Zero-Knowledge Battleship

Aayush Gupta, Nicolaas Kaashoek, Brice Wang, Jason Zhao
{aayushg, nicolaas, bricelw, jzhao7}@mit.edu

1 Introduction

The last decade has seen tremendous growth in both the adoption and understanding of public ledger technologies like Bitcoin and Ethereum. As these cryptocurrencies have risen in popularity, other ideas have also seen a rise in prominence due to a perceived relevance to the blockchain. One example of this is Zero Knowledge Proofs. While they were first conceived of in 1989 by Goldwasser, Micali, and Rackoff[1], they have gained more attention recently. However, there seems to be a lack of consensus as to where they are best used.

We designed an example application whose backbone runs entirely on the blockchain with the support of zero-knowledge proofs. In doing so, we investigated the usefulness, ease of adoption, and practicality of these proofs. For our sample application, we settled on the classic game of Battleship. We implemented our game in an entirely decentralized manner, using zero-knowledge proofs to validate the moves made by each player, and the blockchain as the only "backend" of the system. We successfully achieved our goal of implementing a proof-of-concept that delivers upon these goals, providing evidence to the usefulness of zkps.

Along with our implementation, we developed a strong understanding of the state of zero-knowledge proofs in the modern world, analyzing both their strengths and weaknesses. Our proof-of-concept demonstrates that zero-knowledge applications are viable and practical for certain uses, but also exposed particular problems in using the blockchain for real-time applications. However, we feel that this could be addressed with relatively minor modifications, and does very little to diminish the usefulness of the actual zero-knowledge proofs.

2 Background

Zero knowledge proofs were first proposed in 1989 by Goldwasser, Micali and Rackoff as a way for an untrusted "prover" to demonstrate their knowledge of some piece of information without compromising the information in question. Since then, zero-knowledge proofs have seen no widespread adoption, but have been applied to a number of fields. Examples include using them to enforce ethical behavior in cryptographic protocols, as an alternative to passwords in authentication systems, and even as proof of nuclear disarmament[2]. The rise of the blockchain in recent years has seen zero-knowledge proofs' space in the spotlight grow, with theorized applications including the validation of transaction information. However, they have seen been no large-scale adoption in any of these spaces.

2.1 Games

For our project, we chose games as an application of zkps. Games are played between two or more parties, each trying to achieve some victory condition by following a prescribed set of actions as detailed in the rules of the game. Although cooperative games do exist, we focus on competitive ones. Games lend themselves well to zero-knowledge proofs, as often each player has some information they desire to keep secret from the others. For instance, in a game of Uno, each player's hand must remain secret from all other players. However, the actions each player can take may be limited by the secret information. Returning to Uno, a player may only take an action for which they have the corresponding card in hand. Therefore, zero-knowledge proofs work well in the context of a game, as they allow a player to prove that they can take some action without actually revealing the secret that allows them to do so.

2.2 Decentralization

Another appealing aspect of zero-knowledge proofs is their usefulness in decentralized situations. That is, where there is no backend in the system. Without a trusted server to verify each action taken by a player, zero-knowledge proofs become essential in confirming the validity of each turn. In order to investigate zero-knowledge proofs and their usefulness on the blockchain, this was an essential step for us to take. We deployed our project on the blockchain using smart contracts and zero-knowledge proofs instead of using a server based backend.

3 Previous Work

Both games and zero-knowledge proofs have seen plenty of research attention in the past. Even decentralized games have seen attention in the last five years[3], as building games on the blockchain would alleviate the load placed on the servers that are the backend of most online games today. Platforms like Xaya[4] and Gala[5] exist to help developers take advantage of this, but as of 2020 no large game has been released for either platform. Decentralized games based on zero-knowledge proofs have seen much less attention. However, Wei Jie Koh's zero-knowledge mastermind implementation illustrates the potential in such projects[6].

4 Battleship

We implemented the board game Battleship for our zero-knowledge game, as the entire game is based on hidden states. Battleship is a two player game divided into two phases: a "placement" phase, followed by a "shooting" phase. During the placement phase, each player places 5 ships on a 10x10 grid. Each ship is a rectangle of width 1 and variable height. Once each player finishes the placement phase, the game proceeds to the shooting phase. During this phase, players take turns naming a coordinate on the grid. Their opponent then informs them of whether or not that coordinate contains a ship. If the coordinate is occupied, it is a "hit", and if not it is a "miss". Once all the squares of a ship have been hit, that ship is considered "sunk". Once one player has sunk all of their opponent's ships, that player wins.

