ZK Battleship

Timothy Mathew*, Varun Sangtani*,†, Zachary S. Siegel*, and Aleksander T. Vestlund*,‡

*EECS, MIT, USA, {tjmathew,sangtani,zss,alekstv}@mit.edu
†EEE, ICL, UK, varun.sangtani22@imperial.ac.uk
‡IE, NTNU, Norway, alekstv@stud.ntnu.no

May 12, 2026

Abstract

Decentralized peer-to-peer (P2P) gaming requires a transition from server-side trust to mathematical certainty to ensure fair play without a central mediator. This paper presents a fully P2P, trustless implementation of the classic game Battleship using Zero-Knowledge Proofs to enforce rules while maintaining the privacy of hidden board states. We design and implement arithmetic circuits using the Groth16 proving system and the Circom domain-specific language to facilitate board validation and move verification. Our methodology evaluates two distinct architectural approaches for cryptographic commitment: a naïve circuit design that directly encodes ship positions and an optimized approach using Merkle tree structures. Results demonstrate that while the naïve approach is limited by a hard upper bound on the number of ship coordinates, the Merkle tree implementation successfully scales to fill a standard ten-by-ten board. Performance analysis indicates that the Merkle tree approach nearly eliminates the marginal cost of adding ship coordinates due to its logarithmic scaling, providing a more efficient and practical solution for P2P environments. This work establishes a framework for anti-cheat guarantees in hidden-information games without the latency or overhead associated with blockchain-mediated architectures.

Keywords — Zero-Knowledge, Battleship, SNARK, Merkle tree, Poseidon

Table of Contents

List of Figures

List of Tables

List of Listings

1 Introduction

Decentralized systems require a shift from server-side trust to mathematical certainty. In a peer-to-peer (P2P) setting, ensuring the integrity of the game without a central mediator is a significant challenge. This project utilizes Zero-Knowledge Proofs (ZKPs) to implement a trustless version of Battleship, allowing players to prove compliance with rules without revealing their private board states.

1.1 Zero-Knowledge Proofs

Zero-Knowledge Proofs are a class of cryptographic protocols that allow one party, the *prover* P, to convince another party, the *verifier* V, that a statement is true without revealing any additional information beyond the validity of the statement itself. The concept was first introduced in 1989 by Goldwasser, Micali, and Rackoff [GMR89] in their foundational work on Interactive Proof (IP) systems. Their research established the theoretical basis for proving knowledge while preserving privacy, introducing the idea that correctness and confidentiality can coexist in the same protocol.

In recent years, ZKPs have evolved from a theoretical cryptographic concept into a practical, widely adopted tool used in modern distributed systems. The early Zero-Knowledge (ZK) systems were highly interactive and computationally expensive, limiting their applicability outside academia. Advances in proof systems such as zk-Succinct Non-Interactive Arguments of Knowledge (SNARKs) and zk-Scalable Transparent Arguments of Knowledge (STARKs) enabled significantly smaller proof sizes and faster verification times, making practical deployments feasible. Among modern protocols, $\mathsf{Groth16}$ [Gro16] is widely known for its succinct proof sizes and fast verification times.

A central motivation behind ZK systems is the ability to establish trust in environments where participants may not fully trust each other. Rather than trusting a server or another third party, participants trust the underlying mathematics of the protocol.

This property makes ZKPs particularly compelling in multiplayer gaming environments, such as Battleship, which is the focus of our contribution. Many games require players to maintain hidden information while simultaneously proving that they are following the rules. In conventional online games, this responsibility is typically delegated to a centralized server. Although effective, this model requires that all players trust the

server operator. In a P2P setting, without a trusted authority, ensuring fair play is non-trivial because players may attempt to alter hidden information, forge game states, or reveal information selectively to gain an advantage.

1.2 Battleship

Battleship is a classic two-player board game. Each player arranges a set of ships with lengths in the multiset

$$\mathcal{L} = \{2, 3, 3, 4, 5\} \tag{1.1}$$

on a typically ten-by-ten private grid, hidden from the opponent. The players then take turns guessing coordinates on the opponent's grid, and the opponent truthfully states whether the guess is a hit, if a part of a ship occupies that cell, or a miss. The goal is to sink all of the opponent's ships first.

A correct implementation must guarantee the following:

- **Validity** — Each player's initial board satisfies the game rules. In other words, all ships must be placed contiguously within the ten-by-ten grid such that the required lengths in Equation (1.1) are satisfied and no ships overlap.
- **Honest responses** — An opponent's answer, $z \in \{\mathsf{hit}, \mathsf{miss}\}$, must be consistent with their ship placement. Without this guarantee, the integrity of the game is compromised, as players could claim false outcomes to gain an unfair advantage.
- **Fair reveal** — A player cannot retroactively change their board after seeing the opponent's moves. If this were possible, a player could move ships based on the opposing party's ship-hitting strategy, giving an unfair advantage.

In a P2P implementation without a trusted authority, each of these properties requires cryptographic enforcement.

1.3 Our Contribution

In this project, we implement a fully P2P, efficiently scalable Battleship game where no trusted server mediates the interaction [ast+26]. We design and compile arithmetic circuits covering two main functions:

(1) **Board validation** — Each player generates a proof that their initial ship configuration is legal, detailed in Subsection 3.6. The resulting output commitment is public.

(2) **Move verification** — When a player receives a guessed coordinate, they produce a proof that their public answer $z \in \{\mathsf{hit}, \mathsf{miss}\}$ follows from their committed board state. See Subsection 3.7.

Crucially, our implementation ensures that even after the game concludes, players are not required to disclose their unhit ship coordinates. Therefore, if a player has constructed a "winning" board, it can be kept secret. In our implementation, we leverage Merkle tree structures, explored in Subsection 3.5, alongside the Groth16 proving system, discussed in Subsection 3.1, to construct scalable circuits with concise proofs, making the approach well suited for a P2P setting.

Our implementation is available at: `https://github.com/aleksandervestlund/zk-battleship` [ast+26].

1.4 Overview of Results

To evaluate the practicality of our approach, we conducted a performance analysis of the proving and verification procedures used throughout the game. In particular, we measured the time required to generate and verify the initial board state validation proof. These experiments were performed across varying numbers of ship coordinates in order to analyze how proof complexity scales with game state size. We compare two implementations:

(1) A naïve circuit design that directly encodes ship positions.

(2) An optimized approach that incorporates Merkle tree structures for more efficient representation and verification of board data.

Through these evaluations, we find that using Merkle trees all but eliminates the marginal cost of adding ship coordinates when computing the board commitment. These results are discussed in Section 4.

2 Related Work

Our project sits at the intersection of three lines of research: cheat prevention in multiplayer games, cryptographic techniques for conducting interactive protocols between distrustful parties without a trusted third party, and applying ZKPs to hidden-information games.

2.1 Anti-Cheat in Multiplayer Games

Cheating in online multiplayer games is an important problem in the distributed systems literature. Yan and Randell [YR05] classify a broad threat landscape, including client-side state manipulation, timing attacks, and player collusion. Battleship exposes a narrower but representative version of this problem: a player must keep their board hidden while still being unable to lie about it or change it after seeing the opponent's moves. A trusted server can record and validate this state, but in a P2P setting there is no such referee.

One response to this setting is the "lockstep" protocol [BL01]. In a lockstep protocol, each player commits to their next move using a cryptographic hash. All commitments are exchanged before any moves are revealed, preventing a player from choosing their move adaptively after observing their opponent's actions. Lockstep commits to a single *action* per round and opens it within that same round, whereas our protocol commits to a player's hidden *state* at the start of the game and the commitment is never opened. Each move can thus be verified in ZK that the reported outcome z is consistent with the committed state.

