Verifiable Private Balance Contract

Roman Walch[1]
[1] TACEO
walch@taceo.io

Abstract. In this paper we describe how to build a private token contract with MPC and collaborative SNARKs. This smart contract hides the balances of users, as well as transaction amounts when sending tokens to other participants by only showing commitments publicly. The actual balances corresponding to these commitments are stored – as secret shares – on an MPC network, which performs the necessary computations privately in MPC. The addition of the collaborative SNARK adds verifiability to the system, proving that enough funds are present for each transaction. Our system currently achieves a throughput of 200 transactions per second (including proof generation) on reasonable MPC hardware.

Keywords: MPC · ZKP · CoSNARKs · Private Token

Contents

TODO Discuss related work

1 Introduction

Privacy preserving cryptographic protocols and primitives such as secure multiparty computation (MPC), fully homomorphic encryption (FHE) and zero-knowledge proofs (ZKP) have been heavily improved in the recent years and become more feasible to apply to real world use cases every day. In this document we investigate the application of these protocols, more concretely MPC, ZKPs and their intersection, dubbed collaborative SNARKs (CoSNARKs), to the use case of private on-chain payments. Concretely, we investigate the case of confidential and verifiable on-chain token balances. There, the goal is to add privacy to an existing token (which can for example be ETH, or a stable coin such as USDCS), by adding a smart contract which stores balances to users privately, e.g., by holding cryptographic commitments or (homomorphic) ciphertexts of the balances. Then, when users transfer money amongst each other, the commitments or ciphertexts are updated accordingly, while zero knowledge proofs prove that 1) the sender has enough balance and 2) the commitments where updated correctly. In this way, the current balance of the involved users, as well as the transferred amount, stay private while everyone can verify the computations were done correctly.

The users should also be able to deposit fresh tokens to its private on-chain balance, and withdraw them (if enough funds are present). To this end, when the user deposits some tokens into the smart contract, the smart contract keeps it alongside all other deposited tokens in a pool. Furthermore, when the user withdraws funds, the smart contract pays out tokens from this pool. Thus, the amount of tokens in this pool should always match the total balance of all users in the system.

1.1 Related Work

TODO Talk about FHE, TEE and ZK based solutions

TODO Somewhere compare in depth against the FHE solution as in https://www.notion.so/FHE-based-private-Balance-29352d23baea8056ac77f18e609281cf

1.2 Notation

In the rest of the manuscript, we denote a secret sharing of a value $x \in \mathbb{F}_p$ with a bracket notation $[x]$.

2 High Level Overview

In this section we describe our proposed solution on a high level, for more details on the involved subprotocols we refer to the next sections.

In our solution, we essentially have three different entities involved: 1) A user who deposits and withdraws tokens and wants to initiate token-transfers to others; 2) A smart contract keeping private balances and a pool of deposited tokens; 3) A MPC-network holding secret-shares of the actual balances.

Since it is not a goal of this use case to hide sender and receiver address, the smart contract holds a hashmap (e.g, mapping in solidity), mapping user addresses directly to commitments. Furthermore, the MPC network holds another hashmap, which maps user addresses to secret shares of the balance and the randomness used to compute the commitment which is stored on chain. Whenever the user initiates an action (i.e., deposit, withdraw, or transfer), it has to communicate this to the smart contract. The smart contract will then register the action in a queue which is continuously monitored by the MPC network. The MPC network then computes the action in MPC which leads to updated commitments. To prove it performed the updates correctly, it also generates a zero knowledge proof in MPC (which is called collaborative SNARK, or CoSNARK) proving correctness of the update. The MPC network posts the new commitments alongside the proof on chain, where the smart contract verifies the proof, updates the commitments in its hashmap if the proof was correct, removes the action from the queue, and, in case of withdraw, sends tokens to the recipient.

The advantages of this systems are as follows. First, the client side computations are minimal. To initiate transactions, it only has to post a transaction on chain and, in case of transfer, send secret shares to the MPC network. No expensive client side ZK proofs need to be computed. Second, since the MPC network creates ZK proofs of all its actions, integrity of funds is always given and the MPC network can not misuse funds.