While the rules of battleship are simple, they contain many of the abstractions needed to implement games in general. The concept of a turn exists, the game operates on hidden information, and each action taken by the player is influenced by that hidden state. For this reason, battleship is a good proof-of-concept for implementing zero-knowledge games in general. While other games may contain more complex mechanics, the actual operation of each phase of a game tends to be relatively similar to battleship.

5 zk-SNARKs

The standard model of zero-knowledge proofs, in which a prover and verifier communicate with each other in an interactive protocol is inefficient on the blockchain where everything must be stored on the public ledger, as communication occurs over multiple rounds. To address this issue, zk-SNARKs (**z**ero-**k**nowledge **s**uccinct **n**on-interactive **ar**guments of **k**nowledge) were developed. The math behind zk-SNARKs relies on three key concepts: homomorphic hiding, quadratic arithmetic programs, and blind evaluation of polynomials. The following sections cover each of these concepts.

5.1 Homomorphic Hiding

A hiding scheme E allows a user to hide the actual value of some variable x by transmitting only the hiding $E(x)$. This is similar to encryption, except that hidings don't have any associated decryption scheme. zk-SNARKs rely on hiding schemes at a number of points in the protocol, and require a hiding scheme that is homomorphic over both addition and multiplication.

The specific hiding scheme used in most implementations of zk-SNARKs relies on Tate Reduced Pairings of elliptic curves[7]. Elliptic curves $\mathcal{C}$ consists of the points $(x, y) \in \mathbb{F}_p^2$, where $\mathbb{F}_p$ is the group defined by some prime p, that satisfy an equation the form $Y^2 = X^3 + u \cdot X + v$. As this project is not about elliptic curves nor about Tate reduced pairings, we simply include a reference that provides an in-depth explanation of these complicated topics[8].

For the purposes of zk-SNARKs, it suffices that Tate Reduced pairings give us the ability to construct a weak homomorphic hiding scheme that supports both addition and multiplication. This hiding scheme is used in the Pinocchio protocol, the driving protocol behind zk-SNARKs, as illustrated in section 5.4.

5.2 Quadratic Arithmetic Programs

The core of the zero-knowledge proofs used in this project relies on a concept known as Quadratic Arithmetic Programs (QAPs). As demonstrated by Gennaro, Gentry, Parno and Raykova[9], computations in the form of arithmetic circuits can be translated into polynomials. These polynomials then form the QAP for that computation. As arithmetic circuits are extremely powerful, this means that almost arbitrary computations can be translated into QAPs, as shown by the aforementioned authors.

Arithmetic circuits have some set of inputs $(c_1, c_2, \ldots, c_m)$ that if assigned correctly cause the circuit to be complete. For instance, the statement $(c_1 \cdot c_2) + (c_2 \cdot c_3) = 7$ is fulfilled only if c_1, c_2, and c_3 are assigned values that make the equation true. This also translates to the QAP form of the circuit. A QAP of size m and degree d consists of the following:

1. The set of polynomials $L_1, L_2, \ldots, L_m$
2. The set of polynomials $R_1, R_2, \ldots, R_m$
3. The set of polynomials $O_1, O_2, \ldots, O_m$
4. A target polynomial T of degree d

Given some assignment $(c_1, c_2, \ldots, c_m)$ to the QAP, three new polynomials are defined: $L := \sum_{i=1}^{m} c_i * L_i$, $R := \sum_{i=1}^{m} c_i * R_i$ and $O := \sum_{i=1}^{m} c_i * O_i$. If the assignment is satisfying, then the following is true:

$$\exists P := H \cdot T \mid L \cdot R - O = P$$

Importantly, for some arithmetic circuit, if a party knows the values that complete the circuit, they also know a satisfying assignment to the resulting QAP.

