White-City: A Framework For Massive MPC with Partial Synchrony and Partially Authenticated Channels

***Abstract*—Secure Multiparty Computation (MPC) allows for a set of parties to evaluate a given function on their private inputs without revealing anything but the output. In the last decade, there has been tremendous progress made in improving the concrete-efficiency of MPC protocols. As a consequence, MPC has started to be merchandised and used in real-world applications. However, most existing work assumes that all participants are connected via reliable point-to-point authenticated channels. This assumption limits the range of applications that can benefit from MPC. Indeed, all current industry deployments are only for a small number of parties, where it is possible to generate, in advance, a list of static identities with secure connections between each pair.**

In this work, we present *White-City*, a framework for MPC with a large number of parties. It relies on a trusted setup of a small set of permissioned facilitators, to support distributed secure computation among a large scale set of permission-less parties. Our framework does not assume any prior connection between the MPC parties or any kind of knowledge of the other participants. It allows the parties to run a pre-computation step, where they register and a list of identities is established, after which the joint computation is carried-out over partially authenticated channels. At the heart of our construction lies a Byzantine Fault-Tolerant State Machine Replication protocol (BFT SMR) for maintaining a shared state among the MPC parties. Using this mechanism, we are able to move from a message-based to a state-based computation, which can support large-scale computation with parties that progress at their own pace. With White-City, we solves two, usually overlooked, problems that might be of independent interest. First, we show how to mitigate Sybil attacks, bounding the number of corrupted parties participating in the computation. Second, we allow easy for crash-recovery for parties during the execution.

We implemented our scheme using Tendermint BFT SMR and provide empirical validation for our results.

I. Introduction

In the setting of secure multiparty computation (MPC), a set of parties with private inputs wish to compute a joint function of their inputs, without revealing anything but the output. Protocols for secure computation guarantee *privacy* (meaning that the protocol reveals nothing but the output), *correctness* (meaning that the correct function is computed), and more. These security guarantees are provided in the presence of adversarial behavior. There are two classic adversary models that are typically considered: *semi-honest* (where the adversary follows the protocol specification but may try to learn more than allowed from the protocol transcript) and *malicious* (where the adversary can run any arbitrary polynomial-time attack strategy). Security in the presence of malicious adversaries provides much stronger security guarantees, but is far more challenging with respect to efficiency.

Despite its stringent requirements, it has been shown in the 80s that any polynomial-time functionality can be securely computed with computational security [46], [31], [7] and with information-theoretic security [8], [20]. Since these feasibility results, the problem of constructing efficient protocols for secure computation has gained significant interest. Over the past decade progress has been extraordinarily fast, transforming the topic from a only notion of theoretical interest, into a practical technology that is even being commercialized by multiple companies. Practical applications have been found so far in key management [1], [3], financial oversight [4], MPC secured database [10], market design [11], biomedical computations [23], [21] and even satellite collision detection [34].

Despite this progress, current implementation of MPC (outside of academia) is still limited. In fact, successful deployments can be found, to this date, either for a very small number of participants (e.g., key management with two parties [1], secure database with three parties [2]) or for simple computations (e.g., study of wage disparities [38], Zcash zk-snark CRS [13]).

We argue that the main obstacle towards adopting large-scale MPC by the industry lies in the model and assumptions all existing MPC protocols and frameworks rely upon. While the protocol level continues to improve in efficiency and security, the communication level remains abstract. Specifically, current models assume that all parties are fully connected to each other in a peer-to-peer network of secure communication channels. This means in particular that the parties are assumed to "know" each other in advance and that secure authenticated channels were established between each pair of participants. As a result most open source implementations rely upon non-robust peer-to-peer links between lists of static IPs [33]. This raises several questions. First, how the IP tables are created in the first place. 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 [33], 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[1]. While such assumptions can be considered reasonable for computations of small scale (e.g., two or three parties), it is unlikely to accept these when hundreds or thousands parties are involved in the computation.