3 The MPC Computations

In this section we have a closer look at the computations happening at the MPC network. We list all the involved algorithms in Algorithm 1. The first tow algorithms in Algorithm 1 (`Map.insert` and `Map.get`) represent the access of the actual data structure, which store the secret-shared balance and randomness. Note that these are normal hashmaps. Then, each action (deposit, withdraw and transfer) essentially just create new commitment and stores them in the hashmap. Finally each action concludes by computing a zero knowledge proof in MPC to prove integrity to the smart contract.

Algorithm 1. The computations on the MPC network

1: ▷ Data structure
2: **function** MAP.INSERT($k, [b], [r]$)
3: ▷ Store balance $[b]$ and randomness [r] in the hashmap at key k.

```
4:  end
5:
6:  function MAP.GET(k)
7:    if hashmap has entry([b], [r]) for key k then
8:      return ([b], [r])
9:    else
10:     return (0, 0)
11:   end
12: end
13:
14: function DEPOSIT(k, b)
15:   [r_old], [b_old] ← Map.get(k)
16:   [r_new] ← MPC.rand()                    ▷ Sample new blinding for commitment
17:   [b_new] ← [b_old] + b
18:   Map.insert(k, [b_new], [r_new])
19:   C_old ← commit([b_old], [r_old])
20:   C_new ← commit([b_new], [r_new])
21:   π_deposit ← proof(C_old, C_new, b)      ▷ Collaborative SNARK, see Listing 1
22:   return π_deposit, C_new
23: end
24:
25: function WITHDRAW(k, b)
26:   [r_old], [b_old] ← Map.get(k)
27:   [r_new] ← MPC.rand()                    ▷ Sample new blinding for commitment
28:   [b_new] ← [b_old] − b
29:   Map.insert(k, [b_new], [r_new])
30:   C_old ← commit([b_old], [r_old])
31:   C_new ← commit([b_new], [r_new])
32:   π_withdraw ← proof(C_old, C_new, b)     ▷ Collaborative SNARK, see Listing 2
33:   return π_withdraw, C_new
34: end
35:
36: function TRANSFER(k_sender, k_receiver, [b], [r_b])
37:   [r_s,old], [b_s,old] ← Map.get(k_sender)
38:   [r_s,new] ← MPC.rand()                  ▷ Sample new blinding for commitment
39:   [b_s,new] ← [b_s,old] − [b]
40:   Map.insert(k_sender, [b_s,new], [r_s,new])
41:   C_s,old ← commit([b_s,old], [r_s,old])
42:   C_s,new ← commit([b_s,new], [r_s,new])
43:
44:   [r_r,old], [b_r,old] ← Map.get(k_receiver)
45:   [r_r,new] ← MPC.rand()                  ▷ Sample new blinding for commitment
46:   [b_r,new] ← [b_r,old] ü [b]
47:   Map.insert(k_receiver, [b_r,new], [r_r new])
48:   C_r,old ← commit([b_r,old], [r_r,old])
49:   C_r,new ← commit([b_r,new], [r_r,new])
```

50:
51: $C_b \leftarrow \texttt{commit}([b], [r_b])$
52: $\pi_{\text{transfer}} \leftarrow \texttt{proof}\big(C_{\text{s,old}}, C_{\text{s,new}}, C_{\text{r,old}}, C_{\text{r,new}}, C_b\big)$ ▷ Collaborative SNARK, see Listing 3
53: **return** $\pi_{\text{transfer}}, C_{\text{s,new}}, C_{\text{r,new}}$
54: **end**

4 The Zero Knowledge Proofs

In this section we give details to the zero knowledge proof created by the MPC network (see Algorithm 1).

4.1 Deposit Proof π_{deposit}

The deposit proof (Listing 1) essentially proves that the commitment of the targeted user has been update correctly. It thus can be verified using the old commitment (stored in the smart contract), the new commitment (is put into the smart contract by the MPC network) and by knowing the deposit amount (which is also known by the smart contract at this point). Furthermore, we limit the deposit amount to a reasonable number, which is also part of the ZK proof. The reason for this limit is that withdraw and transfer proofs need to show that the new balance of the sender is still positive. Having a limit on the transfer amount simplifies this range proof.

In our current prototype we use a limit of 80 bit for transaction amounts. In case of ethereum where balances are denoted in Wei, this allows to transact $\frac{2^{80}}{10^{18}} \approx 1.2$ million ETH at once, which should be enough for most use cases.