A more recent approach is to replace the central server with a blockchain. Kalra, Sanghi, and Dhawan [KSD18] use smart contracts to record game state and validate updates, making the game auditable without relying on a single server. This removes one trusted party but introduces the latency and operational overhead of a distributed ledger. Our work studies a more direct approach: cryptographic anti-cheat guarantees in a direct channel between players.

2.2 Cryptographic Protocols Without a Trusted Referee

The general question of how mutually distrustful parties can carry out a hidden-information protocol without a trusted referee is one of the founding questions of modern cryptography. Shamir, Rivest, and Adleman [SRA81] posed and solved an early instance of this problem in their "Mental Poker" protocol, which uses a commutative encryption scheme to allow two players to shuffle and deal cards over a communication channel such that neither can observe the other's hand and neither can cheat about their own. Although mental poker and ZK Battleship differ in mechanics, they share an underlying cryptographic shape: a hidden initial state, a sequence of interactive moves, and a requirement that each player's reported actions be consistent with a state the other party cannot directly inspect.

Commitment schemes provide the binding mechanism behind both lockstep and our protocol. Blum [Blu83] introduced commitments through coin flipping over an untrusted channel, formalizing the idea that a party can bind itself to a value before revealing it. In lockstep, the committed value is later opened. In our protocol, the board commitment remains a public value against which later ZKPs are checked, so the hidden state does not need to be revealed.

One broader theoretical setting our work fits into is secure multiparty computation. Goldreich, Micali, and Wigderson [GMW87] showed that, under standard assumptions, general multiparty functions can be computed without a trusted third party and without revealing private inputs. Battleship could be modeled in this stronger framework, with private boards and private guesses as inputs. Our protocol focuses on a narrower goal: guesses are visible, but the defender cannot change their committed state or lie about the result z. This goal satisfies our threat model and leads to succinct and independently verifiable proofs. For two parties, Yao's garbled circuits [Yao86] are a canonical technique for evaluating such a function, and oblivious transfer [EGL85; Rab81] is the standard primitive used to supply private inputs to the garbled circuit. These tools are most relevant to the stronger version of the problem we discuss in the future work of this project, where one may want to hide not only the board but also the player's targeting strategy.

2.3 Zero-Knowledge Proofs for Hidden-Information Games

The closest prior implementation is *Zero-Knowledge Battleship* by Gupta et al. [GKW+20]. Like our work, it uses Circom [jka+18] and snarkjs [jpx+18] to prove that a battleship response is consistent with a committed private state. The primary difference is architectural: whereas their implementation relies on Ethereum smart contracts to manage state and verify proofs, our protocol exchanges commitments and proofs directly between peers. Table 2.1 summarizes this comparison.

Beyond Battleship, the video game Dark Forest is a prominent deployed example of ZKPs in a hidden-state information game [Dar20a; Dar20b]. It is a real-time strategy game on Ethereum in which players keep the coordinates hidden while proving that the moves are valid. Dark Forest shows that ZK mechanics can support a live game, but it also shares the blockchain-mediated architecture that our project avoids.

Finally, ZK techniques have also been applied to single-player puzzles. The Suguru construction of Robert et al. [RML+22], for instance, proves knowledge of a valid puzzle solution without revealing it. This line of work is adjacent since puzzle protocols prove knowledge of a static solution, while Battleship requires repeated proofs to remain consistent with a hidden state fixed at the beginning of an interactive game.

Table 2.1: Comparison with the closest prior Zero-Knowledge Battleship implementation.

Feature	**[GKW+20]**	**Our implementation**
Network model	Ethereum smart contract	Direct P2P channel
Proof toolchain	Circom / snarkjs	Circom / snarkjs
State commitment	Public board hash	Merkle tree root
Verifier	On-chain contract	Local peer

3 Methodology

Our methodology follows a structured approach: Subsections 3.1 and 3.2 define the cryptographic primitives and tools required for verifiable computation in a P2P environment. After defining our threat model in Subsection 3.3, Subsections 3.4 to 3.8 outline the arithmetic circuits defined to enforce the game rules of Battleship.

3.1 Groth16

Groth [Gro16] introduced a proving protocol in 2016 that has become one of the most popular zk-SNARKs. Building upon Pinocchio [PHG+13], the first widely used zk-SNARK implementation, Groth16 significantly reduces proof sizes while speeding up proof verification. Both protocols follow a similar pipeline, transforming a computation function into a rank-1 constraint system (R1CS), reducing the constraint system into a quadratic arithmetic program (QAP), and generating proving and verification keys, $(\mathsf{pk}, \mathsf{vk})$, during a trusted setup phase.

While the QAP already compresses many circuit constraints into a polynomial identity of the form

$$A(x)B(x) - C(x) = H(x)Z(x),$$

Groth16 further compresses the resulting polynomial consistency checks into polynomial evaluations at a secret trapdoor τ generated during the trusted setup. The prover then constructs three elliptic-curve group elements (A, B, C) that encode these blind evaluations. Verification then consists of checking a constant amount of bilinear pairing equations that together guarantee satisfaction of the original QAP relation. This substantially reduces both proof size and verifier complexity.

While we also explored more recent zk-SNARK protocols such as Permutations over Lagrange-bases for Oecumenical Noninteractive arguments of Knowledge ($\mathcal{PLONK}$) [GWC19], we ultimately chose to use Groth16 since it provides us with the fastest verification time. Since proof verification happens repeatedly throughout P2P gameplay, minimizing verifier latency is crucial for maintaining low-latency user experience.

3.2 Circom and snarkjs

Circom [jka+18] is a domain-specific language for constructing arithmetic circuits used in zk-SNARK systems. A Circom program defines computations over a finite field $\mathbb{F}_p$ using signals and constraints, which are compiled into a R1CS. Each "===" or "<==" directive expands into a quadratic constraint of the form

$$\langle \boldsymbol{a}_i, \boldsymbol{w} \rangle \cdot \langle \boldsymbol{b}_i, \boldsymbol{w} \rangle = \langle \boldsymbol{c}_i, \boldsymbol{w} \rangle$$

over $\mathbb{F}_p$, where $\boldsymbol{w}$ is the witness vector containing all circuit signals. Circom's compiler outputs a `.r1cs` binary, a WebAssembly module that computes the witness from concrete inputs.

snarkjs [jpx+18] implements the remaining Groth16 pipeline. It first ingests the `.r1cs` and performs a reduction to a QAP. The prover must then perform elliptic-curve multiplications to compute the three group elements (A, B, C) from the witness and the QAP representation. snarkjs manages the setup, generating the proving key, pk, and verification key, vk.

During proof generation, snarkjs executes the compiled WebAssembly witness generator, then constructs the Groth16 proof using the pk and the computed witness $\boldsymbol{w}$.

3.3 Threat Model

We consider a P2P setting in which two mutually distrustful players may attempt to cheat by deviating from the rules of the Battleship game. In particular, a dishonest player may move their ships between rounds or falsely report a status z to the attacker. We model both players as probabilistic polynomial-time adversaries. Furthermore, we assume that communication between the two players occurs over an authenticated channel immune to tampering.

Since our setting lacks a trusted centralized server, we do not attempt to defend against Denial-of-Service attacks, premature disconnects, or refusal-to-participate attacks. Instead, we focus on guaranteeing correctness and privacy for players who continue to participate in the protocol. To this end, we assume the hardness of the Elliptic Curve Discrete Logarithm Problem and bilinear pairing-based assumptions that Groth16 relies upon. We also assume that the structured reference string is generated

honestly during a trusted setup phase and that the toxic waste is destroyed after this phase.

Although these assumptions enable us to validate the correctness of the reports z from each player, they do not prevent a dishonest player from cheating between rounds. To defend against a player who cheats by changing the location of their ships between rounds, our protocol requires each player to cryptographically commit to their board state before the first move.

