# Blockchains and Distributed Ledgers – INFR11238

## Smart Contract Programming Coursework Assignment

Andrey Demidenko

12/11/2023

This report supplements a smart contract, which implements a game whereby players A and B each submit a number in range [1,100], and A wins if the sum of the numbers is even whereas B wins if the sum is odd. The winner is rewarded the sum in Wei to their account.

The contract name is *SumGame*. Its code can be found in the Appendix at the end of the report. Its address on Sepolia is 0x9152F8eB07e3FF43dC23F8931Be7b2Eb449AF74C.

# 1. High-Level Description

This section describes all phases of the game and the timing constraint players are subject to. Most of the explanations, why certain patterns and design aspects were chosen, are included in subsequent sections as part of reasoning about security, fairness and efficiency, and any trade-offs between them.

## 1.1. Game Phases

The game emulated by *SumGame* contract has four phases: *Registration*, *Encrypted Bidding*, *Bid Revelation*, and *Reward/Refund*. Every phase involves each player performing a symmetric action, i.e. calling a designated function of the contract corresponding to the game phase.

### 1.1.1. Registration Phase

At the very beginning, anyone who wants to join the game must register. This can only be done if there is at least one vacant spot, i.e. no two players are already registered. Moreover, one cannot join the game until a previous game has finished and both players have withdrawn their rewards. The last joining condition is the registration fee, which is equal to the doubled maximum bid amount (maximum bid is 100, thus the registration fee is 200 Wei). At the end of the game, the winner is rewarded the registration fee plus the sum of the bids, whereas the loser is refunded the registration fee minus the sum of the bids.

For example, let player A's balance at the beginning of the game be $a$, and player B's balance $b$. They join the game, and their balances become $a - 200$ and $b - 200$ respectively. Suppose A bids 40, and B bids 80. The sum $40 + 80 = 120$ is even, thus A wins. At the end of the game, A is rewarded $200 + 120 = 320$ Wei, and B is refunded $200 - 120 = 80$ Wei. The balances become $(a - 200) + 320 = a + 120$ and $(b - 200) + 80 = b - 120$ respectively.

The registration fee has to be twice the maximum bid because at the end of the game, the winner must have "the sum of the bids (in Wei) rewarded to their account". It is quite natural that the amount one player wins should be the same amount the other player loses (otherwise, the contract balance would need to be maintained by the owner to partially reimburse either of the two parties). Therefore, in case both players bid 100, player A wins (100 + 100 = 200 is even) and must have 200 Wei more than they had *before joining the game* (gas fees are not considered here), whereas player B must then have 200 Wei less than before the game. In other words, if A had a balance $a$, and B had a balance $b$ before joining, then after the game, A's balance would be $a + 200$, and B's balance would be $b - 200$. This can only be accommodated if the total "bank" of the game is 400 Wei.

If the players paid the registration fee equal to just the maximum bid (100 Wei each), then the winner's balance at the end would be $(a - 100) + 200 = a + 100$, i.e. only 100 Wei more than their initial balance. This would violate the game specification. The same violation would happen if the players paid the same amount in Wei as their bid (this would happen at the *Encrypted Bidding* phase).

For instance, A bids and pays 40, B bids and pays 80, so the contract balance is now 120 Wei. The sum is even, thus A wins and is paid 120 Wei, and B is paid nothing. But then again, A's balance would only be $(a - 40) + 120 = a + 80$. Moreover, such approach would leak the bid to an adversary because it would be the same as the value of the transaction (which of course cannot be hidden using encryption).

It is important to note here that when the players register, they do not know or choose if they are player A or B. The order in which they join does not matter either, they are just registered as Player1 (who joined first) and Player2 (who joined second). Which one of the players is A, and which one is B, is decided fairly (pseudo-randomly) during the last phase (section 1.1.4). This is to prevent front-running as is explained in section 4.3.2 of the report.

Such design is also a very good incentive for both players not to timeout (accidentally or deliberately) as they would lose an amount of money equal to the maximum potential win.

- To register, one has to call the function $register()$, the transaction value must be 200 Wei.

It is also possible to quit the game for a registered player if no opponent has joined the game. The registration fee is then returned to the player. If both players have registered, and at any later point during the game, it is impossible to quit.

- To quit the game, a player has to call the function $quit()$.

