Battleships

On The Blockchain

Powered by Nebulas
June 5rd, 2018

by Hardly Difficult

With Contributions From

Metjm
Core Concepts

ThePinkFreudian
Words

Twitch Chat
\o/

The Game: https://hardlydifficult.github.io/battleships
The Code: https://github.com/hardlydifficult/battleships

Nebulas Referral Link: https://incentive.nebulas.io/signup.html?invite=2nOH7

Executive Summary

Games on the blockchain offer many benefits over the traditional model to both developers and players by leveraging the network as a decentralized game server with zero downtime, no maintenance costs, an immutable and public history of games played, and a gaming experience that can never be taken from the player should the studio drop the game or go under.

However, blockchains store all of their information publicly and transparently. This presents a challenge when implementing games like Battleships - we don't want to allow your opponent to see your board, but we also need to ensure nobody cheats by moving a ship after the game has started.

This paper and the accompanying open-source game demonstrate how hash codes can be used to solve this problem. We create a decentralized multiplayer game server where everything is public and verifiable without allowing your opponent to spy on your board. And cheaters never win.

Introduction

Our implementation of Battleships addresses a number of known issues with traditional models.

The Problems

Potential for Cheating

When I was a kid, I played my fair share of games of Battleships. And I cheated, especially when playing against my brother. It was simple: if he hit, I would move the ship out of the way under the guise of placing a white peg down.

Unfortunately, players cheat, and research suggests they may be more likely to cheat in an online setting where consequences are scarce. And the problem gets much worse when there are items with real monetary value on the line, as seen with the recent CSGO Skins drama where odds were allegedly being manipulated to help with marketing.

People have a strong desire to win. If they can get paid as well, then some will dedicate serious time to finding a way to maximize their chances.

Server Downtime

Server downtime, whether scheduled or unanticipated, disrupts the player experience. Even in the case of scheduled downtime, players can be quickly frustrated, leaving negative impressions of the game experience and diminishing loyalty over time.

Additionally, many small game studios fail, and when they do they shut down their servers.

Hosting Costs

Battleships is an online, multiplayer game. In a traditional model, we would need dedicated servers to handle game logic, databases, and player accounts. Then these costs are scaled to the size of the player base, which could change drastically and suddenly over time. Even for smaller games, these costs can quickly add up. See the Amazon Web Services latest pricing tiers for cloud gaming servers as an example.

Loss of User Data

Halo 2, Rainbow Six 3, The Sims Online - we've seen over and over what happens when a centralized studio or server stops hosting a game. Loyal players lose their stats, assets, and considerable playtime logged.

Monetization Opportunities

Pay-to-win, skins, and hats are about as good as it gets these days. The blockchain provides several more attractive options for monetization. For example, in Battleships, you play to win Shipcoins!

The Solution

Blockchain. For the first time, we can actually create provably fair games. And by hosting the game on a blockchain, we are guaranteeing that players can continue playing as long as someone else is interested in joining in.

We are leveraging the blockchain as a decentralized multiplayer game server. Players communicate through the blockchain, avoiding the issues discussed above.

In order to ensure that players are not able to cheat like I used to, they need to share the layout of their board before the game begins. We use hash codes so this is possible without allowing the other player to learn anything about our board.

We also demonstrate monetization potential through the use of a limited-edition "Shipcoin" NEP5 token.

Battleships is a simple example, but the concepts here could be applied to other, more interesting games as well.

Technical Design

The dapp is comprised of a front-end interface and a smart contract. Both are written using Javascript. As with any secure application, the front-end cannot be trusted. All gameplay decisions need to be confirmed and enforced by the smart contract.

When the game begins, users lay out their board and then publish hash codes for each tile as explained in the secret management process below. Throughout the game, users may reveal the contents of a tile (e.g. “hit” or “miss”). When they do, they provide enough information for the blockchain to confirm that the contents have not changed since the game began.

The front-end simply polls the blockchain for updates and then visualizes that for the player. When players make a move, the front-end creates a transaction for it and then monitors the blockchain to see when/if the transaction is confirmed.

There is still a hosting fee, but it is paid by the users. Every transaction has an associated fee known as “gas”. The cost for this gas may be insignificant, as is the case with Nebulas, where you can play 100 full games for less than a penny (at current prices).

Sequence of Events

Each of the following steps are smart contract methods.