3.4 Zero-Knowledge-Friendly Hashing Algorithms

The commitments that are used to ensure that the hidden state of the board is not changed throughout the game could be implemented through a standard cryptographic hash function. These functions are considered secure because they provide pre-image, second pre-image, and collision resistance. The Secure Hash Algorithm (SHA)-2 [NIST15] family of hash functions is the industry standard across domains including Secure Sockets Layer/Transport Layer Security certificates, digital signatures, and Bitcoin. Furthermore, modern cryptographic standards are increasingly adopting the SHA-3 family of hash functions [Dwo15].

However, these traditional hashes, such as SHA-256 [NIST15], are optimized for conventional hardware architectures and consequently rely heavily on bitwise operations such as XORs, rotations, and bit shifts. Although these operations are efficient on modern processors, they are expensive to represent within the arithmetic circuits over finite fields used in ZKPs. In particular, bitwise operations typically require a large number of R1CS constraints when encoded into zk-SNARK circuits. To address this issue, "ZK-friendly" hash functions were specifically designed to minimize the complexity of the arithmetic circuit. Rather than relying on binary operations, these constructions primarily use finite-field additions, modular arithmetic, and low-degree polynomial operations that can be represented compactly within arithmetic circuits. This substantially reduces constraint counts and, therefore, decreases proving time during zk-SNARK generation.

There are multiple ZK-friendly hash functions to choose from. These include, but are not limited to, Minimal Multiplicative Complexity (MiMC) [AGR+16], Poseidon [GKR+19], Vision, and Rescue [AAB+19]. Our primary objective when choosing the hash function was to minimize the number of R1CS constraints. In addition, Circom, discussed in Subsection 3.2, at the time of this writing, supports only MiMC and Poseidon.

Furthermore, as Grassi et al. [GKR+19, p. 14] state, the number of R1CS constraints in Poseidon is significantly smaller than that of MiMC. For this reason, our implementation uses Poseidon, designed specifically for efficient use within arithmetic proof systems.

One of the primary limitations of the Poseidon hashing algorithm, also noted by the authors [GKR+19, pp. 8, 29], is the hard upper bound on the number of inputs, which is 16. In the standardized Battleship game, as discussed in Subsection 1.2, the different lengths of the ships must be each of the lengths in the multiset from Equation (1.1), totaling 17 ship coordinates. Because our implementation requires a private salt in each node to achieve Indistinguishability under Chosen Plaintext Attacks (IND-CPA), and each coordinate consists of an (x, y) tuple, the total number of required inputs amounts to

$$\underbrace{2}_{|(x,y)|} \cdot \underbrace{17}_{\sum_{l_i \in \mathcal{L}} l_i} + \underbrace{1}_{\text{salt}} = 35.$$

Therefore, relying solely on Poseidon hashing, our naïve implementation cannot accommodate all ship coordinates, as it is limited to a maximum of

$$\left\lfloor \frac{16-1}{2} \right\rfloor = \left\lfloor \frac{15}{2} \right\rfloor = 7. \tag{3.1}$$

3.5 Merkle Trees

The concept of Merkle trees was first introduced by Merkle [Mer90] in 1990. Merkle defined the function $\mathsf{H}(i, j, \boldsymbol{Y})$, where F is a one-way function, as follows [Mer90, p. 228]:

$$\begin{aligned} &(1) \quad \mathsf{H}(i, i, \boldsymbol{Y}) = F(\boldsymbol{Y}) \\ &(2) \quad \mathsf{H}(i, j, \boldsymbol{Y}) = F\Big(\Big(\mathsf{H}(i, {}^{(i+j-1)}/_{2}, \boldsymbol{Y}), \mathsf{H}({}^{(i+j+1)}/_{2}, j, \boldsymbol{Y})\Big)\Big) \end{aligned} \tag{3.2}$$

In other words, leaf nodes contain the output of the one-way function F applied to its content, while inner nodes are the result of hashing the vector consisting of its two children, usually done by concatenation. "Using this method, only $\log_2 n$ transmissions are required," [Mer90, p. 228] assuming n is a power of two. This is highly relevant to our implementation, as the height of the Merkle tree scales logarithmically with the number of inputs. Consequently, the "Merkle path", the sequence of nodes from a given leaf ℓ_i to the root R, also has a length of $\mathcal{O}(\log n)$.

Consider the Merkle tree with eight leaf nodes ℓ_i given in Figure 3.1. Here, the leaf nodes are the hash of the coordinate of the ship, (x, y), and a private random salt, r_i. In addition, the inner nodes are the hash of its children, which is the second line in Equation (3.2). This means that $h_{1,2}$ is the hash of the vector consisting of ℓ_1 and ℓ_2, $h_{1,4}$ is the hash of the vector consisting of $h_{1,2}$ and $h_{3,4}$, and lastly, $R = h_{1,8}$ is the hash of the vector consisting of $h_{1,4}$ and $h_{5,8}$. Hence, to compute R given the pre-image of ℓ_1, the prover P provides ℓ_2, the verifier V computes $h_{1,2}$, and so on, given only $\left\{\ell_2, h_{3,4}, h_{5,8}\right\}$, which has a cardinality of $\mathcal{O}(\log n)$, as stated previously. In summary, the commitment is now the root R instead.

Applying this to our Battleship implementation not only enables proving time to scale logarithmically with the number of ship coordinates but also circumvents the strict input limits described in Equation (3.1).

3.6 Initial Proof

The public input to the circuit is the board commitment C, while the private witness consists of the board configuration B and commitment randomness r. The prover demonstrates knowledge of a board configuration B and randomness r such that $C = \mathsf{Com}(B; r)$ and B satisfies the Battleship placement rules. We represent each ship as an ordered sequence of occupied coordinates, $((X_1, Y_1), \ldots, (X_L, Y_L))$ where L is the ship length. To verify that a ship is valid, the circuit checks that the coordinates form either a horizontal or vertical contiguous sequence. In the horizontal case, all Y_i values are equal, and the X_i values increase by one at each step. In the vertical case, all X_i values are equal, and the Y_i values increase by one at each step.

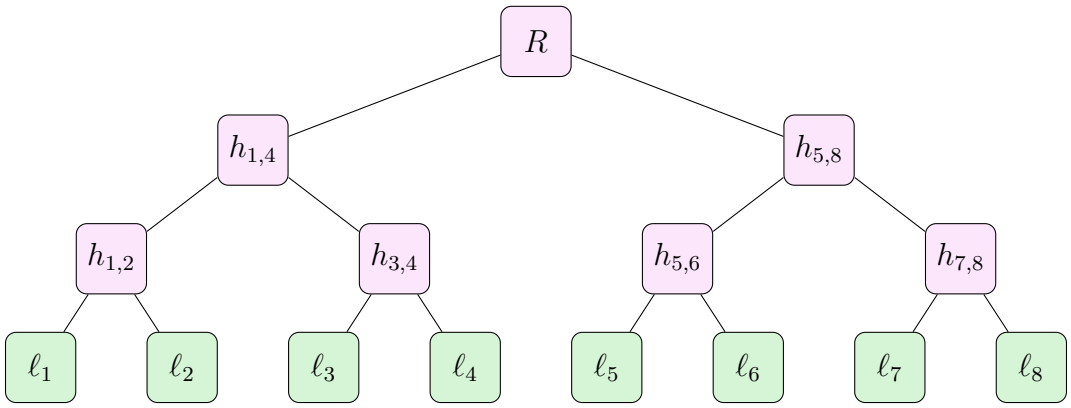

Figure 3.1: Merkle tree example with eight leaf nodes.

Beyond individual ship validity, the initial proof must also enforce global board validity. Each coordinate must lie within the board bounds, and no two occupied ship coordinates may overlap. Bounds checks ensure that every coordinate corresponds to a valid board square, while overlap checks compare each pair of occupied coordinates and reject configurations where both the X and Y coordinates agree. Together, these constraints guarantee that the committed witness corresponds to a legal battleship placement rather than an arbitrary collection of cells.

