White-City: A Secure Substrate for Massive MPC

Omer Shlomovits, KZen Research
Alex Manuskin, KZen Research
Ittay Eyal, Technion, IC3
Denis Kolegov, Bi.Zone, TSU

Abstract

Secure Multiparty Computation (MPC) allows for a set of parties to evaluate a given function while achieving certain security properties such as input privacy. However, MPC protocols require extensive communication. Using direct peer-to-peer channels among the parties implies significant communication complexity and lack of robustness. Adding robustness to a protocol incurs even higher communication overhead and restricts the maximal number of faults in the system to 1/3 of the parties. It is no surprise then that most implementations of MPC are among small number of parties or run in controlled network environments.

We present *White-City*, a framework focused on supporting a flavor of MPC we call Massive Multiparty Computation (MMPC). Our scheme utilizes a Byzantine fault-tolerant replication protocol (BFT SMR) for carrying out a shared *state* among the MPC parties. Using an efficient SMR protocol, White-City allows for a flexible number of SMR participants. It allows for a large number of MPC parties, of which up to an arbitrary threshold are controlled by an active adversary. We argue the move from *message-based* to *state-based* communication is secure, strictly improves communication complexity and enjoys desirable properties for scaling MPC. In the setting of White-City, we treat two, usually overlooked, problems that might be of separate interest. First, we show how to support MPC with unauthenticated parties, made possible due to a model simplification that allows to mitigate Sybil attacks. Second, we show how to use the Noise protocol framework (Noise) for key agreement. Third, we show how to incentivize the SMR participants to run the network for the MPC. Finally, we implement our scheme using Tendermint BFT SMR and the Noise KK pattern and provide empirical validation for our results.

1 Introduction

Multiparty Computation (MPC) provides distrusting parties means to collaborate in order to run a joint computation without exposing each party's private input. MPC is the cryptographic implementation of a trusted party that can compute a function over stakeholders' private inputs and return the result. As such, MPC has high potential to disrupt industries that can benefit from such security properties [35, 8]. Practical applications have been found so far in key management [1, 5], financial oversight [7], MPC secured database [14], market design [15], biomedical computations [24, 23] and satellite collision detection [36].

However, successful decades of fruitful research in MPC have yet to be translated into adoption by industry. MPC is used today outside of academia either with a small number of participants (i.e. key management with two parties [1], secure database with three parties [3]) or for simple protocols (i.e. study of wage disparities [38], Zcash zk-snark CRS [17]). With White-City we wish to expend the usage of MPC outside of controlled testing environments to support large scale secure computations.

BFT SMR Byzantine fault tolerance (BFT) refers to the ability of a computing system to endure arbitrary (i.e., Byzantine) failures of its components while taking actions critical to the system's operation. In the context of state machine replication (SMR) [41, 37], the system as a whole provides a replicated service whose state is mirrored across k deterministic replicas. A BFT SMR protocol is used to ensure that non-faulty replicas agree on an order of execution for client initiated service commands, despite the efforts of f Byzantine replicas. This, in turn, ensures that the $k - f$ non-faulty replicas will run commands identically and so produce the same response for each command.

Distributed Systems researchers have made tremendous progress over the past decades in consensus protocols following the boom in blockchain based cryptocurrencies. PBFT [22] was the first practical BFT replication solution in the partial synchrony model. PBFT introduced a two-phase method to reach consensus decision within two rounds of communication. Each communication round requires $\mathcal{O}(k)^2$ messages when a an honest leader leads the consensus process. PBFT inspired many follow-up protocols that use the same concept, for example SBFT [33] and BFT-SMaRt [13]. Unfortunately, the leader election problem in the two-phase paradigm is rather involved which interfere with scaling the BFT SMR. Newer protocols are avoiding this problem by either moving to a three-phase model (i.e. Hotstuff [45]) or assuming optimistic responsiveness (i.e. Casper [20] and Tendermint [18]). In addition to streamlining the leader election process, by using

threshold signature schemes in modern BFT protocols such as Hotstuff and Tendermint, communication complexity for each consensus round can be reduced to $\mathcal{O}(k)$ [45].

MMPC In this work we put forward the simple yet elusive until now idea of combining state of the art MPC with state of the art BFT SMR to conquer the next frontier in MPC: Scale. We present White-City, a usable scheme for running MPC with large number of parties. Here is a short list of possible applications where MMPC meets the real world.

- **The PhD student.** You are a student doing research. In your research you want to train a ML model over genome sequences to improve personal medicine. You want to collect samples from random 1000 humans but you encounter a problem: while they all believe in your cause no one is willing to give you his private genome sequence. In another case, a student wants to run statistical analysis on passwords but no one wants to reveal to her his personal password. MMPC can be used: an online protocol that will output the model parameters / password statistics while keeping the samples private.
- **The lottery.** You are in a pub together with 249 more people. You suggest a game of lottery but you do not know any of the other people hanging in the pub nor do you have the physical means to run a lottery. You solve both problems by using MMPC: Each party chooses a random number. The parties will jointly generate a new random number and the winner is the party that holds the closest number.
- **The CRS.** You built for your company a blockchain with opt-in confidentiality properties. To do it you used a Non-interactive ZK proofs that require trusted parameters in the form of a so-called common reference string (CRS). If you generate it yourself the blockchain will not be used because you will have a backdoor. Instead you initiate MMPC ceremony with 5000 participants, accepting up to 4000 corruptions.
- **The Validator.** You are one out of 100 validators in an extremely fast PBFT based blockchain. The network value increases but adding more validators to increase the network security makes the PBFT consensus run slow. Instead each validator distributes his voting power to 100 parties where 50 are needed to complete a threshold signature. This Effectively makes the network 50x stronger.
- **The Farmer.** You grow the best carrots in the world. Once a year you put your carrots to public auction. Your carrot auction attracts 10000 bidders from across the globe. For each auction the carrot will go the the highest bidder. All bids should be kept private. You use MMPC with the bids as the parties private inputs and the output will be the highest bidder. You have 1000 carrots so to save time you run all 1000 auctions in parallel.
- **The VDF.** Verifiable Delay functions (VDF) can be useful tool in distributed systems. Some efficient constructions of VDFs are based on group of unknown order [43], such as RSA group. MMPC can be used to run a distributed RSA key generation among many parties such that not a single party or even a large group of parties can learn the order of the group.

The above examples are all useful usages for MPC that require a fast and reliable protocol among hundreds or more parties. In some cases, it makes no sense to run the protocol from static workstations or servers and mobile or IoT devices must be used. Those end points might have low connectivity, low computation power and no authentication to begin with. In this work, we rely on a strong infrastructure between permissioned facilitators to support distributed secure computation between a large scale set of permission-less parties.

The existing model. The classical MPC model assumes a set of n parties fully logically connected in a peer-to-peer network of transport secure channels. Most MPC protocols are progressing in rounds, assuming the communication is synchronous. The secure channels are used to either send a direct encrypted message to another party or to broadcast a message to all parties. A single adversary can control a subset of the parties. Over the years MPC protocols have been designed to provide security against more and more realistic adversarial behavior. However, while the protocol level continued to improve in efficiency and security, the communication level remained abstracted, ignoring faults and progress in distributed computing. As a result most open source implementations are relying on non-robust peer-to-peer links between lists of static IP-addresses [35]. This raise a couple of questions: First, how the IP tables are created at the first place. For example, in the list of potential applications of MMPC stated above it does not make much sense and will take a significant amount of time to build such table. Second, what happens when a channel breaks because of a network fault or if a party suffers a crash. As it turns out in existing general purpose frameworks [35], no recovery option exists and the MPC protocol will terminate prematurely in a non graceful manner; It can take a few timeouts until all parties will exit the program.

A new model. In our model, we assume there is an additional set of k parties, which we will refer to as *nodes*. The nodes are maintaining a *state* using a state machine replication (SMR) algorithm that can tolerate Byzantine faults (BFT). Each one of the n MPC parties can read from or write to the *state* via node's interface. We assume partial synchronous network, as it usually assumed for BFT SMR algorithms, and show how MPC based on synchronous network can run in such network model. The channels are partially authenticated. The nodes are connected to each other via secure channels (e.g., mutually authenticated TLS 1.3 channel). We assume their credentials (authenticated public keys) can be accessed and verified by the n parties using traditional PKI means. However, the channels between the parties and nodes must be securely set up prior to running the actual protocol. This partial authentication assumption is modelling the usage of IoT or new devices in the multiparty computation. For example, in the *PhD student* application above, a DNA sampling device might be used, that was never connected to any network prior to running the computation and therefore do not have a network identification that can be authenticated.

Each MPC party has one physical link and is logically connected to a small set of nodes instead of to all other parties directly. Broadcast if offloaded to the nodes, which results in asymptotic improvement in the overall communication, assuming number of nodes is much smaller than number of parties.

In the rest of the introduction we expend on the different parts of our system.

A distributed system tailored to support massive MPC. We use the nodes to maintain a *ledger*. The ledger contains the current state and immutable history of the communication. To do so we compile the MPC protocol to transform all peer-to-peer messages to broadcast messages.

We define the notion of security for our system design that includes both arguing about the security of the MPC as well as the distributed system of nodes and show the compiled MPC protocol is secure within this notion.

We instantiate our system design using Tendermint core [18] for consensus and threshold signature protocol for MPC. We provide a set of experiments that support our claims.

In particular, we define two notions relevant to our system: *crash-recovery* and *accountability*. In a crash recoverable MPC a party can recover from a temporary crash as long as its private inputs can still be accessed. Accountability gives a party a way to blame another party of adversarial behaviour in a way that can be publicly verifiable given the *state*. We show that our scheme satisfy these two properties.

Registration protocol. Our model assumes that the n parties have no history and are not part of any authority-based trust system like public key infrastructure (PKI) in the following sense: The parties don't have provisioned public key pairs and the corresponding signed certificates for them. At the same time, we assume that parties have all necessary CA certificates to authenticate nodes. This is important in our model to capture a large range of real world situations where MPC might be needed to run between many end devices, some of them might be IoT/mobile phone/stand alone/new. On the other hand, the k nodes are fixed and authenticated with a PKI. In classical MPC setting, the protocol assume secure channels between the parties. Therefore we need to have a registration protocol in place prior to running the MPC, in order to establish cryptographic identities for the parties.

Sybil attack resistance. Our registration protocol is solving the Sybil attack problem. We rely on a model in which the first message by an unidentified honest party is guaranteed to be generated honestly. We define the new model and discuss its validity. Our protocol consists of two main parts. At first, the parties send messages to all the nodes. Second, after some known time span the nodes arrive to consensus on the set of registered parties. An attacker can DoS the protocol by generating Sybil identities but it is guaranteed that if the consensus terminates we have a single list of identified parties of which no more than threshold t are corrupted.

