Zero-Knowledge University (ZKU):

Assignment 2

Tal Derei

14 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`

Privacy & ZK VMs:

1. The state of the current (**public**) transaction model on Ethereum involves [**1**] An account initiates a transaction to update the state of the network, [**2**] Transactions are cryptographically signed using the account's private key, [**3**] Miners execute the transactions, initiating a state change of the EVM that's broadcasted to the entire network (e.g. w/ Gossip Protocol) and validated in blocks (e.g. w/ Consensus Mechanism). Following this control flow, **re-execution** is required for state transitions on the main-chain: every (**full**) node processes/executes every transaction. After miners decide which transactions to package into blocks and mine the block (based on gasprice), every full

node re-executes all contract executions in order to independently verify the security/validity of the block.

In contrast, **ZK-based verification** only requires the verification of a single zero-knowledge proof in order to transition state. For example, ZK-Rollups maintain smart contracts rooted in Ethereum, and move computation and state storage (i.e. the execution) off-chain. In this zero-knowledge environment, sequencers/validators are tasked with computing a proof of validity for a batch of transactions, which is then posted back on the mainchain where a collection of smart contract precompiles will verify the proof and update the state hash accordingly. Rather than having full nodes perform the validation and re-execution, smart contracts perform this verification check **once** in order to transition state.

2. Zero-Knowledge Ethereum Virtual Machines (**zkEVMs**) are *turing-complete*, *EVM-compatible*, and *SNARK-friendly* virtual machines that execute smart contracts on L2 zk-Rollups. In that regard, zkEVMs support general EVM computations (e.g. compilation and execution of solidity smart contracts) on L2. So existing DAAPs and DAOs written in EVM-comptible languages like soldity and deployed on the main-chain can easily port to ZK-Rollups for cheaper gas fees.

Since zkEVM needs to conform to the zero-knowledge environment on L2...there are traditionally two popular approaches to building zkEVMs:

Compiler-Based Implementation vs. **Opcode-Based Implementation**. Some examples include:

- **Zksync 2.0 (by matter Labs):** Compiler-Based Implementation of a zkEVM to maintain solidity compatibility on L2. Zksync developed the "**Zinc**" frame-

work for developing zero-knowledge smart contracts and SNARK circuits. Their **zincVM** (a SNARK friendly VM) transpiles solidity to zinc, compiles zinc into bytecode (w/ zinc compiler), executes the bytecode, and generates a proof of the program trace. [SEE QUESTION 3 FOR MORE DETAILS].

- Hermez: Opcode-based Implementation where the entire set of EVM opcodes are rebuilt for layer-2. Each opcode needs to be reconstructed for L2 (50 opcodes), and some are more complex and expensive to compute (e.g. SHA256/keccak256 hash functions). In that regard, their zkEVM acts as a translation layer that translates existing EVM opcodes into special representations called "micro-opcodes" that are more suitable for L2. Unlike zksync 2.0, they are NOT building a newly designed virtual machine, but rather using the same EVM as L1. Instead they're interpreting L1 EVM-specific opcodes to a more zk friendly set of opcodes under the hood.

- Polygon-Zero: Recursive SNARK-based zkEVM powered by their **Plonk2** Proof System: Recursive SNARKs w/ PLONK + FRI Polynomial Commitments.

[2]. Advantages of **some** zkEVMs are Ethereum compatibility: ability to compile existing solidity code to run on zkEVM. Where Ethereum compatibility is not directly maintained, zkEVMs incorporate their own programming language (e.g. Starknet's "Cairo", Zksync's "Zinc"). Writing code in a ZK programming language is easier than writing application specific circuits (using Circom or Arkworks). A notable disadvantage is the potential risk of vendor lock-in based on stringent terms-of-use and licenses.

[3]. Zksync 2.0 In-Detail:

1. Rewrite smart contract in their ZK smart contract language called Zinc. Al-

ternatively, you use native solidity smart contract that gets transpiled into Zinc using their newly designed virtual machine called zincVM a SNARK friendly VM. So you take solidity code, it's gets translated to zinc, zinc code get's compiled using the zinc compiler than lives in their zincVM.

2. Sequencer (normal Ethereum full node) batches multiple executions (contract calls) and executes them. Validation of a state change is done through zk proofs.

3. Generate a proof of validity. Zksync has a custom ZK Proof System based on PLONK. And then they use recursive snarks to aggregate many proofs in a tree-like structure, and create a single proof that's posted back on mainchain.

4. Optimized precompiled smart contracts on Ethereum perform the proof verification (zksync's custom proof is called a "Redshift" proof based on PLONK (universal trusted setup) with custom gates and lookup tables (UltraPLONK) and Ethereum's BN-254 elliptic curve).