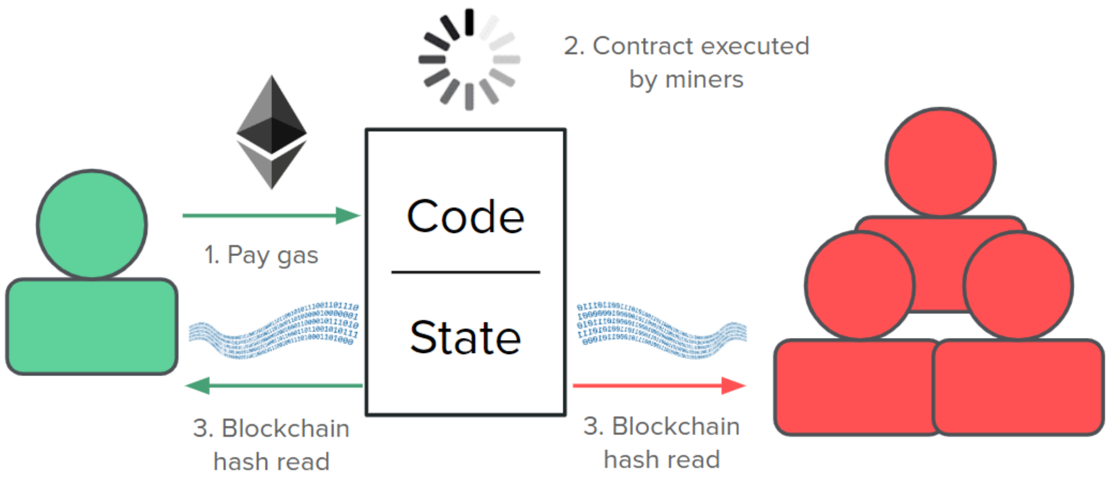

Figure 1: Execution of a sample smart contract.

5.3 Blind Evaluation of Polynomials

In order to make use of QAPs, a party must be able to prove that they know a satisfying assignment. To do so, they can use the protocol for the blind evaluation of polynomials. This relies on the Knowledge of Coefficients Assumption[10], and allows some prover P to prove to a verifier V that they know the coefficients to a polynomial P without revealing the polynomial. For a more detailed description of the protocol see Bayer and Groth's paper on the subject[11]. This relies on the homomorphic hiding schemes discussed previously.

5.4 Pinocchio Protocol

The Pinocchio Protocol[12], proposed by Parno, Howell, Gentry and Raykova relies on both QAPs and blind evaluation as a method to verify nearly arbitrary computations. By translating the computation into a circuit and then into a QAP, a prover can use the blind evaluation of polynomials to demonstrate their knowledge of L, R, O, and H, thereby proving that they have a satisfying assignment to the QAP. This in turn verifies their computation to the verifier in an easily checkable manner, as the verifier need only check that $L \cdot R - O = H \cdot T$.

Of course, all of this has taken place in an interactive proof system, between a verifer and a prover. However, we previously noted that the non-interactiveness of zk-SNARKs is central to their practicality. To make the Pinocchio protocol non-interactive, a common reference string is published for both the prover and the verifier [13]. While doing so requires some extra setup before the proof takes place, it ensures that the proof can be posted publicly and then verified by any third party. This makes it extremely practical for blockchain applications.

6 Smart Contracts

Any logic that needs to be executed in a trusted environment (typically a backend, in this case the ethereum virtual machine) and writes to the data layer (typically a database, in this case the blockchain) must be executed in a smart contract. This is written by a developer and trusted by all who use the relevant frontend, as all the code is public.

The execution of a smart contract begins with the intial caller, the green figure in Figure 1. They pay some gas in the form of a small amount of Ethereum to incentivize miners to execute the contract, and initialize it with some variables (for instance, their proof or balance or other data). The miners then run the contract and commit the result to the blockchain. After consensus by multiple miners, the hash will be written to the blockchain and any party (represented in orange) can read the value. This hash is communicates to all parties that some action was executed.

Because there is actually some cost to execute code and it takes anywhere between 20 seconds and a few minutes to commit to the blockchain, this setup is not as useful in a proof-of-work blockchain like Ethereum 1.0, and thus this implementation is largely a proof of concept. A proof of stake blockchain would significantly speed this process up.

7 Design

Our application consists of three components: a frontend, a smart contract which functions as a backend, and a proof generation and verification integrated into the frontend and backend. As previously discussed, the battleship game consists of two phases: a ship placement phase and a shooting phase. Additionally, the program must enable users to create new games, as well as end games when a user has won. Our program allows an arbitrary number of games between different users to be played at the same time.

