Zero-Knowledge University (ZKU):

Assignment 3

Tal Derei

21 March 2022

Course Registration Email: Tad222@lehigh.edu

Discord Username: Margulus#4273

Github (Personal Repistory): `https://github.com/TalDerei`

Gitub (ZK Research): `https://github.com/TalDerei/Zero-Knowledge`

Dark Forest:

1. *Triangle Jump* Circom Circuit

Codebase (Git Commit):

`https://github.com/TalDerei/ZKU/commit/95e8142b4a9262423b75c22037509cfb6baf827c`

The problem was solved by calculating checking whether area of triangle != 0 using the *isZero()* method. This ensures the points form a triangle. On a side note: dark forest works in a similar way, and the coordinates are kept secure since the number of locations in 2D space are way higher than the integer limit (e.g. the y-location is

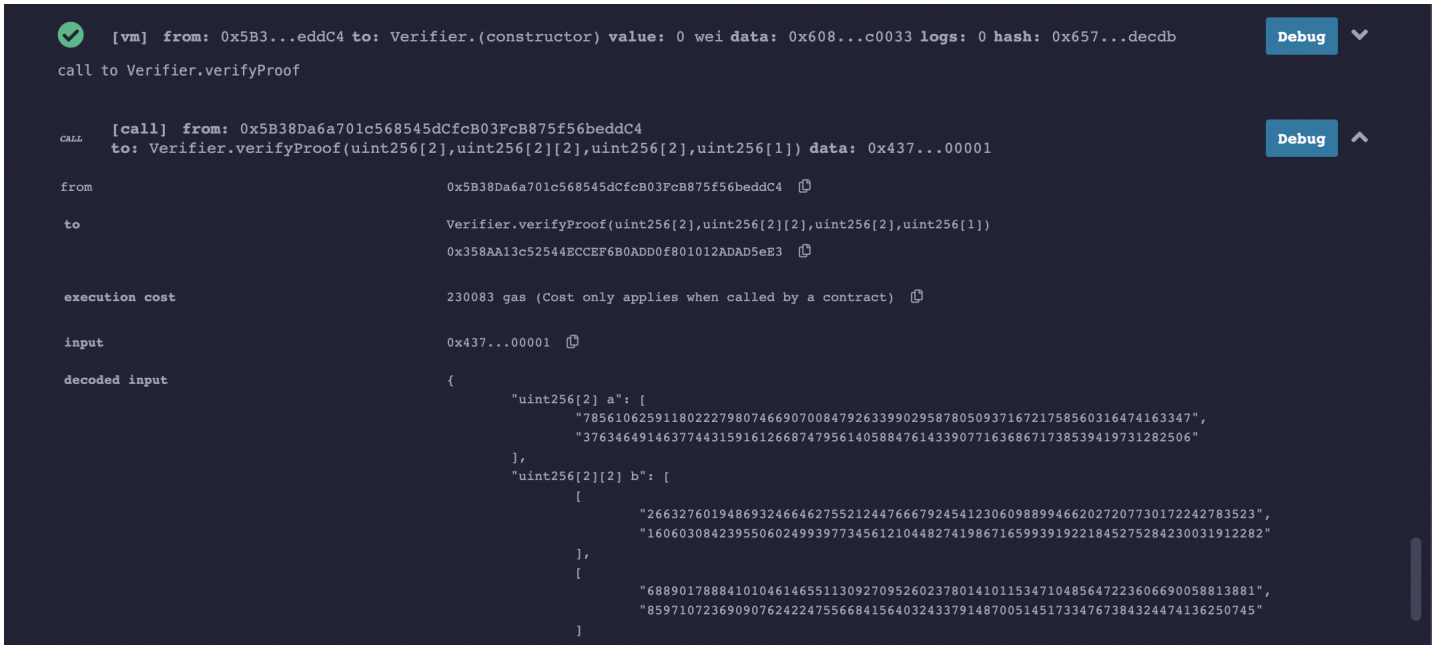

Figure 1: *Smart Contract Proof Verification (verifier.sol)*

Figure 2: *Smart Contract Proof Verification (verifier.sol)*

21888242871839275222246405745257275088548364400416034343698204186575808492485). Additionally, there's a Perlin noise function to add randomness to the hashes to prevent brute force attempts.

2. The git commit contains the circom circuit defined in **darkforest.circom**. After running the power's of tau ceremony and generating a validity proof and public.json (containing the public outputs), a solidity verification smart contract called **verifier.sol** was generated with the command "snarkjs zkey export solidityverifier darkforest0001.zkey verifier.sol". I then ran *"snarkjs generatecall"* to generate the input parameters for Remix, and deployed the verifier.sol contract. The proof was validated using the solidity verification smart contract shown above.