A mechanism design to incentivize the nodes. Operating under a weaker model than previous works, we needed to find a way to incentivize the participants. While the MPC parties utility is well understood, it is unclear how nodes are motivated to participate and how we can make sure they indeed played their role. We introduce an ideal Judge functionality $\mathcal{F}_J$. Given some unit of value the Judge is able to transfer a reward from the parties to the nodes paying for storage and network. The payment is done prior to the online phase of the computation since after the online phase the parties that already got an output have no incentive to pay. On the other hand, the Judge supports a *blame* phase in which any party can present a local transcript for the Judge to decide if a node deviated from running the network honestly. In this case the Judge will punish the specific node. We prove using game theory why $\mathcal{F}_J$ achieves the desired mechanism for the nodes to run the network while protecting from false accusations. Distributed computation incentives is a broad topic and we hope our analysis will provide a first step towards a general framework.

2 Related Work

There are several papers that aim to improve plain mesh-network peer-to-peer communication for secure computation. Badger [31] is a system for distributing RSA signatures in which parties are also nodes in a consensus protocol. Like White-City, they use BFT SMR and instantiate it with Tendermint [18]. However, since the parties are also the nodes in their design, scaling is not achieved since parties are required to handle both the gossip communication to maintain consensus and the MPC communication. The Tendermint nodes need to setup secure authenticated peer-to-peer links between them for each new instance of a protocol which makes the setup process as long as a regular peer-to-peer setup.

MATRIX [9] solves the expensive setup problem using a cloud based service that collects all parties' information and sets all network definitions for them, including secure channels. The parties then communicate and run the protocol via direct peer-to-peer channels. The cloud service is a single point of failure that can suffer DoS attacks and does not provide robustness, accountability or recoverability support, which are required for running a massive long computation. MATRIX uses a specific optimized MPC protocol that can support up to $t = n/3$ malicious parties while our method can be adjusted to dishonest majority MPC protocols as well.

ETHDKG [41] is a distributed key generation protocol that uses the Ethereum blockchain for communication. In ETHDKG all messages are broadcast messages as in our solution. A way to look at it is that ETHDKG is a specific instantiation of our scheme with DKG as the MPC protocol and Ethereum as the BFT SMR. Thus, it lacks the systematic approach put forth in this work. ETHDKG requires the deployment of a smart contract on Ethereum. Each message is a transaction on Ethereum, validated by massive number of nodes. In our solution, the number of nodes is configurable based on the system requirements. The huge number of nodes imply that transactions takes significant time to propagate and confirmed in consensus which will not work for many practical use cases. In fact, since in the Ethereum *gas* model [44] there is specific cost, measured in gas, to each operation and there is a max capacity for gas per block, ETHDKG is highly optimized to fit DKG on Ethereum. Running two DKG protocols or a second MPC protocol in parallel will not be possible. Our scheme is designed to support multiple MPC instances at the same time.

HoneyBadgerMPC [39] is, similarly to our model, using a set of additional k nodes but with a different purpose. While we work in partial-synchronous communication for BFT SMR, HoneyBadgerMPC communication among the nodes is asynchronous to support Reliable Broadcast (RBC) and Common-Subset. In White-City the nodes are in charge of carrying communication between the computing parties. In contrast, with HoneyBadgerMPC the nodes actually participate in the MPC. At start, n parties are providing inputs to the nodes as follows: Each input is masked and secret shared among the nodes, which in turn use common-subset to reach an agreement on the broadcast instances that terminated. The nodes will then run the MPC program and will output the results back to the client parties. The MPC protocol used in HoneyBadgerMPC provides robustness against up to $t = k/3$ malicious nodes. As opposed to HoneyBadgerMPC, in our design the nodes that run the consensuses are never exposed to secret inputs from the parties, the number of nodes running BFT SMR can inflate to withstand the required number of faults, without changing the computation needed from the parties running the MPC. HoneyBadgerMPC increase in nodes means that each client party needs to secret share its private input with more nodes which are then required to run larger computation. HoneyBadgerMPC focus is on robustness and because of the tight coupling between supporting the communication and running the MPC, the MPC protocol is restricted to no more than $t = N/3$ malicious parties. Remembering that the client parties that provide the input might also be dishonest we are left with large overhead to deal with the two types of malicious behaviour.

3 White-City Design Goals

White-City provides a *scalable* solution to support secure communication between parties wishing to run an MPC protocol. White-City do not assume any prerequisites on the parties communication channels. It works in the following phases:

- *Registration:* Each party is a potentially, previously unidentified, honest party for a specific MPC protocol (§5). The parties generate and publish their static public keys. The SMR nodes agree on a list of parties. Each party fetches static public keys of all other parties from the ledger distributed by the connected node.

- *Cryptographic handshake:* The parties run a peer-to-peer (end-to-end) key agreement protocol using the nodes to send and receive handshake messages. The protocol implements Noise KK pattern [2]. As a result of the handshake each party has a pair of transport data symmetric keys per a peer-to-peer channel: One key is for sending the messages and the another one is for receiving. After the symmetric keys were established parties can securely communicate with each other.

- *Online phase:* The parties use the BFT SMR to propagate MPC communication rounds until abort or successful termination (§7).

We introduce the notion of a Judge (§8) that acts throughout the online phase to ensure the SMR nodes are incentivized to support the network.

Registration phase. As a design goal, motivated by the examples in the introduction, the parties in the system are not known before the protocol starts: they don't have cryptographically meaningful identities. The registration process aims to group n parties under a specific protocol identifier *pid* and supply them with one time identifications that will enable them to later run the key agreement in the cryptographic handshake phase.

Definition 3.1 *Specifically, the registration protocol should provide the following properties:*

- Correctness: *The registration protocol, if not aborted, shall output a valid public list of the n identities and corresponding public keys.*
- Sybil Resistance: *Under our model (§4) the final list of size n will contain no more than threshold t parties controlled by the adversary.*
- Parallel Registrations: *The registration protocol can register the same set of parties in multiple MPC protocols at the same time.*

The challenge in registration is to avoid Sybil attacks. Namely, how to restrict an adversary from faking identities and register instead of the honest parties. Intuitively, before a party sends its first message, the party is unknown, therefore the adversary cannot corrupt what it does not know exist. This intuition will be used to construct a Sybil resistant protocol.

Running multiple registrations will enable higher throughput in the online phase by running multiple MPC protocols concurrently, therefore it is a desired property in our scheme.

Cryptographic handshake phase. The cryptographic handshake should satisfy the following modern requirements:

- *Mutually authentication:* The peers are mutually authenticated to each other, if the registration protocol satisfies the requirements defined in 3.1.
- *Forward secrecy:* Each key agreement establish a shared symmetric key pair, providing strong forward secrecy for MPC data.
- *KCI-attack resistance:* Sender authentication resistant to key-compromise impersonation (KCI) attack.
- *Optional post-compromise security:* The protocol implementation should optionally provide post-compromise security [25].
- *Low communication cost:* The handshake should require two handshake messages.

Online phase. After the cryptographic handshake is complete and the transport data keys are derived the parties can run an MPC protocol between them. A messaging format should accomplish the following requirements:

- *Safety and Liveness:* Under our model, the scheme can run securely any MPC protocol π with parameters $\{t, n\}$.
- *Crash-Recoverability:* At any point throughout the online phase a party can go offline and return at a future time to the same point in the computation.
- *Accountability:* At any point throughout the online phase a party can prove with high probability to an outside auditor that it was given a specific message from another party without compromising privacy.

Safety and Liveness are considered the two aspects of *securely* running MPC: Safety means that input privacy is never broken, Liveness means the correctness of the computation; It is guaranteed that the computation will always end, where "abort" is also a valid ending.

Incentive mechanism design. The SMR nodes have no benefit wasting resources to support the MPC parties in the general case. In fact, having control over the communication layer give the SMR nodes the power to DoS some parties running the MPC. The nodes therefore should be compensated on the usage of their resources by the parties and should be punished if they fail to propogate the computation. Concretly, such scheme should meet the following requirements:

- *Dominant-Strategy:* A node's dominant strategy is to run the full computation.
- *Incentive-Independence:* A malicious party cannot affect the node's utility function.

4 Our Model

Communication model. We use the *partial synchrony* model as defined in [29]. Specifically, for each execution there is a bound Δ and a global stabilization time (GST), unknown to the processors, such that the system respects the upper bound Δ from time GST onward. If a correct process p sends a message to a correct process p' at $t >$ GST, then process p' received the message before $t + \Delta$. The partial synchrony model is required to reach consensus among the nodes on each stage of the execution. Supported MPC protocols operate in defined rounds and therefore require synchronization. We show how to define a round period in SMR based on partial synchrony in our compiler described in §7. Lastly, MPC require secure communication which is defined as follows:

Definition 4.1 *[Secure Communication] A secure communication, often called secure message passing, is authenticated communication which in addition the adversary has no access to the contents of the transmitted message.*

Definition 4.2 *[Secure Channel] A secure channel between two communicating peers is the channel that provides the following properties [4]:*

- *Confidentiality: Data sent over the channel after establishment is only visible to the endpoints.*
- *Integrity: Data sent over the channel after establishment cannot be modified by adversaries.*
- *One or both sides of the channel are authenticated.*

Partially authenticated channels. Within the model, we consider the MPC network with partially authenticated secure channels in the following sense [10]:

- Nodes are connected via authenticated channels: Nodes can mutually authenticate each other using asymmetric cryptography and authority-based trust mechanisms (e.g., PKI).
- Parties do not have any known cryptographically meaningful identities prior to the beginning of the protocol.
- The node side of the channel is always authenticated.

As a result we get secure channels where parties can send encrypted data to an authenticated node but the opposite is not always true.

Nodes Fault tolerance. For the BFT SMR the number of SMR nodes k and the number of faulty nodes f hold the relation $k \geq 3f + 1$ or less than third of the nodes are Byzantine. We do not force any relation between the number of SMR nodes k and the number of MPC parties n.

FUNCTIONALITY 4.3 ($\mathcal{F}_{MPC}$)

Functionality $\mathcal{F}_{MPC}$ works with some designated party P_1 and an ideal world adversary $\mathcal{S}$.

- **Init**: Upon receiving $\mathsf{init}(pid, n)$ from P_1, pick sid at random, send $pid' = pid||sid$ to P_1 and store (pid', id_1). Then, send pid' and n to $\mathcal{S}$.
- **Register**:
  - Upon receiving $\mathsf{register}(pid', id_i)$ from $\mathcal{S}$, if the pair (pid', id_i) was not received before, store (pid', id_i).
  - Once a list $(pid', id_1), \ldots, (pid', id_n)$ is stored, then send it to P_1 and $\mathcal{S}$.
- **Input**: Upon receiving $\mathsf{Input}{:}(pid', id_i, x_i)$ from P_1 or $\mathcal{S}$, if x_i was not received before for the pair (pid', id_i), then store (pid', id_i, x_i). If an abort was received from $\mathcal{S}$, then send abort to P_1 and halt. Once $(x_1, \ldots, x_n)$ is stored, then compute $(y_1, \ldots, y_n) = f(x_1, \ldots, x_n)$ and send $y_2, \ldots, y_n$ to $\mathcal{S}$.
- **Output**: Upon receiving $\mathsf{continue}$ from $\mathcal{S}$, send y_1 to P_1.