### 1.1.2. Encrypted Bidding Phase

After both players have registered, the phase of placing an encrypted bid happens. Each player, who is registered and has not yet placed a bid, must submit their bid as follows:

1) Choose a number in the range [1, 100] they want to bid.
2) Choose a nonce, i.e. a string of characters of any length (it serves as a password). Why the nonce (and in general, such commitment scheme) is necessary, is explained is section 4.3.1.
3) Compute the SHA-256 hash function of the chosen number concatenated with the nonce, separated by a hyphen ("-"). The importance of the hyphen is also explained in section 4.3.1.
4) Submit the 32-byte (64 hex digits) output of SHA-256 in the format $0x1a2b3c4d$… (i.e. prepend $0x$ to the value). This is due to the way Solidity expects to receive a *bytes32* input.
   To compute a SHA-256 hash function, one might use an easily-found web service, e.g. https://codebeautify.org/sha256-hash-generator

---

*Example 1*

1) *Player A wants to bid* $\mathbf{75}$.
2) *Player A chooses a nonce* $\boldsymbol{password}$.
3) *Player A computes SHA-256 of* $\mathbf{75}$-$\boldsymbol{password}$*:*
   *bfd1c084da36a8c6de032648b770dcdb6309ae91bf2fb23fe1fe867d83d3a6b6.*
4) *Player A submits their bid as follows:*
   ***0xbfd1c084da36a8c6de032648b770dcdb6309ae91bf2fb23fe1fe867d83d3a6b6***.

---

- To place a bid, a player has to call the function $placeEncryptedBid(encrBid)$, where $encrBid$ is the output of SHA-256 hash function in the format as described above.

### 1.1.3. Bid Revelation Phase

After both players have submitted their encrypted bids, the bid revelation phase takes place. Each player, who is registered and has not yet revealed their bid, must do so by submitting the number and the nonce they chose during the previous phase, with no encryption. The revealed bid must be valid, i.e. in the range [1, 100], otherwise the revelation will fail instantly.

The revelation will only be successful if the player honestly submits the two values they used to compute SHA-256. This is because the revealed values will be hashed again (adding a hyphen in between) and compared with the previously submitted encrypted bid.

*Example 2*

*Assuming the same player A from the example above, they reveal a pair* $(\mathbf{75}, \boldsymbol{password})$*. The hash of the revealed values is computed (*$SHA256(\text{"}75\text{-}password\text{"})$*) and it, of course, matches the previously submitted encrypted bid, so player A's revelation succeeds.*
*If player A revealed* $(\mathbf{75}, \boldsymbol{password1})$ *instead, the computed value of SHA-256 would be cdafa16ab7a6fd6723657e5b0e674a4a568a469495318982144d25a29d87a007 and would obviously not match their encrypted bid, so player A's revelation would fail.*
*If player A had revealed a different bid with the same nonce, revelation would have failed too.*

It is important to note that during the previous phase, it is impossible to ensure that the player's submitted number is valid, i.e. is between 1 and 100. Thus if a player submitted the hash computed using an invalid bid amount, they will not be able to successfully reveal it. Nor will they be able to quit or trick the game to accept an invalid bid, so they will just timeout.

- To reveal a bid, a player has to call the function $revealBid(bid,\ nonce)$.

### 1.1.4. Reward/Refund or Withdraw Phase

To avoid any confusion, the two names of this phase (*"Reward/Refund"* or *"Withdraw"*) are used interchangeably throughout the report.

After both players have successfully revealed their bids, the final phase of withdrawing the rewards starts. Each player, who is registered, must withdraw a reward or a refund, depending if they have won or lost. As explained in section 1.1.1, the winner gets back their registration fee plus the sum of the bids whereas the loser gets the registration fee minus the sum of the bids. Once a player has withdrawn their reward/refund, they are unregistered. When both players have withdrawn, and thus been unregistered, a new pair of players can register for the game.

According to the game specification, the winner is player A if the sum of the bids is even, and B if it is odd. As mentioned earlier, it is only now that it is decided which of the players is A, and which one is B (at the registration phase, they were just recorded as Player1 and Player2 based on the order in which they joined).