Proof $\pi_{\text{deposit}} \leftarrow \texttt{prove}(C_{\text{old}}, C_{\text{new}}, b)$:

- Private witnesses:
  - Old balance b_{old}
  - Old blinding r_{old}
  - New blinding r_{new}
- Public witnesses:
  - Amount deposit b
  - Old commitment C_{old}
  - New commitment C_{new}
- Statements:
  - $b_{\text{new}} \leftarrow b_{\text{old}} + b$
  - C_{old} equals $\texttt{commit}(b_{\text{old}}, r_{\text{old}})$
  - C_{new} equals $\texttt{commit}(b_{\text{new}}, r_{\text{new}})$
  - b is in range (e.g., 80 bits)

Listing 1. The statements proven in π_{deposit}.

4.2 Withdraw Proof π_{withdraw}

The withdraw proof (Listing 2) is conceptually very similar to the deposit proof (Listing 1), with the difference that the balance is subtracted instead of added, and that the resulting balance proven to be ≥ 0. The latter statement is proven by a range check.

In our current prototype we somewhat arbitrarily prove that the resulting statement has at most 100 bit.

Proof $\pi_{\text{transfer}} \leftarrow \texttt{prove}(C_{\text{old}}, C_{\text{new}}, b)$:

- Private witnesses:
  - Old balance b_{old}
  - Old blinding r_{old}
  - New blinding r_{new}
- Public witnesses:
  - Amount withdrawn b
  - Old commitment C_{old}
  - New commitment C_{new}
- Statements:
  - $b_{\text{new}} \leftarrow b_{\text{old}} - b$
  - C_{old} equals $\texttt{commit}(b_{\text{old}}, r_{\text{old}})$
  - C_{new} equals $\texttt{commit}(b_{\text{new}}, r_{\text{new}})$
  - b is in range (e.g., 80 bits)
  - b_{new} is zero or positive (e.g., by checking it is in range of, e.g., 100 bits)

Listing 2. The statements proven in π_{withdraw}.

4.3 Transfer Proof π_{transfer}

Finally, the transfer proof (Listing 3) is essentially a combination of the deposit proof (Listing 1) and withdraw proof (Listing 2). In other words, the proof show that the sender commitments were updated as in a withdraw (including proof of enough balances), whereas the receiver commitments were updated according to a deposit proof.

Proof $\pi_{\textbf{withdraw}} \leftarrow \texttt{prove}(C_{\text{s,old}}, C_{\text{s,new}}, C_{\text{r,old}}, C_{\text{r,new}}, C_b)$:

- Private witnesses:
  - Old sender balance $b_{\text{s,old}}$
  - Old sender blinding $r_{\text{s,old}}$
  - New sender blinding $r_{\text{s,new}}$
  - Old receiver balance $b_{\text{r,old}}$
  - Old receiver blinding $r_{\text{r,old}}$
  - New receiver blinding $r_{\text{r,new}}$
  - Amount transferred b
  - Amount blinding r_b
- Public witnesses:
  - Old sender commitment $C_{\text{s,old}}$
  - New sender commitment $C_{\text{s,new}}$
  - Old receiver commitment $C_{\text{r,old}}$
  - New receiver commitment $C_{\text{r,new}}$
  - Amount commitment C_b
- Statements:
  - $b_{\text{s,new}} \leftarrow b_{\text{s,old}} - b$
  - $b_{\text{r,new}} \leftarrow b_{\text{r,old}} + b$
  - $C_{\text{s,old}}$ equals $\texttt{commit}(b_{\text{s,old}}, r_{\text{s,old}})$
  - $C_{\text{s,new}}$ equals $\texttt{commit}(b_{\text{s,new}}, r_{\text{s,new}})$
  - $C_{\text{r,old}}$ equals $\texttt{commit}(b_{\text{r,old}}, r_{\text{r,old}})$
  - $C_{\text{r,new}}$ equals $\texttt{commit}(b_{\text{r,new}}, r_{\text{r,new}})$
  - C_b equals $\texttt{commit}(b, r_b)$
  - b is in range (e.g., 80 bits)
  - $b_{\text{s,new}}$ is positive (e.g., by checking it is in range of, e.g., 100 bits)

Listing 3. The statements proven in π_{transfer}.