7.1 Game Initialization

In order to support many games being played simultaneously, the smart contract stores the game state of every single game as a list. When a user first accesses the frontend, it asks them to either create a new game or join a preexisting one. Note that a user is only able to join a preexisting game if no one else has joined; battleship is a 2-player game. Upon visiting the website application, a user will be asked to connect their MetaMask wallet, giving the frontend information about the ethereum address of that user and enabling them to send Ether to the contract.
If a user decides to create a new game, the frontend calls the smart contract method that initializes a new game, and pays a nominal amount of gas to drive this function. The smart contract adds a new game to its list of active games. It stores all state information about the game, such as the address of the creator of the game, the current phase of the game, and the hashed board configurations, once they are selected. The smart contract then returns a 6-digit unique number representing the game's ID. The user can share this ID with a friend, who can use it to join the newly-created game.

7.2 Ship Placement

The game begins once a second player joins it. The first phase of battleship consists of ship placement. Both players use the frontend to select the locations of their ships. Once they have confirmed their locations, the frontend serializes and hashs the ship locations and sends them to the smart contract. Hashing the board state ensures that no other parties besides the user themselves knows the true configuration of the board, as all data stored in a Ethereum smart contract is public. Once both players have decided on their respective ship configurations, the smart contract marks this phase of the game as done and proceeds to the second phase of the game. Figure 1 illustrates the game initialization and ship placement portions of the program.

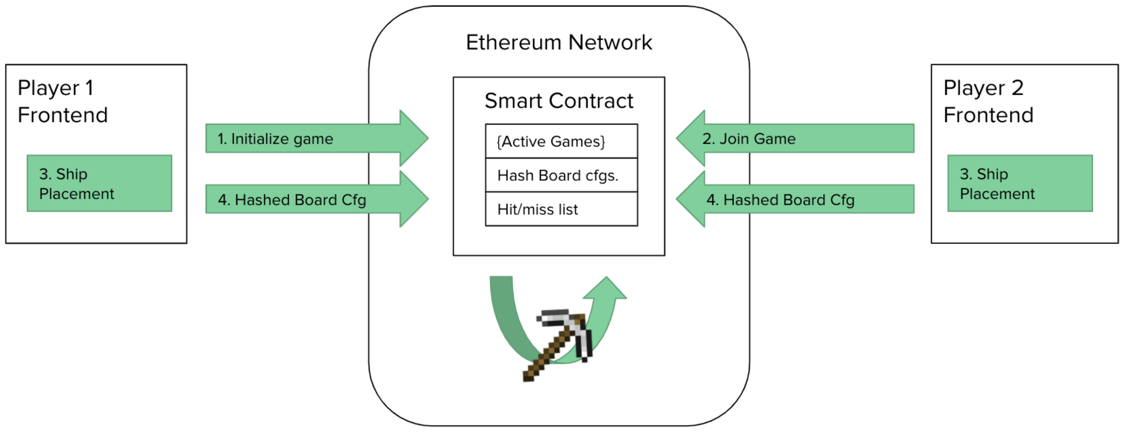

Figure 2: Program flow of users starting a new game and placing their ships.

7.3 Shooting

The second phase of the game consists of each player taking turns shooting a coordinate of their opponent's board. They are notified of whether their shot has hit or missed. If it is player 1's turn, they select a coordinate to shoot via the frontend and send their target coordinate to the smart contract. The smart contract will store this target coordinate for player 2 to read. Once player 2 reads the target coordinate, they check if the coordinate has hit one of their ships and updates their frontend accordingly. Then, they generate a proof demonstrating that their secret board configuration satisfies the hit/miss result they claim, and that their board configuration aligns with the hashed board configuration that is stored in the smart contract.
The smart contract then receives the final shot result from player 2, as well as a corresponding proof. The smart contract first runs the verification method on the proof, ensuring that it is correct. If it is correct, it will store the final shot result in its game state, along with all of the shot results from the previous turns. Player 1 will be able to view the result of his shot from the smart contract and update the frontend accordingly. Figure 2 illustrates the flow of the shooting phase of our application. While we could have used a commitment scheme to commit to ship positions rather than rely on zero-knowledge proofs, we felt that using zero-knowledge proofs was more valuable as a proof-of-concept. As we initially set out to build an application to demonstrate the viability of that technology, we determined that using it here would be a valuable demonstration of its effectiveness.

7.4 Declaring a Winner

Players take turns shooting until one player manages to sink all of their opponent's ships. The smart contract can easily check this win condition after every turn by counting the total number of hit coordinates for every player and checking if it equals the total number of ship coordinates of the game. Once this happens, the smart contract updates the state of the game board to display the winner, at which point each frontend displays the information to the users. Once the game is over, the smart contract waits for a fixed length of time before deleting the game state from the system in order to free up space.