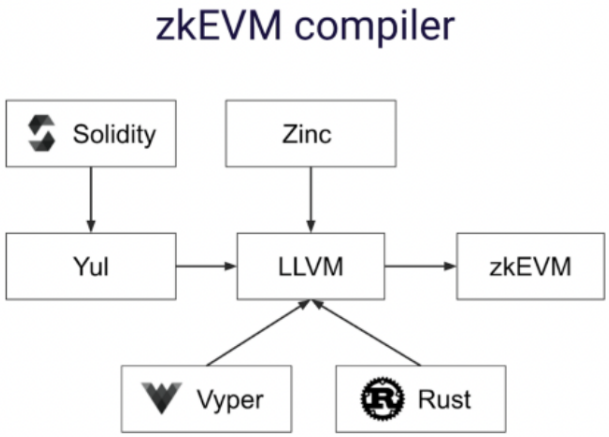

Figure 1: *ZkSync zkEVM*

Semaphore:

1. Semaphore is a privacy layer for Ethereum applications based on zk-SNARKs technology. Semaphore enables "anonymous signaling", allowing users to [1] Register their identity in a smart contract, and [2] anonymously prove their membership of a set without revealing their original identity. These are known as "anonymous proofs of membership" on Ethereum. Registering a user's identity involves sending a hash of their EdDSA public key and two random secrets to the smart contract, which is stored in a merkle tree. The next step, broadcasting a signal, involving generating a zkSNARK proof that verifies a) membership amongst a set of registered users, and b) double-signalling prevention. Some applications include: signaling, private voting, proving set membership, and whistleblowing, mixers, and anonymous authentication (across web2/web3 platforms).

2. Semaphore Repository by AppliedZKP

[**1**] The git commit "**3bce72f**" failed to pass all automation tests. The screenshot below represents the main-branch with 31 test-cases.

[2] Semaphore.circom

The **"Semaphore"** template first instantiates the **"calculateSecret"** template by defining *component calculateSecret*. It takes in a *identityNullifier* and *identityTrapdoor* as inputs (two random secrets to prevent double spending), and runs them through a Poseidon() hash function defined in the **"Poseidon"** template, storing the hash in the *out* output signal. Then the **"CalculateIdentityCommitment"** template is instantiated, and takes in the previous hash as input. It runs the hash through the poseidon hash function again in order to generate the "identity commitment". Then the **"calcu-**