Codebase (Git Commit):

`https://github.com/TalDerei/ZKU/commit/86bf34d544c585a3a8e5dea06b6bef1440920eab`

Fairness in Card Game:

1. The naive cryptography scheme described in the instructions maps each card to a separate number, and then hashes that number. This can easily be brute-forced by precalculating the 52 hashes. Therefore it fails to satisfy the **Sufficient Key-Space Principle**, which states "There must be a sufficient number of keys to make any brute force attack infeasible". Both [1] "Caesar Cipher" (i.e. shifting each card's number by 3), or [2] "Shift Cipher" (replacing "3" with an arbitrary shift-factor k) would not work either in this case as they fall victim to the same brute-force vulnerabilities.

One solution is the "Mono-alphabetic substitution cipher" where instead of mapping a character of plaintext to a character of ciphertext by a shift, use a table to map a

character of plaintext to another character arbitrarily (without two plaintext characters mapping to the same character of ciphertext). In this scheme, the domain lives in 52! space since we have 52 different permutation of cards. Although it satisfies the Sufficient Key-Space Principle, it's vulnerable to statistics! The distribution of characters in the ciphertext will match the distribution of characters in the plaintext if we know the frequency of letters in the plaintext (i.e. with what frequency does the letter e show up) and have a large enough ciphertext document. There are methods to combat this, which is beyond the scope of the assignment.

Therefore my proposed solution is as follows:

Assign a tuple for each card, mapping the number (0-13) and suite (0-3). Hash the tuple, and use a secret **salt** stored off-chain for applying a few rounds of salting to increase the security of the hash. After injecting extra noise in the hash, the result would be a one-way hash that serves as the commitment.

2. The commitment stored on-chain would represent the salted hash of the first card. This hash would serve as a public-input to the circom circuit, alongside another hash of a second card. Storing the hash of the first card on-chain guarantees that the previous state be spoofed, since the state is immutably (and verifiability) stored on the blockchain.

Inside the circom circuit, we're simply running *hash(salt,hash(number,password))* to verify the cards are in the same suit, going through all the possible numbers from (≥ 0 and ≤ 12). You can also pre-compute all the hashes and store them in the circom file, and simply do a comparison using *isEqual()* to check whether the card is in the same suit during runtime. Looking at the *input.json* file, the hash is based on a card with suite **2**.

If you change the number of the card between 0-12, it will compile and generate a valid proof since the computed hash will be equal to the second hash. If the suite of the card is changed, then the hashes will not match and the circuit will fail to generate a proof.

Codebase (Git Commit):

`https://github.com/TalDerei/ZKU/commit/0b46b20fdc2b05bd57976e75a524fb9131489dba`

Additionally, the proof was verified using a verification smart contract on remix.

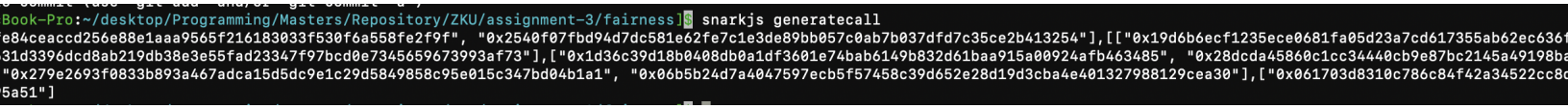

Figure 3: *snarkjs generate*

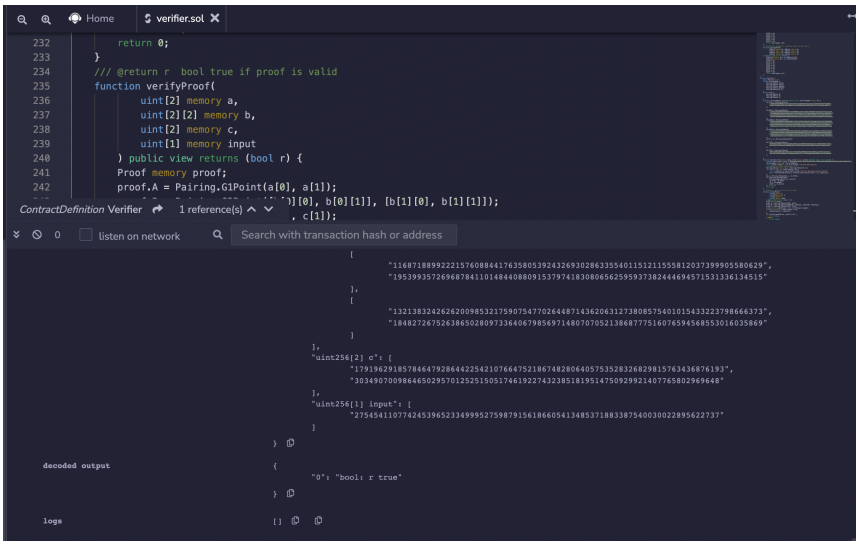

Figure 4: *Smart Contract Proof Verification (verifier.sol)*

Codebase (Git Commit):

`https://github.com/TalDerei/ZKU/commit/faf764f250d2cb68d49d008e566eaed06f5797bc`

3. In order for a player to reveal a particular card without revealing which suit it belongs to, the circuit would compute the hashes for all of the same type of card (e.g.