MPC Security Definition We use the standard definition of simulation based security based on the ideal/real model paradigm [21, 32]. The ideal execution and the formal security definition of MPC can be found in Appendix B. In the real model, a n-party protocol π is executed by the parties. The synchronous network proceeds in rounds and we consider a $\mathsf{rushing\ adversary}$, meaning that the adversary receives its incoming messages in a round before it sends its outgoing message. Secure channels are assumed, definition 4.2. The adversary $\mathcal{A}$ can be malicious; it sends all messages in place of the corrupted parties, and can follow any arbitrary strategy. The honest parties follow the instructions of the protocol. Our security is $\mathsf{with\text{-}abort}$, meaning that the adversary receives the output first and decides which parties will get their output. We assume the adversary is static: a party will be corrupted or honest for the entire execution of the registration and online phases.

Finally, let t be the maximal number of parties controlled by the adversary.

Registration security definition. We define ideal functionality $\mathcal{F}_{reg}$ that works as follows:

FUNCTIONALITY 4.4 (Registration functionality $\mathcal{F}_{reg}$)

Functionality $\mathcal{F}_{reg}$ works with parties $P_1, ..., P_n$ and threshold t.

- Upon receiving $\mathsf{Init}(pid)$ from a party P_i (for $i \in [n]$): pick sid at random and send $pid' = pid||sid$ to all parties.
- Upon receiving $\mathsf{Register}(pid', id_i)$ from all parties P_i (for $i \in [n]$): Send $\mathsf{Register}(pid', \{id_1, ..., id_n\})$ to the adversary $\mathcal{A}$.
- Upon receiving $\mathsf{RegisterOK}(pid', id_1, ..., id_{t'}, id'_1, ..., id'_{t'})$ from $\mathcal{A}$: If $t' \leq t$ and id_i exist, replace id_i with id'_i (for $i \in [t']$). Send the updated list to all parties $P_1, ..., P_n$ together with pid'. Otherwise, abort.

$\mathcal{F}_{reg}$ meets the security design goals as defined in §3.1. Correctness and parallel execution are straight forward. $\mathcal{F}_{reg}$ achieves Sybil-Resistance by accepting all honest parties id's and restricting the Sybil identities to no more than t, chosen by the adversary. The adversary can DoS registration if sending too many requests. We will prove security of our protocol in the hybrid model with ideal functionality for the consensus layer. Justification for using the hybrid model can be found in [21].

Note: for all protocols in the paper we implicitly assume that random numbers are chosen to be of the size of the system security parameter, which is given.

5 Registration Phase

In this section, we put forward a protocol to enable secure communications between n parties to allow for MPC protocol to operate at a later stage. The parties are running initially without any provisioned cryptographic keys or identities.

Distributing keys. For MPC it is required that between each two parties there will be a secure channel. Diffie-Hellman key agreement is a common method to establishe a shared secret key between two parties. It is known, that the basic Diffie-Hellman protocol is vulnerable to man in the middle (MitM) attacks. To be protected against MitM attacks the key agreement protocol in authenticated setting must be employed. In this case a shared secret key is established between two parties who mutually authenticate each other using

their public keys. It should be noted, that a key agreement protocol with static public keys only verifies that the corresponding private keys are possessed by the peer, but it's up to the upper-layer protocol to determine whether the remote peer's static public key is associated with the correct peer [2]. This mechanism is also called long-term key authentication (peer authentication, key validation or key verification). Methods implementing this traditionally include PKI-based authentication, key discovery, tamper-proof audit, pinning, fingerprinting, and others [30]. Having said this, to perform authenticated key agreement each party has to generate a long-term static key pair and securely distribute the public key among other parties.

Let (s_i^{priv}, s_i^{pub}) be a long-term static key pair of a party P_i. We will use the static public key s_i^{pub} of P_i as its identity id_i.

Protocol with a trusted party. We first abstract the consensus layer by presenting our protocol in a hybrid model using a functionality $\mathcal{F}_c$ which we later implement securely using the BFT SMR nodes we already assume to exist in the system.

We first present $\mathcal{F}_c$, functionality 5.1. We introduce a notion of a timer T. In our context it is merely a local estimate of a time frame after which all honest parties deliver their message. If the estimate is too low, we just re-run the protocol with better estimate. We introduce one idea on how to capture time in our system in lemma 7.1 but other possibilities exist (i.e. public blockchain).

FUNCTIONALITY 5.1 (functionality $\mathcal{F}_c$)
Auxiliary Input: timer T.

- Upon receiving $\mathsf{Init}(t, n, r)$ from a party P_i (for some $i \in [n]$): pick sid at random, send $(pid' = pid||sid, t, n, r)$ to all parties and set a $clock$ to 0.
- Upon receiving $\mathsf{Register}(pid', pk)$ from some party, if $clock < T$ add pk to a list L. If $|L| > n + t$ abort.
- Upon receiving $\mathsf{Publish}(pid')$ from some party, If $clock > T$ and $\mathsf{Publish}$ was never called before, send $\mathsf{Publish}(pid', L)$ to $\mathcal{A}$. Upon receiving back $\mathsf{PublishOK}(pid', L')$ from $\mathcal{A}$, check that $|L'| = n$ and that L, L' are different in no more than t entries (Otherwise abort). Send (pid', L') to all parties. If $\mathsf{Publish}$ was called before send L'.

We now move on to present our registration protocol π_{reg}.

PROTOCOL 5.2 (Registration Protocol, Securely computing $\mathcal{F}_{reg}$ in the $\mathcal{F}_c$ hybrid model)

1. A party P_j sends $\mathsf{Init}(pid, t, n, r)$ to $\mathcal{F}_c$
2. Each party P_i, upon receiving (pid', t, n, r) from $\mathcal{F}_c$, sends $\mathsf{Register}(pid', pk_i)$ to $\mathcal{F}_c$.
3. Each party P_i sends $\mathsf{Publish}(pid')$ to $\mathcal{F}_c$
4. Each party P_i upon receiving $(pid', \{pk_1, ..., pk_n\})$ from $\mathcal{F}_c$ checks if its public key exist in the list and if it does, generates a session key with each one of the other parties. Otherwise abort.

Lemma 5.3 *Assuming exactly n honest parties register, π_{reg} securely computes $\mathcal{F}_{reg}$ in the $\mathcal{F}_c$ hybrid model.*

We provide the proof in appendix A.1.1.

Implementing $\mathcal{F}_c$. According to the assumptions of our model, there are k nodes that establish secure channels between each other and run a dedicate network between them. The parties communicate with a small number of nodes over secure channels with authenticated node-side. Protocol 5.4 shows how the nodes can be used to implement $\mathcal{F}_c$

PROTOCOL 5.4 (Securely Computing $\mathcal{F}_c$)

Each node N_j do the following:

1. Upon receiving $\mathsf{Init}(pid, t, n, r)$, choose sid randomly and output $(pid' = pid||sid, t, n, r)$ to all parties $P_1, ..., P_n$. Set a local clock to 0.
2. Upon receiving $\mathsf{Register}(pid', pk)$, if local clock time is less than T: add pk to a list $L^j_{pid'}$. If $|L^j_{pid'}| > n + t$, abort. Once time T has passed: Reach consensus on the entries of $L^j_{pid'}$ and their order. Group the first n entries to an output list $L_{pid'}$. If no consensus on at least n entries was reached, abort.
3. Upon receiving $\mathsf{Publish}(pid')$, send back $L_{pid'}$ if exists and terminate.

Lemma 5.5 *Under our model, described in §4, protocol 5.4 securely computes $\mathcal{F}_c$.*

The consensus required is a multi-shot version of consensus which is the same as the BFT SMR *log replication*. Therefore, the nodes can use the already in place BFT SMR as clients to generate a state with some order of the static public keys. We refer to appendix A.1.2 for a proof of lemma 5.5.

Parallel registrations. The registration protocol can register the same set of parties in multiple MPC protocols at the same time, up to the delay caused by running consensus by the nodes. As can be seen from the protocol a party can initiate independent requests to participate in protocols by changing the *pid* in the request each time. For example, assume n parties wish to run 200 times some DKG protocol, they can agree to send 200 different requests with $pid = 1, ..., 200$.

Complexity analysis. Our first focus is on the *number of messages* sent in registration, irregardless of their size. We motivate this choice later when we analyze the complexity of the online phase.

The comparison to a full peer-to-peer network is not immediate because as we noted before, MPC usually assumes that secure channels are already in place. If it was not the case it is likely that a $\mathcal{O}(n^2)$ messages were needed: a message from each party to all other parties.

In addition we have that when setting up secure channels in a mesh network:

(a) The number of sent messages is equal to the number of received messages, both are $\mathcal{O}(n)$ per party.

(b) All messages are of the same size. We call this size *regular*.

In contrast, analyzing protocol 5.2 while explicitly using the BFT SMR implementation in protocol 5.4 we get that number of messages is now: $\mathcal{O}(kn) + \mathcal{O}(k^2n) + \mathcal{O}(kn)$. The first element is for every party to send a message to all the nodes, the second element is to reach consensus on n values and the last element is for sharing the result with all the parties. For $k << n$ this is roughly $\mathcal{O}(n)$.

Looking into send vs receive and size we get now:

(a) Each party sends $\mathcal{O}(1)$ messages (ignoring the k as constant factor) and receives $\mathcal{O}(1)$ messages.

(b) However, the size of the messages is now different: the *send* message has a regular size, while the *receive* message has n times the size of a regular message.

We conclude that taking the size of messages into consideration the clear improvement of our protocol is by sending one regular message per party instead of n regular messages.

6 Party-to-party Secure Channel Handshake

The Noise Protocol Framework. Noise [2] is a framework for building secure transport protocols based on Diffie-Hellman key agreement. The framework defines a set of fundamental handshake patterns with verifiable security properties specified in a simple language and uses a small set of cryptographic primitives. The handshake patterns are used to develop real secure transport protocols (e.g., Wireguard [6], nQUIC [34]).

The Noise Handshake. A Noise protocol begins with two parties exchanging handshake messages. During the handshake the peers perform an authenticated Diffie-Hellman key agreement according to the specified handshake pattern. The result of the handshake is a shared secret key.

In this model, we use the Noise KK pattern defined as follows:

PROTOCOL 6.1 (Noise KK handshake protocol)

$\rightarrow s$
$\leftarrow s$
...
$\rightarrow e, es, ss$
$\leftarrow e, ee, se$

It is assumed that an initiating peer and a responding peer have already known the static public keys of each other. The first message consists of the initiator's ephemeral public key e; a Diffie-Hellman calculation between that key and the responder's static key; a Diffie-Hellman calculation between the initiator and the responder static keys. The second message consists of the responder's ephemeral public key; a Diffie-Hellman calculation between that key and the initiator's ephemeral key; a Diffie-Hellman calculation between the initiator's static key and the responder's ephemeral key.

Justification of Noise KK. The KK handshake pattern was selected for the following main reasons:

- KK provides security properties for the model.
- KK fits the assumed distributing key model.
- two handshake messages are required to agree a shared secret.