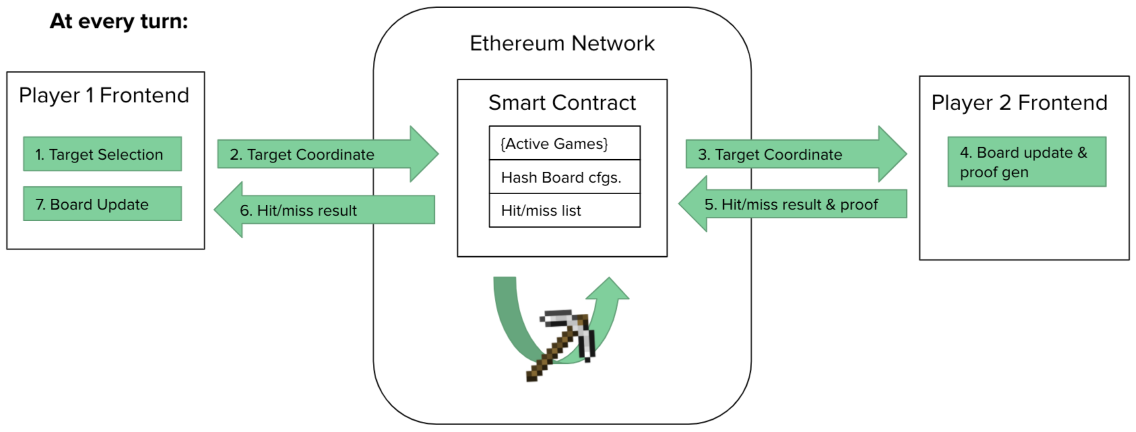

Figure 3: Program flow of a sample turn in which Player 1 shoots a square and player 2 responds.

8 Implementation

We program the battleship client on a ReactJS frontend. Figure 4 shows a screenshot of the front end in action.

