b
UNIVERSITÄT
BERN

Multiparty Computation on Blockchain

MPyC on Tendermint Core

Bachelor Thesis

Gillian Cathomas

from

Bern, Switzerland

Faculty of Science, University of Bern

13. January 2022

Prof. Christian Cachin
Orestis Alpos
Cryptology and Data Security Group
Institute of Computer Science
University of Bern, Switzerland

Abstract

The goal of this thesis is to run Multiparty Computation on a blockchain. A blockchain is a chain of blocks that contain transactions. Every participant of the blockchain has a copy of it. A consensus among those participants is needed in order to add a block and execute the included transactions. Manipulating a single finalized block is currently impossible since this requires changing all the blocks that follow after the manipulated block. This is because every block calculates its own hash using inter alia the hash of the previous block. Furthermore, the manipulation of a block would need to occur in the majority of the participants. Hence a blockchain is a decentralized and distributed framework. The transparency of a blockchain is a significant disadvantage. The blocks often have information about which party did what in a transaction as well as the results. This is why this thesis aims to add privacy by running Multiparty Computation on the blockchain. Multiparty Computation allows multiple parties to calculate a function without revealing any information about their private input. Furthermore, no trusted centralized instance is needed for the computation. In this thesis, Tendermint Core is used as the blockchain, and MPyC is used to facilitate Multiparty Computation. MPyC is a Python package, which allows developers to create Multiparty Computation protocols by providing secure functions and types. Specifically, the goal of this thesis is to run MPyC on Tendermint Core, which we accomplished by implementing multiple classes. One participant can request the start of computation by running the *rpc.py* file, resulting in the Multiparty Computation being done on-chain. The result is included in the block and can also be queried for using the provided RPC endpoints.

Acknowledgement

First and foremost, I would like to thank Orestis Alpos for the continuous support and supervision of my thesis. Furthermore, I would like to thank Christian Cachin for offering me such an interesting topic as a thesis and sharing his expertise. Lastly, I would like to thank my family and friends for supporting me as I worked on this thesis.

Contents

Chapter 1

Introduction

Recently, due to the increased interest in blockchain technology, Multiparty Computation drew also more attention. The goal of Multiparty Computation is to compute functions securely with the input from parties. There is no trust needed between those parties, and no one's secret will be revealed. The result is then reported back to the participants. Hence Multiparty Computation keeps data private. That is the reason why we added it to a blockchain. One of the signigicant disadvantages of blockchains is that they are transparent and thus lack privacy. By running Multiparty Computation on a blockchain, the computation can be hidden and only the result can be written on a blockchain. Additionally, Multiparty Computation and blockchains are both distributed systems making them easily compatible.

The question this thesis answers is "How can secure Multiparty Computation be added to a blockchain?" There are already some existing solutions to this question. The first solution comes from a team of IBM researchers [1]. In their approach, they used Hyperledger Fabric as the blockchain and extended it to run a protocol and a Multiparty Computation primitive on-chain. A limitation of this approach is the small number of protocols that can be run. Another limitation is the helper server that stores all the secrets of the participants. It is thus the most worthwhile target for a potential attacker [1]. A second very elaborate solution comes from Partisia Blockchain Foundation [6]. Their goal is to make Multiparty Computation on blockchain accessible to everyone. Their service can be paid with many different cryptocurrencies. Partisia created a blockchain in order to run Multiparty Computation on it [6]. From our point of view, there are no limitations or disadvantages to Partisia's approach. Considering that the Partisia Blockchain Foundation has multiple researchers and developers working on it for years, it is not surprising that their solution is very sophisticated. Therefore, we could not recreate a system similar to theirs. One difference to Partisia's solution is that we decided to use Tendermint Core, an already existing replicated state machine. A replicated state machine could informally be seen as a blockchain.

The goal for this thesis is that developers can create any kind of protocol by using MPyC and then call those programs using Tendermint Core [10, 25]. MPyC is a Python package allowing developers to create Multiparty Computation programs. The limitations of Hyperledger Fabric don't exist in our solution. No helper server is needed and any program written with MPyC that runs on its own can be run on Tendermint Core. The focus of this thesis is mostly on feasibility.

In our solution, we implemented methods like `check_tx()` and `deliver_tx()` as part of Tendermint. Any MPyC program can be called in the `deliver_tx()` method. The state of the blockchain and the result of past transactions can be queried. Multiple checks are performed on the client's input in order to run MPyC. Furthermore, we will highlight the importance of determinism in the execution of a MPyC program. Our solution has some restricting assumptions. The first is that the nodes running Tendermint Core are also running MPyC. The second is that currently, one node can save the secret input of one client. And finally, message authentication is not taken into account.

The second chapter of this thesis gives an overview of the background. Especially Tendermint Core and Multiparty Computation are explained here. In the third chapter, we present the design of our solution. Furthermore, we compare the existing solution of Partisia Blockchain Foundation to the IBM research teams' solution. The implementation of our solution is described in Chapter 4. There we depict the most

important methods like `check_tx()` and `deliver_tx()`. In the subsequent chapter, we investigate the question of how MPyC's runtime differs between MPyC being called as a separate program or as part of Tendermint. In this chapter, we also examine how the execution time changes for the same program when more validator nodes are added. Finally, in Chapter 6, we conclude and refer to future work.

Chapter 2

Background

2.1 Replicated State Machine

In this section, we refer to information from Schneider [9]. The notion of a replicated state machine is of importance for this thesis because Tendermint Core is a replicated state machine. According to Schneider[9] "The state machine approach is a general method for implementing fault-tolerant services in distributed systems." A distributed system is generally composed of two parts. The first is the client and the second is services. A service consists of one or multiple servers. In the context of this thesis, a service is a state machine, which has a state. That state is encoded by the state variable. The state machine also has commands. These commands are deterministic programs. They can change the state and/or create some output. The execution of a command has to be requested by a client. The client sends a request with all information including the command and any additional information that is required by command. The output of a state machine is determined solely by the order of requests.

A replicated state machine is a state machine whose state is replicated. The state is stored on different processors which are distributed in the system. Those processors will run the commands. Every processor initiates with the same state. Assuming it is not faulty, it will process the client's requests in the same order as the other processors where a replica is run. This is one of two requirements called *order* for *Replica Coordination* needed by a replicated state machine. The second requirement, called *agreement*, requires that all the correct running state machines receive every request. Finally, it will output the same result as the other processors. In Tendermint Core the state we start with is called genesis state. When the replicated state machine is in genesis state, neither a blockchain nor a block exist. Every transaction that gets executed changes the state of Tendermint Core.

2.2 Tendermint Core

Tendermint and Tendermint Core are often used interchangeably. However, Tendermint Core is the software, and Tendermint is the enterprise Tendermint Inc. Tendermint Inc. developed Tendermint Core, Starport, Cosmos Network and the Inter-Blockchain Communication Protocol. Throughout this paper, the terms Tendermint and Tendermint Core are used synonymously for the software unless stated otherwise.

Tendermint is composed of two main parts. The first part is a blockchain consensus engine Tendermint Core. The second part is the generic application interface. This interface is called Application BlockChain Interface or short ABCI [25]. As the name implies, it is an interface, which has methods that are called via request messages and respond with a response message. The ABCI allows a developer to write their application in many languages due to the messages and methods of the ABCI being defined in Google's protocol buffers. The interface is between the developer's application and Tendermint Core. Tendermint is a Byzantine Fault Tolerant "state-machine replication engine", while the developed application is the real state machine [12].

2.2.1 Architecture of Tendermint Core

The architecture of Tendermint has two kinds of applications. There is the end-user application and the ABCI application. The ABCI application has the logic which is run on Tendermint. As mentioned above, the ABCI application and the blockchain consensus engine are separated. This separation can also be seen when looking at the different layers. The application layer is separated from the consensus and networking layers. Those separated layers have different ways of communicating with each other. The end-user application uses RPC endpoints to communicate with Tendermint. Tendermint communicates directly with the ABCI application by calling the ABCI. In order to keep the consensus working, only Tendermint must communicate with the ABCI. This is done by using `/abci_query` to retrieve information from the ABCI application and using `/broadcast_tx_*` to send transactions to the ABCI application. The `abci_query`, as well as the `/broadcast_tx_*`, are RPC endpoints of Tendermint which inter alia can be called using an curl command. Only a working consensus can commit transactions. The commitment of transactions is required for the transactions to get executed by the ABCI. Unless the ABCI application is written in Go, the developer should use UNIX sockets or TCP to talk to Tendermint [13].

We would like to use Figure 2.1 for further explanation and illustration of the architecture. In Figure 2.1, Cosmos Voyager (colored in purple) represents the end-user application. Furthermore, a part called Light Client Daemon can, but does not have to, be used. In our solution, no Light Client Daemon was explicitly implemented. More information about the Light Client Daemon can be found under [19]. The end-user application can communicate directly with the RPC endpoint of Tendermint Core. Transactions that Tendermint Core received over the RPC endpoints will be forwarded to the ABCI. A full node consists of Tendermint Core, the ABCI application, and a validator signer. The nodes communicate with each other by using the peer-to-peer network. In this network, they reach consensus and broadcast transactions. Finally, a JSON dictionary will be sent back to the end-user application where it will be interpreted accordingly [13].

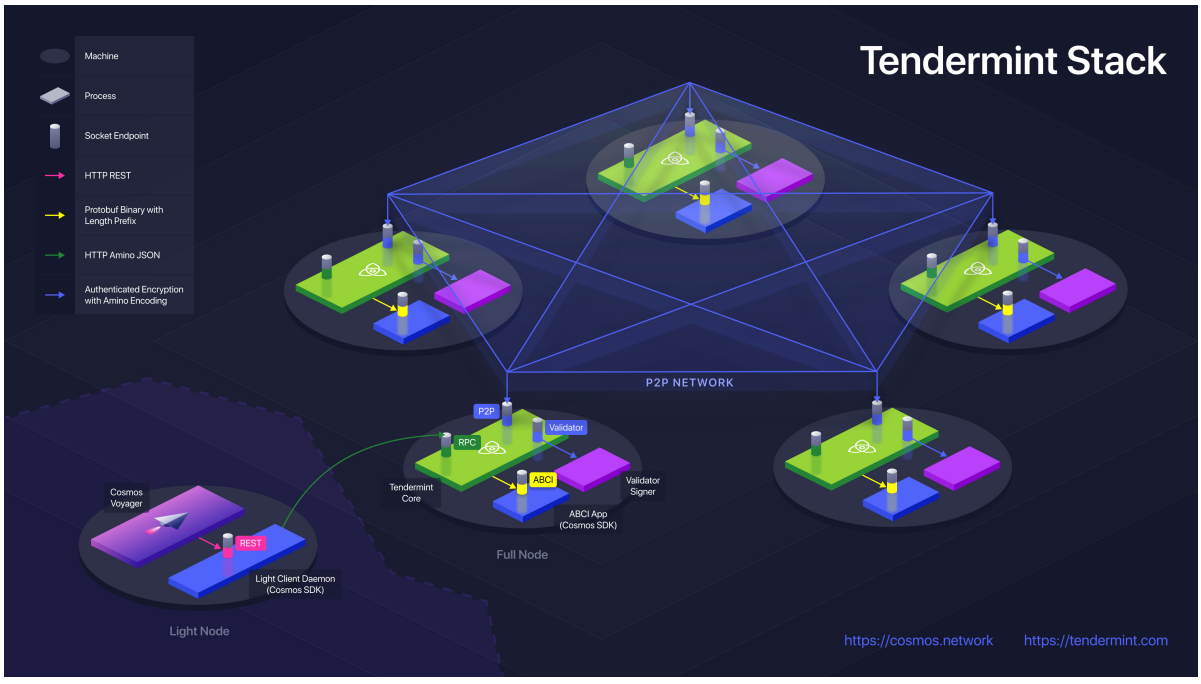

Figure 2.1. Architecture of Tendermint with Cosmos Voyager as end-user application [13]