***Algorithm 1:*** *Decision who is A and B*

Let the player who joined first be $Player1$, and the one who joined second be $Player2$. Let their bids be $b1$ and $b2$ respectively.

$$A \leftarrow Player1,\ B \leftarrow Player2 \quad if \quad (1 \le b1 \le 50\ \ and\ \ 1 \le b2 \le 50)\ or\ (51 \le b1 \le 100\ \ and\ \ 51 \le b2 \le 100)$$
$$A \leftarrow Player2,\ B \leftarrow Player1 \quad otherwise$$

In other words, Player1 is A if ***both*** bids are in range [1, 50] or ***both*** bids are in range [51, 100]. In any case when one of the bids is between 1 and 50, and the other is between 51 and 100, Player2 is A.

As it will be shown in section 4.3.2, there is no incentive for any player to submit a bid from a certain range in order to win. Therefore, the algorithm is always fair to both players.

- To withdraw the reward/refund, a player has to call the function $withdrawReward()$.

## 1.2. Timeout Control

Each phase, except for the registration phase, has a constraint on the time during which it must be completed by both players. In particular, there must be no more than 25 Ethereum blocks (assuming the 12-second mining time of one block, this is approximately 5 minutes) between the start and end of a phase. The timeout mechanism prevents several possible versions of a Denial-of-Service attack, as will be explained in section 4.1.2.

A phase starts when both players complete the actions corresponding to the previous phase. Namely, when the last player creates a transaction to perform the previous phase action, the number of the block, in which that transaction is added, is recorded as the start of the current phase. The end is then set as current block number plus 25. When any of the players tries to perform the current

phase action by creating a transaction, the number of the block in which that transaction is added must be no larger than the previously set phase end. Otherwise, the player has "timed out" and cannot finish neither the current phase nor the game in general.

The only exception is the *Registration* phase, whereby one player can be "sitting in the lobby" for as long as they want. If they decide not to wait anymore, they can quit. Thus there is no point in creating a timeout for the first phase.

---

*Example 3*

*Player1 registered and is waiting for someone to join. Player2 joins via a transaction recorded in block* $X$*. Then* $X + 25$ *is set as the end of Encrypted Bidding phase. Player2 then places their encrypted bid at block* $Y2 = X + 10$*. Player1 places the bid at block* $Y1 = X + 24$*. Both bids were submitted in time, nobody timed out. Moreover,* $Y1 + 25$ *is now recorded as the end of Revelation phase, and each player now has around 5 minutes to reveal their bid*.

---

If one player has timed out, another player can claim the entire bank (or what is left if the opponent timed out during the *Withdraw phase*), given they have not timed out too. Moreover, both players are unregistered, so other people can join the game afterwards.

- To claim the bank, a player has to call the function $claimBankIfOpponentTimedOut()$.

---

*Example 4*

*Assume* $X$ *was set as the end of Reveal phase. Player1 reveals their bid at block* $Y1 = X - 10$ *which is in time. Player2 tries to reveal at block* $Y2 = X + 1$ *(i.e. 1 block after the phase has ended). Player2 cannot reveal because they timed out. Since Player1 did not time out, but their opponent did, Player1 can now call the function* $claimBankIfOpponentTimedOut()$*. Player1 then receives 400 Wei, and both players are unregistered.*

---

If both players have timed out, or if one of them timed out but the other one has not claimed the bank, the owner of the contract (admin) can kick out both players. If both players timed out, the admin will claim the entire bank themselves and unregister the players (this is yet another incentive not to timeout). However, if only one player timed out, the admin will try to send the money (bank) to the player who did not time out. If for any reason the transfer fails, the admin will claim the bank themselves. Both players are unregistered in any scenario.

- To kick out the players, the contract owner has to call the function $adminKickOut()$.
  The function cannot be called by anyone else apart from the owner.

# 2. Low-Level Description

This section describes the data types and structures that were used in the contract code, the mechanism to maintain the game state, as well as what variables and functions are or are not visible to the players and why.

## 2.1. Types and Structures

Firstly, there is a defined *enum Phase* which can take one of the four values: *Register, Bid, Reveal,* and *Withdraw*. Using an enum simplifies the logic of maintaining the game state, as will be explained in section 2.2. It could be substituted with many booleans, but this would require more complicated logic, and thus even more code (for an already very long contract).

Secondly, the contract uses a *struct Player* which has four fields: *player* (payable player's address), *encryptedBid* (submitted SHA-256 hash), *bid* (revealed number), and *phase* (phase player is currently in). How they are used is described in detail in the next section too.

