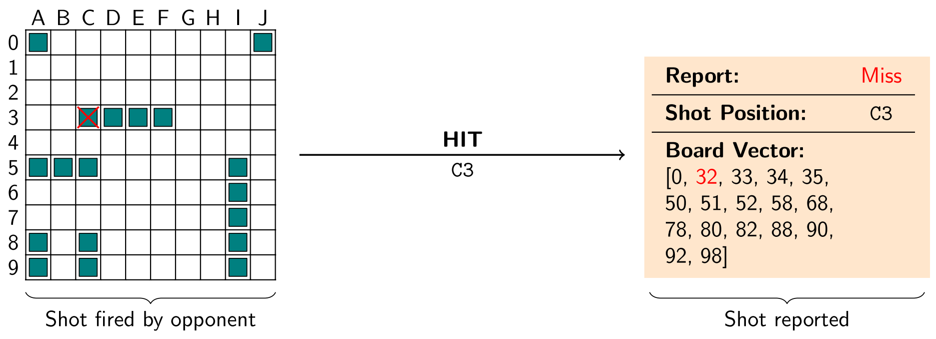


Figure 6. An invalid "Miss" report. Although the player claims a miss at C3, the position is still present in the board vector, indicating a hit. The proof fails and no valid receipt is generated.

If "Hit" is claimed: Similarly to "Miss", the program check if the reported shot position is not present in the board vector, this time, to Verify that the position was removed from the board upon registering the hit position. If the position is found, it indicates that:

- The player either forgot to remove the fleet position from their board or tried to forge the report by keeping the by keeping the ship segment active. If this happens The proof fails and no valid receipt is generated.

If the check passes, the program reconstructs the previous board by re-adding the shot position to the board vector. It then hashes both the previous and updated boards (with the nonce), producing two distinct commitment hashes. Both hashes are committed to the journal to reflect the transition. Figure 7 illustrates a honest hit report:

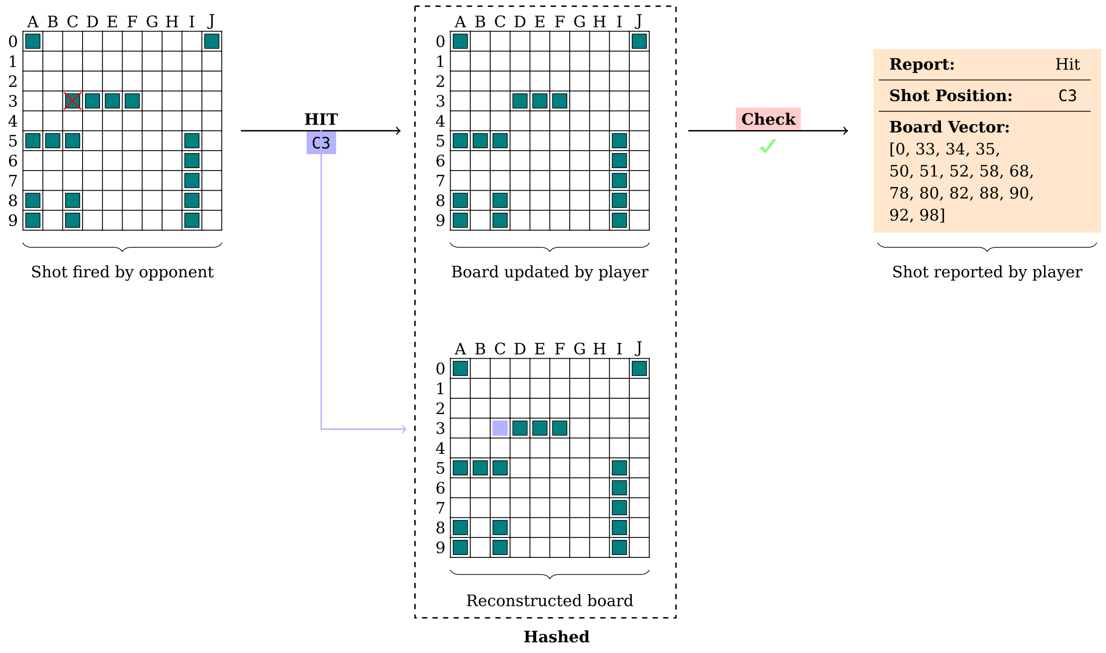


Figure 7. A valid "Hit" report. The opponent fires at C3, and the player correctly removes the hit position from the board vector. The transition is validated and subsequentially, the zkVM reconstructs the original board by re-adding C3 and computes hashes for both states, which are then are committed to the journal.