2.2.2 ABCI methods

We would now like to go into more detail about the methods that the ABCI provides.

init_chain()

If a new blockchain gets created, `init_chain()` is called after the `info()` method. This is the only case when `init_chain()` is called. In this method, the developer can determine what should be loaded and set at genesis time. Furthermore, it allows the developer to set the initial validators. To do so, the validators have to be included in `ResponseInitChain.Validator`. If the validators are not set in the response, the initial validator set defined in the *genesis.json* file will be called [18].

info()

This method is called when the ABCI application is starting. In case the block height is zero, Tendermint will call the above `init_chain()`.The client can also call `info()` by sending the curl command `curl localhost:26657/abci_info`. This command provides information about the application state [18]. Furthermore, the `info()` method helps Tendermint avoid replaying the logged transaction when restarting. This has to be implemented but can be done by looking up the last block height [4].

check_tx()

`Check_tx()` is one of the core methods. It is used to determine whether a transaction is valid according to the current state and gets added to the mempool or not. If a transaction is invalid, it will not be included in a proposal block nor broadcasted. According to the documentation, this method is optional. This method returns a response code zero if everything is ok. Else, the response code is set to one, indicating that an error or an invalid transaction has occurred [18]. It is also the responsibility of `check_tx()` to protect the blockchain against replay [14].

deliver_tx()

`Deliver_tx()` is the second core method, considering it has the responsibility to deliver every transaction of a block to the application. Unlike `check_tx()`, which is technically not obligatory, `deliver_tx()` is. Because `check_tx()` is optional, `deliver_tx()` should still validate the transaction and only return code zero if the entire transaction is valid. Invalid transactions will nonetheless be included in a block. Moreover, if the transaction is valid, it has to be executed, and the changes to the application state should be applied. The execution has to finish before returning to Tendermint. This is very important for Tendermint to reach consensus. If the application doesn't terminate, Tendermint can not reach consensus [18].

begin_block()

`Begin_block()` is called when a new block is created. Consequently, it is called before `deliver_tx()`. The request contains a lot of information. Currently, its header is the same as the block header. Therefore, it has information about the height, timestamp, last block id, last commit hash, data hash, and much more. The last commit info contains information about the round as well as the existing validators. It is also from here that the developer can read which validators signed the last commit. Furthermore, there is the `byzantine_validators` entry, where a list of evidence of malicious validators is provided. That information can be used to reward or punish validators [18].

end_block()

`End_block()` is the counterpart to `begin_block()`. It is called at the end of every block, once all transactions were executed by the `deliver_tx()` method. However, it is called before `commit()`. The last block height can be seen here. The response of `end_block()` can contain a field called `validator_updates`, which allows a developer to update the validator set. The change in the validator set is happening at the block with the current height plus 2. The block after the current height will

only include the `validators_updates` in `NextValidatorsHash` but will still vote with the old set of validators [18].

commit()

`Commit()` is one of the few methods that takes no parameters in their request. Hence it is impossible to pass any data to it. `Commit()` is executed once `end_block()` has finished. Here the state changes should be saved, and in case the block height isn't updated yet, update the block height [18].

There is also the `query()` method that can be used to obtain information about the state of the application. Furthermore, there are four more methods; `list_snapshots()`, `load_snapshot_chunk()`, `offer_snapshot()` and `apply_snapshot_chunk()`. All of these four methods are used in case a new node joins the network and tries to catch up to the other nodes by using existing snapshots of the blockchain. The advantage compared to replaying all the blocks lies in the fast performance and hence facilitates joining the network and becoming a node [18].

2.2.3 Node types

Tendermint knows four different kinds of nodes [20]. The first one is a *seed node*. A seed node does not actively partake in reaching consensus. It connects to a node and helps it find other peers by providing a list of active peers, to which the node can then make a connection. The addresses from the list get stored in the node's address book. From there, the node can connect directly without any more help from the seed node [23]. The seed node will disconnect quickly after the node has received the list with addresses [20]. A seed node should constantly be crawling the network. Thus it is able to find new peers. Once a node has enough addresses, it will not connect to a seed node again. Hence seed nodes are primarily used in the beginning [23].

The second node type is a *full node*. A full node also does not actively partake in reaching consensus but facilitates coming to consensus. For example, a full node can save the blockchain state as well as the application state. Additionally, a full node can help nodes that are at a smaller height to catch up [20]. For this, the full nodes run a *Blocksync Reactor*. If a new node wants to sync up to the current height, it runs the Blocksync Reactor, which provides blocks in the *fast_sync* mode. This means unless the node has caught up, it will constantly ask for new blocks. If the node turns the fast_sync mode off, it indicates that the node caught up. It is at this time that the node will change to use the *Consensus Reactor* [22].

A *validator node* is the most crucial kind of node because it takes part in reaching consensus. They are expected to always be online [24]. Such a node can propose a block or vote on one [23]. It, therefore, needs the highest security [20]. A validator node should only accept incoming connections from trusted validators. Apart from VPN connections to other trusted validators, a validator node should only go through its sentry nodes to talk to the rest of the network [21]. A validator node has voting power and a public-private key pair. The private key is used to sign votes. There are multiple ways for a node to become a validator. The easiest is by configuring it in the *configure.toml*. Another way is by passing the addresses to the `persistent_peer` flag when starting Tendermint. At last, it is possible to use the optional field `validator_updates` in the `end_block` message [24].

A *sentry node* is a full node with the exception that a sentry node has more full nodes and/or validator nodes as private peers. The security that validator nodes need is provided by sentry nodes. Sentry nodes also act similarly to a firewall [20]. Hence, they do not gossip the address of the validator node and they might have higher standards when it comes to other peers. Sentry nodes provide the validators with access to the rest of the network. Since they are the only access for validator nodes to the network, it is of importance that the sentry nodes are well connected. For each two validator nodes that trust each other, their sentry nodes are recommended to have a persistent VPN connection [21].

2.2.4 Consensus

We would now like to go into further detail about what happens once a transaction has passed the local `check_tx()` method and gets broadcasted. Tendermint uses a Byzantine Fault Tolerant consensus. This means that even when some parties are corrupted, it is still possible to reach a consensus. As we will explain later, in Tendermint more than 2/3 of participants have to be honest to have working consensus [8]. A visualization can be seen in Figure 2.2. In order to reach a consensus, the validators go through one or multiple rounds. A round consists of *Propose* followed by *Prevote* followed by *Precommit*. There are different steps such as *NewHeight*, *Propose*, *Prevote*, *Precommit* and *Commit* in those rounds. Every listed step stands for a state of the state machine [15].

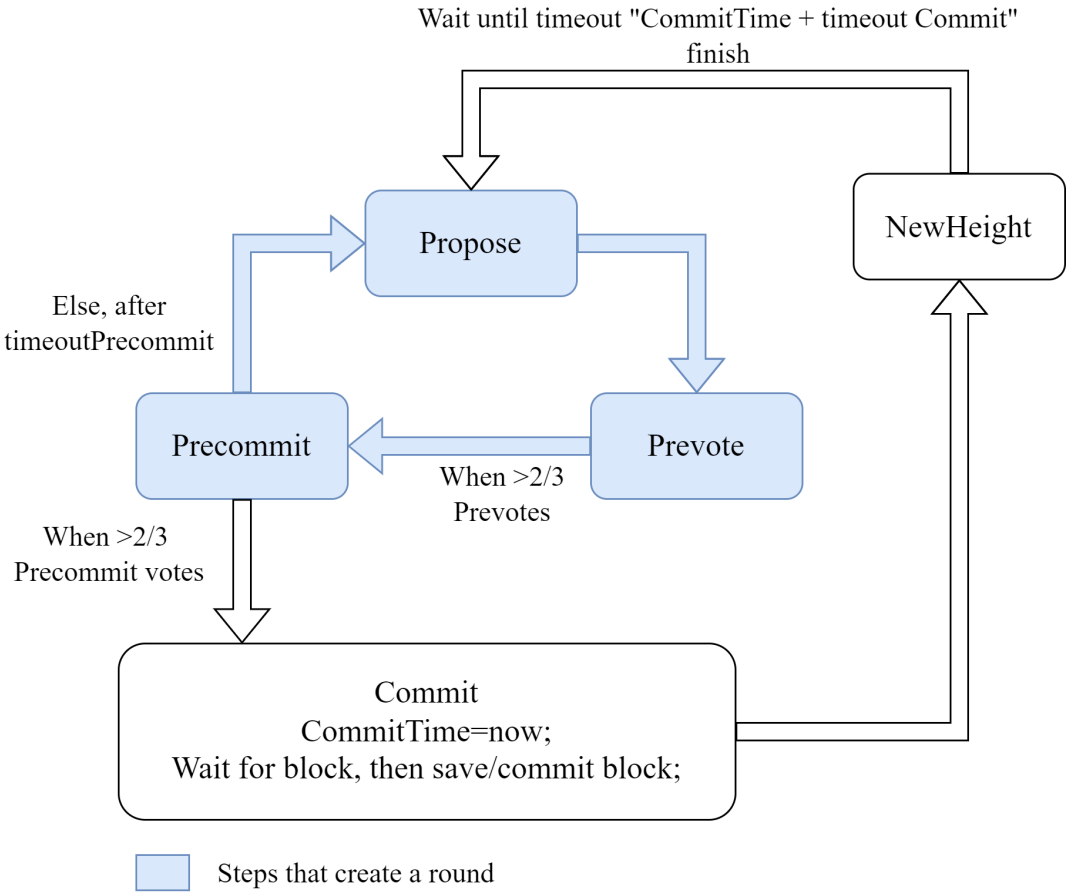

Figure 2.2. Consensus steps according to [15]

Every validator receives the transaction and decides whether it is the proposer for this block. This is determined by using a round-robin selection algorithm. The proposer is proportional to the voting power selected. The selected proposer will then propose the block [15]. Once the nodes have received the complete proposal block, they will start verifying the new block.

According to the Tendermint Core documentation, the verification process is as follows [16]. First, the validity rules of a block and its fields, as well as the versions, are checked. It gets checked that the block version is the same as in the local state; the same applies to the application version. Furthermore, the `chainID's` and the hashes in the header have to match the local state. Then the field `LastBlockID` has to be the same as the current `BlockID`. If it is not the first block of the chain, the `LastCommit` has to be the same as in the state. It is also at this time that the signatures are checked. They have to be the same as in the last block. Next, it is verified that the height is correct. Then it is checked that the proposer is a validator node. Moreover, the block time is validated. This means that the block time is after the last block's time. If this is not the case, the block time is compared to the genesis time. Next, the `medianTime` is calculated and matched against the block's time. Finally, the evidence is validated. It

is acceptable if the evidence is empty [16].

If all the checks pass, the block is deemed valid, and the validator nodes start the prevote step. This validation process is repeated before precommit and before the commit is finalized. In the prevote step, the validators start voting on the blocks and broadcasting their votes. As soon as there are more than 2/3 prevotes, it is locked, and the precommit step is started. Locked means that a *proof-of-lock-change* is obtained, which is the name for the more than 2/3 prevotes. A proof-of-lock-change could also be obtained if a set of more than 2/3 `<nil>` at height H in round R occur. In the precommit step, the nodes again vote and broadcast their vote. Once more more than 2/3 precommit votes have to be obtained to start the commit step. If more than 2/3 precommit votes are not obtained within the `timeoutPrecommit`, a new round will be started. Once the consensus reaches the commit step, the commit is finalized. Now the block is executed as well as the state is committed. Then a timer for the *newHeight* is started, allowing the nodes to receive more precommit votes. At this time, Tendermint also assigns `LastCommit` the `Precommits`. Moreover, while the timer is running, the transactions in the block are indexed. Once the timer times out, a new round is started at the new height [15].