In this work, we present White-City, a framework for MPC with a massive number of parties (e.g., thousands of

Network and Distributed Systems Security (NDSS) Symposium 2021
21-24 February 2021, San Diego, CA, USA
ISBN 1-891562-61-4
https://dx.doi.org/10.14722/ndss.2020.24xxx
www.ndss-symposium.org

[1]It should be noted that a main reason for not taking this issue into consideration is that in existing security models, an honest party is not only assumed to always follow the protocol specification, but also to be always *available*.

parties). Our framework relies on a trusted setup of a small set of authenticated facilitators and does not assume that the MPC parties have any known public identity or any knowledge of other participants prior to the computation. Our communication model assumes only partial synchrony, meaning that the network is unstable and messages can be lost at times. Our framework allows easy recovery for parties who experience a crash and need to restore the current state of the computation. We implemented our framework and provide an empirical validation for the practicality of our results. With White-City we wish to expand the usage of MPC outside of controlled testing environments to support large scale secure computations.

Before proceeding to describe White-City, we highlight several applications that can benefit from a solution for MPC of massive scale in our model:

- **Research over large data.** A researcher wants to train a ML model over genome sequences to improve personal medicine. To this end, it is necessary to collect samples from random 1000 humans but this raises a problem: while they all believe in the cause no one is willing to give his private genome sequence. As an additional example, a security analyst may want to run statistical analysis on passwords but no one wants to reveal his personal password.
- **Auction/lottery.** A company sells some valuable merchandise via public auctions among bidders from across the globe. For each auction, the merchandise will go to the highest bidder. Similarly, one may want to run a lottery game, where 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. In both examples, all bids should be kept private and a computation is carried-out over the private data.
- **Distributed validators.** Suppose we have 100 validators in an extremely fast PBFT-based blockchain. To increase the network security, it would be desirable to increase the number of validators. However, adding more validators will cause the PBFT consensus protocol to run slow. Instead, each validator can distribute his voting power to 100 parties where 50 are needed to complete a threshold signature. This effectively makes the network 50x stronger.
- **Distributed VDF.** Verifiable Delay Functions (VDF) can be a useful tool in distributed systems. Some efficient constructions of VDFs are based on groups of unknown order [44], such as RSA groups. To use them, it is required 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. A very recent construction by [22] uses a trust-less coordinator to carry-out large scale RSA key generation. However, this coordinator is a single point of failure that we would like to avoid.

The above examples are all potential usages for MPC with hundreds or more parties. In these cases it makes no sense to assume that the parties have known identities prior to the computations and are mutually authenticated. Moreover, it would not be reasonable to assume that all parties run the protocol from a static work station or a server. Instead, we would like to allow usage of mobile or IoT devices. End points such as these might have low connectivity, low computation power and no authentication to begin with. Existing MPC solutions do not support such use cases.

A. Our Contributions

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. In particular, 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*. We assume partially synchronous network, as often assumed for BFT SMR algorithms, and show how any MPC protocol in the point-to-point model over a synchronous network can be executed in our network model, with the nodes' support. The channels are partially authenticated; the nodes are connected to each other via secure channels. We assume their credentials (authenticated public keys) can be accessed and verified by the n parties. However, the links between the parties and nodes are not assumed to be authenticated and must be set up prior to running the actual protocol. This partial authentication assumption models the usage of IoT or new devices in the multiparty computation. For example, in the research application above, a DNA sampling device might be used, that was never connected to any network prior to running the computation and therefore does not have a network identification that can be authenticated.

As we do not assume that parties have any known identity before the computation, our framework begins with a registration process, where any party who wishes to join the computation can choose an identity and send a registration request. The list of these new identities is managed and maintained by the nodes by running a consensus protocol. The main challenge in this step is that the adversary can mount a *Sybil attack*, attempting to flood the system with fake identities. To prevent this, we adopt the solution of [35], [5], which uses a Proof-of-Work (PoW) scheme to forces each registering party to invest work before sending the registration request. We present a specific Merkle tree [41] based PoW scheme, and prove that in our setting it indeed bounds the number of identities that can register. Once a list of identities is agreed upon, the parties represented by these identities, can proceed to the next step, where they run an authenticated key agreement. The goal of this step is to have each pair of parties agree on a shared key that will allow them to send private messages in the MPC protocol. As for all parts of our construction, this step is run with the nodes' support, meaning that all messages between two parties are exchanged via the nodes, who constantly run a consensus protocol to maintain the current state.

