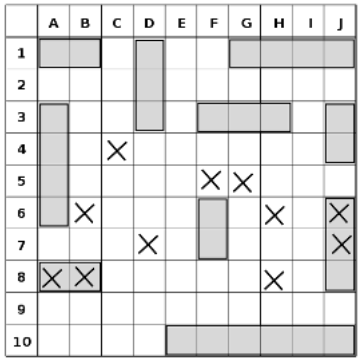
Communications Security Course Distributed

Multiparty Battleship Game
version 3.0

Project
2024/2025
zk-SNARK



1 Goal

The students should learn by experience what it is and how to use "Zero-Knowledge Succinct Non-interactive ARgument of Knowledge" (zkSNARK).

The students will build a distributed application that implements a multi-party version of the popular Battleship game.

2 Game Description

In the well-known two-party battleship game, both parties have a fleet of boats secretly displaced over a square board of 10x10 squares. The boats are of different sizes depending on their type. The Carrier is the biggest one and occupies five squares of the board. The Battleship occupies 4, the Destroyer 3, The Cruiser 2, and the Submarine 1. The number of boats of different types that each player must position on the board differs from version to version, but for the present purpose, the students may assume two submarines, two cruisers, and one of the other types of ships (7 boats total). In this multiparty variant, all the players must define the position of their fleet on its board at the beginning of the game.

In the two-party variant, each player takes turns firing one or a set of shots[1] to some chosen positions at the adversary's board. Because the position of the adversary's fleet is secret, some shots will hit the water, and some will hit the boats. On receiving the shots, the adversary must report accurately which shots hit the water and whose shots hit the vessels. In this multiparty variant, a player may only take shots at the other players' fleets if she is the player who starts the game or if she received a shot. To fire a shot, a player chooses another player's board and takes a shot at that fleet. Upon receiving a shot the player reports the result and takes another shot to some fleet.

The game ends when every player fleet but one is sunk. A fleet is sunk when all the vessels are sunk, and a ship is sunk when some adversary hits every square of the boat.

There are three conundrum situations:

- When a shot hits some vessel's square already hit by some player, the shot should be reported as water. There is no vessel there anymore.
- A player does not need to announce that all his fleet has been sunk. It is the responsibility of the winner to claim victory.
- A player with no fleet cannot shoot other vessels. Therefore, he cannot take a shot when receiving the turn (by another shooting at his board). Instead, he will wave his turn to the player without it for longer. If that player's fleet is sunk, it waves again until someone takes a shot or claims victory.

3 Project Requisites

The project has several restrictions that must be satisfied.

1. There should be no trusted third party.

[1] In some variants each player can fire one shot at a time, in other she may fire three or more.

The traditional way to build a battleship game is to have a trusted service that knows every player's fleet, ensures that each player only shoots on their turn and that the fleet never changes, and reports accurately on every shot fired by the players.

The goal is to achieve the same level of trust without this trusted service.

2. Each player operates a local application that interacts with the other players' applications in a peer-to-peer way.
3. The position of each player's fleet should be known only by the player's application and never revealed to anyone else.
4. Every player must be assured that the other players' fleets are well placed (i.e., all the boats are on the board).
5. Every player must be assured that the other players' fleets never change.
6. Every player must be assured that the player taking the shot has conditions to do so, i.e., some of his boats are not sunk.
7. Every player must be assured that the player receiving a shot accurately reports a hit or a miss.

4 Project Architecture

From an architectural point of view, the most relevant information taken from the game description is that every participating node takes turns performing some actions and that those actions are visible to every other party. It is as if every participating node takes turns publishing a note on a bulletin board, and from that note, it is evident to all who should take the next turn. This kind of distributed architecture pattern is currently implemented on a blockchain.

However, to minimize the efforts in project development, the students will not be required to implement the project on a blockchain. Instead, they will assume that every player's node behaves correctly when approving or rejecting the publishing of a note from some other node. For instance, if someone tries to fire at someone else's board and does not have the turn, then every node will reject it. On a blockchain, it would be required that only a majority of the nodes would behave correctly. Note that although it is assumed that every node behaves correctly in accepting or rejecting the notes published by the other nodes, it does not mean that they will behave correctly when positioning the fleet, firing or reporting the shots, or even publishing notes when they should not. The only simplification is on the matters that the blockchain would solve.

With this simplification assumption, the blockchain may be replaced by a messaging system where every note published is sent to a central service that simulates the blockchain. The blockchain service is not trusted to know the fleets' positions or to accurately report the result of each shot.

The blockchain simulator will print the result of every note published on it by every player. If some player fires a shot and is not his turn a message will appear saying that it was an out-of-order fire. If it has the turn, the service will print the fire on the screen, but it will not validate if it hits water or a ship.

The player is responsible for reading the information on the screen of the blockchain simulator and interacting with his application to report the result. He would reply that the shot taken by player A at position X,Y hit water/boat. The player application will generate a proof and send it to the blockchain simulator to verify.

The blockchain simulator will verify the file and publish the result on screen, at the same time that it says who has the turn. For this purpose, the blockchain simulator must keep a list of open games. For each game, it should keep a list of players in the game, and the player with the turn. For each player, it should keep two commitments. One for the fleet's position and one for the current fleet status (i.e. success shots on the player ships.).

Every player's application will be equal. Each instance of a player application can be in a single game at the same time. The player's application only needs to keep.

- The Blockchain simulation Service defined by a URL
- The name of the game that it is in
- The player's fleet id.
- The fleet position
- The fleet status

All this information should be kept in the browser's local storage. The Host application does not need to contain the players information. This solution simplifies the process of creating a new game or joining an existing one. The player only needs to provide the fleet position and the fleet status. The player's application will generate the commitments and send them to the blockchain simulator. The blockchain simulator will validate the commitments and store them. The player's application will keep a copy of the commitments in the local storage.