So far, we have only considered the host's perspective, which verifies that the report is consistent with the board state and claimed shot. However, a player may still attempt to forge the report by substituting the original shot from their opponent with a different one, then generating a proof for that altered position. To prevent this, the blockchain must intervene by enforcing consistency between the reported shot and the one originally fired.

2.2.2 Blockchain's Perspective

From the blockchain's perspective, the following validations are performed:

1. Ensure that the position reported by the player matches the one previously fired by the opponent. This prevents players from forging the shot location in their report. To do this, the blockchain mantains a log of the last shot fired:

   Game structure in the blockchain
   ```
   pub struct Game {
       pub pmap: HashMap<String, Player>,   // All players in the game
       pub shot_position: Option<u8>,       // Last shot position
       pub pending_win: Option<PendingWin>, // If someone has claimed victory

       // Token authentication
       pub turn_commitment: Option<Digest>,
       pub encrypted_token: Option<String>,
   }
   ```

   If the reported shot position does not match the one recorded by the blockchain, the game action is rejected and the report is considered invalid. The player must then submit a new report using the correct shot position.

2. Confirm that the board commitment sent in the report matches the player's last known committed state. This ensures that the player has not tampered with the board after being shot at.

If both of these checks are passed successfully, the initial commitment is updated with a new commitment which reflects the effect of the shot and the player can now take a new fire game action to progress the game.

3 Announcement Phase

3.1 Win Game Action

When a player believes they've achieved victory (meaning their fleet is still afloat while all others have been sunk) they may invoke the `win` game action.

This action, however, isn't simply a declaration; it must be backed by a valid zkVM receipt proving two key claims:

1. The player's fleet is *not fully sunk*;
2. The state of their board matches the last known commitment stored on-chain.

From the host's perspective, this is enforced by checking (similarly to `join`) that the board is not empty and then generating a fresh hash commitment:

Win condition: fleet must not be sunk
```
assert!(
    !input.board.is_empty(),
    "Your fleet is fully sunk..."
);
```

The host then outputs a journal containing the updated board commitment. This digest is verified by the blockchain against the player's last known on-chain commitment. If the two match, and no action is pending (i.e., `shot_position` is `None`), the system registers a `pending_win`. This struct acts as a public claim: it captures *who* declared victory, *when* they did it, and *what* the final committed board looked like.

Pending win note added to the game

```rust
pub struct PendingWin {
    pub claimant: String, // Fleet ID that claimed win
    pub board: Digest,    // Committed board hash
    pub time: Instant,    // Time when claim was made
}
```

Any deviation, as stated, such as mismatched board hashes, unresolved shots, or invalid receipts, results in rejection of the win attempt.

3.2 Contest Game Action

The `contest` action allows any player to dispute a victory claim by proving that their own fleet is still intact. If successful, this invalidates the most recent `win` and clears the `pending_win` note associated with the respective game.

With a similar construct to other game actions, the zkVM enforces the following during `contest` verification:

1. The challenger's fleet must not be fully sunk (`input.board.is_empty()` must fail);
2. The board hash must match the on-chain commitment.

This mechanism ensures that victory is only granted/registered when no player can prove otherwise, making the win claim subject to peer validation. Since players aren't required to announce when they're sunk, this decentralized contest option becomes a final safeguard against premature or dishonest victory declarations even if by legitimate players.

4 Turn Skipping

4.1 Wave Game Action

While the `fire` and `report` actions drive the core mechanics of Battleship, the `wave` action, besides the immediate purpose, introduces a layer of strategy and flexibility.

A player may issue a `wave` either because their fleet is completely sunk, therefore making them unable to `fire`, or for strategic reasons: to appear inactive/"dead" to other players. This deliberate silence can be used to avoid attention or mislead opponents about their remaining strength (in a multiparty sense). Regardless of motive, we enforce that:

1. The committed board must match the last known on-chain state;
2. Having the turn, there must be no pending shot to report (i.e., `shot_position == None`), even if the player knows the adversary's shot will result in a fully sunk fleet configuration.

From the host's perspective, the zkVM simply recomputes the board commitment (even if empty), and checks that the player is allowed to wave. **Note:** the fleet can be empty, but the action is still considered valid so long as the hash reflects an honest game state.

From the blockchain's perspective, the `wave` action is accepted if the player holds the current turn, no shot is pending, and the submitted board commitment matches the one stored.

If all conditions are met, the wave is accepted, and the turn is passed forward.