Finally, after generating mutual private keys, the parties can carry-out the joint computation. MPC protocols typically proceed in rounds. In each round, each party either sends a *private* message to some other party or sends a *broadcast* message to all the other parties and/or wait to receive a message. In our setting, each sent message is translated to a *write* message sent to the nodes, while receiving messages requires the parties to send a *read* message to the nodes. Using this abstraction, we show that any secure MPC protocol in the point-to-point model can be "compiled" into a secure protocol in our setting of partially synchrony and partially authenticated channels.

The security of all our constructions and protocols is proven via the standard ideal-real world paradigm [30], [16]. To the best of our knowledge, we are the first to provide an end-to-end rigorous treatment for multiparty computation protocols where the participating parties are not authenticated or even known prior to the protocol's execution.

An important property that our framework satisfies is the fact that the parties can advance at their own pace. Moreover, if a party crashes or is offline for a certain time, it can easily rejoin the computation. This is enabled by reading the current state of the computation from the nodes.

An additional interesting feature that is achieved by our framework is reducing communication. Since the parties communicate only with k nodes, then, letting k be constant, the communication grows linearly in the number of parties. From an asymptotic point of view, this improves upon the cost of execution in the point-to-point model, where communication usually grows in a quadratic rate (when an honest majority is not guaranteed).

A major challenge in deploying our scheme is how to incentivize the nodes to support the MPC computation. Clearly, the parties should compensate the nodes, given that the nodes can provide some proof of work. In Section VII we highlight the difficulties in designing an economic model for our scheme, provide an abstraction for such a mechanism, and discuss directions to realize it in practice.

a) Implementation.: We have implemented our system using Tendermint [14] for the BFT state machine replication. We tested the system using AWS infrastructure connected in a LAN setting, for various number of parties and nodes. Each node runs on a dedicated EC2 instance. Parties are oversubscribed on a separate EC2 C5.XLARGE instance with 8GB of RAM and 4 vCPUs. As a proof of concept for an MPC execution, we have implemented parties performing threshold Schnorr signature with dishonest majority using curve25519. We remark that our implementation includes only the MPC execution itself, and not the setup phase to register and agree on pair-wise keys. As the setup step is run only once, and its output can be used to support multiple MPC executions, its cost is amortized away.

Our results demonstrate the practicality of our solution; we are able to run distributed key generation in less than 100 seconds and signing in approximately 450 seconds, with 512 parties. We stress that our framework is designed to support a very large number of parties. Because of the high overhead incurred by the consensus protocol, constantly ran by the nodes, running a small scale computation using our system will be inefficient. Finally, we note that our implementation is written as an open source, and will be available to the benefit of the community.

B. Related Work

There are several papers that aim to improve plain mesh-network peer-to-peer communication for secure computation. Badger [28] 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 [14]. 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 [6] 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 [42] is a distributed key generation protocol that uses the Ethereum blockchain for communication. 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 implies that transactions take significant time to propagate and confirmed through the consensus protocol, which will not work for many practical use cases. In fact, since in the Ethereum *gas* model [45] 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 (HBMPC) [39] attempts to solve the problem of peer-to-peer communication by deploying a set of k nodes, similarly to white-city, with n clients communicating with the nodes to carry-out the joint computation. However, there construction is fundamentally different than ours. In their model, the nodes perform also the MPC protocol itself, over a secret sharing of the n clients' inputs. Sharing the secret input with the nodes implies that if $t > k/3$ nodes are malicious, all secret inputs of participating clients can be revealed. Unlike HBMPC, White-City does not require clients to share their secrets. Nodes operate solely as a communication and reliability layer. HBMPC emphasizes liveliness in an asynchronous environment, but does not guarantee security if $t > k/3$ nodes are malicious. In White-City, $t > k/3$ malicious nodes can prevent the protocol from terminating, but security of the party's private data is maintained. FFR-MPC [29] works similarly to HoneyBadgerMPC and additionally uses a public ledger (EOS blockchain) to incentivize nodes and participants to ensure fairness of the computation.

Finally, we note that none of the above works address the question of how the list of MPC identities is agreed upon to begin with, which we do in this work.