After validating the board, the circuit computes a commitment to the full private board state using a ZK-friendly hash function. All subsequent proofs of z must be generated using a board witness that hashes to the same commitment. Thus, the initial proof establishes both the validity of the player's starting board and the immutable reference point used throughout the rest of the protocol.

3.7 Validating Hits

After the initial board commitment phase, each subsequent move requires the defending player to prove that the reported outcome z is consistent with the previously committed hidden board state. Unlike the initial proof construction, this circuit does not establish board validity from scratch. Instead, it verifies consistency between a queried coordinate and the already committed board witness.

The public inputs to the circuit consist of the board commitment C, the queried shot coordinates (x, y), and the reported outcome bit $b \in \{0, 1\}$ indicating whether the move resulted in a hit or miss. The private witness consists of the defender's board configuration B together with the commitment randomness r.

To validate a move, the circuit iterates through all occupied ship coordinates and performs equality checks against the queried location (x, y). A hit is detected if there exists at least one occupied coordinate matching the queried position. Formally, the circuit computes

$$b = \bigvee_i [(X_i = x) \wedge (Y_i = y)],$$

where (X_i, Y_i) ranges over all occupied ship coordinates in the private witness. The resulting boolean value is constrained to equal the public input z supplied to the verifier.

Finally, the circuit recomputes the board commitment using the private witness (B, r) and enforces equality with the public commitment C. This guarantees that the result z was computed relative to the same immutable board state established during the initial proof phase. The resulting Groth16 proof, therefore, certifies the correctness of the reported gameplay outcome without revealing any additional information about the hidden board configuration.

3.8 Game Sequence

Before gameplay begins, each player commits to their private board state by publishing a Merkle root, R, derived from the board configuration and secret randomness. On each move, the defending player constructs a witness consisting of the private board configuration, randomness used in the commitment, and the queried board location. Using this witness, the prover generates a Groth16 proof demonstrating that the reported outcome z is consistent with the committed board state. At no point during gameplay is the full board configuration disclosed. Instead, correctness is enforced entirely through ZKPs relative to the committed board state. The procedure is shown below.

ZK Battleship

Alice		**Bob**
..... Setup / Commitment Phase		
$B_A \leftarrow \mathsf{PlaceShips}()$		
$r_A \xleftarrow{\mathrm{R}} \{0,1\}^\lambda$		
$R_A \leftarrow \mathsf{Com}(B_A; r_A)$		
	$\xrightarrow{\quad R_A \quad}$	
		$B_B \leftarrow \mathsf{PlaceShips}()$
		$r_B \xleftarrow{\mathrm{R}} \{0,1\}^\lambda$
		$R_B \leftarrow \mathsf{Com}(B_B; r_B)$
	$\xleftarrow{\quad R_B \quad}$	
..... Move Phase		
$x, y \leftarrow \mathsf{ChooseShot}()$		
	$\xrightarrow{\quad x, y \quad}$	
..... Proof / Response Phase		
		$z \leftarrow f(B_B, x, y)$
		$\pi_B \leftarrow \mathsf{Prove}(B_B, r_B, x, y, z, R_B)$
	$\xleftarrow{\quad z, \pi_B \quad}$	
..... Verification Phase		
$\mathsf{accept} \leftarrow \mathsf{Ver}(R_B, x, y, z, \pi_B)$		

4 Results

We benchmark our implementation by measuring the time taken to generate proofs for the setup phase of the game. We first compare the naïve design with the Merkle tree implementation and subsequently benchmark them across varying numbers of ship coordinates. The benchmarks in this section were run on the following machine specifications:

- **Operating system** — Arch Linux (Linux kernel 6.19.11-arch1-1).
- **Central processing unit** — Intel Core Ultra 7 155U, 14 cores, up to 4.8 GHz.
- **Memory** — 16 GiB.

Figures 4.1 to 4.3 show proof generation times with each data point being the mean of one hundred independent runs. To facilitate the scaling and maintain a uniform benchmark, we have simplified the circuit logic by relaxing the standard Battleship schema. Rather than enforcing the traditional multiset of ship lengths from Equation (1.1), the circuit allows for arbitrary ship lengths. To illustrate, consider a board that has 23 coordinates. Then, the multiset $\mathcal{L}$ from Equation (1.1) is defined as $\{10, 10, 3\}$, which fills from the upper left corner in a left-to-right manner. In addition, the benchmarking starts at two ship coordinates to ensure the overlap check is not optimized out of the circuit.

4.1 Naïve Approach

Recall from Subsections 3.4 and 3.5 that the naïve approach has a hard upper bound of seven coordinates. Hence, we are only able to analyze the results by varying the number of ship coordinates from two to seven, inclusively. The circuit for two coordinates is shown in Listings A.1a to A.1c, with minor modifications for the other number of ship coordinates as described in Subsection A.1. The results of the measurements are shown in Figure 4.1. As previously noted, the proving time scales linearly in the number of inputs, and that trend is apparent in the figure. Although the number of coordinates remains fixed in the standard game of Battleship, as described in Subsection 1.2, this structural limitation renders the naïve design unsuitable for other P2P games with larger state spaces. The proving time would be prohibitively slow for the end-users. In

addition, notice that the proving time for only two ship coordinates is 860 ms. This behavior suggests that the total execution time is dominated by a significant constant overhead that is independent of the number of ship coordinates. This overhead likely stems from at least one of the following:

- **Initialization costs** — Setting up the circuit environment.
- **Fixed verification steps** — Certain cryptographic constraints or proof generation steps that must be processed regardless of the specific input size.
- **System latency** — Basic input/output and processing cycles inherent to the benchmarking environment.

4.2 Merkle Tree Approach

The execution times for different numbers of ship coordinates for variations of the circuit displayed in Listings B.1a to B.1d, still adapted to the number of ship coordinates as described in Section B, are illustrated in Figure 4.2. As shown in the plot, the execution time exhibits a shallow growth curve as the number of ship coordinates increases from only two to filling the entire ten-by-ten board at one hundred. Although a slight upward trend is visible, moving from approximately 267 ms at the lower bound to roughly 281 ms at the upper bound, the increase is not proportional to the fiftyfold increase in the number of ship coordinates. This can be attributed to the fact that the verification time of a Merkle tree scales logarithmically, as discussed in Subsection 3.5.

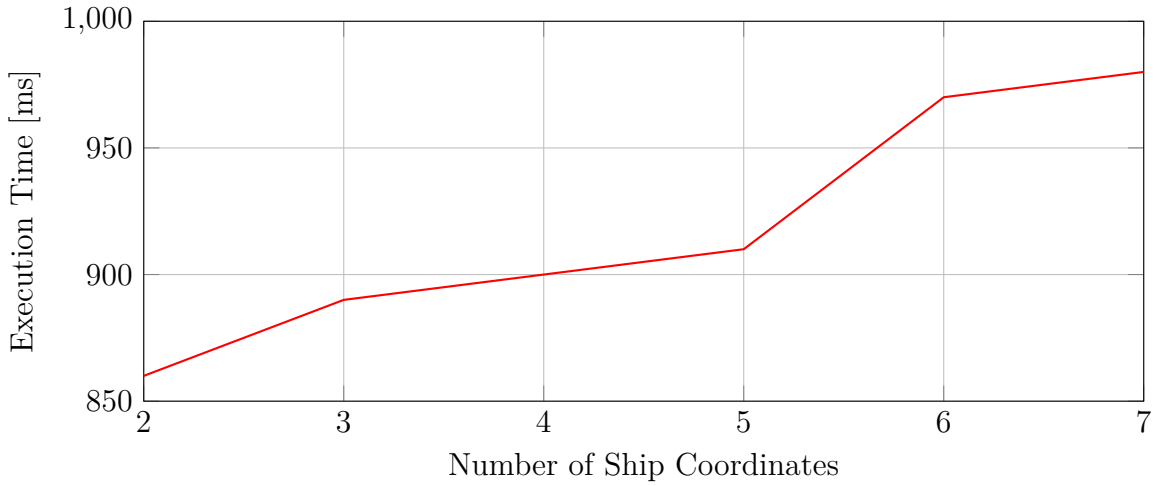

Figure 4.1: Analyzing performance in the naïve approach.