4.4 Batching Proofs

It is possible to batch multiple transfers into one ZK proof to reduce the gas fee for verifying multiple transfers on chain. Furthermore, `deposit` and `withdraw` can be proven by π_{transfer} as well, allowing to batch deposits, withdraws and individual transfers into just one ZK proof which verifies multiple transfers.

To verify a `deposit` via a transfer proof π_{transfer}, we do the following:

- Use the old balance, old blinding and new blinding of the `deposit` as the old receiver balance, old receiver blinding and new receiver blinding in a `transfer` respectively.
- Set the amount blinding to 0.

- Set the old sender balance to the amount b, the old sender blinding to 0, and the new sender blinding to 0 as well.

To verify a `withdraw` via a transfer proof π_{transfer}, we do the following:

- Use the old balance, old blinding and new blinding of the `withdraw` as the old sender balance, old sender blinding and new sender blinding in a `transfer` respectively.
- Set the amount blinding to 0.
- Set the old receiver balance to 0, the old receiver blinding to 0, and the new receiver blinding to the amount b.

5 The Full Protocol

In this section section, we describe the full protocols for reading balances, depositing/withdrawing funds and privately transfering tokens.

5.1 Read Balance

In our proposed system, a user can read balances by asking the MPC network to send over the balance and the randomness used for the commitment in plain. Notice that no zero knowledge proof needs to be involved, since the user can just read the commitment corresponding to itself from the smart contract and verify whether it matches what it received from the MPC network by recomputing the commitment. Consequently, no smart contract transaction is involved as well, only a read of the commitment if a user wants to verify the response of the MPC network. Finally, to keep balances private, the user needs to send a signature to the MPC network, proving its identity. The full protocol is given in Listing 4.

- **The User**:
  1. The user signs a timestamp t to create the signature $\sigma \leftarrow \texttt{sign}(t, \text{sk})$ using its private key sk. The user sends (t, σ, pk) to the MPC network
- **The MPC Network**:
  2. On receiving (t, σ, pk) from a user, the MPC network verifies that the timestamp t is still valid, verifies the signature $\texttt{verify}(\sigma, t, pk)$, and gets $([b], [r]) \leftarrow \texttt{Map.get}(\texttt{address}(pk))$. It then opens $[b]$ and $[r]$ to the user.
- **The User**:
  3. On receiving (b, r) from the MPC network, the user reads its commitment C from the smart contract and checks whether it matches $\texttt{commit}(b, r)$.

Listing 4. The protocol for reading balances.

5.2 Deposit

For a deposit, the user first has to transmit funds into the smart contract. Doing so, the smart contract registers a deposit action in a queue, which gets processed by the MPC network. While processing, the MPC network updates its shares corresponding to the user address, computes a commitment to the new balance, and a ZK proof proving correctness. It then posts the proof on chain, where the smart contract verifies the proof and stores the new commitment in its internal hashmap to show the user balance has been updated. The full protocol is given in Fig. 2.

Note that the deposit does not include any signatures, since the smart contract transaction implicitly authenticates the user already.

- **The User**:
  1. The user submits a transaction to the smart contract, sending b tokens to it.
- **The Smart Contract**:
  2. The smart contract registers a deposit action to its MPC queue, storing the users address k as receiver and the balance b as amount. If b is zero or too large it will revert.
- **The MPC Network**:
  3. The MPC network checks the queue on the smart contract and computes $(\pi_{\text{deposit}}, C_{\text{new}}) \leftarrow \texttt{Deposit}(k, b)$. It posts π_{deposit} and C_{new} into the smart contract.
- **The Smart Contract**:
  4. The smart contract receives π_{deposit} and C_{new}, verifies the proof π_{deposit} using the received C_{new} and the remaining public information which are computable on chain, sets C_{new} to be the new commitment corresponding to k and removes the action from the MPC queue.

Fig. 2. The protocol for depositing tokens.

5.3 Withdraw

For a withdraw, the user first has to post the intent into the smart contract. Doing so, the smart contract registers a withdraw action in a queue, which gets processed by the MPC network. While processing, the MPC network updates its shares corresponding to the user address, computes a commitment to the new balance, and a ZK proof proving correctness. It then posts the proof on chain, where the smart contract verifies the proof and stores the new commitment in its internal hashmap to show the user balance has been updated. If everything was done correctly, the smart contract sends funds to the user. The full protocol is given in Fig. 2.