III. EXTRA FUNCTIONALITIES

1 Digital Signatures

In Star Trek, *Dilithium* is a rare material that cannot be replicated. Similarly, signatures cannot be replicated or forged: each one is unique.

To ensure message authenticity and prevent impersonation in our Battleship-based game, each message exchanged between players is digitally signed using the CRYSTALS-Dilithium2 signature scheme (specified in [3]), a post-quantum cryptography (PQC) National Institute of Standards and Technology (NIST) finalist based on the hardness of lattice problems (module learning with errors and module short integer solution).

The Dilithium process relies on arithmetic over the ring $R_q = \mathbb{Z}_q[X]/(X^n + 1)$ with parameter set tuned for NIST Level 2 (hence, Dilithium2).

At a high level, digital signatures follow three phases: key generation, signing, and verification; as seen below in Figure 8.

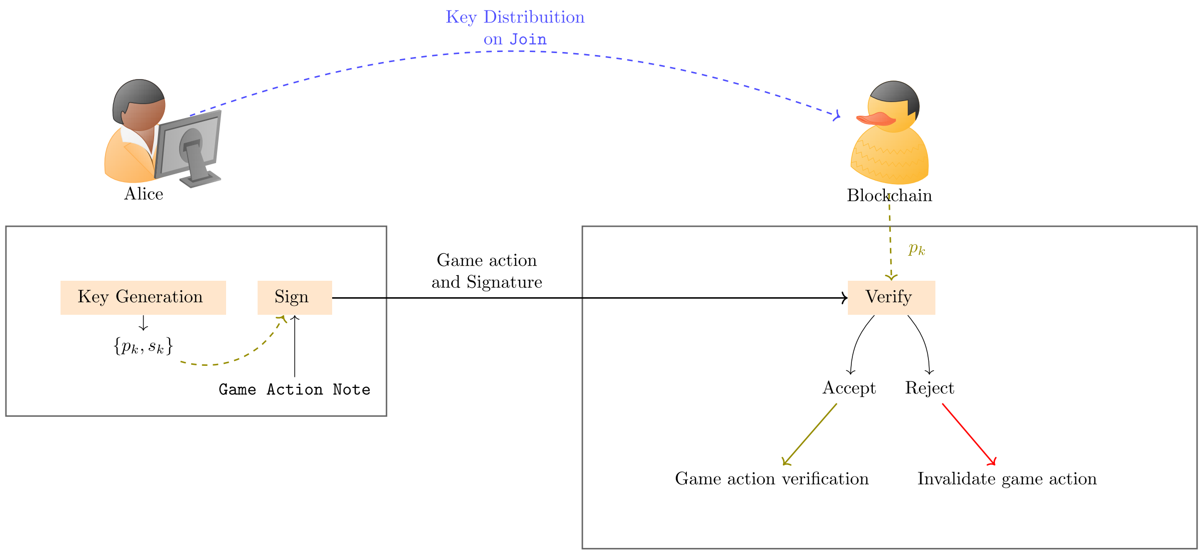


Figure 8. Digital signature logic example between a player and the blockchain.

1. Key generation in Dilithium begins with a random seed that is expanded into a public matrix $\mathbf{A}$ and two short secret vectors $\mathbf{s}_1$ and $\mathbf{s}_2$, used to compute a public vector $\mathbf{t} = \mathbf{A}\mathbf{s}_1 + \mathbf{s}_2$, which is compressed to reduce key size. The public key (p_k) includes the matrix seed and the rounded high-order part of $\mathbf{t}$, denoted as $\mathbf{t}_1$; while the secret key (s_k) includes the original secrets $\mathbf{s}_1$, $\mathbf{s}_2$, the low bits $\mathbf{t}_0$, and hash commitments used during signing.
2. To sign a message, the signer first computes a hash $\boldsymbol{\mu}$ that binds the message $\mathbf{M}$ to their public key. A short masking vector $\mathbf{y}$ is sampled, and the product $\mathbf{w} = \mathbf{A}\mathbf{y}$ is computed. From this, a challenge polynomial c is derived from $\boldsymbol{\mu}\|\mathbf{w}_1$ using a hash function, where $\mathbf{w}_1$ are the high bits of $\mathbf{w}$, following the Fiat–Shamir with aborts paradigm. This replaces an interactive challenge-response protocol (like Schnorr's) with a hash-based challenge.