Furthermore, the data exhibit minor fluctuations, such as the notable dip at eleven ship coordinates. However, since the range across the entire data set is so small, the system demonstrates excellent scalability within this specific parameter range, indicating that the incremental cost of adding one ship coordinate to the circuit is negligible compared to the base cost of the operation itself. Also note that there is a constant overhead as in Subsection 4.1, which stems from the same reasons stated there.

4.3 Comparison of Approaches

Since the circuit in the naïve approach is bounded by a hard limit of at most seven ship coordinates, as discussed in Subsection 3.4, comparisons in Figure 4.3 are made across these values. The first point of comparison would be the linear versus logarithmic scaling exhibited in the plots. As noted, the Merkle tree circuit scales better than the naïve implementation, even for a small number of ship coordinates.

In summary, the Merkle tree approach scales better in our implementation. Even if the naïve implementation offered superior performance, it remains practically infeasible because a standard game requires more than seven ship coordinates.

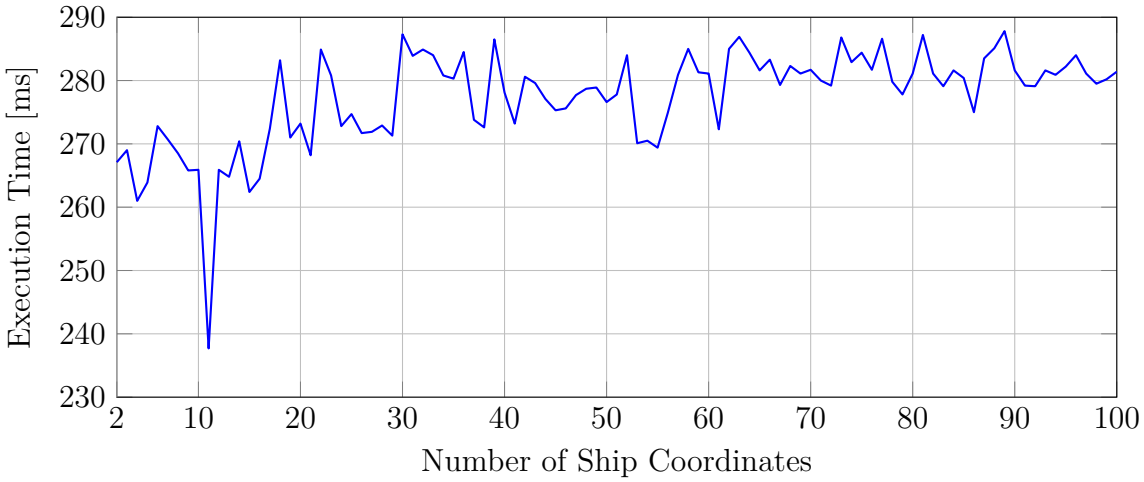

Figure 4.2: Analyzing performance in the Merkle tree approach.

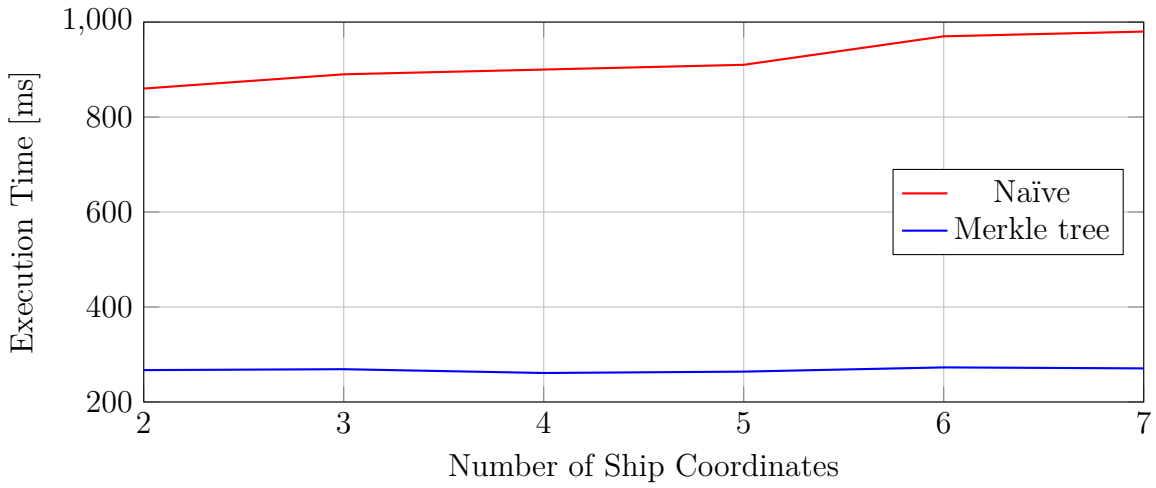

Figure 4.3: Comparing performance in the naïve and Merkle tree approaches.

5 Discussion

The results show a clear difference between the two commitment strategies. The naïve construction is useful as a simple baseline because it commits directly to the occupied ship coordinates, but it is limited by the arity of the Poseidon hash. Since standard Battleship requires 17 occupied coordinates, and each coordinate consists of both an x and a y value, the direct Poseidon commitment cannot represent a full standard board. The Merkle tree construction removes this restriction by committing to a root and proving membership only for the queried square. This makes the Merkle approach the more meaningful result for a realistic version of this game.

The timing results suggest that our prototype is dominated by fixed overheads rather than the cost of adding ship coordinates. The Merkle benchmark shows only small growth over the tested board sizes, which is consistent with the logarithmic Merkle path checked by the circuit. At the same time, however, the absolute running time remains large enough that the current implementation would feel slow in a real-time game. In practice, witness generation and snarkjs invocation likely contribute most to the user-visible latency.

The most important cryptographic limitation of our game is the trusted setup required by Groth16. Before either player can generate or verify proofs, the circuit must be processed into public proving and verification parameters. During this setup procedure, we create secret intermediate randomness. The security of the system depends on the assumption that we discard this randomness and it is not known to

either player. If a malicious party retained it, they could produce proofs that verify even when the underlying Battleship statement is false. Thus, the protocol removes trust in a game server by introducing trust in the setup process. In our case, we assume that the game developer is trusted, but that does not preclude collusion between the developer and one of the players.

The implementation is also limited by timing. Although the Merkle construction scales better than the naïve construction, the current proving pipeline is still too slow to be invisible to the players. Since each move requires the defender to generate a proof before the attacker can accept, proof generation directly affects turn latency. If used on a game that was more real-time than Battleship, this could become a major barrier to deployment. Thus, although our results support Merkle tree design, they do not show this implementation could scale to a real-time game experience. A potential avenue for further research would be to prove a batch of previous game states at regular intervals, amortizing the proof generation cost but requiring that game states are retroactively checked.

The privacy guaranty is also narrower than what one might want from a fully private hidden-information game. Although our protocol hides the unrevealed board squares of the defender, the guesses of the attacker are public. This means that the defender may be able to infer the attacker's search strategy over time, so it does not protect the privacy of how a player chooses moves. A stronger formulation of the problem could hide both the defender's board and the attacker's targeting strategy.

6 Conclusion

This project demonstrates a P2P version of Battleship in which ZKPs replace the trusted game server for central fairness checks. Our scheme requires each player to commit to a hidden board state, prove that the initial board is valid, and then prove that every single reported hit or miss are consistent with the same commitment. The main contributions are the Circom/Groth16 circuits for board and move verification, the P2P proof-exchange structure, and the comparison between a direct Poseidon commitment and a Merkle tree commitment. The Merkle tree approach is the scalable construction because it supports full-board commitments while requiring only a local authentication path for each move.