From a cryptography perspective, the Noise KK pattern provides the following payload security and identity hiding properties according to [2]:

- If the responder's static private key is compromised then an attacker can forge authentication; the attacker can also decrypt any initial packet, but can't obtain the sent identity.
- Sender authentication is resistant to KCI-attack.
- Strong forward secrecy for upper-layer payloads
- Peer identities are not transmitted, but a passive attacker can check candidates for the pair of (responder's private key, initiator's public key) and learn whether the candidate pair is correct.

It should be noted that in this work, we do not develop a custom protocol based on a Noise pattern (like, for example, nQUIC or Wireguard) but use the pure KK pattern defined by the Noise specification. We leave for future work the design and extension of a new handshake protocol related to ratcheting, replay attack prevention, post-quantum resistance, etc.

Cryptographic primitives. For simplicity, and to avoid a negotiation phase we use the following cipher suite:

- Curve25519.
- ChaCha20-Poly1305.
- Blake2s.

7 Online Phase

In the Online phase we assume that parties can established secure point-to-point channels with each other and then can send encrypted and authenticated messages via the SMR layer. We will use n to denote the number of MPC parties and k to denote the number of SMR nodes.

MPC with partial synchrony. Synchronous MPC protocols progress in rounds. A communication round is a finite *known* time bound in which a party is expected to send a message and receive messages from the other parties. In case a message did not arrive, the party can safely determine the sending party faulted. In our model, the SMR assumes partial synchrony and all messages must pass through the SMR as we will later see. This means that we can no longer determine if a party faulted or the network is just experiencing unknown delay. We therefore need to define a communication round in partial syncrony communication model. We first assume that the network is stable enough for the SMR nodes to reach consensus. Under this

network condition we require the nodes to reach consensus over and over again (continuously), even without receiving any requests from clients. We define a *block* as a batch of 0 or more requests to update the state. We note that empty blocks are possible in the case that no requests were introduced to the SMR in the time between two blocks agreements. We define β to be a system parameter that represent the maximum number of blocks it takes a party's request to arrive to the nodes. Under this assumption a round can be defined as β blocks and synchronous MPC with r rounds in the synchronous communication model can be said to be equivalent to SMR approving $\beta \cdot r$ blocks.

Lemma 7.1 *Assuming every block eventually ends up in the state and assuming β is large enough such that a party response to a state change will be included in an updated state within β blocks agreements, than synchronous MPC with r rounds can run in partial synchrony network under $\beta \cdot r$ blocks.*

We justify our assumptions and discuss potential risks an practical considerations of using our method in appendix A.2.

Implementing broadcast channel. Multiparty computation is normally progress via interaction among the participants. The communication can be a broadcast message from one party to the rest or a direct message that should be visible only to a specific receiver and hidden from the rest. Under the old model that assumes a mesh network connected with secure channels, sending a direct message is a straight forward task. Implementing broadcast message on top of peer-to-peer channels is also well studied: We give an example for such protocol in §C. We call a mesh network with secure channels a *message-based* communication.

In our model, on the other hand, instead of assuming secure channels between each subset of two parties, we assume the existence of a broadcast channel. We say this is a *state-based* communication because all communication is public and therefore all parties share the same view or state.

We use BFT SMR with $k << n$ nodes as a layer to which all parties are connected. There are three basic types of requests that a party may issue to a node: a request to update the state with a broadcast message, a request to update the state with a direct message and request to read the state. We use r to indicate the MPC round number and include it in each message. Illegal round number is any $r \notin \{1, .., \rho\}$ where ρ is the last round number for the given protocol *pid*, known from registration phase, protocol 5.4. Formally, the messages are defined as follow:

- *Write broadcast:* Each node upon receiving $\{\mathsf{WriteBC}, pid, r, \mathsf{data}\}$ from party P_s will try to add $\{\mathsf{BC}, pid, P_s, r, \mathsf{data}\}$ to the state.
- *Write direct:* Each node upon receiving $\{\mathsf{WriteP2P}, pid, P_r, r, \mathsf{data}\}$ from party P_s will try to add $\{\mathsf{P2P}, pid, P_s, P_r, r, \mathsf{data}\}$ to the state. P_r is the receiving party .
- *Read:* Each node upon receiving $\{\mathsf{Read}, pid, r\}$ from party P_j, return all BC messages and all $\mathsf{P2P}$ messages intended for party P_j from round r. if round number is illegal return nothing.

We require that each message on both directions, from a party to a node or from a node to a party, to include a digital signature from the sender on the entire message. In our model, nodes are using a PKI and online parties have known post-registration (§5) public keys that will be used for the signing. If a signature on a message from a party is not verified, the message should be discarded.

Since up to $f = k/3$ nodes may be faulty in our model a party is required to send every request to $f + 1$ different nodes. A different subset of $f + 1$ nodes can be used for each message. We are guaranteed that at least one honest node will receive the request, verify the message and pass it to all the other nodes. As we later prove, eventually a message will reach enough nodes to get included in the state.

In addition, signed messages will provide a tool for auditors to authenticate the protocol communication.

Compiling message-based communication to state-based communication. We define a *round* in the compiled protocol to be either a *Write broadcast* followed by a *Read* command or a *Write direct* followed by a *Read*. This corresponds to dividing an MPC protocol to rounds that can be of two types: A broadcast round, where each party broadcast a single message and a direct round, where each party sends directly one message to each other party. *Wlog* we argue that this division captures communication for synchronous MPC protocols. The compiler from message-based to a state-based is given in protocol 7.2:

PROTOCOL 7.2 (Compiler)

We note (s_i^{priv}, s_i^{pub}) and (e_i^{priv}, e_i^{pub}) for a long-term static and ephemeral key pair of a party P_i, and k_{ij} for a shared secret key between parties P_i and P_j. Static public keys of all parties are known and valid. For each party P_i with message m at round r compile:

- **Broadcast round:**
  1. Send $\{\mathsf{WriteBC}, pid, r, m\}$ to $f+1$ nodes
  2. Send $\{\mathsf{Read}, pid, r\}$ to $f+1$ nodes until $n-1$, uniquely signed, $\{\mathsf{WriteBC}\}$ messages from round r are added to the state.
- **Direct handshake round:** If k_{ij} is not set run the handshake phase between P_i and P_j.
  1. P_i sends $\{\mathsf{WriteBC}, pid, r, e_i^{pub}\}$ to $f+1$ nodes. Update the protocol state according to the processing rules for KK pattern.
  2. P_j receives the message. Updates the protocol state according to the processing rules for KK pattern.
  3. P_j sends $\{\mathsf{WriteBC}, pid, r, e_j^{pub}\}$ to $f+1$ nodes. Update the protocol state according to the processing rules for KK pattern.
  4. P_i receives the message. Updates the protocol state according to the processing rules for KK pattern.
  5. P_i and P_j derive $k_{ij}^{send}, k_{ij}^{recv}$.
- **Direct transport round:** For each party $P_{j,j\neq i}$:
  1. Encrypt the message: $c = Enc(k_{ij}^{send}, m)$. Update the protocol state according to the processing rules for KK pattern.
  2. Send $\{\mathsf{WriteP2P}, pid, P_j, r, c\}$ to $f+1$ nodes
  3. Send $\{\mathsf{Read}, pid, r\}$ to $f+1$ nodes until $n-1$, uniquely signed, $\{\mathsf{WriteP2P}\}$ messages from round r are added to the state.

We prove the following lemma in appendix A.3, showing the compiler in protocol 7.2 translates any message based MPC protocol to state based MPC protocol with the same security gurentee.

Lemma 7.3 *For MPC protocol π, call the broadcast only compiled protocol π_c. We say that π_c is secure if π is secure and Enc is semantically secure encryption scheme.*

Lemma 7.3 considers only the MPC adversary. We now take into account the full system setting according to our model where the state machine is distributed between k nodes, $k/3$ of them might be Byzantine. We prove the following properties, defined in §3 about our scheme:

Lemma 7.4 *Parties running MPC protocol π using our scheme are guaranteed: Safety, Liveness, Crash-Recoverability and Accountability as defined in §3.*

A proof for each one of the properties is given in appendix A.4.

Complexity analysis. As with the registration protocol analysis, we start by looking only on the number of messages in the protocol. This metric will present an advantage to our scheme because messages from different parties are aggregated naturally. We show that:

(1) Even taking message sizes into consideration there is asymptotically improvement for our *state-based* scheme over the classical *message-based* MPC for broadcast round.

(2) Direct peer-to-peer round remains on the same order.

In a peer-to-peer mesh network the broadcast protocol used is given in §C. The cost of a broadcast round is therefore $\mathcal{O}(n^3)$.

On the other hand, in our scheme, a broadcast round costs $\mathcal{O}(kn) + \mathcal{O}(k^2 n) + \mathcal{O}(kn)$. The first term is due to each party sending one message to be broadcast by the nodes. The second term is for adding each of the n messages to the state. The third term is for reading the state by all parties. For $k << n$ our scheme complexity is equivalent to $\mathcal{O}(n)$. From this expression it is also clear why for small number of parties, k will become the significant term which will degrade the performance of our system compared to the message based setting. If we take into account the message size we see that the third term is the only one affected where each message was counted as aggregation of n messages. This will result in $\mathcal{O}(n^2)$ which is asymptotically better than the message-based case.

For a peer-to-peer round we require direct messages between every pair of parties. In the message based system this is simply $\mathcal{O}(n^2)$ as each party sends n messages. In our scheme we need $\mathcal{O}(kn^2) + \mathcal{O}(k^2n^2) + \mathcal{O}(kn^2)$. The first and third each can be aggregated to a single long message per party but even so the second term dominates and overall complexity will at best be on par with message based system. We note that we are not using any special batching techniques with both types of systems nor for the SMR layer.

7.1 Optimizations

The basic messaging format can be optimized for better efficiency in several ways which we describe here.

Termination: Knowing when a protocol is terminated is important since some MPC are requiring heavy communication which results in a large state space. The nodes will not be able to clear the state space without clear notion of termination. Without clear termination-signal, erasure is not safe. An *abort* based on timeout after $\beta \cdot r$ blocks (see lemma 7.1) will not necessarily mean that all parties had access to the latest state and therefore erasure is still not safe.

Our main method of signaling safe termination is by letting the SMR nodes be aware to the number of rounds in a specific MPC protocol; As part of registration, see protocol 5.2, the parties register with a protocol *pid* also the parameter r which stands for number of rounds. Only once the state contains a Write *and* Read requests for all rounds $1, ..., r$ for all parties $1, ..., n$ it is safe for the nodes to delete the state as a copy of it can now be reproduces locally with each party.

In dishonest majority MPC the security is with abort. As explained in §4, the adversary can quit early after receiving output. In other case, honest party might early-abort if some cryptographic checks detected malicious behviour. In those cases the protocol run should stop and it is safe to erase the state. To support it we add a fourth request for termination:

Terminate. Each node upon receiving $\{\mathsf{Terminate}, pid\}$ from party P_j will check if all other $n-1$ parties have terminated as well. If true: the node will update the state, delete *pid* protocol history and stop answering queries on *pid*. If false: the node will update the state.