The last general point to mention here is that all variables containing the values of unsigned integers are always declared so as to occupy as little memory as possible. For example, each player's bid is represented as *uint8* because its value will always be in range [1, 100], the sum of the bids is also *uint8* (maximum value 200 can still be stored using 8 bits), whereas the value of the bank is at most 400 (sum of the registration fees), so it is impossible to store it with only 8 bits and it is declared *uint16*.

## 2.2. Game State Control

The game state is maintained using two variables of type *Player*: *player1* and *player2*. They store all data representing each of the players: their address, encrypted bid, revealed bid and the phase they are currently in. The latter should be interpreted as "player is ready to …", e.g. if $player1.phase = Phase.Bid$, it means that the player has registered and is ready to bid.

The contract also contains a *Phase* variable *currentPhase*, which describes the current phase of the entire game, and a *uint256* variable *currentPhaseEnd*, which represents the last block number where the transaction corresponding to the current phase action can be recorded. It is crucial to note here that these two variables are updated when the last player completes the current phase action. Then *currentPhase* changes to the next one ($Register \rightarrow Bid \rightarrow Reveal \rightarrow Withdraw$; then wrap around), and *currentPhaseEnd* is assigned the value $block.number + timeout$ (where $timeout = 25$). If any of the players tries to perform an action after *currentPhaseEnd*, they will fail, and the two state variables will not change.

All variables listed above are used to maintain the timeout constraint of the game. In particular, player1 is considered to have timed out if $block.number > currentPhaseEnd$ and $player1.phase = currentPhase$. In other words, if the current phase has ended, but the player is still in that phase, then they have timed out.

## 2.3. Variable and Function Visibility

All functions described in section 1 are external and thus are only for the players (and admin in case of $adminKickOut()$) to use. They are in fact the interface of the game. Moreover, there are several public variables that are "visible" to the players as they have automatically created getters for them. These variables are the following:

- $minBid = 1$ is the minimum bid amount
- $maxBid = 100$ is the maximum bid amount
- $timeout = 25$ is the phase duration in blocks
- $registerFee = 200$ is the registration fee (doubled maximum bid)
- $currentPhase$ is the current phase of the game, as explained in section 2.2

All these variables provide a convenient user interface by giving enough information about the game. Particularly, players may use $currentPhase$ to view if their opponent timed out.

All other variables are private because the players either ***do not need*** that information or ***must not*** know it as it would give them certain advantage over their opponent.

---

*Example 5*

*At the beginning of the Bid phase, player1 placed a bid (thus $player1.phase = Reveal$ now).*

- *After around 5 minutes, they query the value of $currentPhase$ and observe it is still Bid. This means that their opponent player2 has not placed a bid before the phase end. Thus player1 can now confidently call function $claimBankIfOpponentTimedOut()$.*
- *Alternatively, if after around 1 minute, player1 queries $currentPhase$ and sees that it has already changed to Reveal. This means their opponent player2 has placed a bid in time, so the next phase has started. Then player1 can call $revealBid(bid,\ nonce)$.*

---

In fact, there is no need for a player to check *currentPhase* before trying to complete the current phase action. They can simply try calling a function corresponding to the next phase, and if their opponent still has not made their move, the function will not execute (appropriate modifiers will prevent it), so the gas fees will be returned to the calling player. Thus it just provides an additional way of observing the game state to enhance the user experience (UX).

# 3. Gas Evaluation

This section lists the gas costs of deploying and interacting with the contract, assesses fairness between players in terms of gas used, and describes design aspects used to increase fairness.

## 3.1. Deployment

The cost of deploying the contract on Sepolia testnet is $3{,}659{,}160$ gas. Assuming the normal gas price of 2.5 Gwei (on average it stays around that for Sepolia), the price of deployment is $0.009149$ ETH.

The absolute total price of a transaction can always be calculated using this formula:

$$price[ETH] = gasCost[gas] * gasPrice[\frac{ETH}{gas}]$$

Therefore, interaction gas evaluation (section 3.2) will only list the gas costs of the functions.

## 3.2. Interaction

The cost of interacting with the contract depends on what function is called. Some of the functions are only called by one player (or admin), these are the timeout-related functions and the function $quit()$. To them, the notion of fairness is not applicable because one person simply performs a certain action and spends a certain amount of gas, and so there is no symmetric action performed by another user. These functions are listed as "asymmetric", and for them, just one fixed transaction cost is given (Table 1).