Note that the withdraw does not include any signatures, since the smart contract transaction implicitly authenticates the user already. Furthermore, if the proof does not verify, the MPC network needs to roll back the transaction to keep secret shares of the old balance.

- **The User**:
  1. The user submits a transaction to the smart contract, indicating it wants to withdraw b tokens.
- **The Smart Contract**:
  2. The smart contract registers a withdraw action to its MPC queue, storing the users address k as sender and the balance b as amount. If b is zero or too large it will revert.
- **The MPC Network**:
  3. The MPC network checks the queue on the smart contract and computes $(\pi_{\text{withdraw}}, C_{\text{new}}) \leftarrow \texttt{Withdraw}(k, b)$. It posts π_{withdraw} and C_{new} into the smart contract.
- **The Smart Contract**:
  4. The smart contract receives π_{withdraw} and C_{new}, verifies the proof π_{withdraw} using the received C_{new} and the remaining public information which are computable on chain, sets C_{new} to be the new commitment corresponding to k and removes the action from the MPC queue. It then transmits b tokens to the user address k.

Fig. 3. The protocol for withdrawing tokens.

5.4 Transfer

When a user wants to transfer funds to someone else, it first needs to post the intent into the smart contract. This intent includes the receiver address, as well as a commitment to the amount which later gets linked to the zero knowledge proof posted by the MPC network. Then, the user needs to secret-share the transfer amount, as well as the randomness used to compute the commitments, to the MPC network. To authenticate the shares and prevent other users to post shares for transfers which are not theirs, the user has to sign the secret shares it sends to the MPC network. The MPC network, on reading the queue and receiving the secret shares, updates its balances, computes commitments to the new balances, and computes a ZK proof proving correctness. It then posts the proof on chain, where the smart contract verifies the proof and stores the new commitments in its internal hashmap to show the user balances have been updated.

Note that if the proof does not verify, the MPC network needs to roll back the transaction to keep secret shares of the old balances.

- **The User**:
  1. The user samples a random value r_b and computes the commitment $C_b \leftarrow \texttt{commit}(b, r_b)$. It then submits a transaction to the smart contract, indicating it wants to transfer tokens to k_{receiver}, where the amount corresponds to C_b. It also sends the secret share b_i of b and the secret share r_i of r_b to the i-th MPC party, alongside its user address k_{sender} and a signature $\sigma_i \leftarrow \texttt{sign}(b_i \parallel r_i, pk)$, where pk corresponds to k_{sender}.
- **The Smart Contract**:
  2. The smart contract registers a transfer action to its MPC queue, storing the users address k_{sender} as sender, the address k_{receiver} as receiver, and the commitment C_b.
- **The MPC Network**:
  3. The MPC network receives $([b], [r_b])$ and the signatures σ_i for each party i. Each party i verifies the signature σ_i using its shares b_i and r_i, and matches the shares with the transfer request in the queue of the smart contract. If everything is ok, it computes $\left(\pi_{\text{transfer}}, C_{\text{s,new}}, C_{\text{r,new}}\right) \leftarrow \texttt{Transfer}(k_{\text{sender}}, k_{\text{receiver}}, [b], [r_b])$. It posts π_{transfer} and the commitments $C_{\text{s,new}}$ and $C_{\text{r,new}}$ into the smart contract.
- **The Smart Contract**:
  4. The smart contract receives π_{transfer} and the commitments $C_{\text{s,new}}$ and $C_{\text{r,new}}$ and verifies the proof π_{transfer} using the received commitments and the remaining public information which are computable on chain. It then sets $C_{\text{s,new}}$ to be the new commitment corresponding to k_{sender}, $C_{\text{r,new}}$ to be the new commitment corresponding to k_{receiver} and removes the action from the MPC queue.

Fig. 4. The protocol for withdrawing tokens.