II. Preliminaries

a) Notation.: We use κ to denote the security parameter. We use $[n]$ to denote the set $\{1, \ldots, n\}$. Throughout the paper, we refer to the parties participating in the secure computation using the notation $P_1, \ldots, P_n$, and refer to the nodes that are used to communicate and store the state using the notation $\mathcal{P}_1, \ldots, \mathcal{P}_k$.

A. Communication and Computation Model

In our model, there are parties who wish to engage in some computation, without knowing any other party or having a public identity. Each party has access to an ideal functionality, that will be defined later (and will be realized by the nodes), through which it can communicate with the world (i.e., send and receive messages). Our communication model assumes *partial synchrony* [26]. In this model, the system alternates between being synchronous and begin asynchronous. In particular, there is a bound Δ and a Global Stabilization Time (GST) such that the system is unreliable before the GST, but after this time (which is unknown), each message is guaranteed to arrive before the bound Δ.

Although the parties do not know other entities that exist in the world, we must assume that all parties have an access to some global clock, which allows to advance in the same pace.

While the parties do not have any public identities at the beginning of the computation (which means that no public key for encryption or signing is associated with them), we assume that the channel between each party and the ideal functionality is *partially authenticated*. Specifically, when a party sends a message to the functionality, then the party can encrypt it with the public key associated with the functionality but not vice versa. Similarly, the functionality can sign messages it sends to the parties, which can be later verified using its known public key, but not vice versa.

In addition, each party has an access to a random oracle $\mathcal{H} : \{0,1\}^k \times \{0,1\}^k \cup \{0,1\}^k \to \{0,1\}^k$. To model computational complexity, we bound the number of calls to $\mathcal{H}$ by each party in "one unit" of time (which can be interpreted as the difference between two ticking of the global clock). We call this parameter the *hashrate* and denote it by π. We denote the hashrate of the adversary $\mathcal{A}$ by $\pi_{\mathcal{A}}$ and assume with out loss of generality that $\pi_A = t \cdot \pi$ for some $t \in \mathbb{N}$. This models the fact that the honest parties could use weak devices whereas the adversary may be more powerful. We stress that unlike Bitcoin, in our setting we do not require the majority of the computational power to be honest. We also stress that the assumption that all honest parties have the same hashrate is only for simplicity; our constructions easily adapt to a setting where each party (device) P_i has a different hashrate π_i. We assume that all other local computations performed by each device are immediate and takes no time.

For the nodes that are used to realize the communication ideal functionality, we assume that the channels between the nodes are fully authenticated, whereas the communication is partially synchronized as for the parties.

B. Consensus in the Presence of Byzantine Failures

A key ingredient in our construction is the need of the nodes to reach an agreement on the current state in each round. This is carried out in the presence of a subset of the f failures. In this work, the type of failures we consider is *Authenticated Byzantine*. In this setting, faulty nodes can behave in an arbitrary manner. However, each message is authenticated and so a byzantine node cannot forge messages in the name of other participants. A consensus protcol is defined as follows:

Definition 2.1 (Consensus): Let $\mathcal{P}_1, \ldots, \mathcal{P}_k$ be parties with inputs $x_1, \ldots, x_k$ respectively and let Π be an interactive protocol that receives $x_1, \ldots, x_k$ as inputs and outputs a set Y_i to each party $\mathcal{P}_i$. We say that Π is a consensus protocol in the presence of f authenticated byzantine failures if the following properties are satisfied:

- *Termination*: Each honest party $\mathcal{P}_i$ must receive an output Y_i.
- *Agreement*: For each pair of honest parties $\mathcal{P}_i, \mathcal{P}_j$, we have $Y = Y_i = Y_j$.
- *Unanimity*: If the all honest parties have the same input x, then $x \in Y$
- *Validity*: Each $y \in Y$ must satisfy some predefined condition.

As shown in [26], a consensus protocol in the partial synchronicity setting exists only if $k \geq 3f + 1$. We thus will assume in this work, that $f < \frac{k}{3}$.

a) Byzantine fault tolerance state machine replication (BFT-SMR).: In this work, we use Byzantine consensus in the context of state machine replication [42], [37]. Here 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 [19] 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 [32] and BFT-SMaRt [9]. 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 [47]) or assuming optimistic responsiveness (i.e. Casper [15] and Tendermint [14]). 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 $\mathcal{O}(k)$ [47].