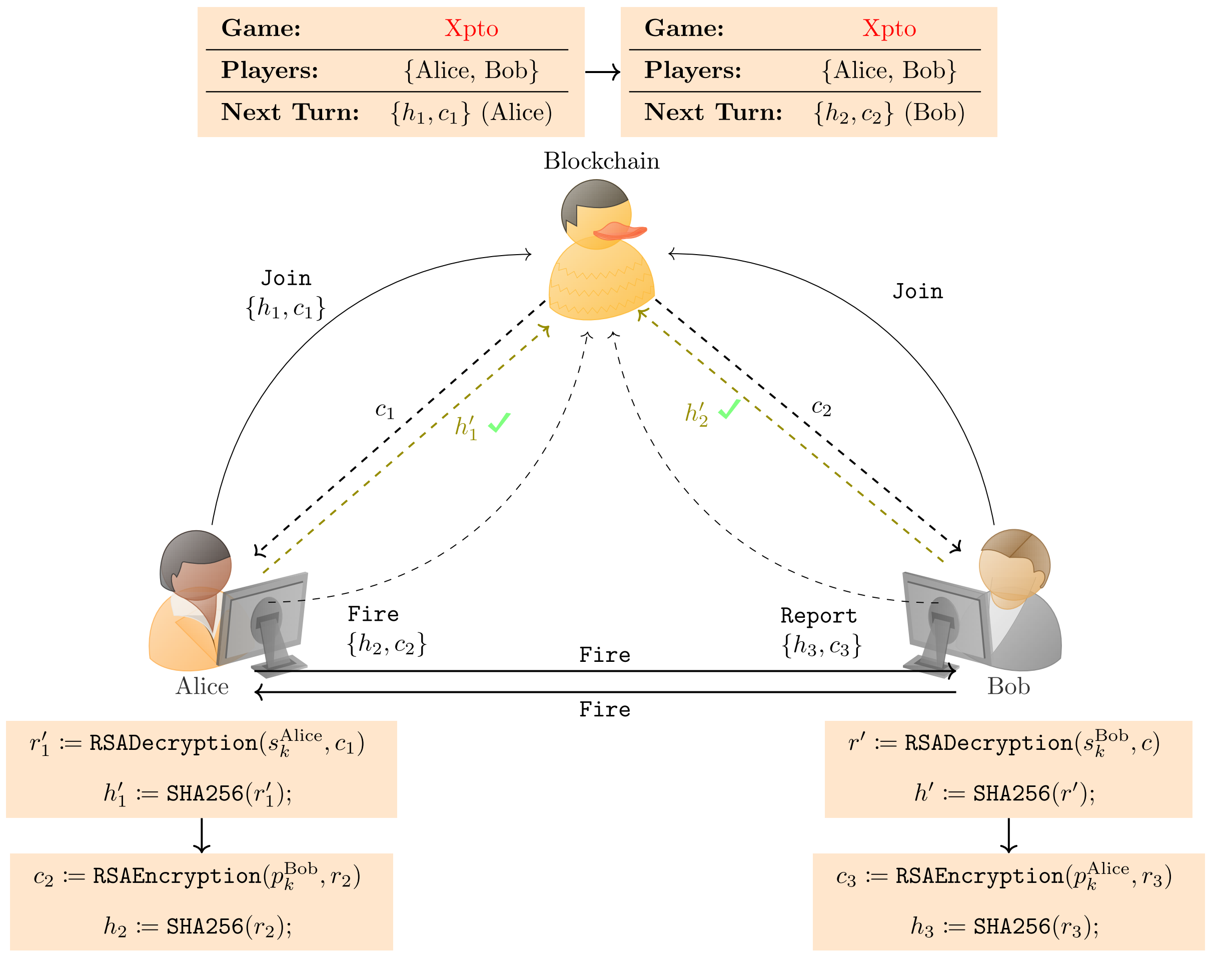
   The signer then computes the response $\mathbf{z} = \mathbf{y} + c\mathbf{s}_1$. If $\mathbf{z}$ exceeds bounds, the process restarts (rejection sampling). A hint $\mathbf{h}$ is added to help the verifier recover necessary information during verification. The signature is output as $\sigma = (\mathbf{z}, \mathbf{h}, \tilde{c})$, where $\tilde{c}$ is a compact encoding of c.

3. To verify a signature, the verifier recomputes the message hash and challenge c, then checks whether the response $\mathbf{z}$ and associated data match expectations. It reconstructs an expected value $\mathbf{w}_1' = \mathbf{A}\mathbf{z} - c\mathbf{t}_1 \cdot 2^d$, applies the hint $\mathbf{h}$, and verifies that the hash of this result matches the original challenge. If all bounds and checks pass, the signature is accepted.