All functions corresponding to the four phases of the game are, on the other hand, "symmetric", i.e. they are performed by both players. Thus for those functions, it is important to consider whether the gas cost is different for the player who calls them first and for the one who calls them second (Table 2).

For all functions listed below, the gas costs are not constant. This is for different reasons, such as using *string* variables (whose length is not known in advance and varies depending on the player's input), or transferring Ether in a function (which can incur arbitrary additional cost). Therefore, the costs given in the tables below are the average costs observed over multiple calls to each of the functions.

| "Asymmetric" Function | Average Gas Cost |
|---:|:---:|
| $quit()$ | 40,921 |
| $claimBankIfOpponentTimedOut()$ | 49,944 |
| $adminKickOut()$ | 47,684 |

*Table 1 – Asymmetric functions' gas costs*

| "Symmetric" Function | Avg. Gas Cost (1st player) | Avg. Gas Cost (2nd player) |
|---:|:---:|:---:|
| $register()$ | 85,919 | 89,606 |
| $placeEncryptedBid(encrBid)$ | 59,856 | 68,442 |
| $revealBid(bid,\ nonce)$ | 53,457 | 62,251 |
| $withdrawReward()$ | 52,267 | 53,273 |

*Table 2 – Symmetric functions' gas costs with respect to the order of players calling them*

As we can see from Table 2, the player who calls a function second during any phase spends more gas. This is because a part of the game state changes only happen when the last player makes a move. It is not ideal, but the gas spending of both players is still very close.

### 3.2.1. Fairness

Fairness is mainly ensured via the PULL design pattern, i.e. each player pays for each phase themselves. In particular, each player withdraws a reward at the end (at their own expense), instead of one player (the first to call) spending gas on transfers to both themselves and their opponent.

Another design aspect that is aimed to increase fairness between players is the idea of making game state changes as "symmetrically" as possible, i.e. so that the same action would be performed by both players, even if it is not meaningful for the first player. For instance, in the *Reveal* phase, the sum of the bids is calculated no matter if both players have revealed their bids. Obviously, this sum calculated by the first player who called $revealBid(bid,\ nonce)$ will just be their own bid, but this operation is still performed to make it fairer for the second player, whose sum calculation will actually matter. Such design pattern is applied wherever possible in the code, but it is impossible to use it all the time.

### 3.2.2. Efficiency and Its Trade-offs

Both techniques described above (PULL and "symmetric" game state changes) increase fairness at the expense of efficiency. The total gas spent by both players could have been much lower if *all* game state changes were only made after the last player's call to a function from a certain phase. Another example of a fairness–efficiency trade-off is the PULL pattern in general, however it is crucial not only to increase fairness, but also to guarantee security, as will be explained in the next section.

The last general reduction in efficiency comes from the use of $register()$ function. It could have been possible to avoid the *Registration* phase by sending the 200-Wei fee together with the hash at the *Encrypted Bidding* phase, thus merging the two phases. However, this is a trade-off between efficiency and the UX. Removing the *Registration* phase would disable the right of a registered player to quit if they do not want to wait for an opponent anymore. It would also make keeping track of players' timeout during the bidding phase infeasible. Moreover, it would appear less natural for a user to submit an encrypted bid immediately, with no preparatory steps.

# 4. Potential Hazards and Their Mitigation

This section provides a thorough list of hazards the contract could be subject to, as well as explains what techniques, patterns and certain game logic aspects are used to mitigate them. The general assumption for an adversary is that they can observe other users' transactions on the network, as well as tamper with block timestamps and prioritise the order of transactions in the block. In other words, an adversary is the most powerful one possible on the blockchain – the miner.

## 4.1. DoS (Denial-of-Service) Attacks

There are several possible versions of a DoS attack that are considered in this subsection.

### 4.1.1. Griefing

This attack is performed by an adversary who deliberately sets their address on the blockchain so as not to receive Ether. It would have worked if the PUSH design pattern was used for the *Withdraw* phase of the contract, i.e. if there was a single function called by one of the players that would send Ether to both players. Thus if an honest player called such function, the transfer to the adversary would fail, and the contract would either get stuck or have to revert to the previous state. Thus a deadlock would occur, so the game would never finish.