C. MPC - Security Definition

We use the standard simulation-based security definition for MPC [30], [16]. Informally speaking, the definition considers an ideal world execution where a trusted party carry-out the computation upon receiving the parties' inputs, and sends back the outputs to the parties. The requirement is that an adversary that controlls a subset of parties, will not be able to do more harm in a real world execution (where a trusted party does not exist) than in an execution that takes place in the ideal world. This can be formulated by saying that for any adversary carrying out a successful attack in the real world, there exists an adversary that successfully carries out an attack with the same effect in the ideal world. However, successful adversarial attacks cannot be carried out in the ideal world. We therefore can conclude that all adversarial attacks on protocol executions in the real world must also fail.

More formally, the security of a protocol is established by comparing the outcome of a real protocol execution to the outcome of an ideal computation. That is, for any adversary attacking a real protocol execution, there exists an adversary attacking an ideal execution, called an ideal-world simulator, such that the input/output distributions of the adversary and the participating parties in the real and ideal executions are essentially the same. If this is the case, we say that the real protocol execution *securely computes* the ideal world functionality.

a) The hybrid model and composition.: We prove the security of our protocols in a hybrid model, where parties run a protocol with real messages and also have access to a trusted party computing a subfunctionality for them. The modular sequential composition theorem of [16] states that one can replace the trusted party computing the subfunctionality with a real secure protocol computing the subfunctionality. When the subfunctionality is g, we say that the protocol works in the g-hybrid model.

Protocols that are proven secure in the universal composability framework [17] have the property that they maintain their security when run in parallel and concurrently with other secure and insecure protocols. In this work, we take any MPC protocol and use in a black-box manner in our setting. Although we prove our protocols in the stand-alone model, it is immediate that if the underlying MPC protocl is UC-secure, then so is the resulted "compiled" protocol.

III. Building Blocks

A. Proof of Work (PoW)

A proof of work (known also as time-locked puzzle) is a protocol with a prover and a verifier. In the protocol, the prover provides a proof that is computational expensive to generate but is easy to verify. This notion was first introduced in [27] to solve the problem of handling spam messages. We follow the ideas of [35], [5] and use it to handle sybil attacks, namely, to prevent the adversary from flooding the system with fake identities. We use the definition for Pow from [5].

Definition 3.1 (Proof-of-work): A proof of work is a scheme that consists of a pair of probabilistic polynomial-time algorithms $(\mathsf{Solve}, \mathsf{VerifyPoW})$. The algorithm Solve takes a challenge $c \in \{0,1\}^{p(\kappa)}$ (where $p()$ is a polynomial) and outputs a solution $s \in \{0,1\}^{\kappa}$. The algorithm VerifyPoW takes c and s as inputs and outputs either 1 or 0 such that for every c it holds that $\mathsf{VerifyPoW}(c, \mathsf{Solve}(c)) = 1$. We denote by $\mathsf{time}_{\mathsf{proof}}$ and $\mathsf{time}_{\mathsf{vrfy}}$, the number of queries to a random oracle $\mathcal{H}$, required by Solve and VerifyPoW respectively. We say that $(\mathsf{Solve}, \mathsf{VerifyPoW})$ is *q-secure with ϵ-error*, if for every malicious prover $\mathcal{A}$, the probability that the output is 1 in the following game is less than $\epsilon(\kappa)$:

1) $\mathcal{A}$ makes polynomial number of queries (in κ) to a random oracle $\mathcal{H}$.
2) $\mathcal{A}$ receives a challenge $c \in \{0,1\}^{p(\kappa)}$.
3) $\mathcal{A}$ makes q more queries to $\mathcal{H}$ and outputs $s \in \{0,1\}^{\kappa}$.
4) The output of the game is 1 if $\mathsf{VerifyPoW}(c, s) = 1$.