1) **Start or Join Game**
   After laying out their board for a game, a transaction is created to kick things off. If there is no game pending, this creates one; if there is a game pending, this will join that game.

   This request includes a hash code for each tile on the board (10x10 or 100 hash codes).

2) **Make Move**
   The first move of the game is the simplest. Who goes first is randomly selected by the smart contract when the game begins. To make a move, the player simply states which tile they are targeting (e.g. “B6”).

3) **Reveal Tile and Make Move**
   Every move after the first must also reveal if their opponent hit or missed. This is done by sharing the salt used to create the hash code for the tile being revealed.

   If your opponent hit and that hit sunk a ship, that sink must also be revealed with your next move.

   The information for the reveal(s) is sent with a “Make Move” request.

4) **Game Over I Lost**
   When the game is over, the loser is going to know this first as the final hit has not yet been revealed. Once your last ship has been sunk, or if you think you are going to lose but don't want to take a beating, you create a transaction conceding the game.

   It's not over yet. You may think you lost, but if your opponent was cheating, we'll discover that next.

5) **Confirm Winner**
   Once a player has conceded, the potential winner must prove that they were not cheating in order to redeem their prize. This is done by revealing their entire board. The smart contract confirms it was a valid layout and any hits or sunk ships during game play were revealed at the correct times.

   Once the winner has been confirmed, tokens are awarded. If the "winner" was actually cheating, they will be unable to confirm and the other player can collect a prize after a timeout period has passed.

Alternative Methods:

- **Redeem Win From Timeout:** After a brief timeout period has passed, you can redeem a win and prize by revealing your board (using the same technique as explained by "Confirm Winner" above), proving that you were not cheating during the game.
- **Cancel Game Which Has Not Started:** If you have a game pending, it may be canceled until an opponent joins in.

Secret Management

"Secret Management" refers to the process for selecting the hash codes shared at the start of the game in a way that prevents your opponent from learning your layout too soon while allowing the smart contact to confirm the layout was not changed during gameplay.

Secret Management Requirements:

- Board layout must not be readable until the game is over. Only the content of tiles which have been targeted are known.
- Moving a ship after the game begins results in a loss.
- Not declaring that a shot has hit or sunk one of your ships results in a loss.
- Pretending a hit was a miss or that the wrong ship was sunk results in a loss.

Secret Management Process:

The board layout is stored at the start of a game as a SHA-256 hash code per tile (referred to as the `tile_hash`). Each `tile_hash` is a hash code of the following data points concatenated:

- `tile_salt`: A SHA-256 hash code of the following points concatenated:
  - A secret word phrase that is randomly selected or entered by the player before the game begins.
  - The tile's position.

- `has_ship`: Binary variable representing whether a ship occupies that tile or not.

After the opponent has taken a shot, the tile is revealed. This is done by sharing the `tile_salt` and `has_ship`.

- Once the `tile_salt` is known, we are able to confirm that `has_ship` was not changed. This is because with SHA-256, a small change to the source data results in a large change to the hash code. It's effectively impossible to find a `tile_salt` which would allow you to change the `has_ship` value without also changing the resulting `tile_hash` value.

If the opponent's shot sinks a ship, the ship is revealed. This is done by sharing the name and position of the ship.

- All ship reveals are stored. If a sunk ship is not revealed right after it was sunk, we will detect this when tiles are later revealed at the end of a game (if not sooner).

In order to claim a win, your entire board must be revealed. This allows the smart contract to confirm that you were not cheating by validating the contents of each tile, confirming that the original board layout was valid, and verifying that any ships sunk during the game were revealed at the correct time.

This solution could be implemented strictly peer-to-peer, but peer-to-peer raises a few issues: sharing your IP address with your opponent, connecting behind firewalls, and enforcing timeout scenarios. This makes strictly peer-to-peer games undesirable.

Front End

We implemented the front-end using Javascript, HTML, and CSS, leveraging Bootstrap 4 and jQuery. The app interfaces with Nebulas via NebPay and the Nebulas Web Extension Wallet. It's a pretty simple front-end, polling for status changes and providing an interface for the game.

The front-end retrieves information from the blockchain via a Nebulas REST server. This server is a free service offered by the Nebulas organization to make creating dapps easy. Anyone could stand up a similar service. This does introduce centralization and a single point of failure for the game; however, it's not a critical component. The front-end could allow the user to select a different server should the Nebulas server go down or if the user does not trust the Nebulas organization.