To prevent such an attack, the PULL pattern is used, so each player is responsible for withdrawing their own reward/refund. It creates a certain UX–security trade-off whereby a player calls more functions (it makes a simple game more complicated), but it is more secure.

### 4.1.2. Registering and Not Playing

It is possible that either one or both players register for the game but do not play. This is the reason a timeout was introduced at each phase, as was explained in section 1.2. It is important to note that it is based on the block numbers rather than on the timestamps (although it creates some imprecision of the phase duration). An adversarial miner can tamper with the block timestamps, but not with the block numbers.

As previously discussed, the function $claimBankIfOpponentTimedOut()$ allows one of the players to claim the bank if their opponent timed out. This makes an individual adversary's DoS attack pointless because the honest player will just claim the entire bank, and the adversary will achieve nothing but lose their money spent on the registration fee.

In case of two colluding adversaries who both register for the game but do not further interact with the contract, the function $adminKickOut()$ lets the contract owner kick both adversaries out, claiming their registration fees.

## 4.2. Re-entrancy Attack

A re-entrancy attack can be arranged by an adversary who created a special fallback function in their smart contract. When a certain function from the *SumGame* contract is called, it sends Ether to its caller's address and calls the malicious *fallback()* function. That *fallback()* function can then call the

*SumGame*'s function again to send more money to the adversary, etc., thus draining the contract's balance until there is nothing left.

There are 4 functions that send Ether to their caller and could thus be subject to a re-entrancy attack: $quit()$, $withdrawReward()$, $claimBankIfOpponentTimedOut()$, and $adminKickOut()$.

To prevent a re-entrancy attack, the contract uses the "Checks–Effects–Interactions" pattern wherever possible. In particular, in $claimBankIfOpponentTimedOut()$, $withdrawReward()$ and $quit()$, the players are first unregistered, and only then Ether is sent to their address. Since all these three functions have a modifier $isRegistered()$, requiring their caller to be a registered player, re-entrancy is not possible.

The pureness of the pattern is slightly violated by the function $adminKickOut()$ where the state variables $bank$ and $currentState$ are updated only after the transfer is completed. This is done because the value of $bank$ is needed to determine the amount of Wei to send to one of the players or the admin, before it is set to zero. Of course, the admin is honest (it is me), but the players might not know it, so the function is designed so that no re-entrancy could be performed by the owner either. In particular, the players who have timed out are unregistered before the money is transferred to the admin, so if the admin's fallback function calls $adminKickOut()$ again, the mandatory checks for the players to have timed out will fail, and the function will not be executed again.

## 4.3. Front-Running

The idea behind a front-running attack performed by an adversary is for their transaction to be placed in a block before the victim's transaction. More specifically, an ordinary blockchain user can set higher gas price for their transaction to be prioritised by the miners (who want to maximise their rewards and record the transactions by gas price in descending order). If an adversary wants their transaction to be executed after the victim's, and they can observe the network, they can set a *slightly* lower gas price to make sure the transaction is recorded after the victim's transaction. In case of an adversarial miner (our global assumption), they can place their own transaction wherever they want in the block.

The techniques used to mitigate all possible forms of front-running are presented as a sequential list of attacks and defences. The list is split into two parts according to the way an adversary tries to cheat – by either adjusting their bid based on their opponent's bid or repeating that bid.

For the purpose of this section, let Bob be the honest player and Eve the miner-adversary. Assume the following initial contract design (which will gradually improve throughout the section):

- The player who registers first becomes player A, and the one joining last becomes B.
- A single $bid()$ function that submits the bid number in the raw form (no *Reveal* phase).

### 4.3.1. Adjusting Bid

***Attack 1***

*Assume Bob is player A, and Eve is player B. Bob submits $X$ via $bid(X)$. Eve observes Bob's bid and submits $Y$ via $bid(Y)$ s.t. $X + Y$ is odd.*

***Defence 1***:

- Encrypt the bids with a cryptographic hash function, e.g. SHA-256.
- Introduce the *Reveal* phase where each player would reveal the raw bid number (its hash is then computed and compared with their encrypted bid).