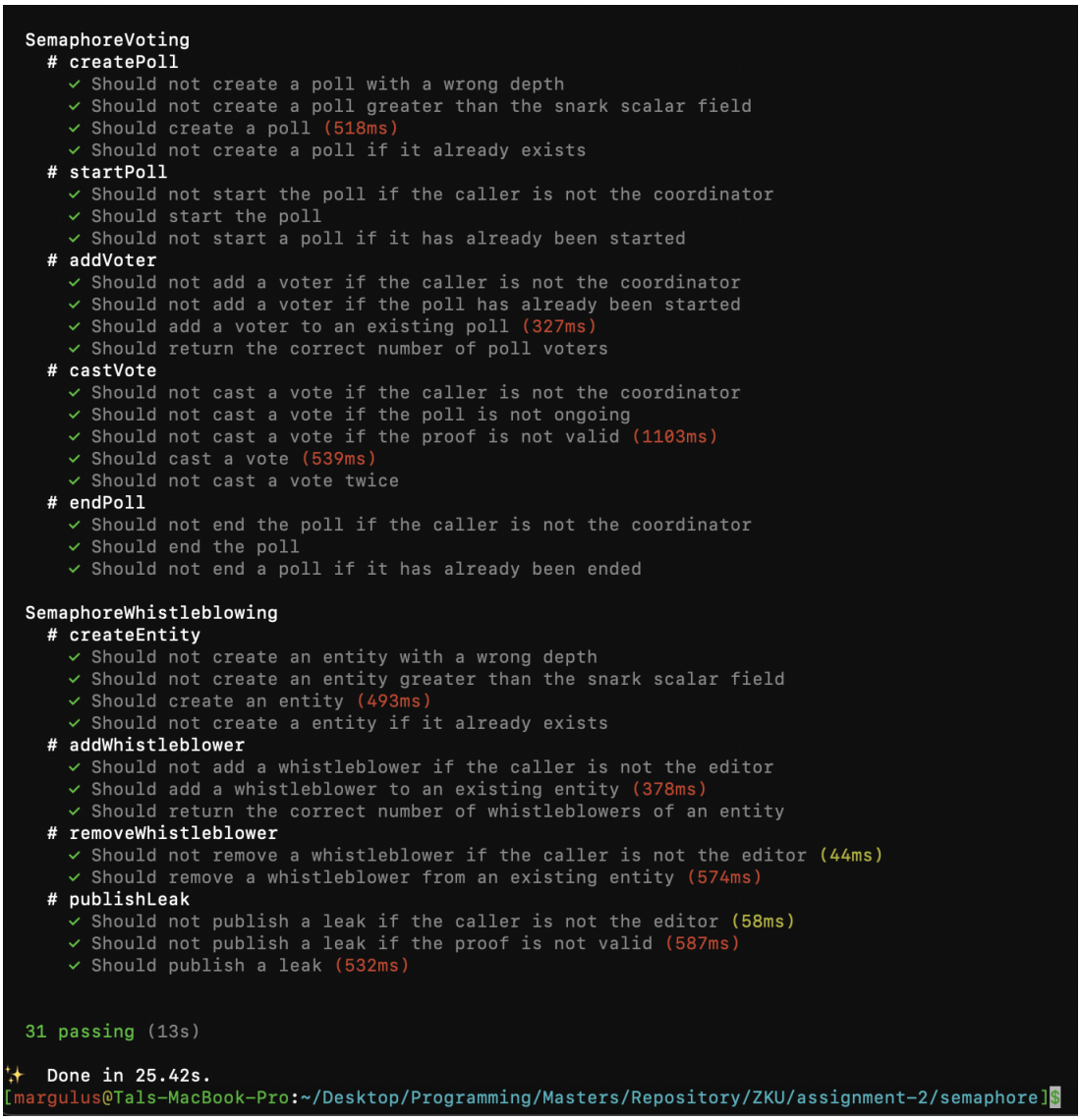

Figure 2: *Semaphore Automation Tests*

lateNullifierHash" template is instantiated, taking in the public input *externalNullifier* (The hash of the service provider's URL) and private input *identityNullifier* and calculates the poseidon hash. Afterwards it instantiates the **"MerkleTreeInclusionProof"** template defined in **tree.circom**, taking in as input [1] the "identitity commitment" (i.e. the poseidon calculated in CalculateIdentityCommitment()) as a leaf node, treeSiblings, and treePathIndices. It builds the merkle tree for nLevels, storing the inclusion/membership proof in the merkle root.

3. Elefria Protocol on the Harmony Testnet

[1] The Elefria authentication protocol inherently incorporates strong privacy with multi-factor authentication to prevent attack vectors such as "stolen credential reuse attacks". But there are other traditional web2 authentication techniques that can be employed to improve security such as implementing weak password checks and credential recovery, since a lot of these API gateways will be traditionally web2-based. Additionally, a user submits a zk proof of membership to the smart contract, and the contract verifies the proof and generates authentication and identity tokens in return. These tokens may be intercepted while in-transit from the server to the client.

[2] The ZK ID is not encrypted in local storage, and only becomes encrypted once the user enters their private password. Encrypting the ZK ID in local storage may mitigate a lot of the potential attack vectors in the future.

Tornado Cash:

1. Tornado-Nova is an experimental upgrade of the **Tornado-Trees**. While Tornado-Trees implements traditional fixed-amount pools (requiring you to specify 0.1/1/10/100 ETH deposits, and withdrawal's based on specific notes), Tornado-Nova implements de-

posits and withdrawals of arbitrary amounts. Another feature are "shielded transfers" to another registered user inside the pool without needing to withdraw from the pool.

2. Tornado-Trees Repository

[**1**] The first step to generating a withdrawal transaction is generating a proof of validity that the full batch of queued withdrawals were inserted into the merkle tree correctly. The "TreeUpdateArgsHasher" template inside the circom file computes the sha256 hash of all inputs (oldRoot, newRoot, pathIndices, instances[nLeaves], hashes[nLeaves], blocks[nLeaves]) packed into a byte array, and generates a proof. This proof is passed to the updateWithdrawalTree() function in the solidity file takes in as input the snark proof of validity, hash of the snark inputs (in order to save gas), current merkle root, updated merkle root (after withdrawl), and merkle path to the batch. The "data" field represents the leaves from which our merkle tree is construct from, and the smart contract is computing the hash recursively until generating a root hash. Then it does some validation on the leaves and the root, and then checks the snark proof using smart contract precompiles on the mainchain.

[**2**] When comparing SHA256 vs. circuit friendly hash like Poseidon, it's important to note that the hash is relatively large as we put a lot of data into it. When considering inserting subtree of 256 elements in a Tornado Cash smart contract...this computation cannot be done in a browser, but instead using specialized machines with a lot of RAM to construct the proof. And building a SNARK circuit like this takes millions of constraints with circom and heavy memory utilization (about 100G in memory). SNARK friendly hashes are expensive on-chain since they contains a lot of contraints, and pay a lot of gas to compute. So if we use Poseidon hash functions, they contain more constraints, and are therefore more expensive to execute.