One direction for future work is to replace Groth16 with a transparent proof system such as a STARK. A SNARK, such as Groth16, gives very short proofs and fast verification, but usually rely on elliptic-curve pairings and setup parameters tied to the circuit being proved. Instead, a STARK is built from hash functions and polynomial testing rather than from a trusted setup based on pairings. Thus, there is no hidden setup secret. Replacing Groth16 with a STARK could address this limitation of the trusted setup directly, but at the potential cost of added latency.

Another direction is to redesign when and how move verification occurs. Instead of blocking the game on every proof, the protocol could allow the gameplay to continue optimistically while the proofs are generated and checked asynchronously. For instance, a move could be tentatively accepted, with the proof verified before the next state transition becomes final. If a proof fails or does not arrive before a timeout, the game could assign a forfeit. This change could hide some proving latency from the player and make the game work better in real-time scenarios.

A final direction is to strengthen the privacy guarantees for the attacking player. Garbled circuits [Yao86] and oblivious transfer [EGL85; Rab81] are natural tools for this, because they allow two parties to compute a function over private inputs, the chosen shot and the committed board information, without revealing them directly. The output would reveal only the result z. Oblivious transfer would allow the attacker to obtain information corresponding to a chosen square without revealing which square was selected, while garbled circuits could enforce the computation z itself. This would move the project from proving honest responses about public guesses toward a stronger protocol for hidden-information games where both the board and the targeting strategy remain private.

7 Contributions

Mathew did research into ZKPs and implemented "cheat mode" for our Battleship implementation. He wrote the majority of Section 3 of this paper.

Sangtani created the different Circom circuits used for benchmarking and recorded the results used in Section 4. In addition, he implemented Merkle trees for board commitments. Additionally, he wrote Section 1 and Subsection 3.2 of this paper.

Siegel implemented the first naïve circuit made by Sangtani into the Battleship game, allowing the later circuits to be easily integrated. He also wrote Sections 2, 5, and 6 of this paper.

Vestlund created the base Battleship game and later implemented the final naïve and Merkle tree circuits made by Sangtani. Furthermore, he wrote the abstract, Section 4, Section 8, and also assisted in writing Subsections 2.3, 3.4, 3.5, and 3.8.

8 Acknowledgements

Thank you to the teaching assistants, Kaiwen He and Simon Langowski, for their guidance throughout the project. In addition, thank you to Professor Srini Devadas for teaching the course.

References

[AAB+19] A. Aly, T. Ashur, E. Ben-Sasson, S. Dhooghe, and A. Szepieniec. *Design of Symmetric-Key Primitives for Advanced Cryptographic Protocols*. Cryptology ePrint Archive, Paper 2019/426. 2019. DOI: 10.13154/tosc.v2020.i3.1-45. URL: https://eprint.iacr.org/2019/426 (visited on May 11, 2026).

[AGR+16] M. Albrecht, L. Grassi, C. Rechberger, A. Roy, and T. Tiessen. MiMC*: Efficient Encryption and Cryptographic Hashing with Minimal Multiplicative Complexity*. Cryptology ePrint Archive, Paper 2016/492. 2016. URL: https://eprint.iacr.org/2016/492 (visited on May 11, 2026).

[ast+26] aleksandervestlund, siegelz, timothy-mathew, and VSangtani. *ZK Battleship: A Zero-Knowledge Battleship implementation*. March 2026. URL: https://github.com/aleksandervestlund/zk-battleship.

[BL01] N. E. Baughman and B. N. Levine. "Cheat-Proof Playout for Centralized and Distributed Online Games." In: *Proceedings IEEE INFOCOM 2001*. Vol. 1. 2001, pp. 104–113. DOI: 10.1109/INFCOM.2001.916692.

[Blu83] M. Blum. "Coin Flipping by Telephone: A Protocol for Solving Impossible Problems." In: *SIGACT News* 15.1 (1983), pp. 23–27. DOI: 10.1145/1008908.1008911.

[Dar20a] Dark Forest Team. *ZKPs for Engineers: A Look at the Dark Forest ZKPs*. September 29, 2020. URL: https://blog.zkga.me/df-init-circuit (visited on May 11, 2026).

[Dar20b] Dark Forest Team. *Zero-Knowledge Proofs for Engineers: Introduction*. August 10, 2020. URL: https://blog.zkga.me/intro-to-zksnarks (visited on May 11, 2026).

[Dwo15] M. J. Dworkin. *SHA-3 standard: Permutation-based hash and extendable-output functions*. Tech. rep. NIST, 2015. DOI: 10.6028/NIST.FIPS.202.

[EGL85] S. Even, O. Goldreich, and A. Lempel. "A Randomized Protocol for Signing Contracts." In: *Communications of the ACM* 28.6 (1985), pp. 637–647. DOI: 10.1145/3812.3818.

[GKR+19] L. Grassi, D. Khovratovich, C. Rechberger, A. Roy, and M. Schofnegger. Poseidon*: A New Hash Function for Zero-Knowledge Proof Systems*. Cryptology ePrint Archive, Paper 2019/458. 2019. URL: https://eprint.iacr.org/2019/458 (visited on May 11, 2026).

[GKW+20] A. Gupta, N. Kaashoek, B. Wang, and J. Zhao. *Zero-Knowledge Battleship*. MIT 6.857 final project. 2020. URL: https://courses.csail.mit.edu/6.857/2020/projects/13-Gupta-Kaashoek-Wang-Zhao.pdf (visited on May 11, 2026).

[GMR89] S. Goldwasser, S. Micali, and C. Rackoff. "The Knowledge Complexity of Interactive Proof Systems." In: *SIAM Journal on Computing* 18.1 (1989), pp. 186–208. DOI: 10.1137/0218012.

[GMW87] O. Goldreich, S. Micali, and A. Wigderson. "How to Play Any Mental Game or A Completeness Theorem for Protocols with Honest Majority." In: *Proceedings of the Nineteenth Annual ACM Symposium on Theory of Computing*. 1987, pp. 218–229. DOI: 10.1145/28395.28420.

[Gro16] J. Groth. "On the Size of Pairing-Based Non-interactive Arguments." In: *Proceedings, Part II, of the 35th Annual International Conference on Advances in Cryptology—EUROCRYPT 2016*. Vol. 9666. Berlin, Heidelberg: Springer-Verlag, 2016, pp. 305–326. ISBN: 9783662498958.

[GWC19] A. Gabizon, Z. J. Williamson, and O. Ciobotaru. $\mathcal{PLONK}$*: Permutations over Lagrange-bases for Oecumenical Noninteractive arguments of Knowledge*. Cryptology ePrint Archive, Paper 2019/953. 2019. URL: https://eprint.iacr.org/2019/953 (visited on May 11, 2026).

[jka+18] jbayline, kobigurk, alrubio, OBrezhniev, udibr, arnaucube, bellesmarta, pertsev, judiciouscoder, ed255, krlosMata, poma, 0xtsukino, paulmillr, and cre-mer. *CircomLib: Library of basic circuits for circom*. Version 2.0.5. October 2018. URL: https://github.com/iden3/circomlib.

[jpx+18] jbayline, phated, xavi-pinsach, unixpi, OBrezhniev, uaoleg, eduadiez, nalinbhardwaj, jesusholoo, bajpai244, weijiekoh, bellesmarta, glamperd, vplasencia, Kolezhniuk, kobigurk, judiciouscoder, kziemianek, kirrya95, 0xbhagi, RogerTaule, Stumble, wp-lai, martyall, ichub, Arvolear, hecmas, ghiliweld, io4, CPerezz, arnaucube, pertsev, yelhousni, krlosMata, mhchia, lispc, jacobrosenthal, obi1kenobi, jlmunoz77, prabal-banerjee, zkronos73, sragss, naps62, VictorColomb, and paulmillr. *snarkjs: zkSNARK imple-*

mentation in JavaScript & Wasm. Version 0.7.6. August 2018. URL: https://github.com/iden3/snarkjs.