Shipcoins

Instead of filling out a form for an airdrop or redeeming tokens from a faucet, players earn Shipcoins for playing the game - whether they win or lose. Then you play again. And again. And so on until all of the Shipcoins are distributed.

These are limited edition NEP5 tokens which will be tradable on a Nebulas DEX[1], once one is created. The winner of the first game gets 42 Shipcoins. With every win, the jackpot grows by 0.1%. Losers get 1% of the payout as well as a thanks for playing. The only real losers are people who rage quit - if you timeout or simply walk away, you get nothing.

Threat Model

This section reviews various ways someone may try to break the game.

Moving a ship after the game begins

Attack: A common cheat used while playing the board game version of Battleships, a player simply moves their ships a little and declares a "miss" instead of admitting it was a "hit".

Mitigation: At the start of the game, a hash for every tile is provided. If a player attempts to move a ship to cheat like this, they would then need to reveal a tile as a "miss" when that tile's hash was originally created with a "hit". The resulting hash won't match and the move will not be accepted. The player can either try again, making an honest move, or timeout and lose the game.

Missing ship(s) from the board

Attack: A player may choose to simply not include all 5 ships when declaring their layout at the start of the game. Their opponent will struggle to find all the ships and may believe they lost.

Mitigation: When the game ends, the winner must reveal their entire board. When this occurs, we require exactly five ships. If one is missing, the reveal will be rejected and that player will

[1] Note that there is no use case for Shipcoins, I would be shocked if anyone placed a buy order. But yes, technically, it *could* happen.

timeout. This allows the honest player, who appeared to have lost, to collect the win for playing fairly.

Too many ships on the board

Attack: A player may choose to play too many ships on the board, hoping to confuse their opponent.

Mitigation: Players will not be able to successfully reveal more than 5 ships; therefore, a player utilizing this strategy would not be able to win a game.

Invalid ship size

Attack: A player may try placing a ship which is too small (or too large).

Mitigation: Players will not be able to reveal a ship which is not a valid size and therefore cannot win a game this way.

Spying on the opponent

Attack: A player may use the blockchain's block explorer in order to try and discover their opponent's board layout.

Mitigation: The blockchain is only aware of hash codes until tiles are revealed, which does not occur until the game ends and the content of the tiles is revealed to both players. It is not feasible to learn anything from the hash code alone.

Taking two shots instead of one

Attack: A player may send multiple transactions, hoping to get multiple shots in before their opponent's turn.

Mitigation: The blockchain is sequential, so in this scenario one of the moves came first. The smart contract accepts that move and rejects the rest.

Declaring a "hit" was a "miss" or vice-versa

Attack: A player pretends a hit was actually a miss or vice-versa in order to mislead their opponent.

Mitigation: The reveal for that tile will fail. The player must re-submit, revealing that it was a hit, or timeout and lose the game.

Not declaring a sunk ship, or declaring it sunk too soon

Attack: A player honestly reveals a hit, but does not reveal that the hit sunk the ship. Or, they reveal that it was "sunk" by pretending the ship is smaller than it actually is.

Mitigation: The dishonest player here can never win after this occurs, as they will be unable to successfully reveal the board. When they try, the smart contract will see the undeclared or incorrectly declared sunk ship.

Timeout

Attack: A player takes too long, the browser crashes, or they rage quit, leaving the other player hanging.

Mitigation: Each turn gives the player 45 seconds to make a move. After this time has passed, the other player may redeem the win and a prize by simply proving they were not cheating during the game.

Not conceding a lost game

Attack: In order to drag out the game, the opponent never concedes and continues to make shots (so that a timeout does not apply).

Mitigation: By doing so, the honest player will eventually have all of their boats sunk. Once this occurs, they concede. However, the cheater will be unable to reveal, forcing them into a timeout. Therefore, the honest player will be able to collect the win.

Invalid moves

Attack: A player attempts an invalid move, such as out-of-bounds or targeting a cell which has already been targeted.

Mitigation: The smart contract knows the rules of the game and simply will not accept invalid actions. If they try, the transaction will fail and the player can submit a different move.

Break it once and that attack can be reused

Attack: Brute force SHA-256 to find a secret which allows for changing data after the start of the game. If discovered, the same data could be used game after game.