3. Tornado-Nova Repository

[1] Tornado Nova Automation Tests

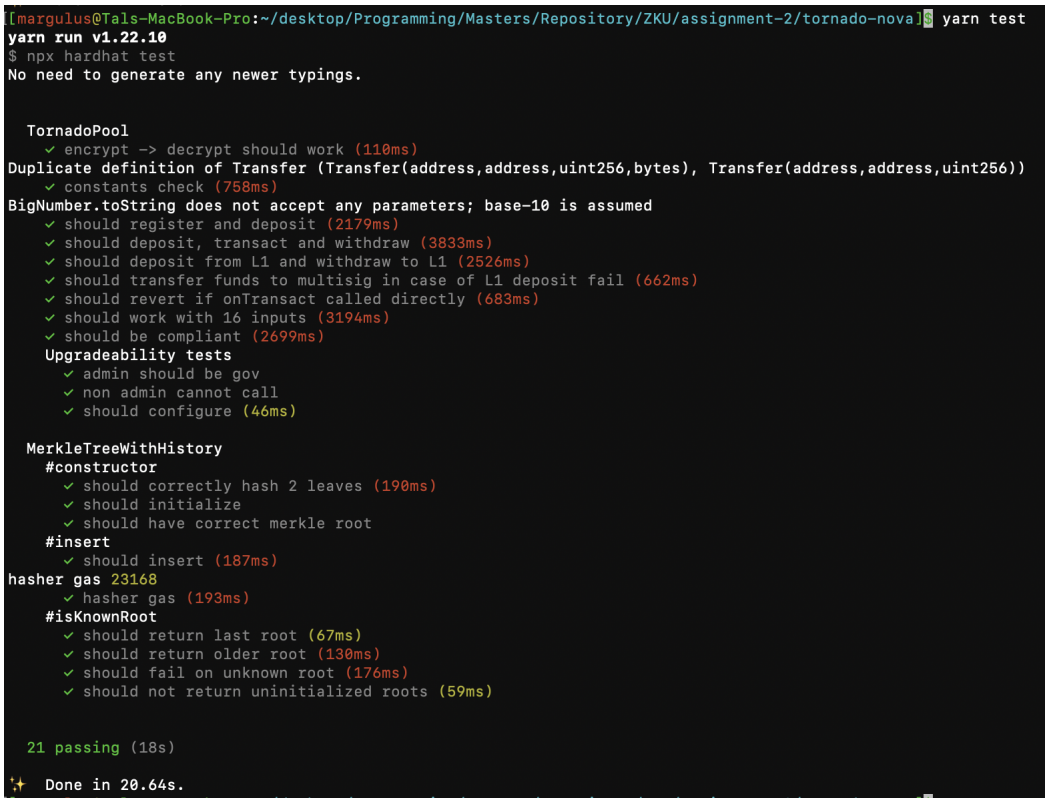

Figure 3: *Tornado Nova Automation Tests*

[2] Codebase (Git Commit)

`https://github.com/TalDerei/tornado-nova/commit/b6d0d19f99bfb6a4efe7aebf54343b216cbde58`

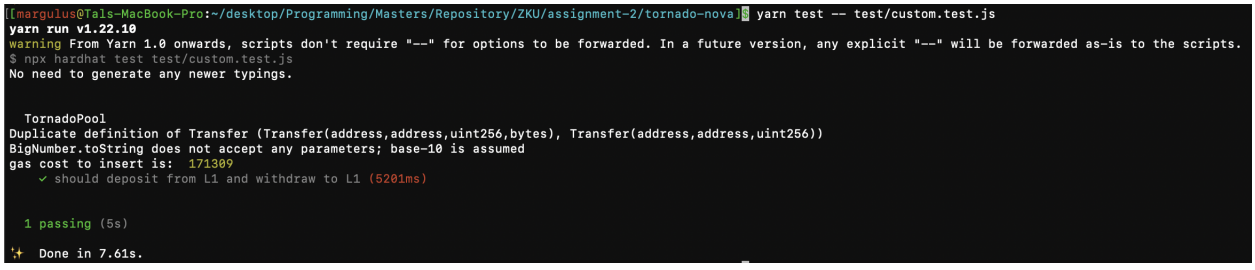

Figure 4: *Custom.test.js Automation Tests*

Tornado Pool is on L2 and omni bridge is on L1. When you deposit from L1, the balance is actually bridged to L2.

Thinking In ZK:

1. What are the minimum specifications changes necessary for the protocols to generate more complex payment types like interest/dividend payments? And can on-chain mixers be created using **Tornado Cash** that interact and bridge privately with DeFI protocols, allowing you to directly take out a loan for example.