   This ensures the signer knew $\mathbf{s}_1$, without revealing it (similar in structure to Schnorr identification protocol, but made non-interactive via the Fiat–Shamir transformation).

2 Token-based Turn Management

To securely enforce turn-taking in our multiparty Battleship game, we implemented a token-based mechanism inspired by commitment schemes and public-key encryption. The goal is to ensure that only the correct player can act on their turn, while preserving privacy and maintaining verifiability on-chain. The token-based turn mechanism is illustrated in Figure 9.



Game: Xpto
Players: {Alice, Bob}
Next Turn: $\{h_1, c_1\}$ (Alice)
Game: Xpto
Players: {Alice, Bob}
Next Turn: $\{h_2, c_2\}$ (Bob)
Blockchain
Join $\{h_1, c_1\}$
Join
c_1
h_1' ✓
h_2' ✓
c_2
Alice
Bob
Fire $\{h_2, c_2\}$
Report $\{h_3, c_3\}$
Fire
Fire
$r_1' :=$ RSADecryption$(s_k^{\text{Alice}}, c_1)$
$h_1' :=$ SHA256(r_1');
$c_2 :=$ RSAEncryption(p_k^{Bob}, r_2)
$h_2 :=$ SHA256(r_2);
$r' :=$ RSADecryption(s_k^{Bob}, c)
$h' :=$ SHA256(r');
$c_3 :=$ RSAEncryption$(p_k^{\text{Alice}}, r_3)$
$h_3 :=$ SHA256(r_3);

Figure 9. Token-based turn management example for two players, as explained below for the join action and two subsequent turns of fire and report actions.

Each turn is authorized by a token, represented as a random nonce $r \in \{0,1\}^{256}$. The player who initiates the turn generates this token (each action generates a fresh nonce), hashes it and encrypts it with the public key of the intended next player:

$$h := \mathsf{SHA256}(r), \quad c := \mathsf{RSAEncryption}(p_k^{\text{next}}, r).$$

The encrypted token and its hash (used as a commitment) are then published to the blockchain.

1. On game setup, the first player joining generates the first token, encrypts it with their own public key, and stores the encrypted blob along with its hash on-chain.
2. Once other players join, and the first shot can be fired, the first player joining generates a new token and encrypts it with the public key of the player being fired at. This encrypted token becomes the embedded capability for the next move. The targeted player must report the result of the shot, and since they are the one responding, they will also act next, so the token both authorizes and hands over the turn.

Only the intended recipient of the turn token (i.e., the target player) can decrypt it using their private key to recover the raw nonce

$$r' := \mathsf{RSADecryption}\left(s_k^{\text{target}}, c\right).$$

This decrypted nonce, r', is then passed privately into the RISC Zero zkVM, where:

- The nonce is hashed inside the zkVM

$$h' := \mathsf{SHA256}(r');$$

- The hash is matched against the on-chain commitment (i.e., the published hash),

$$h' \overset{?}{=} h;$$

- If correct, the zkVM permits the action (i.e., fire, report hit/miss, wave) and produces a valid zero-knowledge proof.

Token ownership validation inside zkVM

```
let hash = Sha256::digest(&auth.token); // This is the hash of r' (decrypted token)
let digest = Digest::try_from(hash.as_slice()).expect("Invalid hash size");
assert_eq!(
    &digest, &auth.expected_hash,
    "Token mismatch: you do not own the turn"
);
```

The zkVM outputs a verifiable receipt showing that:

- The action was computed correctly and that the player was authorized to act.;
- The token matched the expected commitment.

This proof is posted to the blockchain, and any player or observer can verify that the correct participant took their turn, without seeing the actual token (secret).

IV. REFERENCE DOCUMENTS

[1] M. Petkus, “Why and How zk-SNARK Works.” Accessed: Jun. 13, 2025. [Online]. Available: https://doi.org/10.48550/arXiv.1906.07221

[2] RISC Zero, “RISC Zero: Zero-Knowledge Virtual Machine.” Accessed: Jun. 13, 2025. [Online]. Available: https://risczero.com/

[3] S. Bai, L. Ducas, E. Kiltz, T. Lepoint, V. Lyubashevsky, P. Schwabe, G. Seiler and D. Stehlé, “CRYSTALS-Dilithium: Algorithm Specifications and Supporting Documentation.” Accessed: Jun. 13, 2025. [Online]. Available: https://pq-crystals.org/dilithium, Version 3, submission to NIST post-quantum cryptography standardization project