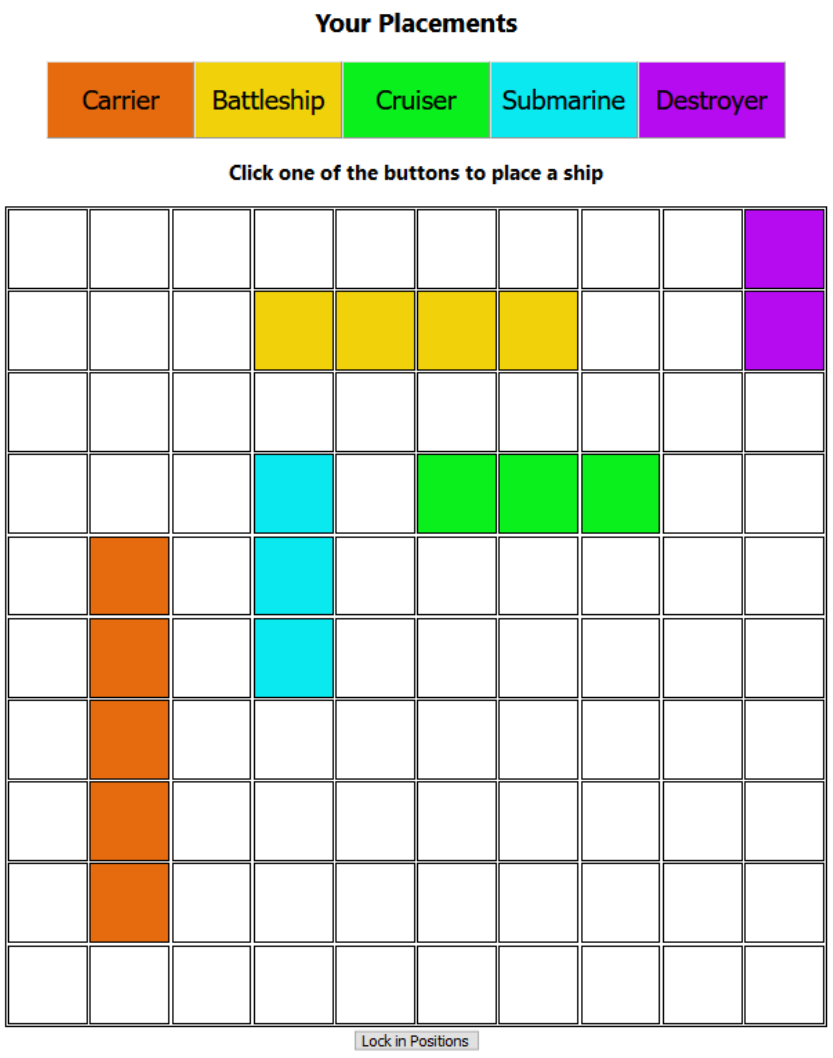

Figure 4: ReactJS Frontend

We programmed our smart contract in Solidity, and used Truffle and Drizzle to connect the smart contracts with the frontend. Our circuit code was written and compiled with circom, which outputs a snarkJS proof. This proof is compiled once, and then connected to the smart contract and the frontend to allow each user to verify the hit/miss responses of the other user. As of now, all of this code is in a working state. The code for the smart contract can be found in Appendix A.1.

8.1 Circuit

The circuit code is compiled into the proof used as the zk-SNARK for verifying the hit/miss result returned by each player during the firing phase of the game. The circuit verifies the hit/miss calculation against the public hash of the board stored on the smart contract. It takes a number of inputs, the most important of which are the hit/miss result, the public hash of the board state as stored in the smart contract, and the private record of what the board state actually is. The circuit then confirms that the hit/miss result actively reflects the state of the private board.

After doing so, it confirms that the private board correctly hashes to the public hash posted in the smart contract. We hash the board state by generating an array of length 136, where every 8 spaces in the array contain the binary representation of an (x, y) coordinate occupied by a ship. We then turn this array into a 136 bit binary number, and pass that as the argument to the hash function, in this case a Pedersen hash[14]. This hash is stored in the smart contract, and recomputed by the circuit to verify an accurate hit/miss calculation. Code for the circuit can be found in Appendix A.2.

The circuit is compiled into a snarkJS proof which can be verified for any set of inputs. This allows each user's frontend to feed the relevant inputs for a given turn to the proof, and the other user to verify that proof. After proof verification, the user can be confident that the hit/miss result returned by their opponent accurately reflects the public hash stored in the smart contract at the start of the game.

9 Conclusion

Using smart contracts in conjunction with zero-knowledge proofs, we were able to successfully develop a working implementation of the game of Battleship. In doing so, we demonstrated the feasibility of zero-knowledge proofs for developing decentralized games. However, we did run into trouble building the game on the blockchain. Because each turn must be added to the smart contract, the transactions representing those turns must be added to the blockchain. Currently, this takes around 20 seconds. While this is relatively quick, it is still too high latency to actually build an enjoyable game. This problem can easily be addressed by storing the moves on some central server that publishes all of its data to a public frontend, however, and shouldn't be taken as a negative for zero-knowledge proofs. In general, we are confident about using zero-knowledge proofs for other applications in the future, as they open up interesting new design paradigms, especially for playing games online.

References

[1] S. Goldwasser, S. Micali, and C. Rackoff, "The knowledge complexity of interactive proof systems," *SIAM Journal on Computing*, vol. 18, no. 1, pp. 186–208, 1989.

[2] A. Glaser, B. Barak, and R. J. Goldston, "A zero-knowledge protocol for nuclear warhead verification," *Nature*, vol. 510, no. 7506, pp. 497–502, 2014.

[3] D. Pollock, "Blockchain Technology Can Give Billion Dollar Gaming Industry A Decentralized Leg Up," 2019.

[4] "xaya.io."

[5] D. Takahashi, "Zynga cofounder builds Gala network for decentralized gaming," 2020.

[6] "GitHub - weijiekoh/zkmm: A Mastermind game using zk-snarks."

[7] P. S. L. M. Barreto and M. Naehrig, "Pairing-Friendly Elliptic Curves of Prime Order," in *Selected Areas in Cryptography* (B. Preneel and S. Tavares, eds.), (Berlin, Heidelberg), pp. 319–331, Springer Berlin Heidelberg, 2006.

[8] S. D. Galbraith, *Mathematics of Public Key Cryptography.* Cambridge University Press, 2012.

[9] R. Gennaro, C. Gentry, B. Parno, and M. Raykova, "Quadratic span programs and succinct nizks without pcps." Cryptology ePrint Archive, Report 2012/215, 2012. https://eprint.iacr.org/2012/215.