a) PoW from Merkle trees [5].: The proof-of-work used in Bitcoin works by requiring the prover to find an input to an hash function that produces an output starting with a string of zeros. This idea is less suited to our purpose because the variance of the work required by the parties is high, as lucky provers might find a solution very quickly. In the context of sybil resistance, we are looking to a scheme where the computational effort needed by the prover is constant. Thus, we use in this work a scheme that was suggested in [5] and is based on Merkle trees [41]. In this scheme, given a challenge c, the prover is asked to build a merkle tree with L leaves, where the value of the ℓth leaf is $\mathcal{H}(\ell, c)$. Denote the root of the obtained tree by r. Then, the prover computes $(b_1, \ldots, b_\kappa) = G(r)$ where G is modeled as another random oracle. Then, it sends the root r together with Merkle-proof for the leaves indexed by $b_1, \ldots, b_\kappa$ (recall that a Merkle-proof consists of the siblings of each node on the path from a leaf to the root).

Observe that in this scheme the prover's running time $\mathsf{time}_{\mathsf{proof}}$ is $2L$ (as there are $2L$ nodes in the tree and each node require one call to $\mathcal{H}$) and thus the verifier's running time $\mathsf{time}_{\mathsf{vrfy}}$ is $\kappa \cdot \log(2L)$. In [5], the following lemma was proved:

Lemma 3.2 ([5]): The Merkle-tree based PoW described in the text is q-secure with error $q \cdot (\frac{q+1}{\mathsf{time}_{\mathsf{proof}}})^{\kappa} + \eta(\kappa)$ where $\eta(\kappa)$ is negligible in κ.

B. Signatures and Threshold Signatures

A signature scheme consists of three probabilistic polynomial-time algorithms $(\mathsf{KeyGen}, \mathsf{Sign}, \mathsf{VrfySign})$. The algorithm KeyGen takes as input a security parameter 1^{κ} and outputs a pair of keys (pk, sk), where pk is the public key and sk is the secret key. The algorithm Sign takes as input the private key sk and a message m (from some message space) and output s signature σ, and we write this as $\sigma \leftarrow \mathsf{Sign}_{sk}(m)$. Finally, the algorithm VrfySign takes as input the public key pk, a message m and a signature σ, to output $b \in \{0,1\}$. We write this as $b = \mathsf{VrfySign}_{pk}(n, \sigma)$. We say that σ is a valid signature if $\mathsf{VrfySign}_{pk}(m, \mathsf{Sign}_{sk}(m)) = 1$. The security requirement is that no adversary could forge a signature without knowing the private key with more than a negligible probability.

In threshold signature scheme, there are k parties and a threshold f. Then, KeyGen is an interactive protocol, with the output of the protocol held by P_i is (sk_i, pk). We cal sk_i the share of sk held by P_i and the requirement is that no subset of f parties can learn any information on sk, whereas a subset of $f+1$ can use their shares to reconstruct sk. Sign is an interactive protocol where the input of party P_i is sk_i and m, and outputs a signature σ. It may be possible also for signatures to be produced by a subset of $k+1$ parties (i.e., unlike key generation, it is not necessary that all parties will be available to run the Sign protocol successfully). Since pk is a public key, VrfySign remains unchanged. The security requirement is that no adversary controlling up to t parties will be able to generate a valid signature with more than a negligible probability.

IV. Setup Phase: Choosing the Participants and Establishing Secure Channels

In this section, we present the setup phase, which takes place before the joint computation is carried-out. The phase begins with n honest parties (devices) and ends with (at most) $n+t$ parties which will run the actual MPC protocol. The phase is divided into two steps. First, a registration step is executed to create an agreed list of participants. Each participant is identified by a signing public key that it distributes among all other parties, and is used to authenticate all messages sent by him from this moment on. The main challenge here is to prevent the adversary from mounting a Sybil attack where the system is flooded with fake identities. To overcome this challenge, we use a PoW mechanism that enforce each device to spend some computational power in order to create a new identity. In the second step of this phase, the chosen participants run pairwise handshake protocols to establish secure channels for the MPC protocol. At the end of this step, each two parties hold a joint symmetric key which is used to encrypt private messages sent from one party to the other.

