ZKU Assignment 3

Name: Shyam Patel
Discord: L_guy#4297
Email: shyampkira@gmail.com
GitHub Repo: [Link]

Question:1 Dark Forest:

In DarkForest the move circuit allows a player to hop from one planet to another. Consider a hypothetical extension of DarkForest with an additional 'energy' parameter. If the energy of a player is 10, then the player can only hop to a planet at most 10 units away. The energy will be regenerated when a new planet is reached.

Consider a hypothetical move called the 'triangle jump', a player hops from planet A to B then to C and returns to A all in one move, such that A, B, and C lie on a triangle.

1. Write a Circom circuit that verifies this move. The coordinates of A, B, and C are private inputs. You may need to use basic geometry to ascertain that the move lies on a triangle. Also, verify that the move distances (A → B and B → C) are within the energy bounds.

Link to circuit: [Link].

2. [Bonus] Make a Solidity contract and a verifier that accepts a snark proof and updates the location state of players stored in the contract.

Link to Verifier Contract: verifier.sol [Link].
Link to Solidity Contract: Triangle.sol [Link].

All the generated files, inputs and proofs can be found here: [Link] .

Question 2. Fairness in Card Games

1. Card commitment - In DarkForest, players commit to a location by submitting a location hash. It is hard to brute force a location hash since there can be so many possible coordinates.

In a card game, how can a player commit to a card without revealing what the card is? A naive protocol would be to map all cards to a number between 0 and 51 and then hash this number to get a commitment. This won't actually work as one could easily brute force the 52 hashes.
To prevent players from changing the card we need to store some commitment on-chain. How would you design this commitment? Assume each player has a single card that needs to be kept secret. Modify the naive protocol so that brute force doesn't work.

For designing something like this we need to

- Create a circuit that takes as private inputs: the card number (0 to 51) and a random number - the so-called salt - to be concatenated with the card number to produce a protected hash of the card number.

Note: To increase the security we can use salt several times. But for our use case, taking a sufficiently large range for random number generation will make brute force almost impossible (i.e. int254).

- The circuit can be used to hash (MiMCSponge) the card number and the salt together and produce an output hash. i.e.

```
signal output out
out <== MiMCSponge.outs[0]
```

- Commit to a smart contract the following: the proof generated from the above circuit and the salted hash of the card number.

2. Now assume that the player needs to pick another card from the same suite. Design a circuit that can prove that the newly picked card is in the same suite as the previous one. Can the previous state be spoofed? If so, what mechanism is needed in the contracts to verify this?

Design a contract, necessary circuits, and verifiers to achieve this. You may need to come up with an appropriate representation of cards as integers such that the above operations can be done easily.

Link to CardCommit circuit: [Link]
Link to CardCommitter Contract: [Link]
Link to verifier Contract: [Link]

Deployed Contract addresses on rinkeby Test Network:

```
verifier.sol: 0x2edc30af4e7c1cf955a6901d56a2a344f934a346
CardCommitter.sol: 0x233cbbbcdbb3c5b26b113bfec26a981dd44457ca
```

Question 3. MACI and VDF

1. What problems in voting does MACI not solve? What are some potential solutions?

MACI can provide collusion resistance only if the coordinator is honest. Although a dishonest coordinator can neither alter nor tamper with its execution, it knows all actions taken by each voter. Current MACI implementation cannot prevent the dishonest coordinator from coordinating the collusion behavior. Ideally, we'd like a situation where the coordinator is responsible only for anti-collusion and doesn't know which user took what action.

MACI does not solve the problem of network takeover. If a single person wants to implement a 51% attack, they can still generate enough nodes to take over a network and flood it with many votes that they know are biased to one decision. A quick solution to this is, unfortunately, KYC. In the long-term, it is a problem that could be solved with proof of identity.

One possible solution is to use VDF to delay the user action proof generation until the end of the vote and use smart contracts to verify and compute results. This eliminates the need of a centralized coordinator.

2. How can a pseudorandom dice roll be simulated using just Solidity?

This pseudorandom can be easily implemented by using (pseudo) random sources, like timestamp, blockhash. For example: dice_outcome = block.timestamp % 7.