6 MPC Benchmarks

In this section we give some benchmarks of our currently implemented prototype. In all the benchmarks we use the 3-party replicated secret sharing protocol [MR18]. As commitments we use hash-based commitments using the Poseidon2 hash function [GKS23]. Concretely, we use Poseidon2 with an internal statesize of $t = 2$ in feed-forward mode of operation. In other words, a commitment to x with randomness r is computed as $C = \texttt{Poseidon2.permute}(x + d, r)[0] + x$, where d is a domain separator. We use the Noir[1] language to write the ZK circuits, use a

[1]https://noir-lang.org/
[2]https://github.com/TaceoLabs/CoNoir-to-R1CS
[3]https://github.com/TaceoLabs/co-snarks/tree/main/co-circom

compiler[2] to translate it to an R1CS system and use Co-Circom[3] to create a Groth16 [Gro16] proof in MPC. Thereby, Groth16 is instantiated over the BN254 pairing friendly curve.

In all benchmarks in this section, we give numbers for different subprotocols. Thereby, `No Proofs` refers to all MPC computations without verifiability. In other words, only the data structure is updated and commitments are computed in MPC. On the contrary, `Full` refers to the full MPC protocol including the CoGroth16 proof. Furthermore, we give benchmarks for the two major parts involved in `Full`, namely `WitExt` (witness extension) which includes modifying the data structure and creating the witness for R1CS constraint system in MPC[4], and `CoGroth16` which translates the R1CS constraint system (including secret shared witness) into a zero knowledge proof.

6.1 Single Benchmarks

In this section we give benchmarks for processing one individual deposit, withdraw, and transactions. Runtimes are given, in Table 1, the corresponding throughput in actions per second in Table 2, while Table 3 gives the communication between the parties. All benchmarks in this section were performed on a co-located setup where each party is a m7a.4xlarge AWS instance with 16 CPU cores and a network bandwidth of 12.5 GBit/s.

Table 1. Runtime in `ms` for all the steps involved in deposit, withdraw and transfer.

Protocol	No Proofs	WitExt	CoGroth16	Full
Deposit	4.4	5.9	8.1	14.0
Withdraw	4.1	6.2	8.6	14.8
Transfer	4.3	6.9	12.3	19.2

Table 2. Throughput in `1/s` for all the steps involved in deposit, withdraw and transfer.

Protocol	No Proofs	WitExt	CoGroth16	Full
Deposit	227.3	169.5	123.5	71.4
Withdraw	243.9	161.3	116.3	67.6
Transfer	232.6	144.9	81.3	52.1

[4]This also implicitly creates the commitments.

Table 3. Communication in `kB` for all the steps involved in deposit, withdraw and transfer.

Protocol	No Proofs	WitExt	CoGroth16	Full
Deposit	35.1	37.8	0.2	38.0
Withdraw	35.1	41.0	0.2	41.2
Transfer	58.1	64.3	0.2	64.5

The tables show, that processing an individual action is reasonably fast, where a singlethreated execution without proof takes roughly 4 `ms`, which gets increased to 19 `ms` for the transfer action executed with ZK proof[5] Furthermore, communication is less than 65 `kB` in any case. 'This leads to a throughput of 230 wihtout proof and about 50 with proof.

6.2 Batched Benchmarks

In this section we give benchmarks for a multithread and batched version of the protocol (see Section 4.4). Concretely, we batch 96 transaction. Thereby, `No Proofs` and `WitExt` compute each individual transaction in a separate thread, whereas `CoGroth16` produces on ZK proof which verifies 96 transactions internally.[6] The benchmarks are given in Table 4 for two different machine setups. Furthermore, Table 5 gives the corresponding throughput of transactions per second.

For the two machine setup we benchmark the case where each MPC party is one m7a.4xlarge AWS instance with 16 CPU cores and a network bandwidth of 12.5 GBit/s (same as for the benchmarks in the previous section), and the case where each MPC party is a m7a.24xlarge instance with a bandwidth of 37.5 GBit/s. The latter machine has 96 CPUs which matches the batch size we benchmark.

Table 4. Runtime in `ms` for all the steps involved in a batch of 96 transfers.

Machine	No Proofs	WitExt	CoGroth16	Full
m7a.4xlarge	22.9	197.8	595.8	793.6
m7a.24xlarge	15.3	185.9	273.0	458.9

[5] Witness extension is singlethreated, while CoGroth16 uses multithreading internally.
[6] CoGroth16 uses multithreading internally to compute the multiscalar multiplications more efficiently.