Threshold signatures. Recall that each request sent from a party is sent to $f+1$ nodes to circumvent Byzantine faults in up to $f = k/3$ nodes. For messages of type Read, which constitute half the messages in the compiled protocol, this can be optimized. The first option is for a party to communicate with a single node that will answer with the state signed with $f+1$ different signatures from $f+1$ different nodes. The downside is that the size of the message will be dependent in the number of the nodes. To further optimize it we suggest to utilize $\{f+1, n\}$ threshold signature, i.e. with $t = f+1$, on each state update between the nodes. The threshold signature will be attached to every message sent to the party. This enables a constant size message and communicating with a single node. Implemented with BLS signatures [16] the process of threshold signing can be extremely efficient in comparison to SMR complexity.

Message Ack. Currently, a party in the compiled protocol has no way to tell if its messages were added to the state. In case of a fault, outside the scope of our model, a lack of acknowledgment on the message received correctly and added should trigger the party to resend. We remedy this by adjusting the compiler, protocol 7.2, and Read message format. For every Read the nodes will return also the messages sent previously from the sender party in that round. The party will check upon Read request, in addition to $n-1$ unique signed messages, that its own message appear in the state as well, otherwise it will resend it.

Enforcing message integrity. There are no limitations on the number of messages a party can write to the state in a given round. The nodes have the tools to enforce a one message per round per party policy. This will help protect against several kinds of faults, DoS attacks and will save space in the state. A node, post registration, is aware to the number of rounds, parties and the list of public keys that are valid for a given protocol, as depicted in protocol 5.4. For any Write message to have a correct form and accepted by the node, the round number must be specified and the party id should be validated using the digital signature attached. The node will try to add a message to the state only if the message is the first verified message sent from that party in that round.

Hybrid designs. Having a BFT SMR in place allows us to consider few advanced options that use the nodes in multiple ways. One such option is *optimistic message delivery* which uses the nodes to directly relay messages between parties before adding them to the state, potentially decreasing the running time in case the nodes are honest. Another idea is to use the nodes to backup the parties secret data by secret sharing a party's private data between the nodes. Finally, under certain conditions the nodes might be able to support

pre-processing phase for MPC protocols that require it. We discuss these ideas in more details in Appendix D.

8 Economic Model for Nodes

We direct the reader to Appendix E for a discussion on the problem of SMR nodes incentives and existing solutions.

We introduce a Judge Functionality $\mathcal{F}_J$, functionality 8.2. $\mathcal{F}_J$ is an ideal functionality connected to all the nodes and all the parties that took part in registration for a specific MPC protocol *pid*. The Judge abstracts how the nodes are rewarded and punished; it is assumed that there exists a fungible unit of value for reward, i.e. a cryptocurrency, and there exists a unit of value for punishment, i.e. reputation. We will briefly discuss a concrete instantiation later. To keep the solution simple we make the following helpful assumptions:

1. All SMR nodes are equal and should be rewarded equally.
2. All MPC parties are equally motivated to complete the protocol.
3. The MPC protocol is known in advance.

The first assumption can be removed afterwards, giving the nodes different weights based on some predefined criteria. Otherwise, it simplifies our analysis to focus on reward and punishment for a single node. The second assumption removes the potential complexity of parties that are not interested in the output or fair execution. We differ the treatment of different client strategies to future work. Finally, the third assumption is coherent with the way we constructed the registration phase in this work and the way MPC works, i.e. the to be computed function is known to all. Here we take it one step further and assume that each party knows *exactly* the structure of the final state before the protocol has started and can estimate the amount of work it takes to a node to get to this state.

Based on the third assumption we chose to operate in a *Gas* model [44] in which there is a fixed, known price per operation given in units of gas. The computation will continue as long as there is enough gas to pay for it. In this way the parties can transfer gas to the Judge that will use it to pay the nodes for computation. In functionality 8.2 we use η for the the mapping of each operation to some given unit of value. We use α as the amount of units required per party to run a computation, which is known in advance. The function Reward is used for the parties to signal the judge they want to run a protocol *pid* with ρ rounds and willing to pay the nodes a total of $n\alpha$ units for it.

After reward is defined and secured by the nodes, the parties and nodes will run the computation as specified in the online phase in §7. If no dispute - we are done. Since the nodes are paid in advance the only dispute that can happen is if a node denies service for a party and not providing it with the full state. A wrong state can occur only with negligible probability under our model assumptions, see lemma 7.4. In the case of a dispute, $\mathcal{F}_J$ can be called with function Blame. The input to this function is the view (transcript) of the accusing party. Here the Judge have two options: If the party is right the Judge will punish the accused node using a predicate called **Punish** we keep abstract. If the party is wrong, the Judge will provide the most updated state. Using the Crash-recoverability property, the state is global and each party can use it to get its respective view. The Judge therefore needs to maintain only the global state that can be updated by *any* of the nodes. As we prove, it is in the nodes best interest to update the Judge constantly when state is updated. They can do it via the interface UpdateState. The Judge keeps an internal representation of the state and if the new received state is a super set of the old state the Judge will update its internal view accordingly. The judge will punish under several conditions:

1. The input transcript is authentic
2. The input transcript includes the Judge internal state S
3. The input transcript is missing at least the last round ($S_\rho \notin transcript$)
4. There is enough gas for the nodes to continue the computation

The first condition can be checked using the digital signatures of the nodes in the transcript, the same way the Judge will verify a state update from a node before updating it internally. The second condition is trivial to check. The third condition is possible to check because at the time of setting the reward the parties declared the number of rounds ρ. Finally the last condition can be checked using the mapping η and a simulation of the internal state, compared to the initial funding αn.

We argue the following:

Lemma 8.1 *Functionality 8.2, parameterized by* ***Punish*** *predicate, satisfies Dominant-Strategy and Incentive-Independence properties as defined in §3.*

See appendix A.5 for a proof. We note that the Judge must keep in memory all previous *pid*'s and corresponding *state*'s to avoid parties replaying an old *transcript*.

We do not discuss a specific implementation for the ideal functionality and leave it for future work. One good candidate solution is to use a smart contract with privacy property to implement $\mathcal{F}_J$ and cryptocurrency as the reward *and* punishment. Another idea can be to use secure enclave that punish using a reputation penalty for registered nodes. A full analysis is out of scope for this paper.

FUNCTIONALITY 8.2 (Judge Functionality)

Functionality $\mathcal{F}_J$ works with parties $P_1, \ldots, P_n$ and nodes $N_1, \ldots, N_k$. The functionality maintains a state S. $S_r^{\{i\}}$ is the state at round r as received from node N_i, which is initialized to empty for all indices. $S^{\{i\}} = \sum_r S_r^{\{i\}}$

Auxiliary Input: A mapping η between operations (send, store etc..) and costs. Punishment predicate **Punish**.

- Upon receiving $\mathsf{Reward}(pid, \alpha, \rho)$ from all parties $P_1, \ldots, P_n$, save ρ and add $\beta = \alpha n/k$ to each node $N_1, \ldots, N_k$ balance.
- Upon receiving $\mathsf{UpdateState}(pid, state, r)$ from some node N_i, if $r \leq \rho$ and $S_r^{\{i\}} \subset state$ then update $S_r^{\{i\}} \leftarrow state$ with all new verified messages.
- Upon Receiving $\mathsf{Blame}(pid, transcript, N_i)$ from some party P_j, compare *transcript* to $S^{\{i\}}$. If $transcript \subset S^{\{i\}}$, send $S^{\{i\}}$ to P_j. Otherwise, if transcript is signed by the nodes, $\eta(S^{\{i\}}) < \beta$ and $S_\rho^{\{i\}} \notin transcript$, call **Punish** with input N_i.

9 Implementation

We have implemented our system using Tendermint for BFT state machine replication.

Each node comprises a Tendermint node [18] and an *application server*, in our case a White-City (WC) server. The Tendermint node is responsible for the peer-to-peer gossip protocol, for inter-node communication communication and for achieving consensus. The application determines which transactions are valid and keeps track of the state of the system.

The Tendermint node communicates with the application server via Application BlockChain Interface (ABCI) protocol. If a transactions is deemed to be valid by the application, it is propagated among the nodes. When consensus is reached, the transactions is appended to the distributed log and can later be queried.

A client, in our case a party running MPC, communicates directly with the Tendermint node, using the API for posting transactions and querying information. In our construction we give each node an equal voting power.

Each node has a public key, known to all elements in the system. MPC parties send requests to read or write the state via a RESTful protocol. Each party can make a request to any of the known nodes.

We have implemented the WC application server in Rust [40], using a Rust ABCI protocol implementation to communicate between the Tendermint node and the WC server. Our implementation is available as open source for the benefit of the community [1] As a proof of concept for an MPC party, we have implemented a client performing a $\{n, n\}$ threshold Schnorr signature using curve25519. The protocol is separated into a distributed key generation protocol (DKG), and a distributed signing protocol. DKG is performed once after registration. The parties can then perform the distributed signing protocol multiple times on various messages. A general overview of the system and connected clients is depicted in figure 1.

Deployment We tested the performance of our system using AWS infrastructure connected in a LAN setting. Each node runs on a dedicated EC2 instance. Parties are oversubscribed on separate EC2 C5.XLARGE instance with 8GB of RAM and 4 vCPUs. We tested various configurations of nodes and parties for DKG and signing protocols. Each measurement was performed 5 times with the average result taken as the measurement and the standard deviation as the error.