1. What are the issues with this approach?

The problem with this approach is the random result is generated by the individual miner before committing to the block chain and the miner thus can take this advantage to only commit the number that favors his benefit. The random number is stored on-chain and thus is public. This means the number should not be generated until after the entry into the (lottery) game has been closed. Otherwise, other players can take benefitable action based on earlier published numbers.

2. How would you design a multi party system that performs a dice roll?

- **Phase 1: Random Number Generation**
    - The coordinator sets aside a reward for a random number. A reward is necessary to foster competition among participants.
    - Each Participant generates their own secret random number N.
    - Participants create their commitment by hashing their N with their address: `hash = sha256(N, msg.sender)`
    - Participants send their hash to the contract.
    - Submissions continue until sufficient participants have joined.
- **Phase 2: Commit and Reveal**
    - Once the submission round has ended, each participant submits their random number N to the contract.
    - The contract verifies that the `sha256(N, msg.sender)` matches the original submission. If a participant does not submit a valid N in time, his deposit is forfeit.
- **Phase 3: Nested Random Number Generation**
    - The Ns are all hashed together: `sha256(N1, N2...Nn)` to generate the resulting random number.
    - Valid participants receive the reward. (Reward / num of valid participants). This gives more incentive to be an honest participant since hiding the secret can kick-out dishonest participants and thus win more rewards.

3. Compare both techniques and explain which one is fairer and why.

The second technique is fairer because compared with the simple hash of block.timestamp, these 3 phases approach makes the prediction harder (although not completely impossible, see description below) and not beneficial for dishonest participants.

4. Show how the multi party system is still vulnerable to manipulation by malicious parties and then elaborate on the use of VDF's in solving this.

The above multi-party system is vulnerable to last-revealer-attack. In phase 2, the last revealer can still leverage all the already revealed numbers on chain and decide whether he wants to reveal his own number, thus benefiting from avoiding the losing game. One workaround is to have a coordinator to be the last revealer, but this makes such coordinator a central point of trust.

To overcome this problem, we can introduce VDF in phase 3 (also illustrated in the below figure, borrowed from harmony [Link]),

- Once all Numbers are committed on a chain, smart Contract calculates `pRnd` with a hash of all random numbers: `pRnd = Hash(N1...Nn)`
- A VDF is executed to generate the final random number: `Rnd = VDF(pRnd, T)` and its proof.

- Each participant then validates VDF proof and makes it the result committed to the block chain.

The VDF is used to provably delay the revelation of final Rnd and prevent the last participant from biasing the randomness by choosing to submit the wrong random number. Because of the VDF, the last participant won't be able to know the actual final randomness before pRnd is calculated and committed to the blockchain. By the time Rnd is computed with the VDF, pRnd are already committed in a previous block so any participant cannot manipulate it anymore. Therefore, the most a malicious participant can do is to either blindly reveal a random number or try to delay phase 3 by not revealing its random number. The former is the same as honest behavior. The latter won't cause much damage as the timeout and competition mechanism will be triggered to forfeit this process or kick-out the dishonest participant.

Question 4: InterRep

1. How does InterRep use Semaphore in their implementation? Explain why InterRep still needs a centralized server.

Interep provides special groups that can be used by DApps or services to verify users' reputations without exposing their identities. In order to join groups each user must create a unique identifier using an Ethereum account and Semaphore. Semaphore, then, allows users to prove that their identifier is part of a specific group. Their approach sacrifices some decentralization to gain more privacy, since they use a centralized bridge.