The names of the games can be requested from the Blockchain simulator as if it were a real blockchain, or they could be read by the player from the Blockchain simulation screen and entered by hand whenever necessary.

The number of players in the current game can also be requested from the blockchain service or read from the screen of the blockchain service.

The position and status commitments of the fleet can also be taken from the blockchain or generated appropriately by the player's application, whenever necessary.

Initially, the application asks the user if he wants to start a new game or join an existing one. Every game has a name. If a player wishes to join a game, she will use the name to identify the game. The players know the game names

by reading the blockchain screen or by having their application request those numbers from the blockchain (if that choice is implemented). For simplicity, after creating or joining a game, the player can only exit after the game ends.

When creating or joining a game, the player must provide the position of her fleet. This can be done using the graphical interface provided in the project skeleton.

On receiving a game creation or joining message, the blockchain application updates a record of every player participating in every game.

When the player receives the turn, she can fire at some fleet or wave her turn. If the player chooses to fire, she should provide the other player's identifier and the coordinates x and y of the shot. To fire the player must prove to have unsunk boats, but to wave no proof is necessary.

The player receives the turn to fire when it receives a shot from a legitimate player. The player is legitimate if it is her turn to send a note. The blockchain simulator keeps a record of the player with the turn, so it can validate incoming notes.

In the beginning, the player who created the game has the turn to fire. After firing at some fleet, the receiver fleet has the turn to fire, but only after reporting the result of the received shot. Because the Blockchain simulator sees every message it is always capable of knowing who has the turn.

The applications may publish five notes:

1. Create Game

   This note should contain the type of the note, the identifier of the sender, a random identifier for the game, and evidence of the correct position of the fleet.

2. Join Game

   This note should contain the type of the note, the identifier of the sender, a random identifier for the game, and evidence of the correct position of the fleet. If it is the first player to join the game the game is created, before joining the game.

3. Fire one shot.

   This note should contain the type of the note, the game's identifier, the sender's identifier, the targeting player's identifier, the shot's coordinates x and y, and evidence that the sender's fleet has not been sunk.

4. Report on Shot

   This note should contain the type of the note, the identifier of the game, the identifier of the sender, the identifier of the shooter, the coordinates x and y of the shot, the result of the shot (hit/miss), and evidence that the response is correct.

5. Wave Turn

   This note should contain the type of the note, the identifier of the game, and the identifier of the sender.

6. Claim Victory

   This note should contain the type of the note, the identifier of the game, the identifier of the sender, and evidence that his fleet is not entirely sunk.

5 Evidence of Correct Behavior

Four of the five types of notes that each node may publish require evidence of correct behavior.

The "Join Game" note must provide evidence that the position of the player's fleet is correct and that it will not change during the game without disclosing the actual position of the fleet. This can be done by sending an encryption of the fleet position and proving in zero-knowledge that the encrypted fleet's position is valid, thus preventing the player from proposing an invalid fleet's position or changing it without changing the encryption store by each other node. However, this encryption would never be decrypted; thus, it may be changed by a commitment. For instance, generating the hash of the fleet position together with some nonce and proving in zero-knowledge that the prover knows a secret nonce and a secret position that produces that hash

$$h = Hash(nonce, fleet)$$

.

The nonce is needed because the number of possible fleet positions is less than 1000, and it would be straightforward for someone who knows the hash to calculate the fleet position by brute force.

zkSNARKs are zero-knowledge proofs of the correct computation of functions, usually the verification functions of NP problems, and may be used to generate the required evidence.

Let

$$f(nonce, fleet) = (True, h) = \ (check(fleet), Hash(nonce, fleet)$$

be a function that returns a pair of values. The first value is the result of validation calculation over the fleet, and the second is the hash of the nonce and the fleet. Using a zkSNARK is possible to generate a proof π ensuring that the tuple $(True, h)$ is the result of the computation of the function $f()$ over some secret values *nonce* and $fleet$ known by the prover. Thus, sending the hash h and the zkSNARK proof π is enough.

The evidence required by the "Report note" is quite similar. The prover must provide a proof π to the function

$$g(x, y, nonce, fleet) = (r, h) = (Report(x, y, fleet), h = Hash(nonce, fleet))$$

was correctly computed, where $r = Report(x, y, fleet)$ is a function that tests if there is a boat on the coordinates x, y, and returns "hit" if it is and "miss" otherwise.

On receiving the tuple (r, h, π), the verifier checks: i) that the hash h is the same as the player initially reported, ii) that the value r is the reported value, and iii) that the proof $\pi(r, h)$ it is valid.

The "Fire" and "Claim Victory" notes require proof that the fleet is not sunk. To do this, the application will need to record every shot at a ship and provide evidence that it updated its state. The design of the rest of the proof system is left to the students.

6 Implementation of zkSNARK

To simplify the process of proof generation, the students will use the framework RISCZERO (https://risczero.com)

This framework greatly simplifies the process of creating and verifying the proof. The proof is written as a program in RUST. It is then compiled into a proof and verified.

7 Implementation of the Players Node

The students should follow the project skeleton available at https://github.com/ribeirocn/comm-security.git

For details on how to use the skeleton, see the videos at

- Part 1: https://tinyurl.com/4rwn999k
- Part 2: https://tinyurl.com/yrn8rumb

8 Extra Points

The students may earn extra points by providing additional security features to the project, for instance:

1. Add signatures to every message sent by the nodes [Easy]
2. Submit a proof with every message, proving it is the player's turn. [Hard, Not Recommended]