[KSD18] S. Kalra, R. Sanghi, and M. Dhawan. "Blockchain-based real-time cheat prevention and robustness for multi-player online games." In: *Proceedings of the 14th International Conference on Emerging Networking EXperiments and Technologies*. CoNEXT '18. Heraklion, Greece: Association for Computing Machinery, 2018, pp. 178–190. ISBN: 9781450360807. DOI: 10.1145/3281411.3281438.

[Mer90] R. C. Merkle. "A Certified Digital Signature." In: *Advances in Cryptology—CRYPTO' 89 Proceedings*. Ed. by G. Brassard. New York, NY: Springer New York, 1990, pp. 218–238. ISBN: 978-0-387-34805-6.

[NIST15] National Institute of Standards and Technology (NIST). *Secure Hash Standard (SHS)*. Tech. rep. FIPS PUB 180-4. NIST, 2015. DOI: 10.6028/NIST.FIPS.180-4.

[PHG+13] B. Parno, J. Howell, C. Gentry, and M. Raykova. "Pinocchio: Nearly Practical Verifiable Computation." In: *2013 IEEE Symposium on Security and Privacy*. 2013, pp. 238–252. DOI: 10.1109/SP.2013.47.

[Rab81] M. O. Rabin. *How to Exchange Secrets by Oblivious Transfer*. Tech. rep. TR-81. Aiken Computation Laboratory, Harvard University, 1981.

[RML+22] L. Robert, D. Miyahara, P. Lafourcade, L. Libralesso, and T. Mizuki. "Physical Zero-Knowledge Proof and NP-Completeness Proof of Suguru Puzzle." In: *Information and Computation* 285 (2022), p. 104858. DOI: 10.1016/j.ic.2021.104858.

[SRA81] A. Shamir, R. L. Rivest, and L. M. Adleman. "Mental Poker." In: *The Mathematical Gardner*. Ed. by D. A. Klarner. Prindle, Weber & Schmidt and Wadsworth International, 1981, pp. 37–43.

[Yao86] A. C.-C. Yao. "How to Generate and Exchange Secrets." In: *27th Annual Symposium on Foundations of Computer Science*. 1986, pp. 162–167. DOI: 10.1109/SFCS.1986.25.

[YR05] J. Yan and B. Randell. "A Systematic Classification of Cheating in Online Games." In: *NetGames '05: Proceedings of the 4th ACM SIGCOMM Workshop on Network and System Support for Games*. ACM, 2005, pp. 1–9. DOI: 10.1145/1103599.1103606.

Acronyms

FIPS	Federal Information Processing Standards	24
INFOCOM	International Conference on Computer Communications	22
NIST	National Institute of Standards and Technology	22, 24
$\mathcal{P}\mathfrak{lonK}$	Permutations over Lagrange-bases for Oecumenical Noninteractive arguments of Knowledge	7, 23
PUB	Publication	24
SHA	Secure Hash Algorithm	9, 22
SIAM	Society for Industrial and Applied Mathematics	23
SIGACT	Special Interest Group on Algorithms and Computation Theory	22
SIGCOMM	Special Interest Group on Data Communication	24
SNARK	Succinct Non-Interactive Argument of Knowledge	1, 7–9, 20, 23
STARK	Scalable Transparent Argument of Knowledge	1, 20
Wasm	WebAssembly	24

Initialisms

ACM	Association for Computing Machinery	22–24

EECS	Department of Electrical Engineering and Computer Science	1
EEE	Department of Electrical and Electronic Engineering	1
ICL	Imperial College London	1
IE	Faculty of Information Technology and Electrical Engineering	1
IEEE	Institute of Electrical and Electronics Engineers	22, 24
IND-CPA	Indistinguishability under Chosen Plaintext Attacks	10
IP	Interactive Proof	1, 23
MiMC	Minimal Multiplicative Complexity	9, 10, 22
MIT	Massachusetts Institute of Technology	1
NTNU	Norwegian University of Science and Technology	1
P2P	peer-to-peer	1–4, 6–8, 15, 19
QAP	quadratic arithmetic program	7, 8
R1CS	rank-1 constraint system	7–10
SHS	Secure Hash Standard	24
UK	United Kingdom	1
USA	United States of America	1
ZK	Zero-Knowledge	i, iii, 1, 4–6, 9, 12, 14, 22, 23
zk	Zero-Knowledge	*see* ZK
ZKP	Zero-Knowledge Proof	i, 1, 4–6, 9, 13, 19, 20, 22–24

Appendix

As stated previously, all code is available at `https://github.com/aleksandervestlund/zk-battleship` [ast+26].

A Naïve Circuits

A.1 Initial Proof Circuit

```
pragma circom 2.1.6;

include "circomlib/circuits/comparators.circom";
include "circomlib/circuits/poseidon.circom";

template ValidShip(L) {
    signal input X[L];
    signal input Y[L];

    component isYEq[L - 1];
    component isXInc[L - 1];
    signal isHorizontal[L];
    signal horizStep[L - 1];
    isHorizontal[0] <== 1;

    component isXEq[L - 1];
    component isYInc[L - 1];
    signal isVertical[L];
    signal vertStep[L - 1];
    isVertical[0] <== 1;

    for (var i = 0; i < L - 1; i++) {
        isYEq[i] = IsEqual();
        isYEq[i].in[0] <== Y[i];
        isYEq[i].in[1] <== Y[i + 1];

        isXInc[i] = IsEqual();
        isXInc[i].in[0] <== X[i] + 1;
```

Listing A.1a: Naïve commitment for two coordinates.

```
        isXInc[i].in[1] <== X[i + 1];

        horizStep[i] <== isYEq[i].out * isXInc[i].out;
        isHorizontal[i + 1] <== isHorizontal[i] * horizStep[i];

        isXEq[i] = IsEqual();
        isXEq[i].in[0] <== X[i];
        isXEq[i].in[1] <== X[i + 1];

        isYInc[i] = IsEqual();
        isYInc[i].in[0] <== Y[i] + 1;
        isYInc[i].in[1] <== Y[i + 1];

        vertStep[i] <== isXEq[i].out * isYInc[i].out;
        isVertical[i + 1] <== isVertical[i] * vertStep[i];
    }

    isHorizontal[L - 1] + isVertical[L - 1] === 1;
}

template BoardSetup(N) {
    signal input privShipX[N];
    signal input privShipY[N];
    signal input privSalt;

    signal output pubCommitment;

    component lessX[N];
    component lessY[N];

    for (var i = 0; i < N; i++) {
        lessX[i] = LessThan(4);
        lessX[i].in[0] <== privShipX[i];
        lessX[i].in[1] <== 10;
        lessX[i].out === 1;

        lessY[i] = LessThan(4);
        lessY[i].in[0] <== privShipY[i];
        lessY[i].in[1] <== 10;
        lessY[i].out === 1;
    }
```

Listing A.1b: Naïve commitment for two coordinates (continued).

```
    component eqX[N][N];
    component eqY[N][N];

    for (var i = 0; i < N; i++) {
        for (var j = i + 1; j < N; j++) {
            eqX[i][j] = IsEqual();
            eqX[i][j].in[0] <== privShipX[i];
            eqX[i][j].in[1] <== privShipX[j];

            eqY[i][j] = IsEqual();
            eqY[i][j].in[0] <== privShipY[i];
            eqY[i][j].in[1] <== privShipY[j];

            eqX[i][j].out * eqY[i][j].out === 0;
        }
    }

    component ship = ValidShip(N);

    for (var i = 0; i < N; i++) {
        ship.X[i] <== privShipX[i];
        ship.Y[i] <== privShipY[i];
    }

    component hasher = Poseidon(2 * N + 1);

    for (var i = 0; i < N; i++) {
        hasher.inputs[i] <== privShipX[i];
        hasher.inputs[i + N] <== privShipY[i];
    }

    hasher.inputs[2 * N] <== privSalt;
    pubCommitment <== hasher.out;
}

component main = BoardSetup(2);
```

Listing A.1c: Naïve commitment for two coordinates (continued).