2.3 Secure Mulitparty Computation

As the name implies, Secure Multiparty Computation is a way for two or more people to evaluate a computation together without disclosing their secret input. Secure Multiparty Computation is often abbreviated with MPC, irrelevant to the number of participants. Multiparty Computation is a secure way for a group to anonymously and cooperatively perform a computation. Usually, a function will be calculated using all the inputs from the group members. No trust between the group members is needed because no private information will be disclosed to the rest of the group. Multiparty Computation started with Andrew Yao in the 1980s and piqued the interests of researchers at the beginning of the 21st century. The easiest and most used example to explain Multiparty Computation is that it can solve Yao's Millionaires Problem. Given two millionaires, they both want to find out who is wealthier without telling how many assets they have. The solution to this problem is to calculate the function $f(x_1, x_2) : x_1 <= x_2$ where x_1 is the first millionaire's input and x_2 is the second millionaire's input [5].

2.3.1 Security

MPC is about providing more privacy and security. In this section, we would like to elaborate, which security properties MPC provides. In addition, we also define security using the "real-ideal paradigm".

Real-ideal paradigm In the real-ideal paradigm, we compare the ideal world to the real world to determine the security of a protocol. In an ideal world, all security guarantees are included. Furthermore, there is a trusted party T to which all the parties $P_1, ..., P_n$ securely send their secret input. This trusted party T, which is also called *functionality*, will then compute the function $F(x_1, x_2, ..., x_n)$. The result will be returned to every party. This world is ideal because no completely trustworthy third party exists in the real world. Even in this ideal world, an attacker can take over any participating party, except the *functionality T*. Nevertheless, the attacker would not gain any information more than the output because the input of the honest parties is independent of the attacker's input choices [5, Section 2.3].

Since there is no trusted party in the real world, the parties use a protocol π to communicate. The "next-message" function π_i for every $i \in \{1, ..., n\}$ is given by the protocol π. The input this "next-message" function needs is a security parameter, the private input x_i, a random tape, as well as all received messages of that party. It allows the parties to create a "next-message". This message is either sent to the correct parties or it is an order to this party to halt with some output. If an adversary corrupts a party at the start of the protocol, it is the same as if the party were a malicious adversary from the beginning. If an attacker is able to do the same actions in the ideal world as it does in the real world, a protocol is considered secure [5, Section 2.3].

Semi-honest and Malicious behavior There are two types of behavior for a corrupted party. The first one is semi-honest behavior. This means the corrupted party fulfills the protocol correctly but passes any information or values to the adversary. This is also called "passive" corruption because the faulty party does not actively work against others and the protocol. An example could be a "read-only" attack on a server. The second behavior is malicious behavior. In this case, the party behaves unpredictably and deviates from the protocol. This is "active" corruption because the faulty party takes action against the honest parties and the protocol. The goal is to break security. Those attacks are coordinated by the adversary, who corrupted the malicious parties [5, Section 2.3].

In this thesis, we assume that a secure channel exists between every two participants of MPC. Furthermore, in the context of MPC, parties and participants are interchangeable. In the following subsections, we would like to mention some of the main primitives of MPC.

2.3.2 Secret sharing

Secret sharing is one of the most used and, therefore, most important primitives in MPC. Given n participants, we share one secret s among those participants. On the one hand, if an adversary were to collect $t-1$ shares no information would be revealed. On the other hand, t shares are enough to reconstruct the secret s. This is called a (t, n)-secret sharing scheme. To define secret sharing Evans et al.[5, Section 2.2] first define a secret sharing algorithm as a function.

Definition 2.3.1 (Secret sharing[5])**.** *$Shr : D \to D_1$ where D denotes the domain of secrets and D_1 the domain of shares. Then he defines $Rec : D_1^k \to D$ as the reconstruction algorithm. Lastly, a (t, n)-secret sharing scheme is defined as a pair of the before-defined algorithms (Shr, Rec). That pair has to fulfill the following two properties:*

1. *Correctness: [5, Section 2.2] Let $(s_1, s_2, ..., s_n) = Shr(s)$. Then,*

$$Pr[\forall k \geq t, Rec(s_{i1}, ..., s_{ik}) = s] = 1$$

2. *Perfect Privacy: [5, Section 2.2] If someone gets access to a set of up to $t-1$ shares, the person would not gain any information about the secret. Let $a, b \in D$ and any possible vector of shares $v = v_1, v_2, ..., v_k$, such that $k < t$,*

$$Pr[v = Shr(a)|_k] = Pr[v = Shr(b)|_k]$$

where $|_k$ denotes appropriate projection on a subspace of k elements.

2.3.3 Oblivious Transfer

The acronym OT stands for Oblivious Transfer. Oblivious Transfer is often used in Multiparty Computation and is an asymmetric primitive. Consequently, it uses public and private keys [5, Section 3.7]. The idea behind OT is that a receiver R who has a choice bit c where $c \in \{1, ...n\}$ can get one element x_c from a sender S. The sender has a database with elements $x_1, ..., x_n$. Since the receiver gets one element and the sender has n elements, this OT is called an $\binom{n}{1}$-OT. Important in OT is, that S does not learn which c the receiver chooses, and that R does not learn any information about the other elements x_i where $i \neq c$. Oblivious Transfer can be used to access databases privately or most commonly as a part of a more complex secure protocol [2, Section 11.7].

We would now like to show how a $\binom{2}{1}$-OT protocol from ElGamal works. We assume a semi-honest model. Let's assume Alice and Bob would like to run this protocol. Let Alice, the sender, have two plaintext secrets $m_0, m_1 \in \{0,1\}^n$ and Bob, the receiver, a selection bit $c \in \{0,1\}$. Since OT uses public and private keys, Bob has to create two public keys y_0, y_1 so that only he knows the private key for y_c [11].

Definition 2.3.2 (Oblivious Transfer [11])**.** *The given parameters of a Discrete Logarithm Problem-based cryptosystem are*

$$G =< g >$$
$$|G| = q$$
$$H : G \to \{0,1\}^n$$

where H is the hash function, q is prime, G is a group, where the discrete logarithm is hard, and g is the generator of G. Also let EG_Enc(pk,m) denote the ElGamal encryption algorithm, where pk is the private key and m is the message. Furthermore, EG_Dec(sk,c) is the ElGamal decryption algorithm with sk as the secret key and c the encrypted message [2, Section 11.5].

$Alice\ is\ S(m_0, m_1)$		$Bob\ is\ R(c), for\ c \in \{0,1\}$
$r \leftarrow \mathbb{Z}_q$		
$t \leftarrow g^r$	$\xrightarrow{t}$	$x_c \leftarrow \mathbb{Z}_q$
		$y_c \leftarrow g^{x_c}$
if $y_0 * y_1 \neq t$ *then halt*	$\xleftarrow{y_0, y_1}$	$y_{\bar{c}} \leftarrow t/y_c$
$(R_0, C_0) \leftarrow EG\_Enc(y_0, m_0)$		
$(R_1, C_1) \leftarrow EG\_Enc(y_1, m_1)$	$\xrightarrow{(R_0, R_1, C_0, C_1)}$	$z \leftarrow EG\_Dec(x_c, R_c, C_c)$
		return z

This protocol is complete because Bob, the receiver, will know x_c. It is also secure for both parties. For Alice, the sender, security means that Bob will only know either the discrete logarithm of y_0 or y_1. However, Bob can't know both because y_0 and y_1 are chosen such that $t = y_0 * y_1$ holds and t is chosen by Alice. Moreover, the security of ElGamal assures Alice that Bob will not get any information about $x_{\bar{c}}$. The security for Bob is that y_0 and y_1 are identically distributed and hold that $t = y_0 * y_1$. Hence he has unconditional security [11].

2.3.4 Yao's Garbled Circuit Protocol

There exist many different MPC protocols. We would like to detail about the protocol that started MPC. This is explained in [5, Section 3.1]. The goal of Yao's Garbled Circuit protocol is to compute a function $f(x, y)$, where x is the secret input of party P_1 and y is the secret input of party P_2. We assume that the function is given as a logical circuit. Every gate has two input wires w_i and w_j, and one output wire w_t. The output wire is $w_t = G(w_i, w_j)$, where G represents the gate. The sender will now choose two random keys, k_i^0 and k_i^1, per wire. Those keys correspond to the possible input values. Since there are two input wires with two possible values, there are four possible input pairs. For every gate, a table is created where the output is encrypted with the input keys. Below we can see a generic table for one gate:

$$T_G = \begin{pmatrix} Enc_{k_i^0, k_j^0}(k_t^{G(0,0)}) \\ Enc_{k_i^0, k_j^1}(k_t^{G(0,1)}) \\ Enc_{k_i^1, k_j^0}(k_t^{G(1,0)}) \\ Enc_{k_i^1, k_j^1}(k_t^{G(1,1)}) \end{pmatrix}$$

Every table gets permuted randomly. Such a table is also called garbled table. We need more tables for the final output. Otherwise P_2 will not be able to read the output in plaintext. This decoding table only has two rows and looks as follows:

$$T_0 = \begin{pmatrix} Enc_(k_0^{\emptyset}, \emptyset) \\ Enc(k_0^1, 1) \end{pmatrix}$$

This table has to be created for every output wire. All the tables, including the decoding table, are then sent to the evaluator together with the input keys of P_1. Hence the receiver has precisely one key per gate. The other key, which represents his input, will be obtained by doing a $\binom{2}{1}$ -OT. For an input of length l, the OT has to be done l times. Finally, the receiver will obtain the result of the protocol by evaluating the circuit and decoding the last output.

This paragraph is according to lecture notes of the lecture cryptographic protocols at University of Bern by Cachin [5, Section 3.1]. This protocol is complete since it calculates the given function. Additionally, the result will be the same if the same decisions were made as before. The security for the sender is given through the security of the $\binom{2}{1}$ -OT. Therefore, the receiver is not able to conclude anything about the other values on the not chosen input wires. The only conclusion possible is from the calculated function and its input $f(\bullet, y)$. Security for the receiver is also given thanks to the Oblivious Transfer. Due to the fact that only OTs are used, the sender will not be able to gain any information about the choice bit. Yao's Garbled Circuit protocol needs $|y|$ public key operations and hence lies in $\mathcal{O}(|y|)$. There are $|C| \cdot \lambda$ bits sent for communication, where $|C|$ is the size of the circuit and λ is the security parameter. Thus Yao's Garbled Circuit protocol lies in $\mathcal{O}(|C| \cdot \lambda)$. Lastly, the number of rounds and hence the latency is constant in Yao's Garbled Circuit protocol because only three rounds are needed to send messages.

2.3.5 Ben-Or, Goldwasser and Wigderson protocol

The Ben-Or, Goldwasser and Wigderson protocol, or short BGW protocol, is another important protocol that is a core part of MPyC. This is why we would like to go into more detail about it. This section is according to the lectures notes of the lecture cryptographic protocols at University of Bern by Cachin [5, Section 3.3] [3, Section 3.3].

Definition 2.3.3 (BGW protocol [5],[3])**.** *An important part of the basis of the BGW protocol is secret sharing. We assume that there are secure channels between all parties and a secure broadcast channel. Unlike Yao's Garbled Circuit protocol, the BGW protocol computes $y = f(x_1, x_2, ..., x_n)$ with an arithmetic circuit. Therefore, the protocol uses addition and multiplication gates. Let there be n parties $P_1, P_2, ..., P_n$ of which f may be corrupted. We assume they are semi-honest. Furthermore, every party P_i has an input x_i, where $x_i \in GF(q)$ where $GF(q)$ is a finite field of order q. Then a random polynomial a is chosen by a dealer. The polynomial has to fulfill the following conditions:*

$$a(0) = x_i$$
$$a(y) = \sum_{i=0}^{f} a_i y^i$$
a has at most degree f

So, a share $x_{ij} = a_i(j)$. Every party P_i secret-shares its secret. Since $x_i = w_i$ where w_i is the i-th input wire, every party holds a share of w_i. The parties hold shares of w_i, which is denoted as $[w_i]$. It furthermore follows that: $x_{ij} = a_i(j) = w_{ij}$. As a next step, the gates have to be evaluated.