A. Registration Step

We begin this subsection by defining a registration protocol. The aim of the protocol is to generate one public list of identities. The requirements from the protocol are that all parties will hold the same list of public keys and that the number of identities generated by the adversary is bounded. This is formalized in Definition 4.1. Note that we do not require that all honest parties will register successfully. This is due to the the fact that in the partial synchronous setting, we cannot guarantee that their registration requests will arrive at time. Nevertheless, we do require that if messages sent by an honest party arrive at time, then its public key will be in the final list.

Definition 4.1 (Registration Protocol): Let $P_1, \ldots, P_n$ be honest parties holding devices with hash rate π and consider the following protocol Π. At the beginning of the execution, each party P_i runs $(sk_i, pk_i) \leftarrow \mathsf{KeyGen}(1^\kappa)$. At the end of Π's execution, each party P_i holds a list $\mathcal{L}_i = (pk_1, \ldots, pk_{n'})$. We say that Π is a secure registration protocol in the presence of a malicious adversary $\mathcal{A}$ who has a device with hash rate $\pi_A = t \cdot \pi$ for some $t \in \mathbb{N} \setminus \{0\}$, if the following conditions hold:

- *Agreement*: for each pair of honest parties P_i and P_j, it holds that $\mathcal{L}_i = \mathcal{L}_j$.
- *Partial validity*: for each honest party P_i, if all messages sent by P_i in the protocol were sent after GST, then $pk_i \in \mathcal{L}_i$.
- *Resistance to sybil attacks*: the number of identities in $\mathcal{L}_i$ generated by $\mathcal{A}$ is at most t except with probability negligible in κ.

1) The ideal Functionality $\mathcal{F}_{\text{coordinate}}$: The ideal functionality $\mathcal{F}_{\text{coordinate}}$ coordinates the registration protocol by receiving registration requests from the parties with hash rate π, handing them a challenge, verifying their solutions and publishing the final list of participants to the parties. The functionality is given two deadlines, T_{req} and T_{sol}, which are assumed to be publicly known. The former is a deadline for sending registration requests, whereas the later is a deadline for sending solutions to the challenge. In addition, it works with an ideal world adversary $\mathcal{S}$, who is allowed to add t identities to the final list. When we will describe how to realize the functionality, $\mathcal{S}$ will be replaced with a real world adversary with hash rate $\pi_A = t \cdot \pi$, and we will show that $\mathcal{A}$ cannot produce more fake identities in the real world than $\mathcal{S}$ is allowed in the ideal world execution.

FUNCTIONALITY 4.2 (The ideal functionality $\mathcal{F}_{\text{coordinate}}$):

Let $(\mathsf{KeyGen}, \mathsf{Sign}, \mathsf{VrfySign})$ be a signature scheme and let $(\mathsf{Solve}, \mathsf{VerifyPoW})$ be a PoW scheme as defined in Definition 3.1.

The functionality works with n parties with hashrate π and an adversary $\mathcal{S}$.

Let $T_{req}, T_{sol} > \Delta$ be two public parameters known to functionality, honest parties and the adversary.

The functionality initializes two empty lists $\mathcal{L}, \mathcal{L}'$ and works as follows:

- Until T units of time have passed: upon receiving the message (pk, σ) from some honest party, $\mathcal{F}_{\text{coordinate}}$ checks that $\mathsf{VrfySign}_{pk}(pk, \sigma) = 1$. If not, or if $pk \in \mathcal{L}$, then it ignores the message. Otherwise, it sets $\mathcal{L} \leftarrow \mathcal{L} \cup \{pk\}$. In addition, $\mathcal{F}_{\text{coordinate}}$ sends each message (pk, σ) received from an honest party to $\mathcal{S}$.
- The functionality $\mathcal{F}_{\text{coordinate}}$ chooses a random challenge $c \in \{0,1\}^k$, sends c to all honest parties P_i for which $pk_i \in \mathcal{L}$ and to $\mathcal{S}$.
- Let $T = \mathsf{time}_{\mathsf{proof}}/\pi + T_{sol}$. Then, until T units of time have passed: upon receiving (pk, s, ρ) from an honest party, if $pk \notin \mathcal{L}$ then $\mathcal{F}_{\text{coordinate}}$ ignores the message or if $\mathsf{VrfySign}_{pk}(s, \rho) = 0$ or $\mathsf{VerifyPoW}(s, c||pk) = 0$, then $\mathcal{F}_{\text{coordinate}}$ ignores the message. Otherwise, it sets $\mathcal{L}' \leftarrow \{pk\}$. In addition, $\mathcal{F}_{\text{coordinate}}$ sends each message (pk, s, ρ) received from an honest party to $\mathcal{S}$.
- The functionality $\mathcal{F}_{\text{coordinate}}$ receives up to t additional identities $(pk_1, \ldots, pk_t)$ from $\mathcal{S}$ and adds them to $\mathcal{L}'$.
- The functionality $\mathcal{F}_{\text{coordinate}}$ sends $\mathcal{L}'$ to the honest parties and $\mathcal{S}$ and halts.