***Attack 2***

*Assume Bob is player A, and Eve is player B. Bob submits $X$ via $placeEncryptedBid(SHA256(X))$. Eve observes Bob's bid, looks at the list of precomputed hashes of all numbers from 1 to 100 and figures out what number Bob encrypted. Eve bids $Y$ via $placeEncryptedBid(SHA256(Y))$ s.t. $X + Y$ is odd.*

***Defence 2***:

- Introduce a commitment scheme: players must encrypt the concatenation of a bid and nonce. During the *Reveal* phase, they must then reveal both these values. It both hides the bid value from the adversary and binds the player to the hash they have committed to, so they cannot later reveal a bid number different from the one they used to compute the hash.

***Attack 3***

*Assume Bob is A, Eve is B. Eve submits the following value:* $placeEncryptedBid(SHA256("65eve"))$ *at the Bid phase. Bob also places his bid. At the Reveal phase, Bob reveals the values* $(50, "bob")$*. Eve observes that and reveals her values as* $(65, "eve")$ *and wins because sum is odd. If Bob instead reveals the values* $(51, "bob")$*, Eve would observe it, reveal her values as* $(6, "5eve")$ *and thus also win.*

*So, Eve can always bid a hash with a 2-digit number where one digit is even, and the other is odd. Eve then reveals either only the first digit as the bid (second digit plus nonce is then the revealed nonce) or the original 2-digit number with the original nonce, depending on the Bob's revealed number.*

***Defence 3***:

- Add a delimiter, e.g. a hyphen ("-"), between the bid and the nonce when encrypting them.

### 4.3.2. Repeating Opponent's Bid

With *Defence 3* in place, it is no longer possible for Eve to adjust her bid number based on the observed bid value revealed by Bob. What she still can do is simply repeat what Bob submits as his encrypted bid and then repeat his revelation of the raw bid and the nonce. The attack is explained below:

***Attack 4***

1. *Eve observes Bob's registration transaction and registers first by putting her transaction before Bob's in the block (or setting a higher gas price if she was not a miner). So, Eve is now A, Bob is B.*
2. *Bob places his bid via* $placeEncryptedBid(X)$ *where* $X = SHA256("[bid]\text{-}[nonce]")$*. Eve places her bid the same as Bob (without knowing what the bid and nonce are):* $placeEncryptedBid(X)$*.*
3. *Bob reveals his values via* $revealBid(Y, Z)$*, Eve repeats that too.*

*Thus the outcome is that Bob and Eve have both bid* $Y$*. Then the sum of their bids is* $2Y$ *which is even, so Eve wins because she is player A.*

*This attack would also work if the first player to register would be player B. Then Eve would ensure she joins after Bob (put her registration transaction after Bob's in the block) and again becomes player A.*

***Defence 4***:

- There must be no dependence of who is A and B on the order in which players register.

Intuitively, this can be achieved through randomness, i.e. who joins first becomes either A or B with 50% probability. But most of the sources of randomness in Solidity can be manipulated by a dishonest miner (block timestamps, block hashes, etc.) or can simply be viewed by anyone on the blockchain (private "random" seeds). Future block hashes are not a good solution either as a malicious miner can (with some probability) guess the number of a block in the future whose hash to take as the seed, especially since the phase duration is quite short – only 25 blocks.

Therefore, as explained in *Algorithm 1* (section 1.1.4), it is only decided during the *Withdraw* phase which of the players is A and which one is B. There is no reason for the players to bid in either of the ranges [1, 50] or [51, 100] as it does not increase their chances of becoming player A or B (and thus winning). The proof is below:

*Proof of Fairness of Algorithm 1*

*Assume there are 2 players: Player1 and Player2. We need to show there is equal chance of becoming A and B for Player1 (and thus for Player2 if we swap them) if their bid is in range [1, 50] or [51, 100].*

- *If Player1 bids a number in range [1, 50], then Player2 bids [1, 50] or [51, 100] with 50% likelihood, so Player1 becomes A or B respectively with 50% probability each.*
- *If Player1 bids a number in range [51, 100], then Player2 bids [1, 50] or [51, 100] with 50% likelihood, so Player1 becomes B or A respectively with 50% probability each.*

*In both cases, Player1 has 50% probability of becoming player A or B.*

Irrespectively of what bid each player places, they are equally likely to become A and B. Repeating the opponent's bid loses its point because the likelihood of *Attack 4* succeeding becomes the same as the probability of winning honestly, which is 50%.

Thus, the combination of the players' bids is used a source of "randomness", which both keeps secret who is A and who is B and maintains the game's fairness to both players.