Addition gate

$$\begin{aligned}
[w_j] + [w_k] &= (w_{j1}, ...w_{jn}) + (w_{k1}, ..., w_{kn}) \\
&= (a_j(1), ..., a_j(n)) + (a_k(1), ..., a_k(n)) \\
&= (a_j(1) + a_k(1), ..., a_j(n) + a_k(n)) \\
&= (w_{j1} + w_{k1}, ..., w_{jn} + w_{kn}) \\
&= [w_j + w_k]
\end{aligned}$$

Hence addition can be done locally.

Multiplication-by-a-constant gate *This is similar to the addition gate.*

$$\begin{aligned}\alpha \cdot [w_j] &= \alpha \cdot (w_{j1}, ..., w_{jn}) \\ &= (\alpha \cdot w_{j1}, ..., \alpha \cdot w_{jn}) \\ &= [\alpha \cdot w_j]\end{aligned}$$

Hence multiplication by a constant can also be done locally.

Multiplication gate *The party P_i locally computes $h_{ti} = w_{ji} \cdot w_{k_i}$. By computing this for every $i \in \{1, ..., n\}$ the parties get $h_{t1}, ..., h_{tn}$. Those are the points of a polynomial $b_t()$. This polynomial can have a degree of $2f$, so a degree reduction is needed. It can be concluded that*

$$b_t(i) = h_{ti} = w_{ji} \cdot w_{ki} = a_j(i) \cdot a_k(i).$$

Hence it also follows that for $i = 0$,

$$b_t(0) = a_j(0) \cdot a_k(0) = w_j \cdot w_k = w_t.$$

Degree reduction *We assume that $n > 2f$. So let $n = 2f + 1$. In order to reduce the degree an interactive protocol among $P_1, ..., P_n$ is needed.*

We know that Lagrange coefficients exist such that

$$w_t = b_t(0) = \sum_{i=1}^{2f+1} \lambda b_t(i) = \sum_{i=1}^{2f+1} \lambda w_{ji} \cdot w_{ki} = \sum_{i=1}^{2f+1} \lambda h_{ti}$$

To evaluate this expression yet another "shared" computation protocol is used. First every party P_i shares h_{ti} and gets $(h_{ti1}, ..., h_{tin})$. Hence the parties hold $[h_{t1}], ..., [h_{tn}]$. Therefore, we can now do linear computation as follows:

$$[w_t] = \sum_{i=1}^{2f+1} \lambda [h_{ti}] = [\sum_{i=1}^{2f+1} \lambda h_{ti}]$$

This is again a $(f + 1)$ of n sharing because $(h_{ti1}, ..., h_{tin})$ is already an $(f + 1)$ of n sharing. Since for every output wire 0 the parties hold $[w_0]$, every party has to broadcast its w_{0i}. This will lead to party P_i receiving at least $f + 1$ shares of w_0 to reconstruct w_0.

In the BGW protocol, no public-key operations are required. Furthermore, $|C| \cdot n^2$ bits are needed for communication, where $|C|$ is the size of the circuit and n is the number of parties. Thus it lies in $\mathcal{O}(|C| \cdot n^2)$. The BGW needs the depth of the circuit for many communication rounds. It, therefore, lies in $\mathcal{O}(depth(C))$.

Chapter 3

Design

3.1 Existing solutions

In this thesis, our goal is to run Multiparty Computation on top of a blockchain. Since there are some existing solutions to this problem, we start by comparing two approaches. One solution comes from a team of IBM researchers and the other is from Partisia Blockchain Foundation [1] [6]. The main difference between these two solutions is that IBM's solution runs the Multiparty Computation on-chain rather than off-chain.

MPC on Hyperledger Fabric Let's start with the solution which was designed by a team of IBM researchers in 2018. This paragraph is according to information from their conference paper [1]. Their approach uses Hyperledger Fabric as their blockchain. This has the advantage that Hyperledger Fabric's facilities can be utilized for secure Multiparty Computation. Hyperledger Fabric already has a bit of privacy from non-member peers by having channels. Every channel can be seen as a separate ledger, and only members of that channel can see the data on it. Nevertheless, this privacy is insufficient.

The components are the client and the Hyperledger Fabric as the blockchain, where the private data of involved parties is stored with symmetric key encryption. Moreover, there are peers, which are the participants. The participants are nodes that have access to the ledger. The ledger is one of the facilities that MPC uses. Then there is a helper server, and smart contracts are used. Those are programmable logic that is implemented in Go. Smart contracts and chaincode are often used synonymously in the context of Hyperledger Fabric. Smart contracts are invoked for every transaction and they are executed by peers.

There are two phases in their approach. The first phase starts with a transaction proposal. This proposal is sent by the client to one or multiple peers. Those peers are then requested to execute and endorse the received proposal. The proposal must be the same for all the peers that received it. If this is not the case, the proposal will be rejected in the second phase. The proposal triggers a smart contract to be executed by the peers that received the proposal. The chaincode has access to the current ledger and the details of the new transaction. That information is used in the smart contract to decide whether the transaction should be authorized and, if so, what would change on the ledger. It is during this phase that the Multiparty Computation is executed. According to the IBM team[1], "once sufficiently many endorsements are obtained, the client sends the endorsed transaction to an ordering service that imposes a linear order on the transaction and then actually adds them to the ledger." At the initialization of the ledger, it is decided how many endorsements are required for a transaction. This number is part of the endorsement policy.

The way Hyperledger Fabric is built, every peer has access to its copy of the ledger and instance of the chaincode. On the ledger, the secrets are stored in encrypted form. Smart contracts are needed so the peers can communicate with each other. For the communication to work, the chaincode in one peer communicates with its other instance in other peers via the helper server.

The helper server has two main responsibilities. First, it helps facilitate communication. And second, it stores the local parameters as well as the secrets of every peer. This makes the solution somewhat

insecure. It was decided by the team to still use the server in order not to change the architecture of Hyperledger Fabric.

In this approach, the execution of the secure MPC protocol is incorporated into the smart contract. Often private data is needed as input for the MPC protocol. For the smart contract to receive that data, the party with the corresponding key decrypts the data and uses it as input in decrypted form. The team is convinced that it is an advantage to have the same peers as MPC nodes. They argue that they can more easily align the trust models since the nodes are the same and hence the system becomes more secure and easier to manage. Another advantage in their eyes is that by running the MPC on-chain, the facilities for identity management and communication can be reused since Hyperledger Fabric already provides them.

In their demo, they simulate an auction with three parties. The highest bid wins, given that it is bigger than the reserve price the seller asks for. First, a seller uploads an advertisement of the item to be sold. The information recorded on the ledger consists of category, description, start price, as well as an encrypted reserve price that only the seller can see. All that information is sent to the chaincode. Afterward, two buyers see the advertisement and start bidding. Each bid is secret and therefore encrypted. It is also sent to the chaincode. Now the owner of the item starts the auction. A peer of each party is contacted, and the transaction is endorsed. As mentioned before, this is also the phase where the Multiparty Computation takes place. In their solution, the IBM researchers decided to use Yao's Garbled Circuit protocol between the two bidders to find out whose bid is higher. Here they solve the Millionaires Problem with bids. Then two 1-out-of-2 OTs are run to compute the actual number of the highest bid. Lastly, one of the bidders shares its shares with the seller, while the seller and the other bidder go through another Yao's Garbled Circuit protocol. Their goal is to determine, whether the reserve price was smaller than the maximum bid. To conclude, the second bidder sends its shares to the seller and the first bidder, who can then calculate the winning bid and the winner.

MPC on blockchain by Partisia Blockchain Foundation The second solution was developed by a team from Partisia and Sepior [6]. It is currently run by the Partisia Blockchain Foundation. As mentioned before, this approach runs the Multiparty Computation off-chain. The actors and components are the clients, the totally ordered broadcast channel, the smart contracts, Partisia's blockchain and the Multiparty Computation Nodes Pool. This pool consists of all the nodes, which can run an MPC. It is important to know that the TOB, short for Totally Ordered Broadcast, contains the events. Those events are contained in blocks, which have a unique sequence. Every block gets proposed by a sequencer. Those new blocks are signed off by validators. A block is final if more than 2/3 of the validators signed off on it.

In Figure 3.1, we can see the simplified process. First, the buyer posts a Buy event on the totally ordered broadcast. A Buy event indicates that the client would like to run an MPC on Partisia's blockchain. Then the nodes are selected from the MPC Nodes Pool, and a smaller group called MPC Group is created. A smart contract will be orchestrating everything from collecting input, which the MPC Group will get, to the computation, to receiving the result and informing the client about the result.

Let us now go into more details on what happens when a buyer wants to run MPC using Partisia's system. First, clients can create Buy events and post them on the totally ordered broadcast. This event contains all the information about the set-up such as how many nodes the client needs, as well as which kind of nodes, which protocol, all the payment information, etcetera. At this point, no input has been communicated. For the event to be performed, a block has to be produced.

A block holds a stack of transactions and events. Transactions can either call a method to create a smart contract, interact with a contract, or remove a contract. After the production of a block, the events and transactions being executed, the result is stored as the current state. Then the validator committee will validate the block according to the consensus protocol.

So, the transaction, in the beginning, is likely to trigger the creation of a smart contract, which is then deployed. Partisia uses smart contracts to organize the control flow. Furthermore, smart contracts can also generate events on the TOB.

At this time, a protocol is being run, which randomly selects the requested number of nodes from the MPC Node Pool. This creates the MPC Group, which will perform the MPC as well as receive the secret

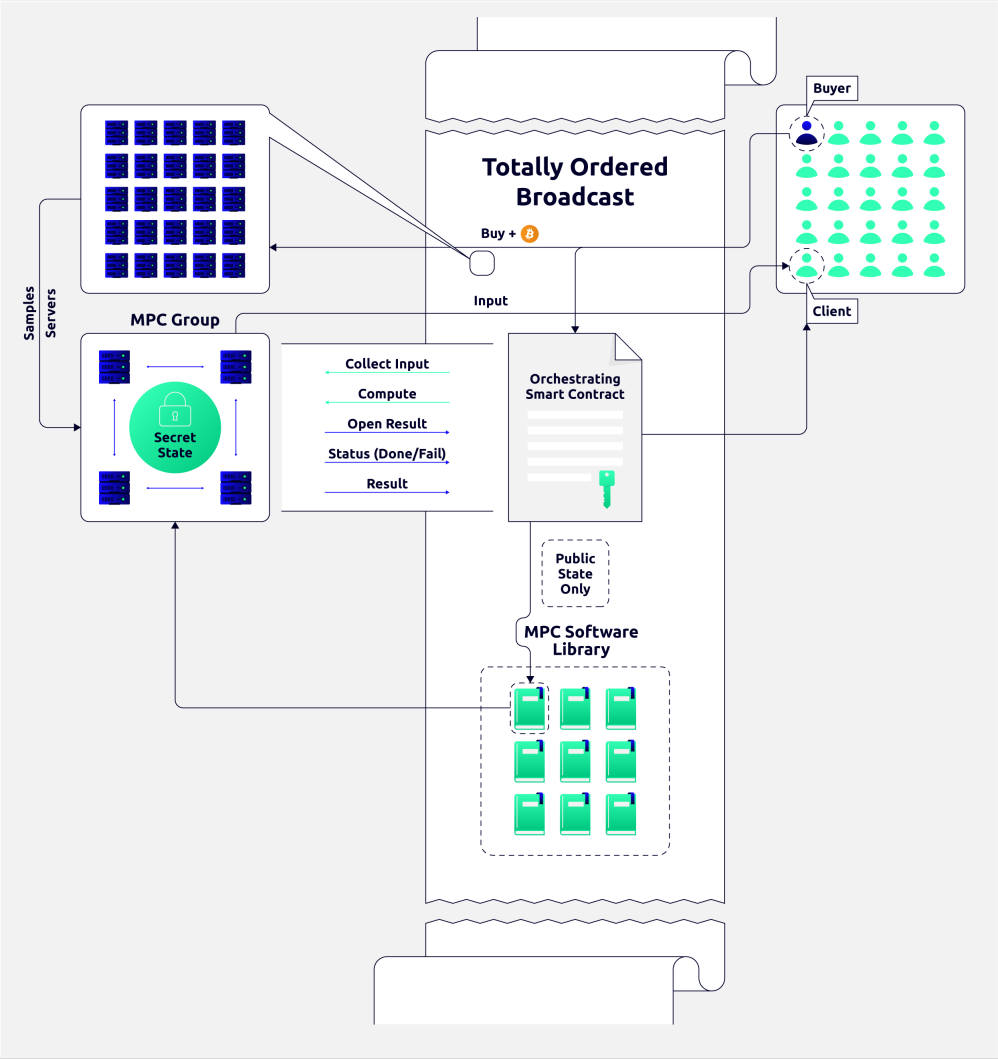

Figure 3.1. Involved parties and process of Partisia Blockchain Foundation solution [6]

input. The MPC Group also listens to the events on the TOB, which were generated by the smart contracts. During this phase, they learn where every node of the MPC Group is located. They communicate with each other using events on the Totally Ordered Broadcast.