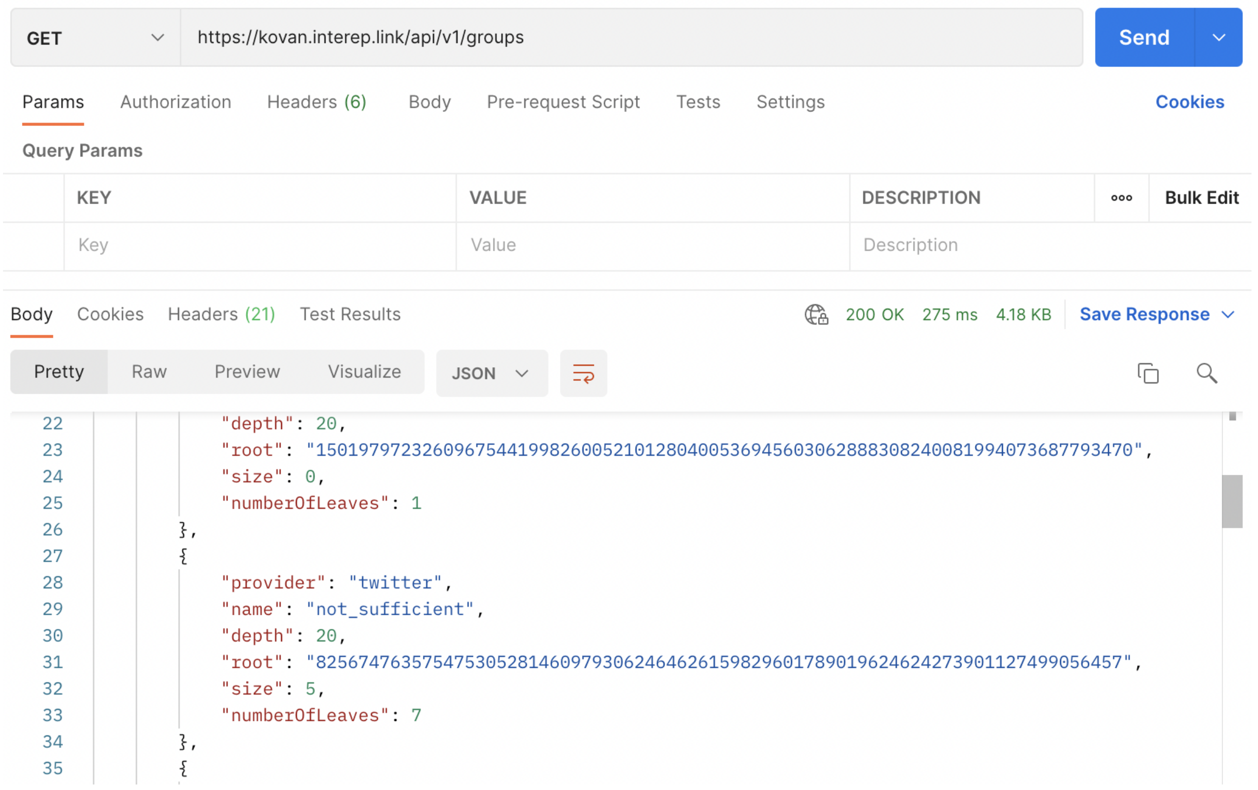
2. Clone the InterRep repos: contracts and reputation-service. Follow the instructions on the Github repos to start the development environment. Try to join one of the groups, and then leave the group. Explain what happens to the Merkle Tree in the MongoDB instance when you decide to leave a group.

When we leave a group, the database prunes the corresponding leaf from the merkle tree which it stores the commitment on.

3. Use the public API (instead of calling the Kovan testnet, call your localhost) to query the status of your own identityCommitment in any of the social groups supported by InterRep before and after you leave the group. Take the screenshots of the responses and paste them to your assignment submission PDF.

```
{
    "network" : "kovan",
    "Poseidon" : "0x4495098f5C12824655971BE2127f9e27700fB0eb",
    "IncrementalBinaryTree" :
```

```
"0x54a7dC50Cf154c2B114aeFF3D9739E8f229b6634",
    "Interep" : "0x05c43e444413fb11dcec9279D51EBa7Afaa36DAf"
}
```



GET https://kovan.interep.link/api/v1/groups

Send

Params | Authorization | Headers (6) | Body | Pre-request Script | Tests | Settings | Cookies