Ace's), and then hash the input and see if the hash's are the same.

MACI and VDF:

1. Minimal Anti-Collusion Infrastructure (MACI) serves by making it impossible for voters to demonstrate how they actually voted, mitigating the effectiveness of bribery schemes. MACI uses zkSNARKs under the hood to [1] ensure computation + provide voting secrency using encryption. One vulnerable area is the concept of multiple identities (by the same person) to increase their voting influence. One solution is integrating MACI with a proof of humanity scheme in order to verify that user is indeed unique, and they still can't prove how they voted in a particular election / DAO proposal for example.

2. Pseudorandomness

[**1**] Solidity is not capable of generating random numbers, and subsequently random dice rolls. Smart contracts are deterministic, and trying to generate randomness inside a contract using seeds or timestamps for example will make you vulnerable if the user figures how your contract produces randomness. Using on-chain quantities in a smart-contract is therefore vulnerable to manipulation by miners, as they can decide to discard an unfavorable result with a simple majority. Therefore, pseudorandom dice roll can be simulated in Solidity using a *"Verifiable Delay Function (VDF)"* that injects some level of randomness to the computation. It's a common source of randomness in Ethereum's Beacon Chain (i.e. Consensus Layer) for securing sampling leader and committee elections, and Shard Chains (i.e. Application Layer) to decide which computation is assigned to which shard. VDFs are usually off-chain mechanisms that bring randomness into a smart contract using Oracles.

[**2**] Multi-Party Computation (MPC) Ceremonies generate the parameters that kick off SNARK-based systems. Outcome is reference string and set of proving keys (for generating proofs of private transactions) and verifying keys (for verifying proofs on chain). Notable example used in this course is Powers of Tau (MPC) developed by Zcash Protocol. MPC ceremony to perform a dice roll would involve many (independent) users each running some computation and injecting their own entropy. This is done sequentially (where the next user continues the computation from the previous user) in some order. If the "toxic waste" is eliminated, and at least one participant in the MPC acted honestly, the ceremony has succeeded in generating a valid reference string and set of proving/verification keys. The reference string can serve as a randomized input to a roll() function, and produce a purely randomized dice roll.

[**3**] Both MPC Ceremonies and VDFs are fairer by design since there are multiple, independent participants introducing their own entropy/randomness. Therefore MPCs ceremonies cannot be untrustworthy unless *every* single member in the ceremony colluded.

[**4**] As previously stated, MPC Ceremonies are ONLY corrupted if every participant in the ceremony is dishonest and fails to discard the "toxic waste" generated that can be used to reconstruct the proving/verificatin keys. VDFs can serve as an extra layer of randomness if a user's entropy is "predictable".

InterRep:

1. Interep utilizes the Semaphore Protocol along with a centralized server to verify users' reputations without exposing their identifies. Recall, Semaphore uses ZK to prove the membership of a set without revealing your original identitiy. Interep implements a Javascript function to create Semaphore identities derived from Ethereum signed messages. Interrep also still needs a centralized server because they need a way to serve in-

coming API requests. Decentralizing this involves implementing an incentive mechanism for full nodes to run these servers and serve these API requests, which usually involves a token as a form of monetary compensation.

2. When adding a new user to the group, it hits the

/api/v1/groups/github/bronze/132529513238338995771523416639353042310321174737304807484928625298362038235 6... endpoint and generates the treeroot batch with a root hash. When you *leave* a group, the *hasJoinedAGroup* field in the oAuthAccount collection switches to 'false', and the *treeRootBatches* collection gets updated with another object. This object contains a new ObjectId identifier, and it's important to note it does *not* contain a hash. This means the user is no longer in the group, since the merkle tree can't be queried to check the specific hash assigning the user to the group.

3. IdentityCommitment in *Reputation Service's* yarn logs (see next page):

13252951323833899577152341663935304231032117473730480748492862529836203823 56 is the identity commitment.