Partisia uses the Beaver trick to do the MPC [7]. To get a faster computation, some pre-computation is needed. This pre-computation creates the Beaver triple and is done by the MPC Group. Once enough material is generated, the computation starts. While the MPC has started, more material is still generated by a so-called Fuelling Group. While this group is bigger than the MPC Group, it still includes the MPC Group. By already running the computation but simultaneously having other nodes with minimal involvement from the MPC Group generate more material, Partisia is able to improve its performance.

As explained before, secret sharing is an important primitive of Multiparty Computation, hence it is not surprising that Partisia uses it to get the secret input to the MPC Group before running the MPC protocol. Partisia expects the performance of computers to increase and finally be able to decrypt encrypted secrets, which are stored on the TOB. Therefore, letting the client encrypt their secret input share for every node with the node's public key and post the ciphertext on the TOB is not a solution. Partisia doesn't want to keep secrets for long on the blockchain. The solution to this weakness is using a random value R, which is unknown to the MPC Group. So every node encrypts its share of R with the client's public key. This encrypted share is then sent to the client, who can decrypt it with its secret key. Once the client gets all the shares, it receives R with which it then calculates $D = V - R$, where V is the secret input of the client. D is then posted on the TOB.

The MPC Group then runs the protocol, and when it is done, the nodes post on the TOB that the computation is done. Once a certain number of reports are posted on the TOB, the event is considered done. The result is stored in secret variables which can be opened to the client or on the TOB. For the client to get the result, the same process happens as at the beginning with the random R, but this time the client receives the output.

Insufficient reports might be posted on the TOB on time, and a timeout occurs. This means that the computation failed. All secret variables are deleted, so no information is leaked. In some cases, the failed computation might trigger a new computation with a longer timeout and/or a more robust protocol. This way, the Partisia Blockchain can have a high performance, and in case the computation fails, it becomes more robust and/or slower.

Another failure occurs when the output has failed. There is a fairness problem since some might have seen the output while others didn't. This could end with some parties having an advantage. Partisia Blockchain solves this problem by using more servers and specialized protocols during output in order to improve robustness [6].

3.2 Our solution

First, we would like to explain three similar designs. All designs are about how many secret inputs are given to a node and where the secret sharing of these inputs occurs.

In the first design, each participant in the MPC computation has a node. Therefore, the number of nodes equals the number of participants in the MPC computation. In the second design, multiple participants can send their input to one node. This node would then secret-share their inputs with the other nodes. This design is less restrictive and consequently more realistic. But it holds the challenge to redistribute the responsibilities between the ABCI application and the MPyC program. This challenge will be described in Chapter 4.6. In the last design, a client secret-shares its secret input. Hence the advantage of this design is that the client can be sure that no node knows its secret. This design also presents similar challenges as the second one. The motivation for this last design is slightly different compared to the other two. The difference is that in the third design the focus lies more on the privacy of the secret from any node. Whereas in the other two designs, the motivation is more about the number of secrets that can be given to a node.

In this thesis, we have chosen the first design, which we would now like to explain. Our idea is to call MPyC within ABCI. Let us first give a quick overview using Figure 3.2 about how a client can run an MPyC program on Tendermint.

First, the client and all the other participants are required to start *mpc.py*, which is our ABCI application. As a next step, the client starts Tendermint. Given Tendermint has been run before on this particular machine and the client would like to start with a new blockchain, the command *tendermint unsafe_reset_all* should be used. At this point, the client should also delete all the *appstate.json* from the nodes, followed by the command *tendermint init*. In both cases, whether Tendermint was paused or started a new chain, the client can now start Tendermint with the command *tendermint start*. The client should, at this point, check if the nodes are running by looking at the Tendermint console and, in case of problems, troubleshoot them. Given that everything works, one of the parties can start the computation by calling the *rpc.py* file if no input is needed. Otherwise, every participant calls the *rpc.py* file and enters the following *input:/yourInput*. We recommend that all parties send their input before the computation starts. We will now go into further details on the different steps.

The command *tendermint unsafe_reset_all* should only be used if the old chain can be deleted, otherwise, all the previous information is lost. In case the client calls *tendermint unsafe_reset_all*, he also has to delete the *appstate.json* from every node, else the *mpc.py* will not run. If Tendermint starts for the first time or after deleting the old blockchain, the next command for the Tendermint is *tendermint init*. In newer versions of Tendermint, the client has to choose whether it wants to initialize a validator node, a peer node, or a full node. No matter if it is a new chain or not, the client can start Tendermint with the command *tendermint start*.

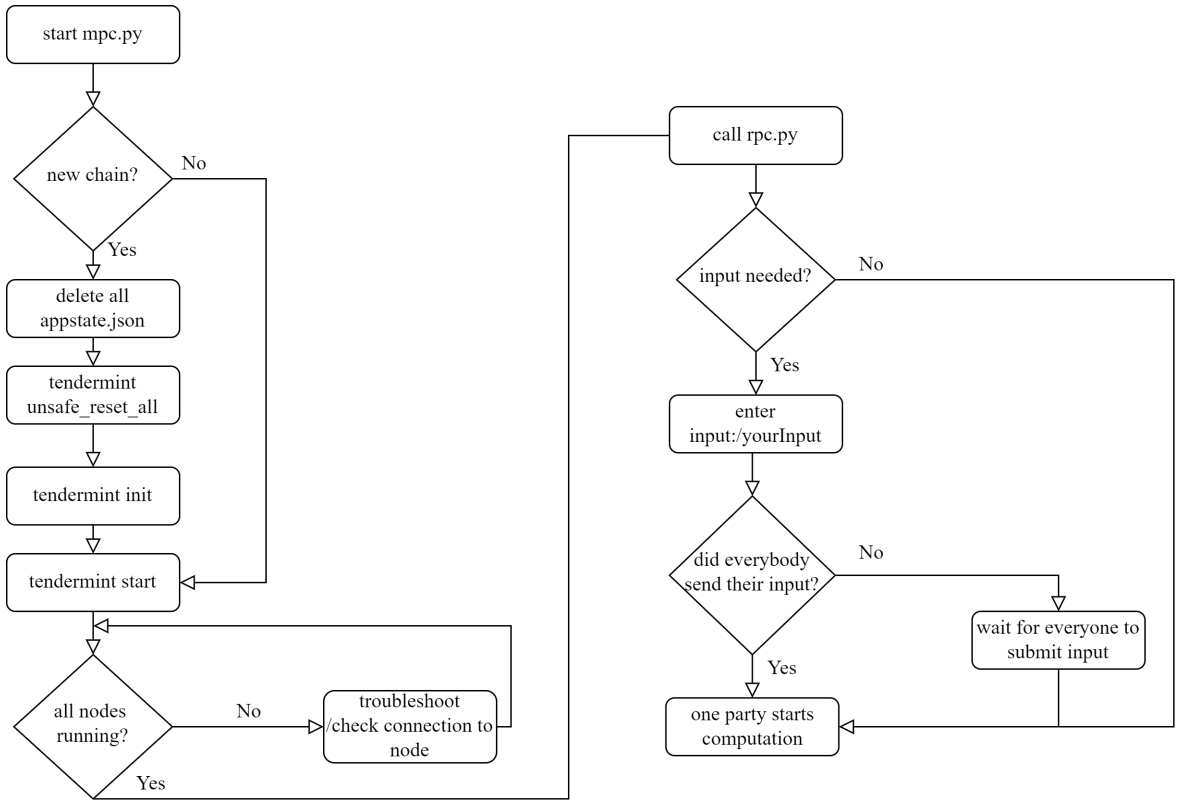

Figure 3.2. These are the condensed steps a client has to go through to run an MPyC program on Tendermint.

Provided the blockchain was started anew, and the validator nodes are not added to the *config.toml*, the client must provide Tendermint with the addresses of all the other validators. The client can choose to add `nodeID@ip:port` to the flag `--p2p.persistent_peers` [24]. Other useful flags are `--consensus.create_empty_blocks=false` as well as `--log_level="debug"`. As the names already imply, the first flag stops Tendermint from creating empty blocks, whereas the second flag logs more information in the console. This is particularly helpful when looking for bugs or expanding one's knowledge of how Tendermint works.

As a next step, we recommend checking some nodes whether they started correctly. Usually, looking at the consoles where Tendermint runs is enough. This is because Tendermint informs the developer about the nodes it has a problem connecting to or on which it has stopped. A problem could be that a node is a virtual machine, and a connection could not be established. In that case, we recommend checking the CPU usage and, in the worst-case, rebooting it. Other problems could be that *mpc.py* is still running in a different thread, causing the current *mpc.py* to shut down since *mpc.py* can only run once per node.

The next step for the client is to run the *rpc.py* file. The *rpc.py* file is the end-user application and handles given input. This will let the client know how to enter different kinds of inputs. In our design, we pass the secret input by calling the `query()` method. The reason for sending the secret input using queries is that the information is not broadcasted and stays local. So before a client can start a Multiparty Computation with inputs, every party has to send a query to their node. In order to send a query the client can use a curl command, however, the safer alternative is running the *rpc.py* file. To let the script know that the client wants to give input, the following information *input:/the_clients_secret_input* in the given format has to be entered. It will ensure that the input is in the correct format, hence the app will not crash.

Once every participant sent their secret input to their node, it is enough for one client to start the computation. It is the client's responsibility to know when all the inputs are given. If the computation is started earlier, it is up to the MPyC program whether an error message is returned or some assumptions about the input are made. For example, the demo file *unanimous.py* of MPyC assumes that no input is

the same as sending a 1 and hence agreeing. Again, the client should call the *rpc.py* file, but this time entering *program/number_of_participants/ip1/ip2/...../ipn*. Only one party has to start the computation. Once the computation finishes or as soon as an error occurs, the response of Tendermint is printed in the console.

The response of Tendermint is a JSON dictionary. Given the computation completed within a reasonable time, the returned JSON dictionary has sub-dictionaries called `check_tx` and `deliver_tx`. In case of a valid transaction and successful computation, the keys "code" in `check_tx` and `deliver_tx` are both 0. Furthermore, the dictionary `deliver_tx` has another sub-dictionary called "events", which contains the result of the computation under the key "value". The result is base64 encoded and has to be decoded to be human-readable. Lastly, if the execution is successful, the height at the end of the JSON dictionary is not 0. In case of a failed check in the ABCI application, the code of `check_tx` or `deliver_tx` is 1, respectively, to where the failure occurred. This process requires having a working consensus.

Besides, it is theoretically possible to run a Multiparty Computation locally on only one machine. To do so, the client only enters *program/number_of_participants* when running the *rpc.py* file. Technically, the client could also send the transaction via a curl command. The advantage is that the client can choose between sending it with `broadcast_tx_async`, `broadcast_tx_sync` or `broadcast_tx_commit`. The `broadcast_tx_commit` waits for the answer of the `check_tx()` and `deliver_tx()` method before returning. While the `broadcast_tx_sync` waits only for the `check_tx()` to return, the `broadcast_tx_async` returns without waiting for any method. The disadvantages are the same as when sending the input via curl; fewer checks and, therefore, less safe. Furthermore, requests are sent by HTTP to Tendermint by default. We changed this to HTTPS to have slightly more security.