Table 5. Throughput in `1/s` for all the steps involved in a batch of 96 transfers.

Machine	No Proofs	WitExt	CoGroth16	Full
m7a.4xlarge	4192.1	485.3	161.1	121.0
m7a.24xlarge	6274.5	516.4	351.6	209.2

Notice that multithreading has a profound effect on the performance. A batch of 96 transaction requires between 500 `ms` and 800 `ms`, depending on machine setup, for modifying the data structure including ZK proof. Without proof the runtime is roughly 20 `ms`. Throughput, thus is increased to roughly 200 transactions per second with proof, and 6000 for a non-verifiable variant.

Notice, that there is not much difference between the runtime of the witness extension (`WitExt`) in the two different machine setups. We believe that the small difference is mostly due to the larger network bandwidth of the m7a.24xlarge instances, and that the performance of the witness extension is mostly network bound.

Proof generation, on the other hand, is clearly CPU bound, where the increase of number of CPUs from 16 to 96 improves performance by more than two-fold.

7 Issues and Future Work

In this section we discuss some issues of the currently described protocol and potential future work.

7.1 Implications of Batching Transactions

A prototype implemented from the batched version of our protocol (see Section 4.4 and Section 6.2) has the advantage that a batch of transaction (e.g., 96) only creates 1 proof, reducing on-chain gas fees for proof verification. However, it comes with some drawbacks as well.

First, when creating a ZK proof for a single transaction, the smart contract and the MPC network immediately know if the transaction was valid or faulty by verifying the proof. In a batched version of the proof, however, if one transaction fails, the whole batch fails. Currently, there is also no method implemented to detect which of the transactions in a batch were valid or faulty. We think of two potential solutions to this issue.

The first solution is checking in MPC whether the range checks in the zero knowledge proofs (see Listing 3) would be valid or not and open the resulting valid-bit. Based on this bit, the MPC network would then decide to not include the transaction in a batch. To cleanup the on-chain action queue, the MPC network would then call a removal function exposed by the smart contract to remove the faulty transaction from the queue. While this solution is simple and should induce only small performance overhead, it comes with the disadvantage that the

existence of a removal function for the on-chain action queue gives the MPC network the possibility to ignore transactions and censor specific user addresses.

The second solution is modifying the zero knowledge proofs to be valid even for faulty transactions, but outputting a bit indicating whether the transaction was valid. In other words, the range constraints are modified to output 0 if in range and 1 if not in range. This bit is an extra input to the smart contract and the proof verification, and if the bit is set, the smart contract removes the faulty action from the queue.

If the latter solution is implemented, it is probably also required to let the smart contract enforce the order of the processed transaction to prevent letting the MPC network just ignoring some transactions indefinitely.

7.2 Concurrent Transactions and Sharding

The current prototype implements multithreading by first modifying the data structure in sequence, and then spawning a thread to compute the witness extension fully in parallel. This has the advantage that we do not need to care about multiple transactions modifying the same commitments in our hashmap concurrently which would lead to faulty results. Furthermore, modifying the data structure is simple and requires no communication, so no performance is lost by doing it in sequence.

This also has the advantage that we can scale up our solution with sharding computations to multiple servers. First on thread takes care of modifying the data structure and then delegates the witness extension and proof generation to a different MPC setup.

The downside of such an approach, however, is that if the MPC network detects faulty transactions, it needs to rewind all the transactions that came after the faulty one, since they might have been using a faulty commitment as input. This would not be the case if a batch only contains a disjoint set of senders and receivers.

References

[MR18] P. Mohassel and P. Rindal, "ABY3: A Mixed Protocol Framework for Machine Learning," in *CCS*, ACM, 2018, pp. 35–52.

[GKS23] L. Grassi, D. Khovratovich, and M. Schofnegger, "Poseidon2: A Faster Version of the Poseidon Hash Function," in *AFRICACRYPT*, in Lecture Notes in Computer Science, vol. 14064. Springer, 2023, pp. 177–203.

[Gro16] J. Groth, "On the Size of Pairing-Based Non-interactive Arguments," in *EUROCRYPT (2)*, in Lecture Notes in Computer Science, vol. 9666. Springer, 2016, pp. 305–326.