This circuit is available here: `https://github.com/aleksandervestlund/zk-battleship/blob/master/circuits/board_commit2.circom`. For the other numbers of

ship coordinates, the 2 on the last line in Listing A.1c is replaced with the respective number of ship coordinates.

A.2 Validating Hit Circuit

```
pragma circom 2.1.6;

include "circomlib/circuits/comparators.circom";
include "circomlib/circuits/poseidon.circom";

template BattleshipHit(N) {
    signal input pubGuessX;
    signal input pubGuessY;
    signal input pubCommitment;
    signal input pubReportedHit; // 1 (Hit) or 0 (Miss)

    signal input privShipX[N];
    signal input privShipY[N];
    signal input privSalt;

    component hasher = Poseidon(2 * N + 1);

    for (var i = 0; i < N; i++) {
        hasher.inputs[i] <== privShipX[i];
        hasher.inputs[i + N] <== privShipY[i];
    }

    hasher.inputs[2 * N] <== privSalt;
    pubCommitment === hasher.out;

    component eqX[N];
    component eqY[N];
    signal hitMatch[N];

    signal hitAccumulator[N + 1];
    hitAccumulator[0] <== 0;

    for (var i = 0; i < N; i++) {
        eqX[i] = IsEqual();
```

Listing A.2a: Naïve hit validating circuit for two coordinates.

```
        eqX[i].in[0] <== pubGuessX;
        eqX[i].in[1] <== privShipX[i];

        eqY[i] = IsEqual();
        eqY[i].in[0] <== pubGuessY;
        eqY[i].in[1] <== privShipY[i];

        hitMatch[i] <== eqX[i].out * eqY[i].out;
        hitAccumulator[i + 1] <== hitAccumulator[i] + hitMatch[i];
    }

    pubReportedHit === hitAccumulator[N];
}

component main {public [pubGuessX, pubGuessY, pubCommitment,
↪  pubReportedHit]} = BattleshipHit(2);
```

Listing A.2b: Naïve hit validating circuit for two coordinates (continued).

This circuit is available here: `https://github.com/aleksandervestlund/zk-battleship/blob/master/circuits/battleship_hit2.circom`. For the other numbers of ship coordinates, the 2 on the last line in Listing A.2b is replaced with the respective number of ship coordinates.

B Merkle Tree Initial Proof Circuit

```
pragma circom 2.1.6;

include "circomlib/circuits/poseidon.circom";
include "circomlib/circuits/comparators.circom";

template ValidShip(L) {
    signal input X[L];
    signal input Y[L];
```

Listing B.1a: Merkle tree commitment for two coordinates.

```
    component isYEq[L - 1];
    component isXInc[L - 1];
    signal isHorizontal[L];
    signal horizStep[L - 1];
    isHorizontal[0] <== 1;

    component isXEq[L - 1];
    component isYInc[L - 1];
    signal isVertical[L];
    signal vertStep[L - 1];
    isVertical[0] <== 1;

    for (var i = 0; i < L - 1; i++) {
        isYEq[i] = IsEqual();
        isYEq[i].in[0] <== Y[i];
        isYEq[i].in[1] <== Y[i + 1];

        isXInc[i] = IsEqual();
        isXInc[i].in[0] <== X[i] + 1;
        isXInc[i].in[1] <== X[i + 1];

        horizStep[i] <== isYEq[i].out * isXInc[i].out;
        isHorizontal[i + 1] <== isHorizontal[i] * horizStep[i];

        isXEq[i] = IsEqual();
        isXEq[i].in[0] <== X[i];
        isXEq[i].in[1] <== X[i + 1];

        isYInc[i] = IsEqual();
        isYInc[i].in[0] <== Y[i] + 1;
        isYInc[i].in[1] <== Y[i + 1];

        vertStep[i] <== isXEq[i].out * isYInc[i].out;
        isVertical[i + 1] <== isVertical[i] * vertStep[i];
    }

    isHorizontal[L - 1] + isVertical[L - 1] === 1;
}

template BoardSetup() {
    signal input privShipX[2];
```

Listing B.1b: Merkle tree commitment for two coordinates (continued).

```
    signal input privShipY[2];
    signal input privSalt;
    signal output pubCommitment;

    signal leaves[4];

    component lessX[2];
    component lessY[2];

    lessX[0] = LessThan(4);
    lessX[0].in[0] <== privShipX[0];
    lessX[0].in[1] <== 10;
    lessX[0].out === 1;

    lessY[0] = LessThan(4);
    lessY[0].in[0] <== privShipY[0];
    lessY[0].in[1] <== 10;
    lessY[0].out === 1;

    leaves[0] <== privShipX[0] * 10 + privShipY[0] + 1;

    lessX[1] = LessThan(4);
    lessX[1].in[0] <== privShipX[1];
    lessX[1].in[1] <== 10;
    lessX[1].out === 1;

    lessY[1] = LessThan(4);
    lessY[1].in[0] <== privShipY[1];
    lessY[1].in[1] <== 10;
    lessY[1].out === 1;

    leaves[1] <== privShipX[1] * 10 + privShipY[1] + 1;
    leaves[2] <== privSalt;
    leaves[3] <== 0;

    component eq[1];
    eq[0] = IsEqual();

    var pairIdx = 0;

    for (var i = 0; i < 2; i++) {
```

Listing B.1c: Merkle tree commitment for two coordinates (continued).

```
        for (var j = i + 1; j < 2; j++) {
            eq[pairIdx].in[0] <== privShipX[i] * 10 + privShipY[i];
            eq[pairIdx].in[1] <== privShipX[j] * 10 + privShipY[j];
            eq[pairIdx].out === 0;
            pairIdx++;
        }
    }

    component ship0 = ValidShip(2);
    ship0.X[0] <== privShipX[0];
    ship0.Y[0] <== privShipY[0];
    ship0.X[1] <== privShipX[1];
    ship0.Y[1] <== privShipY[1];

    signal nodes[3][4];
    nodes[0][0] <== leaves[0];
    nodes[0][1] <== leaves[1];
    nodes[0][2] <== leaves[2];
    nodes[0][3] <== leaves[3];

    component hasher[3];
    hasher[0] = Poseidon(2);
    hasher[0].inputs[0] <== nodes[0][0];
    hasher[0].inputs[1] <== nodes[0][1];
    nodes[1][0] <== hasher[0].out;

    hasher[1] = Poseidon(2);
    hasher[1].inputs[0] <== nodes[0][2];
    hasher[1].inputs[1] <== nodes[0][3];
    nodes[1][1] <== hasher[1].out;

    hasher[2] = Poseidon(2);
    hasher[2].inputs[0] <== nodes[1][0];
    hasher[2].inputs[1] <== nodes[1][1];
    nodes[2][0] <== hasher[2].out;

    pubCommitment <== nodes[2][0];
}

component main = BoardSetup();
```

Listing B.1d: Merkle tree commitment for two coordinates (continued).

This circuit is available here: `https://github.com/aleksandervestlund/zk-battleship/blob/master/circom_scripts/test/autogen_circuit2.circom`. It was generated using the following script: `https://github.com/aleksandervestlund/zk-battleship/blob/master/circom_scripts/gen_circom_1.py`. For the other numbers of ship coordinates, the Merkle trees need to be constructed such that the tree is complete.