For everything to work, consensus and the termination of consensus are particularly important in our design. If we had not any consensus protocol, no decentralization would be possible. The reason for this is that some mechanism is needed to decide when a valid block is added to the blockchain. Without a consensus protocol, another mechanism to determine whether the block can be added would be needed. This would result in requiring a centralized instance, which is not the goal of this thesis. Furthermore, the consensus is needed so the servers can agree on a set of valid transactions and add a block to the blockchain. Moreover, since Tendermint is our replicated state machine, it is crucial to have consensus because in Tendermint the consensus guarantees us that all the transactions are received in the same order on every replicated state machine. Therefore, the consensus is also vital so all servers keep the same state. If consensus does not terminate, our solution is unable to process any other transactions. Reasons for this could be that when the nodes don't agree because some field in the request differs from the expected field, it is impossible to find consensus. In that case, the nodes would be stuck and the execution of transactions would come to a halt. Another reason why consensus would not terminate is that Tendermint is stuck calling an ABCI method multiple times and is therefore blocked from continuing. For example, consensus failures in Tendermint can be forced by setting the cache size to 0 in more than 2/3 of the nodes. This results in the nodes going multiple times through `check_tx()` and thus not committing the second transaction properly. Hence the propose step fails because the proposal block is invalid due to the `Block.Header.LastResultsHash` being incorrect. Once consensus fails, it is impossible to execute blocks nor do Multiparty Computation on the blockchain. Finally, if the ABCI application does not terminate, consensus can not be reached anymore since the nodes are still occupied with the execution of the application. Meaning, the nodes are blocked infinitely and can not move on to the next block. So if we don't have a consensus, the whole system can not be used and has to be reset.

If we compare our solution to the solutions of IBM and Partisia Blockchain Foundation, we find some major differences as well as some similarities. Firstly, similar to IBM's approach we have chosen to use an existing blockchain. Although we don't use Hyperledger Fabric but Tendermint. Additionally, in our solution the MPC is computed on-chain. Furthermore, comparable to IBM's approach, our blockchain is permissioned too. The nodes that start MPyC will also be involved in running the program, and it is MPyC's responsibility to orchestrate communication during the execution of the program. This could be compared to the smart contracts, which are used for communication in IBM's approach. Moreover,

during the program's execution, the nodes can not be used otherwise and are blocked. Besides, like IBM's solution, our solution also needs the number of nodes to match the number of participants of the MPC. In our solution, there is no need to let the nodes know where to find each other given the network already exists. This is unlike both existing solutions. Like Partisia's, our solution has a separate consensus and network layer, but ours is provided by Tendermint. Our solution requires one node to see the secret input of the client but will not share the input. This is an aspect that Partisia has solved more elegantly, while Hyperledger has an even less privacy-preserving solution by using the helper server. Unlike in Partisia's approach, where the delivery time to the TOB is assumed to be finite but not measured, the delivery may not take more than 10 seconds in our solution. And the execution of the transaction may not take more than 7 seconds. Another difference to Partisia is that our solution does not need any preprocessing. However, our solution, like Partisia's approach, can run multiple different protocols, while IBM's approach only mentioned two. The only requirement for protocols is that the programs are written in Python using MPyC and that they are saved where the demos currently are. Otherwise, the path needs to be adjusted. Similar to the solution with Hyperledger Fabric, we currently neither use any gas nor forms of payment. Another difference to both approaches is that our result is written on the blockchain, whereas Partisia's could but does not have to. In the Hyperledger Fabric's approach, it is the responsibility of the nodes to inform each other of the result. Lastly, similar to Partisia's approach, more than three nodes can create a network. So far, we use up to 10 nodes, but we believe it is more scalable.

Chapter 4

Implementation

4.1 Software

Tendermint has many dependencies, hence, a lot of software had to be downloaded. First, we installed Go version 1.17.1 and downloaded Tendermint version 0.34.11. Once Tendermint was installed, and all PATH variables were set, ABCI was downloaded and installed. We used version 0.17.0 in our implementation. The next step was to download MPyC version 0.7.7 as well as install Python 3.9. For MPyC to run faster, we also installed the gmpy2 package. Ultimately, we downloaded py-abci, which was the limiting factor here. The reason for this is that py-abci needs Tendermint version 0.34.11, which is only a couple of months older than the newest version. Since MPyC is written in Python, the assumption was made that the ABCI server has to be as well. Hence py-abci was downloaded to work as the ABCI server. It later turned out that a Go server would also have worked. Visual Studio Code was used as an IDE with the following extensions: the ”Remote-SSH” and ”Remote-SSH: Editing Configuration” extension from Microsoft. Furthermore, Python extensions like Python for VSCode, Python and Pylance were installed. Lastly, an SSH key pair was generated to log into the virtual machines.

The implementation of this thesis was done using 4 DigitalOcean Virtual Machines, each with Ubuntu 20.04.3 Focal Fossa. Each Virtual Machine has 1GB of Memory, 25 GB of Disk and is located in Frankfurt, Germany. The following IPs and ports were used to listen for the peer-to-peer communication.

4.2 MPyC

As mentioned above, MPyC was used in this thesis [10]. MPyC is an open-source Python package that allows developers to create MPC protocols and programs. It provides secure types such as secure integers, floats, fixpoints, numbers, objects, lists and many more. Many logical, arithmetical and conditional statements, as well as known Python methods like `all()`, `any()`, `sum()`, `min()`, etcetera, are implemented securely in MPyC. MPyC lets the involved parties communicate via peer-to-peer connections. Protocols that are written using MPyC run uncorrupted as long as up to half of the participants are malicious. This is a weaker restriction than what Tendermint needs to reach consensus, where more than 2/3 of the nodes have to be honest. Furthermore, those protocols are based on Shamir’s threshold secret sharing scheme. One of the core protocols used in MPyC is the BGW protocol. Most cryptographic protocols used in MPyC are hidden by overloaded Python methods. There are about 17 different demos in the MPyC package, which is available on GitHub [10].

The Oblivious Transfer is one of the demos. In this thesis, we changed another existing demo called *unanimous.py*. Our goal was to create a protocol that is closer to the real world. So we implemented a voting protocol where participants could vote yes or no, which is called *voteyn.py*. We created a second protocol where one could vote for people. A use case could be when there are elections for the parliament. According to their documentation, MPyC can be called using a console command. The methods `start()` as well as `shutdown()` are used in every MPyC program to start the MPyC runtime respectively to shut it

down. Commonly used functions are `input()`, which secret-shares the input provided by the command. To receive the output of the secret computation, `output()` has to be used. Another important method is `run()`. Without it, the coroutines are not run, and hence it would not be possible to receive the output, for example.

4.3 Adding validators to create a network

We created a network where all four virtual machines are validators. We are aware that other nodes like sentry nodes would also be needed to build a complete network. However, since our blockchain is permissioned and the focus of this thesis lies on the feasibility, we did not add any other types of nodes. It has to be mentioned that all four VMs had the same installation of Tendermint and the same software installed. To create a network, we added validators. First, a public-private key pair has to be created, where the private key will be the signing key for the consensus. In Tendermint, such a pair can be generated by calling *tendermint gen_validator* or *tendermint gen-validator* for newer versions of Tendermint. This would return a key pair similar to this:

```
{"Key":
        {"address":"53D1772BB97BC23456247753A2CB0D8208F19E52",
        "pub_key":{
                "type":"tendermint/PubKeyEd25519",
                "value":"stL7bD+Ilrp4/WMNwBRd1ZtYvaoyGJ7xAUoncHpoimw
                   ="
        },
        "priv_key":{
                "type":"tendermint/PrivKeyEd25519",
                "value":"0Bv9uGoT8dSv0xuLamoSk6vK1rtlA/sxBV7arXzm1xC"
        }
        },
"LastSignState":{
        "height":"0",
        "round":0,
        "step":0
        }
}
```

It is similar because for the sake of readability we shortened the value of the private key. Everything from the curly brace before the address to the second curly brace after the private key will then be copied into a file called *priv_validator_key.json*. Another file called *node_key.json* should only have the private key. At last, in the *genesis.json* file, the validators list should be extended with information such as the address, the public key, and the voting power of the other nodes. In the above case, one would add the part from the bracket before the address to the first bracket after *pub_key* value and then add `"power": "10", "name": ""}`. The name and power of 10 is just a simple example and can be chosen by the developer. Further, in the *config.toml* file, the external address should be set to the IP address followed by `:port`. At this point, the nodes will not connect automatically to the peers given in the *genesis.json* file. There are two options to let Tendermint know how to connect to those peers. We tried both of them. The first option uses the persistent peers flag `--p2p.persistent_peers` followed by a comma-separated list of their `nodeID@ip:port`. This option was mentioned prior but is mentioned again for the sake of completeness. This needs to be repeated whenever Tendermint is started. The other longer-lasting option is to add the peers in the *config.toml* under p2p configuration options `persistent_peers`. There again, a comma-separated list of their `nodeID@ip:port` is needed. In both cases we would add: `53d1772bb97bc23456247753a2cb0d8208f19e52@165.22.26.96:26656`

When adding the validator to the persistent-peers list, it is essential to change the uppercase letters to lowercase ones. Otherwise, when running Tendermint, it will report that an error occurred when dialing the peer. The message will say that the `nodeIDs` mismatch.

4.4 ABCI methods

The following methods are implemented in the *mpc.py* file.

4.4.1 check_tx()

Since `check_tx()` receives the transaction as bytes, we split the transaction into the different parts using the forward slash. Then the timestamp is checked, whether it has the correct format. If it does not, an error is returned, else the timestamp is removed. Afterward, it is checked that the program received enough information. The given input has to be longer than just one word. Again, an error saying not enough data was provided will be returned otherwise. As the next step, the program and the number of parties get validated. It will then differ between running the MPyC program locally or with other nodes. If the computation runs locally and the length of the input is two, and the first two parts are valid, a code zero is returned. In case the Multiparty Computation has to be run with different participants, our app collects all validator's IPs by looking them up from the *adrbook.json* in the *config* folder. Then it is checked that an IP is not given more than once by adding each IP to a list. We use `sorted(set(list),key=index))` to keep the order but at the same time remove any element that existed twice. It is important that every IP only exists once in the list passed to the MPyC program, otherwise, the application will not terminate. The reason for not terminating is that in `deliver_tx()` we call the MPyC program only once per node. If the IP is given more than once, the MPyC expects to be run more than once on that node. Consequently, MPyC is waiting infinitely for the same node to start the MPyC program. Subsequently, the IPs are validated, meaning it is checked that their format conforms to the format of IPs. It also is checked that more than one IP is given. Lastly, it is reviewed that only validator nodes call the MPyC program. If this check would not exist, one could request a random IP to take part in the MPyC program, and with a high possibility, this IP would not have MPyC installed and running with the same commands as the rest of the Tendermint network. Given the IPs are valid and this node is partaking in the Multiparty Computation, a code zero is returned. In case one of the checks fails, an error code is returned.