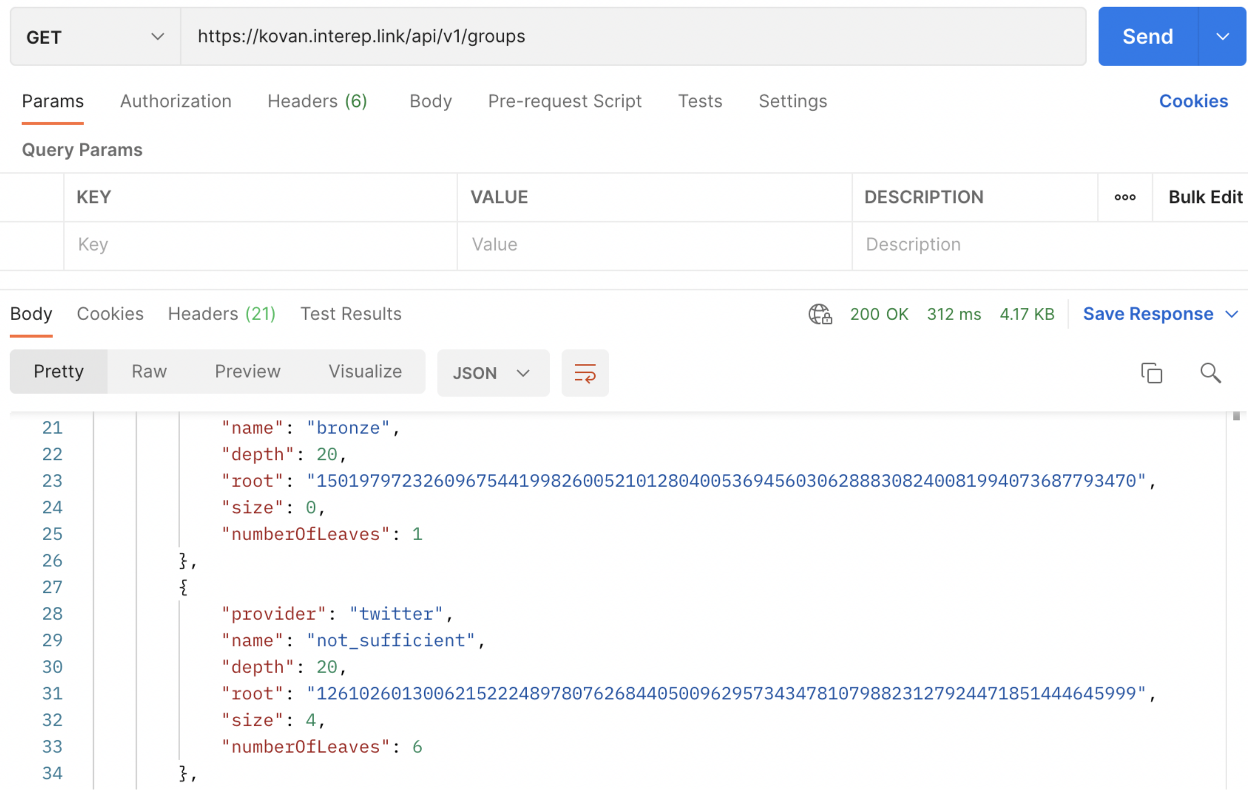
Query Params

	KEY	VALUE	DESCRIPTION
	Key	Value	Description

Body | Cookies | Headers (21) | Test Results — 200 OK 275 ms 4.18 KB — Save Response

Pretty | Raw | Preview | Visualize | JSON

```
            "depth": 20,
            "root": "15019797232609675441998260052101280400536945603062888308240081994073687793470",
            "size": 0,
            "numberOfLeaves": 1
        },
        {
            "provider": "twitter",
            "name": "not_sufficient",
            "depth": 20,
            "root": "8256747635754753052814609793062464626159829601789019624624273901127499056457",
            "size": 5,
            "numberOfLeaves": 7
        },
        {
```



GET https://kovan.interep.link/api/v1/groups

Send

Params | Authorization | Headers (6) | Body | Pre-request Script | Tests | Settings | Cookies

Query Params

	KEY	VALUE	DESCRIPTION
	Key	Value	Description

Body | Cookies | Headers (21) | Test Results — 200 OK 312 ms 4.17 KB — Save Response

Pretty | Raw | Preview | Visualize | JSON

```
            "name": "bronze",
            "depth": 20,
            "root": "15019797232609675441998260052101280400536945603062888308240081994073687793470",
            "size": 0,
            "numberOfLeaves": 1
        },
        {
            "provider": "twitter",
            "name": "not_sufficient",
            "depth": 20,
            "root": "12610260130062152224897807626844050096295734347810798823127924471851444645999",
            "size": 4,
            "numberOfLeaves": 6
        },
```