Mitigation: This is practically impossible. To improve security, we could add the `game_id` to the hash creation. However, we don't do this as it adds one more transaction to the flow, slowing an already slow game.

Playing yourself

Attack: Someone creates multiple accounts, opens a second browser, and plays themself over and over to farm tokens.

Mitigation: We could limit how many times each account may play, or consider the move count to help detect when people are not really playing. Alternatively, a small fee to play the game could help discourage this.

Issues & Possible Improvements

There are some issues that this implementation still faces.

Gameplay Speed

Issue: Each move is a separate transaction to increase game integrity and eliminate the possibility of cheating. In a game that may take over 100 moves to complete, at an average transaction speed of 10 - 15 seconds, this can be a lengthy process.

Improvement: Deploy to a faster blockchain. There are several sidechain solutions in the works which promise to offer very high performance at a low cost, ultimately secured by a reputable main chain. ZombieChain is one such sidechain in the works for Ethereum targeting gaming use cases.

User Expense

Issue: In our implementation, users only pay for gas and are rewarded with tokens. Even though it's very cheap to play a game, it's not entirely free. There is a cognitive load associated with approving transactions and considering the cost of gas, and it's a turnoff for users. Having to pay anything will significantly reduce the number of players who try the game.

Improvement: An alternative blockchain which pushes the costs back to the developers, such as EOS, may make the experience more attractive for players.

Tedious

Issue: Each move is a transaction, and each transaction must be approved by the user manually. This is the design used by the wallet extension and is generally important to maintaining the user's security: The user can trust one wallet extension instead of having to trust each individual dapp they interface with.

Improvement: If the game became successful or we were otherwise a reputable brand, users might be willing to trust our game. If we are trusted, users can share their keys with the app (like they do with the wallet extension today). Then transactions could be processed automatically without prompting for confirmation with each action.

Gameplay Experience

In order to simplify development, some commonly expected features were not included. These features were excluded for dev-time considerations, but should be possible to add on.

- **Matchmaking.** Something more interesting than simply matching one person to the very next one who plays.

- **Reveal Experience.** For an improved play experience, the reveal action could be made right away. Then the player considers their next move and takes a shot. Implementing this would require another transaction in the process, and may make the game even longer.

- **Challenging.** If you strongly suspect your opponent is cheating, there should be an option to challenge them. Once a challenge has occurred, the game is over. If your opponent was not cheating they win; otherwise, the win goes to the challenger.

Other Technical Considerations

Considerations which could be addressed but have been excluded from this implementation for simplicity:

- **Chain reversals or orphan blocks may occur.** This is a normal part of the consensus process for blockchains. The impact to dapps is data may appear and then a few seconds later disappear. The smart contract will roll with this gracefully, but the front-end would need some additional capabilities to handle this smoothly.

- **Clock drift.** Time is difficult for networks. During consensus, different entities will be producing blocks, and their clock may not align perfectly with others. Currently, the game uses time for the timeout feature, and with the current implementation it is possible for the countdown to go up and down (e.g. T-5, 4, 5, 3, 1, etc.).

- **Sniping.** A player can watch the mempool for pending transactions, attempt to glean some information from it, and then post a transaction with a higher gas price so it makes it into the chain before the original transaction. For example: I allow a timeout to occur and then my opponent submits a transaction to claim the win. That transaction includes their entire board. If I see this in the mempool and then quickly make a move that's accepted into a block first… well, then it's easy to win. This could be mitigated by adding another step to the timeout process.

- **Merkle Tree.** Revealing the board is an expensive transaction. Using a Merkle Tree could simplify that final, full board reveal.

Other Possible Applications

The main capability that Battleships On The Blockchain demonstrates is the ability to establish player selections or other secret information which can be broadcast to a public blockchain, preventing players from cheating while ensuring that no-one can spy on that information prematurely. Throughout a game, select pieces of information may be verifiably revealed, and at the end of the game we can complete a final check confirming that there was no cheating along the way.

There are several types of games which could be implemented using these techniques. We discuss some simple examples below.

Example: Rock Paper Scissors

The challenge with Rock Paper Scissors is ensuring that neither player could possibly know the other's selection before making their choice. Using the techniques outlined above, we could implement this in a provably fair way as follows:

1) Bob selects "Paper," hashes it along with a salt, and publishes the hash to the blockchain.
    - Jane has not yet made a selection for Bob to try and spy on.