4.4.2 deliver_tx()

`Deliver_tx()` uses the same helper methods as `check_tx()`. The reason for this is that `check_tx()` is optional and `deliver_tx()` is not. All those checks are to ensure that our application terminates. Furthermore, we added a timer started in a separate thread to ensure that the MPyC program would not take longer than 10 seconds. Ten seconds is the duration of the propose timeout. Reaching this timeout results in receiving an internal error. The response would be that it timed out waiting for the transaction to be included in the block. More on this error can be found in Section 4.6. In both cases, whether we call MPyC locally or with other IPs, this thread is stopped if the MPyC computation is successful. If it were not stopped after a successful computation, it might result in killing a process that did not time out yet. If the called program does not finish within 7 seconds, the computation is killed, hence ensuring that the ABCI application stays deterministic. Additionally, `deliver_tx()` checks if the secret input was saved locally and if so, stores it, so it can be used when calling MPyC. Again, we have two cases. First, if the Multiparty Computation is run locally, it will check if the transaction length is two. If that is the case, call the MPyC locally. If the computation isn't done locally, all the checks on the IPs are done as described in `check_tx()`. If the IP checks return valid, the MPyC program is called with the correct IPs and with input if given. The result of the MPyC is then saved to the response events, which will not influence the consensus. This makes it also possible to subscribe/index for the result. At this time, the secret input is reset. Any information that will not be available later in the `commit()` method, such as the result of the MPyC and the program that was run, is written into the state. It is also checked that if IPs were used,

they are saved. Else, if the program was run locally then participants will be set to an empty list. This prevents locally run MPyC programs from having the same participant IPs as the program run with IPs before. Now Tendermint will also set the flag whether the transaction was valid. This way, the `commit()` method knows it is okay to save the state later on in the program. Finally, a code zero is returned in a JSON dictionary with the result information in the events part of the response. That information will be displayed base64 encoded in the client's terminal.

4.4.3 commit()

In our `commit()` method, we check that the transaction is not in the very first block of the chain and that it is valid by looking at the `validTx` flag. If both conditions are fulfilled, the block height and the transaction number are updated, a transaction is created and saved to the state. Since the `commit()` method does not receive the transaction, no transaction information can be updated here. The `validTx` flag is then reset to false. If the transaction was invalid or if it was the first block, only the block height is increased. The reason for the block height to be increased if the transaction was deemed invalid is that in `deliver_tx()` a transaction is still included in a block but is marked as invalid. Finally, an empty `ResponseCommit` is returned.

4.4.4 query()

The client has two options when sending a query. On the one hand, it can send a query with its secret input for the Multiparty Computation. On the other hand, it can query for past valid transactions, which have certain parameters. If the client decides to send its secret input, it is crucial that its query starts with *input:/*. Otherwise, the `query()` method will not be able to recognize it. The reason for this particular form is that the *rpc.py* chooses the correct command, which will be sent to Tendermint. Hence the correct syntax is of importance. If the input is not an integer, an error will be returned. Else, the input will be saved, and a code zero, as well as the value encoded with base64, are returned. The second case is if the client wants to retrieve some information from the current or past height. It has the option to send params/key/value to Tendermint. Possible keys are *program*, *txnumber*, *result*, *blockheight*, or *participants*. In this method, we check that only those keys are passed. Assuming the client would like to query for more than one participant, it has to enter the IPs separated by a comma. For example, the client would like to query for the program *ot.py* and participants *46.101.233.157* and *139.59.138.151*. Its input in *rpc.py* would then look like *params/program/ot/participants/46.101.233.157,139.59.138.151*. Since every value is separated from the next key with a forward slash, the number of words used for a query can not be an even number, even when querying for multiple participants. For example, *params/program/helloworld/participants /46.101.233.157,161.35.202.162* would be counted as five words and not 6. This is checked before a lookup in the state occurs. It is important to note that if one queries for a result, the whole output should be given, otherwise, our application will not be able to find it. It is hence recommended to query with the other parameters. If the checks are okay, then the method collects all the keys, values and queries the state class. The state class has access to the *appstate.json* file, where the past valid transactions are saved. If the state cannot find a file, an error is returned. Otherwise, the found transactions encoded with base64 are returned. It is acceptable to return empty if no transaction was found.

4.4.5 init_chain()

In the `init_chain()` method, we first load the state and retrieve the node's IP. This is currently done by a console command but could also be done by reading from the *config.toml* file on line 181. Using the command line is less error-prone compared to the second option, where the developer has to set the external address manually in the *config.toml* file on line 181. The node needs to know its IP. On the one side, the IP is needed to compare to the given IP from the client and find out if the IPs run Tendermint with MPyC. On the other hand, we need it to determine the index. That means we need to find out which

number the node is in the MPyC computation. This information has to be passed to the MPyC. After assigning the IP to a global variable, an empty response is returned.

4.4.6 info()

In this method, we try to read the current height. This is done by reading from *appstate.json*. If the file is not found, the `last_block_height` is set to 0. This means Tendermint will call `init_chain()` and start a new blockchain. Otherwise, the state is loaded, and the block height as well as the transaction number are set. Since, in this case, the `init_chain()` method will not be called, the node's IP has to be set here.

4.5 MPyC and Tendermint

To explain how MPyC is run as part of the ABCI application, we will first explain how MPyC is run without Tendermint. MPyC is run by entering *python3 program.py -M numberOfParties -P IP1 -P IP2 -Index (Input)* in the console. This command runs an MPyC program with 2 IPs. More IPs can be added by appending *-P IP* for every additional IP. At least two IPs are required in order not to call MPyC locally. The input is optional, and it depends on the MPyC program, whether it is required or not. The *rpc.py* file requests all the needed input like the program, the number of parties, IPs, and separately input if needed. As mentioned before, the input is validated and if an MPyC computation should be started, the following lines in *rpc.py* are executed:

```
payload={'tx': f"\"{inputNoEndOfLine}/{now}\"" }
result= requests.get('https://localhost:26657/broadcast_tx_commit',
                params=payload, verify=False)
```

`InputNoEndOfLine` refers to the input given from the console without any line breaks and everything in lower cases. `Now` is the current timestamp. The information is sent to Tendermints RPC endpoint `broadcast_tx_commit`, where Tendermint forwards them to the ABCI application. Methods like `check_tx()` are called and after reaching consensus also `deliver_tx()`. In `deliver_tx()` MPyC is called using the subprocess package:

```
result=subprocess.run(f"python3 {program}.py -M{numOfParties}
                {ipAddresses} -I{index} {input}",
                cwd="/usr/local/lib/python3.8/dist-packages/mpyc/
                   demos",
                shell=True,capture_output=True).stdout
```

The result that MPyC prints in the console is saved in the variable `result`, which will be added to the response of Tendermint as explained in 4.4.2.

4.6 Encountered difficulties

Throughout this thesis, different kinds of problems arose. Some of them could not be fixed within the timeframe of this thesis, while others could.

4.6.1 Transaction already exists in mempool cache

In the beginning, we were not able to call an MPyC program twice. We assumed it was an error on our part. After reading the documentation, we realized that Tendermint has a mempool cache. This mempool cache serves as replay protection. It has a default size of 1000 transactions that can be stored. If a transaction is the same as a transaction in the mempool cache, Tendermint does not execute nor broadcast it. The mempool cache receives new transactions, which will then be validated by the ABCI method `check_tx()`. It will also be checked if the transactions are not already contained in the mempool, and if

not, the transactions will be broadcasted [17]. The first attempt was to set the mempool cache size to 0, which can be done under cache in the *config.toml*. This worked only temporarily. Once a network was created, sending transactions resulted in infinite loops of calling `check_tx()`. Therefore, the cache size was reset to 1000. Two possible solutions were taken into consideration to make a transaction unique. The first one was adding a transaction count after every transaction, which would be the client's responsibility. The second solution was to add a unique timestamp automatically after every transaction. It was decided to proceed with this solution since it does not require the client to keep track of the transaction count. Furthermore, a timestamp is less prone to being the same for two transactions since they always differ in milliseconds.

4.6.2 Timed out waiting for transaction to be included in a block

A reoccurring problem was the error response of Tendermint, stating that it timed out waiting for the transaction to be included in a block. It was unclear whether the consensus round was completed and whether the transaction was included in the block. We let a *helloworld.py* transaction run locally, which did not time out, and compared the debug entries in Tendermint to the transaction which timed out. Furthermore, we tried to find more information about the error in the documentation and GitHub issues. We found out that the consensus does complete because the commit step is entered. Furthermore, we realized that calling MPyC with IPs, which don't belong to the network, results in MPyC not being able to complete the computation and waiting infinitely. This affects the consensus since Tendermint requires the ABCI application to terminate. The first solution was to compare the given IPs to the IPs of the validator nodes. The IP addresses of other nodes can be obtained by reading from the address book, which can be found in the *config* folder. If one of the IPs does not match any validator's IP, an error is returned. We then conducted some tests, whether the problem only occurred when sending transactions using `broadcast_tx_commit`. It turned out that only `broadcast_tx_commit` returns the error, while the other two options returned a JSON dictionary more quickly. However, we also found out that even if Tendermint responds quicker, the consensus will still not be reached unless the transaction finishes. That is why we added the second solution a timer, which kills the MPyC program, in case it takes too long to finish. The `timeoutPropose`, initially was set to 3 seconds, was increased to 10 seconds. This gives a transaction up to 10 seconds to complete. The timer, which runs in a separate thread, was set to 7 seconds. After this time, the MPyC program is killed. It is important to stop the timer after a successful MPyC computation. Otherwise, if the same program is computed again, it might be killed from the old unstopped thread. Hence, by calling only the validator's IP, we increase the probability that MPyC finishes quicker. And secondly, by adding the timer, we ensure that MPyC computation is bounded in time. Therefore, the error that a transaction could not be included in a block should not occur anymore.

4.6.3 One node, multiple secret inputs

As mentioned in Section 3.2, there were some difficulties with the second and third designs. We focused more on solving the problems of the second design. There, we assumed that one node could take multiple inputs from different participants of the MPC computation. However, since the third design is similar to the second design, similar problems would have been encountered. In both designs, this would mean that the MPyC program accepts a list of secret shares. Then the MPyC would compute without secret-sharing them again and return the result.

Our first attempt at finding a solution was to use MPyC's logic of the `input()` and `distribute()` method to obtain secret sharing outside the ABCI application. It did not work because by calling the `input()` and `output()` methods of MPyC in one file results in the secrets being displayed in plain-text in an array, which is not secret-shared. The next idea was to not use the `output()` method and to only use the `input()` method to secret-share. In theory, a valid approach but realistically, it returned `<mpyc.sectypes.SecInt10 object at 0x7fd5f328a820>` as a string. This could not be used once the external program stopped. The next approach was to integrate secret sharing into the ABCI application. Here we had some problems as well. First, we weren't able to import MPyC as

a package. This could be solved by installing MPyC for Python3.9 using *python3.9 -m pip install git+https://github.com/lschoe/mpyc*. Other small problems arose when trying to understand the code of the `distribute()` method. However, by slowly adding more and more code from MPyC the problems could be solved. The only exception was that our ABCI application would not await the calculation of the shares. This was realized when trying to send a query request to another node in a separate thread. This caused a *ConnectionError*.

Assuming those two problems could be resolved when everything works together and gets the correct input, we tried calling MPyC with a list of secret shares. This presented us with major difficulties. Firstly, the shares are returned in the form of `<mpyc.sectypes.SecInt10 object at 0x7fd5f328a820>`, even when called in the same file. This can not be passed to a command, expecting it to know how to find this variable. Secondly, when calling *unanimous.py*, for example, which is one of the only demos that takes input, the method `all()` is used. This method reshares the input and leads to a `ConnectionResetError` even when called locally. When we commented that part out and just let the program give the output, it never finished. We waited for 5 minutes and then stopped the computation since MPyC programs as *unanimous.py* usually take a couple of seconds at most to terminate. Furthermore, by having shares as input, it is harder to use methods like `count()`, which looks for a specific number to be counted.

Our last attempt was to call MPyC for every input a node has. Unfortunately, the time we had to work on this thesis was not enough to explore this solution further, but we expect it to work from our experience. Hence, we propose sending the input to one node, storing the secrets as a list, and calling MPyC multiple times. This would then allow multiple clients to send their secrets to one node. Some adjustments would need to be made in the code. One would be to add the own IP to the command as many times as there is an input for that node. Hence one node would run multiple MPyC instances. One disadvantage of the solution would be that it would hold more weight in the computation if the node was malicious. As a result, the security of the transaction might not be given if the node has many secrets proportionally to the other nodes.

