

ZK Minesweeper

Minesweeper with Zero Knowledge Proofs

05.01.2026

Bruno M. F. Ricardo, Alexandre Almeida

1 Introduction

What is it about?



This project implements a Minesweeper game in a client/server paradigm.

It uses Zero-Knowledge Proofs to allow the client to verify that the server is not cheating, without the server having to reveal the mines' locations.

But how can the server cheat in a Minesweeper game?

How to cheat



The server has several ways to cheat:

1. When the client digs in a spot the does not have a mine, the server straight up lies and says that there is a mine. The server can also make up any valid layout of mines that does not contradict any previous information. Of course, this can only be done in cases where the client cannot be sure that there is no mine.
2. The server can generate a map that forces the client to make multiple guesses, making it more likely that some of the guesses are wrong.
3. A continuation of the last cheat: if the server is forced to generate maps in a random manner, it can do it with many different seeds and only select one when the map is difficult for the client to solve.

Naturally, our solution handles all these cheating cases.

2 Solution

Requirements



These are the overall requirements:

- The map must be randomly generated. This handles cheating case 2.
- The randomness source must have input from the client. This handles cheating case 3.
- The generated map cannot be changed during the game. This handles cheating case 1.
- The initial spot that the client digs must not have any adjacent mine. This can be guaranteed by doing the whole map generation procedure after the first click.
- When the client digs a spot, the server must prove whether it is a mine or not, and in case it is not, the server must prove how many adjacent mines there are.

Zero Knowledge Model



In order to use Zero Knowledge Proofs in our protocol, we need to define what our ZKP can do:

- **Commit:** it allows committing a secret integer from a small set (in our case it is 0 or 1). After being committed, this secret value cannot be changed.
- **Reveal the sum:** it allows revealing the sum of some secret values and prove that it matches the sum of the commits.
- **Reveal:** it allows to reveal the secret value of a commit and prove that they match.

Protocol



- Client and server agree on map size and mine count.
- The client selects the first spot to dig. The 3x3 square surrounding it must be clear of mines, so only the remaining spots will be generated.
- The server distributes the mines in those spots. It generates a commit for each spot, where 0 indicates that there is no mine, and 1 that there is. These commits are sent to the client (note that the client knows the place in the grid where each commit is).
- The server proves that the sum of all committed values is equal to the mine count.
- The client selects a permutation of the remaining spots and sends it to the server. All remaining map spots will be permuted according to it.
- The map generation procedure is finished.
- For each dug spot, the server reveals its commit. If it is not a mine, the server reveals how many adjacent mines there are and proves it by summing the adjacent commits.

3 Implementation

Preliminaries



The ZKP implementation is based on the discrete logarithm problem. As such, it chooses a group with prime order p in which the discrete logarithm is hard.

The Schnorr Identification Scheme is also used. It will not be explained here; for more information see the paper <https://web.stanford.edu/class/cs259c/lectures/schnorr.pdf>

Two generators G and H must be chosen, such that the discrete logarithm relation between them is unknown.

We use the additive notation for group operations. Group elements are described by uppercase letters and integers by lowercase letters.

Commit and reveal



In order to commit to an integer v , the prover generates a random secret nonce integer s in $[0, p)$ and computes the commit

$$C = sG + vH$$

To reveal it, the prover simply reveals s and v , so the verification procedure is trivial.

It is easy to show that if the prover commits to a value, it cannot change it. By contradiction, suppose that it can find two different pairs $(s_1, v_1), (s_2, v_2)$ such that $C = s_1G + v_1H = s_2G + v_2H$. So, the following equation holds:

$$G = \frac{v_2 - v_1}{s_1 - s_2} H$$

Which allows the prover to solve the discrete logarithm problem. Since we supposed it is a hard problem, we conclude that the prover cannot change the committed value.

Revealing the sum



To prove the sum of some committed values, first let C be the sum of the corresponding commits, s the sum of the random nonces and v the sum of the values. All integers sums are taken over mod p . By definition, we know that

$$C = sG + vH$$

This can also be written as

$$C - vH = sG$$

Which can be interpreted as a Schnorr key pair, with $P = C - vH$ being the public key and s the private key.

Now the prover can use the Schnorr identification scheme to prove that they know the value of s in a zero knowledge fashion.

Group



For efficiency reasons, we used elliptic curves. The chosen one was P-256, but we do not use its generator.

The group generators are chosen in a random manner from the curve coordinates. An initial seed is agreed upon, a random x coordinate is selected and the respective y is calculated. If there is no point at this x coordinate, then the process is repeated until one is found. This has to be done in order to be sure that the discrete logarithm of H on base G is unknown.

Software



We use Python 3, with the `fastecdsa` package for the elliptic curve implementation.