Fixed k, varying n We test the effect of increasing the number of parties participating in the computation, where the number of nodes remains constant. We analyzed the time to complete the computation by all participating parties. Figure 2 shows the computation time to complete the $\{n, n\}$ threshold signature on a

[1] Reference to implementation omitted for anonymity

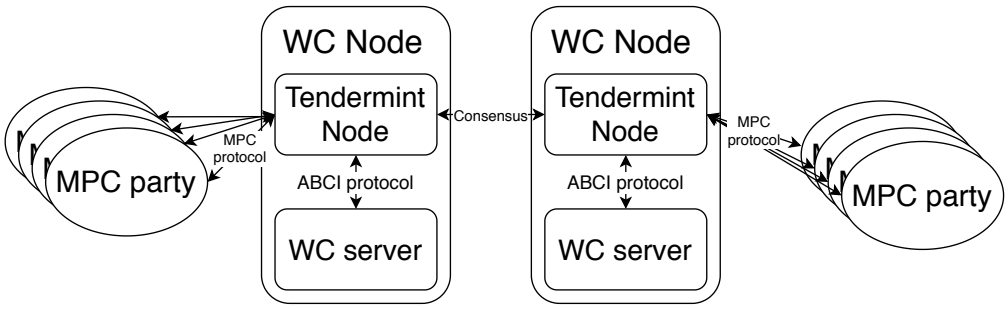

Figure 1: Overview of White-City nodes and connected parties

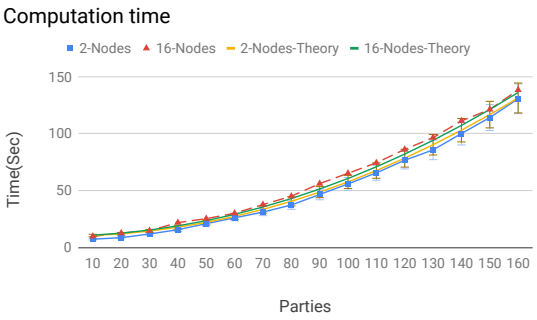

Figure 2: Computation time for varying number of parties

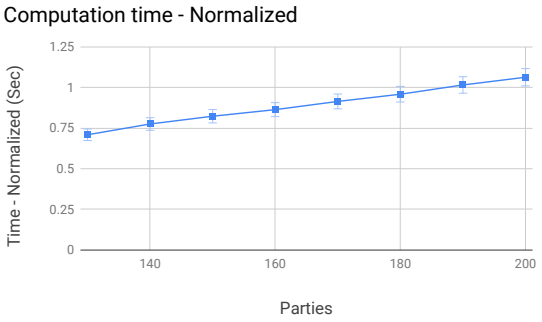

Figure 3: Computation time, normalized by the number of parties

known message by a varying number of parties, with 2 and 16 nodes. As expected, the total computation time grows qudratically as a function of the number of parties. In addition, the figure shows a theoretical analysis of the expected computation time. We expect computation to behave as $A \cdot n^2 + B \cdot k \cdot n$ for some constants A and B, when network conditions are stable. The exact values of A and B depend on variables such as network bandwidth, message size and computation complexity. We observe the measured results are inline with our theoretical analysis.

Figure 3 shows execution time, normalized by the number of participating parties. We observe that for a large enough number of parties, increase in message complexity and computation time is linear with the number of parties. As the number of parties increases relatively to the number of nodes, communication complexity of the parties becomes the dominant part of the computation.

Fixed n, varying k Next we tested the effect of the number of White-City nodes on computation time, when the number of parties is constant. Figure 4 shows computation times for a constant number of parties and varying number of WC nodes. As an example, experiments with 10 and 100 parties are presented. As expected, when $k << n$, adding WC node does not increase the total computation time significantly, all the while making the system more robust. When n is small, the cost of running BFT SMR is relatively more significant.

Support for multiple protocols To demonstrate the capabilities of the system to run Massive MPC, we have tested our system with as many as 512 parties and 16 WC nodes. Measurement results are presented in figure 5. The figure presents computation times for both DKG and co-signing a message by all parties. In this case, DKG is complete within a single round, while it takes 4 rounds to complete signing. At the moment, the largest documented MPC protocols are 128 parties AES [42], 256 parties ECDSA [28] and 500 parties computing mean and varaiance over secret inputs [9]. All demonstrated in academic literature only. We hope that using the scheme in this paper, MMPC will become possible in real world applications such as the above.

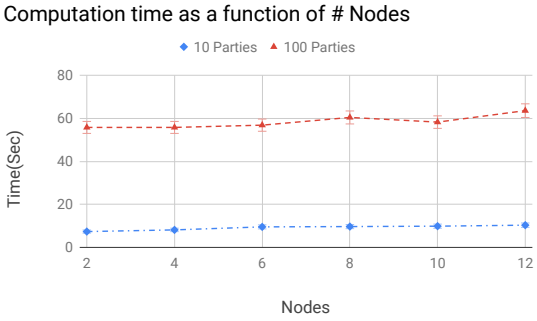

Figure 4: Computation time for varying number of nodes

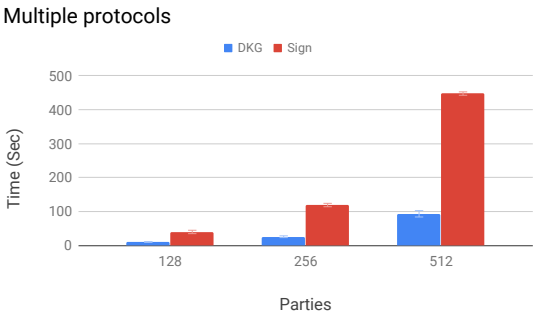

Figure 5: Comparing DKG and signing protocols using White-City

References

[1] Kzen networks. www.ZenGo.com. Accessed: 2019-09-01.

[2] The noise protocol framework. http://www.noiseprotocol.org/noise.html. Accessed: 2020-26-01.

[3] Sharemind. www.sharemind.cyber.ee/secure-computing-platform/. Accessed: 2019-09-01.

[4] The transport layer security (tls) protocol version 1.3. https://tools.ietf.org/html/rfc8446. Accessed: 2020-26-01.

[5] Unbound tech. www.unboundtech.com. Accessed: 2019-09-01.

[6] Wireguard: Next generation kernel network tunnel. https://www.wireguard.com/papers/wireguard.pdf. Accessed: 2020-20-01.

[7] Emmanuel A Abbe, Amir E Khandani, and Andrew W Lo. Privacy-preserving methods for sharing financial risk exposures. *American Economic Review*, 102(3):65–70, 2012.

[8] David W Archer, Dan Bogdanov, Yehuda Lindell, Liina Kamm, Kurt Nielsen, Jakob Illeborg Pagter, Nigel P Smart, and Rebecca N Wright. From keys to databases—real-world applications of secure multi-party computation. *The Computer Journal*, 61(12):1749–1771, 2018.

[9] Assi Barak, Martin Hirt, Lior Koskas, and Yehuda Lindell. An end-to-end system for large scale p2p mpc-as-a-service and low-bandwidth mpc for weak participants. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, pages 695–712. ACM, 2018.

[10] Boaz Barak, Ran Canetti, Yehuda Lindell, Rafael Pass, and Tal Rabin. Secure computation without authentication. In *Annual International Cryptology Conference*, pages 361–377. Springer, 2005.

[11] Carsten Baum, Ivan Damgård, and Claudio Orlandi. Publicly auditable secure multi-party computation. In *International Conference on Security and Cryptography for Networks*, pages 175–196. Springer, 2014.

[12] Carsten Baum, Emmanuela Orsini, and Peter Scholl. Efficient secure multiparty computation with identifiable abort. In *Theory of Cryptography Conference*, pages 461–490. Springer, 2016.

[13] Alysson Bessani, João Sousa, and Eduardo EP Alchieri. State machine replication for the masses with bft-smart. In *2014 44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, pages 355–362. IEEE, 2014.

[14] Dan Bogdanov, Riivo Talviste, and Jan Willemson. Deploying secure multi-party computation for financial data analysis. In *International Conference on Financial Cryptography and Data Security*, pages 57–64. Springer, 2012.

[15] Peter Bogetoft, Dan Lund Christensen, Ivan Damgård, Martin Geisler, Thomas Jakobsen, Mikkel Krøigaard, Janus Dam Nielsen, Jesper Buus Nielsen, Kurt Nielsen, Jakob Pagter, et al. Secure multiparty computation goes live. In *International Conference on Financial Cryptography and Data Security*, pages 325–343. Springer, 2009.

[16] Dan Boneh, Ben Lynn, and Hovav Shacham. Short signatures from the weil pairing. In *International Conference on the Theory and Application of Cryptology and Information Security*, pages 514–532. Springer, 2001.

[17] Sean Bowe, Ariel Gabizon, and Ian Miers. Scalable multi-party computation for zk-snark parameters in the random beacon model. *IACR Cryptology ePrint Archive*, 2017:1050, 2017.

[18] Ethan Buchman. *Tendermint: Byzantine fault tolerance in the age of blockchains.* PhD thesis, 2016.

[19] Ethan Buchman, Jae Kwon, and Zarko Milosevic. The latest gossip on bft consensus. *arXiv preprint arXiv:1807.04938*, 2018.

[20] Vitalik Buterin and Virgil Griffith. Casper the friendly finality gadget. *arXiv preprint arXiv:1710.09437*, 2017.

[21] Ran Canetti. Security and composition of multiparty cryptographic protocols. *Journal of CRYPTOLOGY*, 13(1):143–202, 2000.

[22] Miguel Castro, Barbara Liskov, et al. Practical byzantine fault tolerance. In *OSDI*, volume 99, pages 173–186, 1999.

[23] Erika Check Hayden. Extreme cryptography paves way to personalized medicine. *Nature News*, 519(7544):400, 2015.

[24] Hyunghoon Cho, David J Wu, and Bonnie Berger. Secure genome-wide association analysis using multiparty computation. *Nature biotechnology*, 36(6):547, 2018.

[25] Cremers C. Cohn-Gordon, K. and L Garratt. On post-compromise security. In *In Computer Security Foundations Symposium (CSF)*, pages 164–178. IEEE, 2016.

[26] Robert Cunningham, Benjamin Fuller, and Sophia Yakoubov. Catching mpc cheaters: Identification and openability. In *International Conference on Information Theoretic Security*, pages 110–134. Springer, 2017.

[27] Ivan Damgård, Valerio Pastro, Nigel Smart, and Sarah Zakarias. Multiparty computation from somewhat homomorphic encryption. In *Annual Cryptology Conference*, pages 643–662. Springer, 2012.

[28] Jack Doerner, Yashvanth Kondi, Eysa Lee, and Abhi Shelat. Threshold ecdsa from ecdsa assumptions: The multiparty case. In *Threshold ECDSA from ECDSA Assumptions: The Multiparty Case*, page 0. IEEE.

[29] Cynthia Dwork, Nancy Lynch, and Larry Stockmeyer. Consensus in the presence of partial synchrony. *Journal of the ACM (JACM)*, 35(2):288–323, 1988.

[30] N. Unger et al. Sok: Secure messaging. In *International Conference on Financial Cryptography and Data Security*, pages 232–249. IEEE Symp. Secur.Privacy, 2015.

[31] Naomi Farley, Robert Fitzpatrick, and Duncan Jones. BADGER – blockchain auditable distributed (rsa) key generation. *IACR Cryptology ePrint Archive*, 2019:104, 2019.

[32] Oded Goldreich. *Foundations of cryptography: volume 2, basic applications.* Cambridge university press, 2009.

[33] Guy Golan Gueta, Ittai Abraham, Shelly Grossman, Dahlia Malkhi, Benny Pinkas, Michael Reiter, Dragos-Adrian Seredinschi, Orr Tamir, and Alin Tomescu. Sbft: a scalable and decentralized trust infrastructure. In *2019 49th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 568–580. IEEE, 2019.

[34] Mathias Hall-Andersen, David Wong, Nick Sullivan, and Alishah Chator. Nquic: Noise-based quic packet protection. In *Proceedings of the Workshop on the Evolution, Performance, and Interoperability of QUIC*, EPIQ'18, page 22–28, New York, NY, USA, 2018. Association for Computing Machinery.

[35] Marcella Hastings, Brett Hemenway, Daniel Noble, and Steve Zdancewic. Sok: General purpose compilers for secure multi-party computation. *SoK: General Purpose Compilers for Secure Multi-Party Computation*, page 0, 2019.