Chapter 5

Benchmarks

We will now compare MPyC's runtime with the runtime of an MPyC transaction in our program. We called MPyC separately and as a part of our ABCI application to see the difference in time. We ran different programs. *Helloworld.py* was run with four nodes, whereas *ot.py* and *unanimous.py* were run with three nodes. The reason for this is that *ot.py* and *unanimous.py* should run correctly and therefore need an odd number of participants. Both programs can be called with even nodes, but a message saying they need an odd number of participants is returned. In that case, neither *unanimous.py* nor *ot.py* start a computation. *Unanimous.py* was only run with input 1, leading to a unanimous agreement among the parties. The data for the separate MPyC computation was collected by running the computation multiple times. A script was created not to distort the timing. The script opened and connected to the DigitalOcean VMs and started the MPyC program with the needed information. Then the elapsed time provided by MPyC for each node was used. This was done multiple times, and the average time of the nodes per round was taken. From there again, the average was calculated to preclude distortion. The computation, where Tendermint was involved, was started manually. The duration was calculated before Tendermint was called in the *rpc.py* file and after the JSON dictionary was returned. Therefore, the waiting time of manually starting did not affect the calculated times. In Figure 5.1, we can see that a separate MPyC computation only takes about half to a third as long as an MPyC computation where Tendermint is involved. We expected this since Tendermint needs time to reach a consensus and execute the transaction.

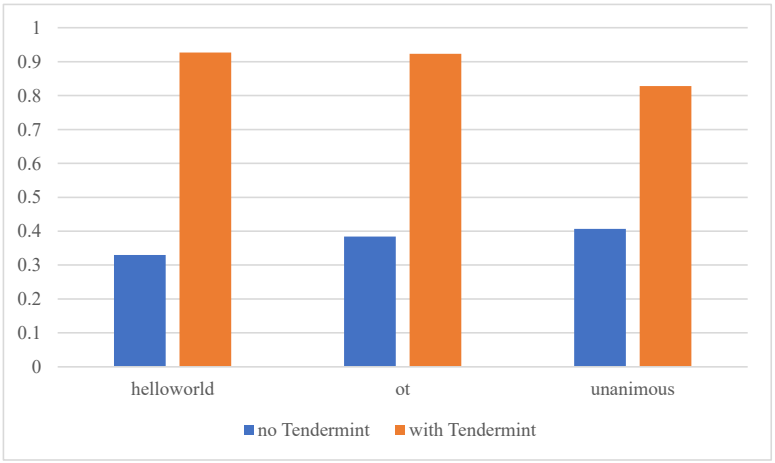

Figure 5.1. Execution time in seconds of MPyC called without Tendermint and with Tendermint

Then the programs *unanimous.py* and *helloworld.py* were run with increasingly more validator nodes in the network. Here again, the execution time in seconds was compared. As a first step, six additional VMs were created. They had the same specification as the first four. The used IPs and ports for peer-to-peer communication can be seen in the table below.

They were iteratively added to the network as described in Subsection 4.3. *Unanimous.py* was chosen as it is one of the programs that takes input. *Helloworld.py* was chosen due to the inability of *unanimous.py* to run properly with an even number of nodes. Hence only odd numbers of nodes can be seen in Figure 5.2 in order not to distort the graph. The even number of nodes would have led to drops in the graph which can be attributed to the fact that *unanimous.py* only returns the message mentioned above. Nevertheless, the needed time to execute *unanimous.py* as part of Tendermint increased with the number of nodes, which was expected.

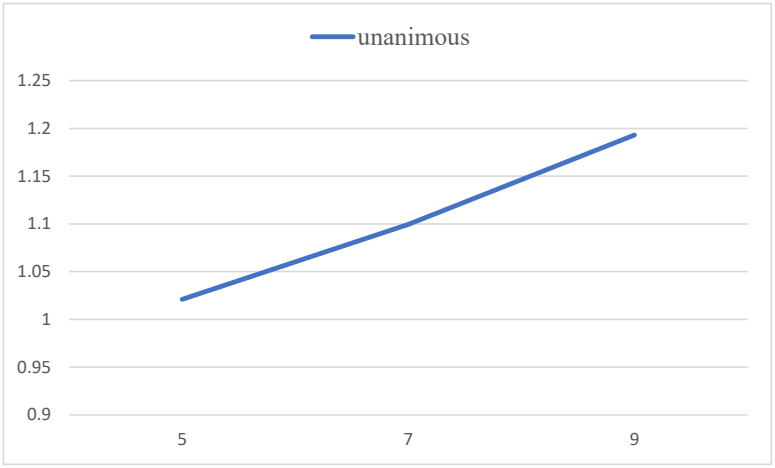

Figure 5.2. Execution times in seconds of *unanimous.py* program run with four up to 10 nodes

Figure 5.3 shows that the increase of nodes results in an almost linear increase in execution time.

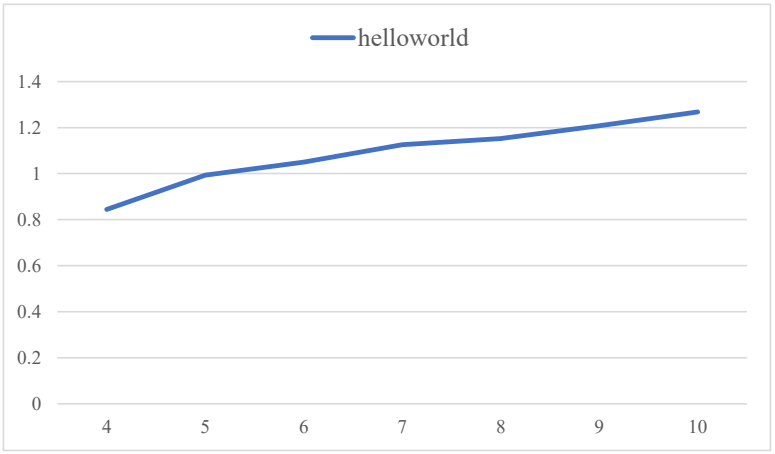

Figure 5.3. Execution times in seconds of *helloworld.py* program run with four up to 10 nodes

Chapter 6

Conclusion

The answer to the question we wanted to solve is, by using Tendermint Core as a replicated state machine and by calling MPyC in the `deliver_tx()` method, it is possible to do MPC on a blockchain. Some limitations of this thesis are currently the strict assumption that only one client can send input to one node. It was found that creating shares calling the methods provided by MPyC works. Unfortunately, it was not possible to pass them to MPyC and change MPyC in the remaining time of this thesis to accept the shared inputs when being called. Another missing aspect is the security of the messages passed from the client to Tendermint to the ABCI application. Even though we use HTTPS, the transactions could still come from a malicious person impersonating a client. Hence, our solution is currently missing message authentication, but it could be added using standard techniques.

Future work could consist of implementing the second less strict assumption that multiple people can use one node. A theoretical idea for such an approach was provided in Section 4.6.3. Furthermore, it might be interesting to try and implement a similar solution as ours with Cosmos SDK. The reason is that Cosmos SDK offers more security, especially for signing messages, making the communication between the client and the ABCI application more secure. It further would not limit us to Tendermint version 0.34 because Tendermint is already a part of Cosmos SDK.

Bibliography

[1] F. Benhamouda, S. Halevi, and T. Halevi, "Supporting Private Data on Hyperledger Fabric with Secure Multiparty Computation," in *2018 IEEE International Conference on Cloud Engineering, IC2E 2018, Orlando, FL, USA, April 17-20, 2018*, pp. 357–363, IEEE Computer Society, 2018.

[2] D. Boneh and V. Shoup, *A graduate course in applied cryptography*. Self-publishing, 2020.

[3] R. Cramer, I. B. Damgård, and J. B. Nielsen, *Secure Multiparty Computation and Secret Sharing*. Cambridge University Press, 2015.

[4] Davebryson, "py-abci/application.py at master · davebryson/py-abci." `https://github.com/davebryson/py-abci/blob/master/src/abci/application.py`.

[5] D. Evans, V. Kolesnikov, and M. Rosulek, "A pragmatic introduction to secure multi-party computation," *Foundations and Trends® in Privacy and Security*, vol. 2, no. 2-3, pp. 1–64, 2017.

[6] J. B. Nielsen, "MPC Techniques Series, Part 10: MPC-as-a-Service — the Partisia Blockchain Infrastructure." `https://medium.com/partisia-blockchain/mpc-techniques-series-part-10-mpc-as-a-service-the-partisia-blockchain-infrastructure-9b4833e77965`.

[7] J. B. Nielsen, "MPC techniques series, part 4: Beaver's trick." `https://medium.com/partisia-blockchain/beavers-trick-e275e79839cc`.

[8] G. Santhosh, "How to write Tendermint Applications using Python, or in any other language." `https://medium.com/coinmonks/how-to-write-tendermint-applications-using-python-d8dde304e339`.

[9] F. B. Schneider, "Implementing fault-tolerant services using the state machine approach: A tutorial," *ACM Computing Surveys (CSUR)*, vol. 22, no. 4, pp. 299–319, 1990.

[10] B. Schoenmakers, "MPyC (version 0.7)." `https://lschoe.github.io/mpyc/`.

[11] N. P. Smart, *Cryptography made simple*. Springer, 2016.

[12] Tendermint Inc., "ABCI." `https://docs.tendermint.com/master/spec/abci/`.

[13] Tendermint Inc., "Application Architecture Guide." `https://docs.tendermint.com/master/app-dev/app-architecture.html`.

[14] Tendermint Inc., "Applications." `https://docs.tendermint.com/master/spec/abci/apps.html`.

[15] Tendermint Inc., "Byzantine Consensus Algorithm." `https://docs.tendermint.com/master/spec/consensus/consensus.html`.

[16] Tendermint Inc., "Data Structures." `https://docs.tendermint.com/master/spec/core/data_structures.html`.

[17] Tendermint Inc., "Mempool Functionality." https://docs.tendermint.com/v0.32/spec/reactors/mempool/functionality.html.

[18] Tendermint Inc., "Methods and Types." https://docs.tendermint.com/master/spec/abci/abci.html.

[19] Tendermint Inc., "Node Client (Daemon)." https://docs.cosmos.network/master/core/node.html.

[20] Tendermint Inc., "Overview." https://docs.tendermint.com/master/nodes/#node-types.

[21] Tendermint Inc., "Peer Discovery." https://docs.tendermint.com/master/spec/p2p/node.html.

[22] Tendermint Inc., "Reactor." https://docs.tendermint.com/master/tendermint-core/block-sync/reactor.html#reactor.

[23] Tendermint Inc., "Using Tendermint." https://docs.tendermint.com/master/tendermint-core/using-tendermint.html.

[24] Tendermint Inc., "Validators Tendermint Core." https://docs.tendermint.com/master/nodes/validators.html.

[25] Tendermint Inc., "What is Tendermint." https://docs.tendermint.com/master/introduction/what-is-tendermint.html.

Erklärung

Erklärung gemäss Art. 30 RSL Phil.-nat. 18

Ich erkläre hiermit, dass ich diese Arbeit selbstständig verfasst und keine anderen als die angegebenen Quellen benutzt habe. Alle Stellen, die wörtlich oder sinngemäss aus Quellen entnommen wurden, habe ich als solche gekennzeichnet. Mir ist bekannt, dass andernfalls der Senat gemäss Artikel 36 Absatz 1 Buchstabe r des Gesetzes vom 5. September 1996 über die Universität zum Entzug des auf Grund dieser Arbeit verliehenen Titels berechtigt ist.

Für die Zwecke der Begutachtung und der Überprüfung der Einhaltung der Selbständigkeitserklärung bzw. der Reglemente betreffend Plagiate erteile ich der Universität Bern das Recht, die dazu erforderlichen Personendaten zu bearbeiten und Nutzungshandlungen vorzunehmen, insbesondere die schriftliche Arbeit zu vervielfältigen und dauerhaft in einer Datenbank zu speichern sowie diese zur Überprüfung von Arbeiten Dritter zu verwenden oder hierzu zur Verfügung zu stellen.

Bern 13.01.22
Ort/Datum

Unterschrift