[10] M. Bellare and A. Palacio, "The Knowledge-of-Exponent Assumptions and 3-Round Zero-Knowledge Protocols," in *Advances in Cryptology – CRYPTO 2004* (M. Franklin, ed.), (Berlin, Heidelberg), pp. 273–289, Springer Berlin Heidelberg, 2004.

[11] S. Bayer and J. Groth, "Zero-Knowledge Argument for Polynomial Evaluation with Application to Blacklists," in *Advances in Cryptology – EUROCRYPT 2013* (T. Johansson and P. Q. Nguyen, eds.), (Berlin, Heidelberg), pp. 646–663, Springer Berlin Heidelberg, 2013.

[12] B. Parno, C. Gentry, J. Howell, and M. Raykova, "Pinocchio: Nearly practical verifiable computation." Cryptology ePrint Archive, Report 2013/279, 2013. https://eprint.iacr.org/2013/279.

[13] E. Ben-Sasson, A. Chiesa, E. Tromer, and M. Virza, "Succinct Non-Interactive Zero Knowledge for a von Neumann Architecture," in *23rd {USENIX} Security Symposium ({USENIX} Security 14)*, (San Diego, CA), pp. 781–796, {USENIX} Association, aug 2014.

[14] S. Micali, M. Rabin, and J. Kilian, "Zero-knowledge sets," in *Proceedings of the 44th Annual IEEE Symposium on Foundations of Computer Science*, FOCS '03, (USA), p. 80, IEEE Computer Society, 2003.

A Code

We include the two most interesting pieces of code here: the code for the smart contract and for the circuit. For access to all code for the project, please reach out to the authors and we would be happy to share the git repo.

A.1 Smart Contract

```
contract Battleship {

    uint64 constant n_shipsquares = 17;
    uint64 constant INVALID_SHOT = 999;
    uint64 constant g = 69;
    uint64 constant p = 999983;

    uint64 n_games = 0;
    uint64 n_activegames = 0;
    uint64 last_id = 1;

    struct Game {
        address p1;
        address p2;
        uint64 board1;
        uint64 board2;
        uint64[100] revealed1; // 0=unrevealed; 1=miss; 2=hit
        uint64[100] revealed2;
        uint64 candidateShot; // INVALID_SHOT when no candidate, [0, 99] for shot
        uint64 turn; // 1 for p1, 2 for p2
        bool active;
        bool exists; // bool defaults to false
        uint64 winner; // 0 for noone, 1 for p1 wins, 2 for p2 wins
    }

    // return values not allowed within functions that modify state
    // so we need to use events to notify game creator of the game id
    event GameIDs(address indexed from, uint gameID);

    mapping(uint => Game) games;

    function declare() external {
        // Create new game for a user and give them gameID

        // pseudorandom gameID
        uint64 gameID = (last_id * g) % p;
        while (games[gameID].exists) {
            gameID = (gameID + 1) % p;
        }
        last_id = gameID;

        Game storage game = games[gameID];

        game.p1 = msg.sender;
        game.board1 = 0;
        for (uint64 i = 0; i < 100; i++) {
            game.revealed1[i] = 0;
        }
        game.turn = 1;
        game.active = false;
        game.exists = true;
        game.candidateShot = INVALID_SHOT;

        n_games++;

        emit GameIDs(msg.sender, gameID);
    }

    function join(uint gameID) external {
        // Allow player to join valid game if they present gameID and noone else is playing
        Game storage game = games[gameID];
        require(game.exists && !game.active);

        game.p2 = msg.sender;
        game.board2 = 0;
        for (uint64 i = 0; i < 100; i++) {
            game.revealed2[i] = 0;
        }
        game.active = true;

        n_activegames++;
    }

    function place(uint gameID, uint64 board) external {
        Game storage game = games[gameID];
        // check that game has 2 people, and that the sender is a player w/empty board
        require(game.active);
        require((game.p1 == msg.sender && game.board1 == 0) || (game.p2 == msg.sender && game.b


        // player 1 attempting to set on empty board
        if (game.p1 == msg.sender) {
            game.board1 = board;
        }
        // player 2 attempting to set on empty board
        else if (game.p2 == msg.sender) {
            game.board2 = board;
        }
    }

    function fire(uint gameID, uint64 square) external {
        Game storage game = games[gameID];
        // check ships placed, it's sender's turn, and the coordinate is valid
        require(game.active && game.board1 != 0 && game.board2 != 0);
        require((game.p1 == msg.sender && game.turn == 1) || (game.p2 == msg.sender && game.tu
        require(game.candidateShot == INVALID_SHOT && square < 100);

        game.candidateShot = square;
    }

    // verify proof and end current turn. potentially end game
    function resolveShot(uint gameID, uint64 proof, uint64 result) external {
        Game storage game = games[gameID];
        // check ships placed, it's opponent's turn, opponent already shot, result is valid
        require(game.active && game.board1 != 0 && game.board2 != 0);
        require((game.p1 == msg.sender && game.turn == 2) || (game.p2 == msg.sender && game.tu
        require(game.candidateShot != INVALID_SHOT);
        require(result == 0 || result == 1);
        // verification code is generated by snarkJS upon circuit compilation
        bool verified = verify(proof, result);
        if (verified) {
            // process shot result
            if (game.turn == 1) {
                game.revealed1[game.candidateShot] = result + 1;
            } else if (game.turn == 2) {
                game.revealed2[game.candidateShot] = result + 1;
            }
            // end current turn
            game.turn = 3 - game.turn;
            game.candidateShot = INVALID_SHOT;
        }
        checkGameEnd(gameID);
    }

    // check game end condition based on # of hits
    // end the game if the game has finished
    function checkGameEnd(uint gameID) internal returns(uint64){
        Game storage game = games[gameID];
        uint64 p1_hits = 0;
        uint64 p2_hits = 0;
        for (uint64 i = 0; i < 100; i++) {
            if (game.revealed1[i] == 2) {
                p1_hits++;
            }
            if (game.revealed2[i] == 2) {
                p2_hits++;
            }
        }
        if (p1_hits == n_shipsquares) {
            game.winner = 1;
        }
        if (p2_hits == n_shipsquares) {
            game.winner = 2;
        }
    }

    function killGame(uint gameID) external {
        Game memory game = games[gameID];
        require(msg.sender == game.p1 || msg.sender == game.p2);
        delete games[gameID];
    }
}
```