[36] Brett Hemenway, Steve Lu, Rafail Ostrovsky, and William Welser Iv. High-precision secure computation of satellite collision probabilities. In *International Conference on Security and Cryptography for Networks*, pages 169–187. Springer, 2016.

[37] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, 1978.

[38] Andrei Lapets, Nikolaj Volgushev, Azer Bestavros, Frederick Jansen, and Mayank Varia. Secure multiparty computation for analytics deployed as a lightweight web application. Technical report, Computer Science Department, Boston University, 2016.

[39] Donghang Lu, Thomas Yurek, Samarth Kulshreshtha, Rahul Govind, Rahul Mahadev, Aniket Kate, and Andrew Miller. Honeybadgermpc and asynchromix: Practical asynchronous mpc and its application to anonymous communication.

[40] Nicholas D Matsakis and Felix S Klock II. The rust language. In *ACM SIGAda Ada Letters*, volume 34, pages 103–104. ACM, 2014.

[41] Philipp Schindler, Aljosha Judmayer, Nicholas Stifter, and Edgar Weippl. Distributed key generation with ethereum smart contracts.

[42] Xiao Wang, Samuel Ranellucci, and Jonathan Katz. Global-scale secure multiparty computation. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pages 39–56. ACM, 2017.

[43] Benjamin Wesolowski. Efficient verifiable delay functions. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pages 379–407. Springer, 2019.

[44] Gavin Wood et al. Ethereum: A secure decentralised generalised transaction ledger. *Ethereum project yellow paper*, 151(2014):1–32, 2014.

[45] Maofan Yin, Dahlia Malkhi, Michael K Reiter, Guy Golan Gueta, and Ittai Abraham. Hotstuff: Bft consensus in the lens of blockchain. *arXiv preprint arXiv:1803.05069*, 2018.

A Proofs

A.1 Registration Protocol π_{reg} Security

To prove security of the registration protocol we need to prove the two lemmas given in the paper.

A.1.1 lemma 5.3

Assuming exactly n honest parties register, π_{reg} securely computes $\mathcal{F}_{reg}$ in the $\mathcal{F}_c$ hybrid model.
Proof: We construct a simulator S who invokes the real model adversary internally and simulates an execution of the real protocol π_{reg}, while interacting with $\mathcal{F}_{reg}$ in the ideal model. S simulates $\mathcal{F}_c$ to the real model adversary. In the ideal model all honest parties send Register requests to the ideal functionality. S plays $\mathcal{A}$ in the ideal model and therefore receives from $\mathcal{F}_{reg}$ the list of honest parties identities. S works as follows:

1. S invokes the real model adversary with input (pid', t, n, r) where pid' and r are chosen randomly.
2. S accepts Register(pid', pk) requests from the adversary for a period of time T.
3. Upon receiving Publish request from the real model adversary, if S clock shows that T has passed, S sends RegisterOK$(pid', id_1, ..., id_{t'}, id'_1, ..., id'_{t'})$ to $\mathcal{F}_{reg}$, taking t' public keys as identities from the registrations requests it got from the adversary and replacing them with t' identities at random indices. S send back to the adversary (pid', L') where L' contains the modified list of honest identities.

We use pk and id interchangeably as under the DH-TLS scheme security they are equivalent. One difference between the simulated and real execution is that in the ideal model all honest parties send registration message to the ideal functionality, while in the real model it is not guaranteed. However, following our intuition for honesty of first message of unregistered parties we assumed in the lemma that all honest parties send the Register message successfully[2]. Other than that the view of the adversary is identical in the two models, including the ability of the adversary to DoS the parties in the protocol. ■

A.1.2 lemma 5.5

Under our model, protocol 5.4 securely computes $\mathcal{F}_c$.
Proof: We construct a simulator S who invokes the real model adversary internally and simulates an execution of the real protocol:

1. Upon receiving Init(pid', t, n, r) from the ideal functionality pass it to the adversary and receive Register requests from the adversary for time period of T.
2. Upon receiving Publish(pid', L) from the ideal functionality, compose a list with the adversary requests and complete it with honest requests taken from L to generate list L' of size n. return PublishOK(pid', L')

The only difference between the real and ideal executions is that in the ideal model the simulator chooses the output list arbitrarily and in the real model we use consensus to decide. We first note that the nodes decision is that the same set of values *exist* in all nodes, in *any* order. The order of the public keys selected in the consensus is usually a by-product of the mechanics of the consensus (i.e. the leader at a given time can choose). Eventually, in our protocol, this order determines the n public keys that will be selected and outputted. Even if Byzantine nodes are setting the order completely there is no way to distinguish a given order in the real model from the simulator determined order in the ideal model, as the simulator strategy is a valid strategy for determining order. The only option left to distinguish between real and ideal is if the consensus fails. However, since we assume that all Register message of honest parties arrive to all nodes we are guaranteed that at least n public keys of the honest parties to be inputted to the consensus by at least $2k/3$ nodes. Therefore, the consensus will terminate and from the validity property we have that at least all honest public keys will be decided. We conclude that using consensus in the real model protocol will produce indistinguishable output from any valid pick of n public keys from a list L, which is the strategy that the simulator and therefore the ideal model employs. ■

[2]We leave for future work a more formal model to capture this assumption

A.2 Synchronous MPC in Partial Synchronous network

In lemma 7.1 we rely on several assumptions. The first is that SMR nodes reaches consensus under the given network conditions. This is justified because if the network conditions are not allowing for consensus to be reached the SMR will not be able to perform log replication and no MPC will be possible either way.

The second requirement is that all parties agree on β. This constant can be determined by the nodes as part of registration. In case β needs to be renegotiated, it can be decided with consesus of the nodes (see first assumption) and read as part of the state by the parties.

Finally, we assume β is large enough for a message to arrive to the nodes with high probability. We note that a larger value of β decreases the chance of a false positive identification of a faulty party, but might be abused to introduce greater latency to the system. However we show that at the worst case only *Liveness* is affected: If a message was delayed more than the expected β blocks the party would be assumed to be non responsive and the protocol will abort. In fact, the nodes will know to signal the assumed fault to the other parties and the protocol will start from the beginning or from some checkpoint.

Given these trade-offs, β should be determined based on estimates of network conditions and adjusted adaptively.

A.3 State Based Compiler

We prove lemma 7.3:

For MPC protocol π, call the broadcast only compiled protocol π_c. We say that π_c is secure if π is secure and Enc is semantically secure.

Proof:

To prove the compiled protocol is secure we need to construct a simulator that invokes the adversary internally and provides him a view indistinguishable from a real view. The simulator construction is as follows:

PROTOCOL A.1 (Simulator S)

Run the simulator of π. There are few types of messages:

1. **Honest to honest:** The simulator generates a random string with the same length of the encrypted message and simulates a broadcast message of the right type.
2. **Corrupt to honest:** The simulator receives ciphertext c from the adversary, decrypts it and passes it to the simulator of π.
3. **Honest to corrupt:** The simulator encrypts the message that π produces from the honest party to the adversary using the right session key. The simulator sends the ciphertext to the adversary.

We remark that the message length is known from the specification of the MPC protocol and of the compiler.

We will prove the lemma by a sequence of games where each game is computationally indistinguishable from the previous game. Each game describes a simulator.

Game 0: The simulator H_0 simply executes the real protocol.

Game 1: The simulator H_1 is the same as H_0 but all Honest to Honest messages are now replaced with random string with the appropriate length. H_1 is indistinguishable from H_0 otherwise a distinguisher algorithm can break the semantic security of *Enc*.

Game 2: The simulator H_2 is the same as H_1 but all Honest to Corrupt messages are replaced with messages from the simulator of π. Any adversary that can distinguish H_1 and H_2 can be used to break the security of protocol π, therefore H_2 and H_1 are computationally indistinguishable.

Game 2 is the simulator of π_c described in protocol A.1 and Game 0 is the real execution. We conclude that the real model and ideal model are computationally indistinguishable. ■

A.4 Online Phase Properties

We provide proof sketches, showing that each of the following properties exist in our scheme described in §7.

- **Safety:** Our broadcast channel implementation introduces a Byzantine adversary controlling up to f of the node for $f = k/3$. Safety in the context of MPC is that the parties inputs remains private and never get leaked. The challenge is to handle the two types of adversaries at the same time. In the worst case we have a single adversary $\mathcal{B}$ that controls both the Byzantine nodes and the MPC adversary $\mathcal{A}$. We show that such an adversary will have no extra power over an adversary $\mathcal{A}$ alone which we already

proved in appendix A.3 to be secure. Concretely, because of the restriction on f Byzantine nodes, the only way $\mathcal{B}$ can add to the state is if it can fake a digital signature of one of the honest parties which it cannot do any better than $\mathcal{A}$. Updating the state with $\mathcal{A}$ input is already possible even without a single Byzantine node because the nodes cannot tell apart an adversary controlled party from an honest one as long as the message is structured correctly which is something $\mathcal{A}$ can do because it know the signing keys for the parties under its control. We conclude that $\mathcal{B}$ cannot change the state anymore than $\mathcal{A}$ and because the state is public and contains only broadcast messages it cannot learn anything more than $\mathcal{A}$ can, which as we proved before in A.3 is indistinguishable from ideal model and therefore security holds.

- **Liveness:** The key insight here is that *all* requests from honest parties are sent to *at least* one honest node. The following two statements are true:

  1. We have more than t honest parties
  2. We have at least $2f/3$ honest nodes

  By combining both statements we are guaranteed to have enough inputs from MPC parties to propagate the rounds and enough nodes to reach an agreement and propagate the SMR.

- **Crash-Recoverability:** The nodes are acting as a replicated memory for the entire computation. A protocol can be seen as a series of steps starting from a private input and a random tape. By making *only* honest requests a party will move to a new round only after reading all previous rounds and making sure it received validated and correct data from all other parties. Successful "Read" requests mean that the state maintained by the nodes is updated to the latest round and contains all previous rounds data from all parties. The state is immutable since under our model there are not enough Byzantine nodes at the same time to rewrite a state (we need $f+1$ nodes for that). The state is public access given access to a $f+1$ node. The state can be translated into the unique view of each one of the parties provided the party knows a private key that corresponds to its public key from registration. As such, a party with a private key and private input can replay the entire MPC protocol until reaching the tip. It can be done from the first round in case of wiped memory or from arbitrary round if the party suffered a temporary crash.

  We note that while we discuss the option of erasing the state to save nodes memory in §7.1, it will never happen before getting acknowledgment from all parties, including the crashed party.

- **Accountability:** To make our argument complete we start with a short background on accountability.

  Accountability is a problem well studied in MPC literature [26, 11, 12]. It is conveyed through a collections of properties, such as Identifiable-Abort, Public-Verifiablity Openability, Audiatbility, Guaranteed-Output, Fairness and more. We can look at these properties as providing different levels or aspectes of *Accountability*. In most of these works (see [26, 11, 12]) a bulletin-board is used to enhance the peer-to-peer network model. Some of the communication is required to be written to the bulletin-board. We claim that our scheme covers the same functionality for all mentioned works and thus the relevant accountability properties for each can be achieved on top of our model. Using a blockchain like Bitcoin for MPC is not feasible today and therefore we believe our solution can be used with these existing protocols to provide them with the necessary network model that can support a full blown MPC in real life.