Thinking in ZK:

1. Specifically in regards to dark forest, I was wondering if the game can implement more advanced game mechanics to take advantage of zero-knowledge (beyond simple *init* and *move* operations)? Extending this question, can zk be used in the context of creating "teams" with anti-collusion and privacy mechanics in place to make sure teams are participating honestly (i.e. for example in a mission or quest).

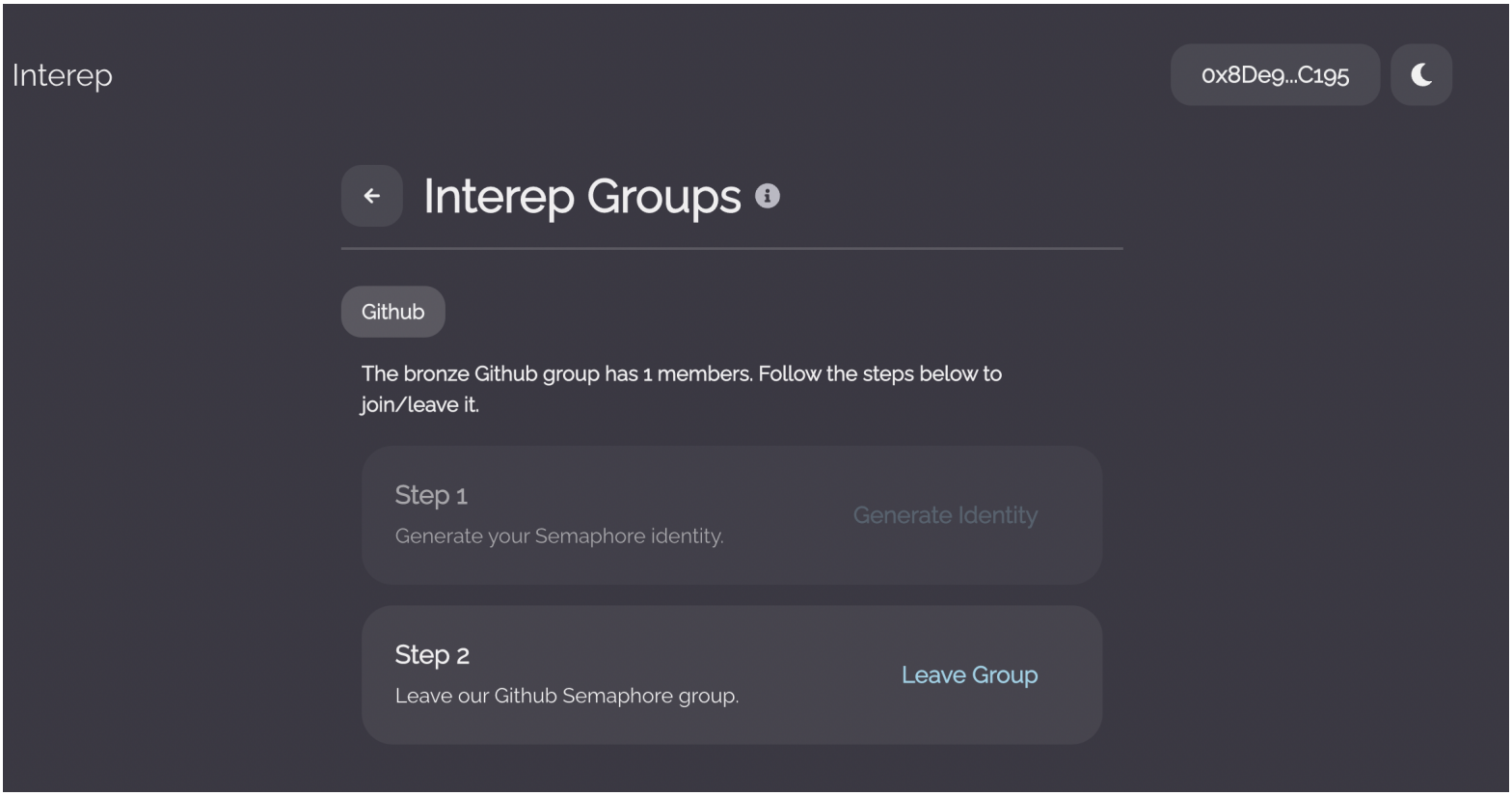

Figure 5: *Interep Groups*

Figure 6: *Interep identityCommitment*

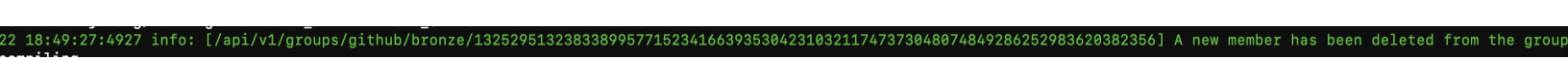

Figure 7: *Interep identityCommitment*