2) Jane selects "Rock" and publishes that to the blockchain.
    - Jane can see Bob's hash, but has no idea which selection it represents.

3) Bob submits his selection and the salt used to create the hash, proving its validity.
    - By using hash codes, Bob is not able to change his selection after seeing Jane's. The worst he could do is rage quit.

Example: Hearthstone ®

For a game like Hearthstone ® or Magic: The Gathering ®, we need to prevent the other player from knowing which cards are in your deck, and we need a way to shuffle the cards fairly. We could do this as follows:

1) Each player privately prepares the following:
    a. **`cardid_to_card:`** A list of the cards in their deck so they may be referenced by `cardid`.
    b. **`orderid_to_cardid:`** A list representing a random order of `cardid` which defines the shuffle order for their opponent.

2) Each player publishes the hash codes of their selections for the `cardid_to_card` and `orderid_to_cardid` lists to the blockchain so we can prove integrity later on.
    a. Hash codes must include some salt information which can later be revealed.

3) To draw a card:
    a. Jane reveals the next (or first) `cardid` from the `orderid_to_cardid` list she has published.
        i. This is done by sending both the `cardid` and the salt used to create the hash in the `orderid_to_cardid` list.
    b. Bob looks up the card from his private `cardid_to_card` list.

4) To play a card:
    a. Bob reveals the card by sending both the card and the salt used to create its hash in the `cardid_to_card` list.

In order to fairly select cards from a shuffled deck, your opponent is effectively selecting a specific order of cards without knowing what the cards are, just as though they were selecting cards from a face-down pile. After this is complete, we can fairly draw cards, keeping the information known to just one player until they are ready to play it.

You could even run with these concepts to reveal only a specific attribute from a set of attributes. In Hearthstone ® , for example, I draw a card with an on-draw effect that does something. My opponent needs to know about the attribute so we can process the on-draw effect, but I do not want them to know which card it is yet. Using the techniques above, another list of hash codes representing `cardid_to_attribute` could be created in order to verifiably reveal only specific attributes as appropriate.

Conclusion

Our goal is to help people understand how dapps work a little better and to get you thinking about new possible applications.

Let’s be honest: the play experience here is not ideal. A few of the suggestions above could help, but unless the game has something real at stake players will not want to sacrifice the benefits of the typical centralized server solution - namely, a free and fast expeniece.

In the future, blockchain performance will improve, and the experience could get closer and closer to what players expect today while adding in benefits blockchains provide, such as the ability to create provably fair games.

Thanks for taking the time to read this paper. Please do give the game a shot as well. And if you start to create a game of your own, let us know.

Appendices

Disclaimer

You may want to consult a lawyer before launching your own dapp, particularly if gambling or other regulated activities may be involved.

Note that every transaction is a taxable event in some countries.

Definitions

Term	Technical Definition
Blockchain	A **distributed network** which reaches consensus on an absolute ordering of events or transactions. This game was implemented using the Nebulas Blockchain specifically, but the techniques used would work on any blockchain with **Smart Contract** support.
Dapp	Short for "decentralized application," a dapp runs back-end logic on a peer-to-peer network in place of a traditional centralized server.
Decentralized Network	A distributed network is decentralized if it does not depend heavily on a specific company or entity.
Distributed Network	A network is distributed if there is no single point of failure. This is often achieved with a mesh network of connected peers.
Hash code	A hash code is the result from executing a hash function. Hashing is the process of taking an arbitrary amount of information and creating a fixed amount of information representing it, the hash code. A small change to the source data results in a large change to the hash code.
Immutable Content	Content which cannot be lost, corrupted, or modified once it has been established.
NEP5	The Nebulas standard for offering tradable tokens. These tokens will be easily managed by Nebulas wallets and tradable on an Nebulas exchange.

Merkle Tree	Merkle trees are a way of representing a large amount of information with a tree of hash codes such that a subset of the real data can be verified without considering all the data.
Salt	A salt is arbitrary information used as an additional input for a hash function to defend against dictionary attacks.
SHA-256	A specific implementation of a hash function. This is the one we used, but there are many other valid options to consider.
Smart Contract	Program logic / code which is hosted on and executed by a blockchain network.