A.2 Circuit

Included below is the code for the circom circuit used to drive the zero-knowledge proofs used in our system.

```
include "./boardhash.circom"
include "./hash.circom"
include "../node_modules/circomlib/circuits/pedersen.circom";
include "../node_modules/circomlib/circuits/bitify.circom"
include "./hash.circom"
template point2num() {
    signal input x;
    signal input y;
    signal output out;

    var n = 256;

    component xBits = Num2Bits(n);
    xBits.in <-- x;

    component yBits = Num2Bits(n);
    yBits.in <-- y;

    component resultNum = Bits2Num(n);
    for (var i=0; i<256-8; i++) {
        resultNum.in[i] <-- yBits.out[i];
    }

    for (var j=256-8; j<n; j++) {
        resultNum.in[j] <-- xBits.out[j];
    }

    out <-- resultNum.out;
}

template BoardHash() {
    signal input shipXs[17];
    signal input shipYs[17];

    var n = 136;
    signal output hashed;
    component hash = Pedersen(n)

    var index = 0;
    for (var i = 0; i < 17; i++) {
        for (var j=0; j < 4; j++) {
            hash.in[index+j] <-- (shipXs[i] >> j) & 1;
        }
        index = index+4;
        for (var j=0; j < 4; j++) {
            hash.in[index+j] <-- (shipYs[i] >> j) & 1;
        }
        index = index+4;
    }
    component p2n = point2num();
        p2n.x <== hash.out[0];
        p2n.y <== hash.out[1];
        hashed <== p2n.out;
}

template BattleshipShot() {
    signal input guessX;
    signal input guessY;
    signal input pubSolnHash;
    signal input hitPub;
    signal input missPub;
    signal private input shipXCoords[17];
    signal private input shipYCoords[17];

    signal output solnHashOut;

    var isHit = 0;
    var isMiss = 1;
    for (var i=0; i < 17; i++) {
        if (shipXCoords[i] == guessX) {
            if (shipYCoords[i] == guessY) {
                isHit = 1;
                isMiss = 0;
            }
        }
    }

    isHit === hitPub;
    isMiss === missPub;

    component boardhash = BoardHash();
    for (var i = 0; i < 17; i++) {
        boardhash.shipXs[i] <-- shipXCoords[i]
        boardhash.shipYs[i] <-- shipYCoords[i]
    }

    solnHashOut <== boardhash.hashed;
    pubSolnHash === boardhash.hashed;
}

component main = BattleshipShot();
```