  In our scheme, the state contains all messages for a specific protocol including sender and receiver identities and the protocol identification. An additional digital signature is attached to each message to attest in irrevocable way to the identity of the sender.

  An auditor needs: The registration list of public keys, and the protocol public transcript which is basically the *state*. To get these he can just ask $f+1$ nodes and make sure the state is the same for all. Under our assumption of BFT this will indeed be the case.

  We leave the specifics of how to use this framework with different *accountable* MPC protocols for future work but we claim that for most it is a drop-in replacement for their communication layer.

A.5 $\mathcal{F}_J$ Security

We wish to prove lemma 8.1. We first give a game theoretic argument for what would be a *Nash-equilibrium* for the nodes playing the game. Otherwise, *Dominant-Strategy* is a security property for the correctness of the Judge functionality.

The utility function for a node is $U = \sum Reward - \sum Punish$. Getting reward is a deterministic, independent outcome of a node participation in the game (taking part in registration). Under our model of BFT, a new state will always exist until completion of the protocol. Each node's best chances for not getting punished, is to participate in the SMR and update the Judge when a new state is agreed upon. Because each node is judged individually, this strategy is independent of what other nodes chosen strategies are: If there are not enough nodes to run the SMR securely because of crash faults, Byzantine faults or because some nodes strategy is being idle nodes, saving resources while other nodes run the network, than running the SMR honestly is the best strategy for the node to increase the odds of successful completion of the protocol. If there are enough nodes running the SMR algorithm, the node would still prefer to run it as well because there are no guarantees the other nodes will share with it the correct or most updated state in time. This makes the strategy of running the SMR and update the Judge the rational choice for a node as this is the Nash equilibrium. A *full* computation is required because the Judge is aware for the number of rounds and will punish on all rounds until the last one.

Interesting to note that the Judge must manage in memory k states which are basically identical. The reason is that if we let the Judge handle only a single instance of the state which *any* node can update, Nash equilibrium will be for the idle strategy; We leave for future work to make the Judge more memory efficient.

The nodes may cooperate and choose a strategy where in each game only a single node is in charge of storing the state in memory and all other nodes can query this node. We do not analyze this case and assume instead there is no safe way for the node to cooperate.

We now move to prove that even if all n MPC parties are malicious they cannot change the utility function of an honest node. We do it by enumerating over all the options that malicious parties can try do to.

Corrupting the Judge by one of the MPC parties is possible only via the *Blame* interface to $\mathcal{F}_J$: The *updateState* interface is accessible only to nodes and the *Reward* interface is triggered only by input from *all* parties and because of our threshold adversary assumption, an adversarial behaviour of not paying the same as others will be detected easily.

We will now cover all the actions a malicious party can take and show why they will not affect the Judge:

- **Abort and blame:** If a party aborts the protocol and then blames a node, an honest node will always be able to produce $S^{\{i\}}$ such that $transcript \subset S^{\{i\}}$.
- **False transcript:** It is infeasible to fake a transcript. An adversary that can do it can break the security of the nodes signature scheme.
- **Missing input to MPC:** If a malicious party try to cheat in the MPC protocol by not sending some of its inputs, The missing data will not get included in the state and therefore cannot be used for blaming the nodes.

B MPC Formal Definition

Let us start by defining MPC ideal execution: for any (possibly reactive) functionality $\mathcal{F}$, receiving inputs from $P_1, \ldots, P_n$ and providing them outputs. Let $I \subset \{1, \ldots, n\}$ be the set of indices of the corrupted parties controlled by the adversary. The ideal execution proceeds as follows:

- **Send inputs to the trusted party:** Each honest party P_j sends its specified input x_j to the trusted party. A corrupted party P_i controlled by the adversary may either send its specified input x_i, some other x_i' or an abort message.
- **Early abort option:** If the trusted party received abort from the adversary $\mathcal{A}$, it sends $\perp$ to all parties and terminates. Otherwise, it proceeds to the next step.
- **Trusted party sends output to the adversary:** The trusted party computes each party's output as specified by the functionality $\mathcal{F}$ based on the inputs received; denote the output of P_j by y_j. The trusted party then sends $\{y_i\}_{i \in I}$ to the corrupted parties.
- **Adversary instructs trusted party to continue or halt:** For each $j \in \{1, \ldots, n\}$ with $j \notin I$, the adversary sends the trusted party either abort_j or $\mathsf{continue}_j$. For each $j \notin I$:
  - If the trusted party received abort_j then it sends P_j the abort value $\perp$ for output.
  - If the trusted party received $\mathsf{continue}_j$ then it sends P_j its output value y_j.
- **Outputs:** The honest parties always output the output value they obtained from the trusted party, and the corrupted parties outputs nothing.

Let $\mathcal{S}$ be a ideal-world adversary controlling parties P_i for $i \in I$. Let $\text{IDEAL}_{\mathcal{F},\mathcal{S}(z),I}(x_1,\ldots,x_n)$ denote the output of the honest parties and $\mathcal{S}$ in an ideal execution with the functionality $\mathcal{F}$, inputs $x_1,\ldots,x_n$ to the parties and auxiliary-input z to $\mathcal{S}$.

The real model. Let $\mathcal{A}$ be a probabilistic adversary controlling $t < \frac{n}{k}$ parties [3]. Let $\text{REAL}_{\pi,\mathcal{A}(z),I}(x_1,\ldots,x_n)$ denote the output of the honest parties and $\mathcal{A}$ in an real execution of π, with inputs $x_1,\ldots,x_n$, auxiliary-input z for $\mathcal{A}$.

Definition B.1 (Computational Security) *Let $\mathcal{F}$ be a n-party functionality, and let π be a n-party protocol. We say that π* computes *f* with abort and computational security in the presence of an adversary controlling $t < \frac{n}{k}$ parties*, if for every non-uniform probabilistic polynomial-time adversary $\mathcal{A}$ in the real world, there exists a non-uniform probabilistic polynomial-time simulator/adversary $\mathcal{S}$ in the ideal model with $\mathcal{F}$, such that for every $I \subset \{1,\ldots,n\}$ with $|I| < n/2$,*

$$\left\{\text{IDEAL}_{\mathcal{F},\mathcal{S}(z),I}(x_1,\ldots,x_n,\kappa)\right\} \overset{c}{\equiv} \left\{\text{REAL}_{\pi,\mathcal{A}(z),I}(x_1,\ldots,x_n,\kappa)\right\}$$

where $x_1,\ldots,x_n \in \mathbb{F}^*$ under the constraint that $|x_1| = \cdots = |x_n|$, $z \in \mathbb{F}^*$ and $\kappa \in \mathbb{N}$. We say that π computes f with computational security in the presence of an adversary controlling $t < \frac{n}{k}$ parties with statistical error $2^{-\sigma}$ *if there exists a negligible function $\mu(\cdot)$ such that the distinguishing probability between the two distributions is less than $2^{-\sigma} + \mu(\kappa)$.*

C Non Terminating Broadcast Used in MPC

In the case of dishonest majority MPC, which is what we use in our model, MPC do not guarantee termination therefore a broadcast protocol do not have to terminate as well. Broadcast is implemented on top of peer-to-peer channels using the following simple protocol [27]:

1. The broadcaster sends message x to all $n-1$ parties.
2. Each party sends to all other parties what it received in the previous step.
3. Each party checks that it received the value x from all parties. If, so output x, otherwise abort.

This protocol message complexity is $\mathcal{O}(n^2)$. We do not consider any optimization such as batching.

D Hybrid Designs

Optimistic Message delivery. A WriteP2P message is intended to a single recipient. If a node would had a direct link to this recipient it would have been more efficient to first send the message to the recipient and then add it to the state. The node will be used here as trustless coordinator between the parties because the message is authenticated and encrypted. In case the node has no direct link to the recipient it can maintain a table of which party is connected to which nodes and first transfer to this nodes the message directly:
Write direct: Each node upon receiving $\{\mathsf{WriteP2P}, pid, P_r, r, \mathsf{data}\}$ from party P_s will:

1. Find the shortest path to party P_r and transfer the message through this path
2. Try to add $\{\mathsf{P2P}, pid, P_s, P_r, r, \mathsf{data}\}$ to the state. P_r is the receiving party index ($ssid$).

Potential offline phase and other usages for consensus nodes in the context of MPC. Some state of the art MPC protocols are working in the pre-processing model [27]. In this model there are data elements, such as Beaver multiplication triples and sharings of random numbers, that are generated offline, before the parties need to use their secret input to calculate some function. It is tempting to use the nodes infrastructure for MPC related computations during offline times and not only as nodes in a consensus protocol. Another use case is recovery of secrets: a party can use the nodes to store its secret data in a distributed manner, i.e. secret sharing of a secret share. We take advantage of the consensus protocol and our assumption on the network already in place between the nodes: We do not assume that the nodes are fully connected in peer-to-peer secure channels. For example Tendermint is using Gossip protocol [19]. Each node will also play a role of a party in this type of computations. Communication will play out the same as in the online phase. Since the nodes are not aware to the number of parties before registration it is required that a registration will happen before pre-processing can happen.

[3] k is usually 1 or 2

E SMR Nodes Economic Incentive: Problem Statement and Existing Solutions

From a game theoretic prospective, while it is clear what utility the parties gain with our scheme, it is unclear what incentivize the nodes to prefer the strategy of participate and run the network in the most efficient way. Logically, the parties should compensate the nodes provided the nodes can produce a proof of work and in a way which will be profitable considering the costs of running a node. However, this turns out to be far from trivial. To begin with, we try to avoid expensive setup time for the parties. Having the parties validate and pay nodes seems counter to this goal. An even bigger issue is how the nodes can actually provide *proof of work* that they participated in the BFT SMR network in honest way. To give a concrete example: the nodes provide SMR as a service. If a payment is to be given for nodes that participated in a computation and hold the state, a possible *free riding* attack would be for a node to simply copy the final state which will make it indistinguishable from a node that actually contributed resources and communicated messages to reach to that state.

There are two existing solutions, both are far from perfect for our most general case and can be used in some specific use cases.

The first is to use a public ledger: BFT SMR reduces to an application on top of the public ledger, i.e. a smart contact on Ethereum as in [41]. This solution will have the shortcomings discussed before which are mainly: lack of flexibility in setting the number of nodes, slowness in completing rounds because of long time to get to consensus, network fees not related to the actual resource consumption of the nodes and sharing the ledger with other applications that may effect the cost of doing computation on the ledger.

The second solution is to have a correlation between the nodes and the parties; for example: each party will run a single node. This solves the incentive problem because incentive is now aligned between the parties and the nodes. The downside is that we are losing the independence of the two layers. Each party will have to maintain at least one stable and authenticated node. This might turn out to be not cost effective for parties that only wish to carry out a particular computation small number of times.