2) The Registration Protocol: We next present our registration protocol which works in the $\mathcal{F}_{\text{coordinate}}$-hybrid model, by having the parties send their registration requests and PoW solutions to $\mathcal{F}_{\text{coordinate}}$.

PROTOCOL 4.3 (The Registration Protocol in the $\mathcal{F}_{\text{coordinate}}$-Hybrid Model):

Let $(\mathsf{KeyGen}, \mathsf{Sign}, \mathsf{VrfySign})$ be a signature scheme and let $(\mathsf{Solve}, \mathsf{VerifyPoW})$ be a PoW scheme as defined in Definition 3.1. The protocol works as follows:

1) Each party P_i runs $(pk_i, sk_i) \leftarrow \mathsf{KeyGen}(1^\kappa)$. Then, it runs $\sigma_i \leftarrow \mathsf{Sign}_{sk_i}(pk_i)$ and sends (pk_i, σ_i) to $\mathcal{F}_{\text{coordinate}}$.
2) Upon receiving c from $\mathcal{F}_{\text{coordinate}}$, each P_i runs $s_i \leftarrow \mathsf{Solve}(c||pk_i)$. Then, it computes $\rho_i \leftarrow \mathsf{Sign}_{sk_i}(s_i)$ and sends (pk_i, s_i, ρ_i) to $\mathcal{F}_{\text{coordinate}}$.
3) Upon receiving $\mathcal{L}_i$ from $\mathcal{F}_{\text{coordinate}}$, party P_i outputs $\mathcal{L}_i$ and halts.

Theorem 4.4: Protocol 4.3 is a secure registration protocol in the $\mathcal{F}_{\text{coordinate}}$-hybrid model as defined in Definition 4.1.

Proof: We show that the properties in Definition 4.1 are satisfied. Agreement follows directly from the definition of $\mathcal{F}_{\text{coordinate}}$. For partial validity, note that since $T = \mathsf{time}_{\mathsf{proof}}/\pi + T_{sol}$ (recall that T is the amount of time given to the parties to provide a solution to the challenge), then each honest party has enough time to run Solve and so it is guaranteed that if its messages are received by $\mathcal{F}_{\text{coordinate}}$ before time T passes, then its public key will be part of the output list. Finally, the resistance to sybil attack follows since the ideal world adversary $\mathcal{S}$ is allowed to add only t identities to the final list. ■

3) Realizing $\mathcal{F}_{\text{coordinate}}$: We next show how a cluster of k nodes can securely compute the $\mathcal{F}_{\text{coordinate}}$ functionality. This is described in Protocol 4.5. The nodes run a consensus protocol to agree on an initial list of participants, choose jointly a challenge and then, run a consensus protocol again to agree on the final list of participants, given the solutions sent by the parties. In Theorem 4.6 we show that using the Merkle-tree PoW scheme from Section III-A, with carefully chosen configuration, we can bound the number of malicious identities to being t, except for a negligible probability.

PROTOCOL 4.5 (Computing $\mathcal{F}_{\text{coordinate}}$ - Coordination Protocol):
Let $(\mathsf{KeyGen}, \mathsf{Sign}, \mathsf{VrfySign})$ be a signature scheme, let Π_{cons} be a consensus protocol as defined in Definition 2.1 and let $(\mathsf{Solve}, \mathsf{VerifyPoW})$ be a PoW scheme as defined in Definition 3.1.
Let $T_{req}, T_{sol} > \Delta$ be two public parameters known to functionality, honest parties and the adversary.
Finally, let $G : \{0,1\}^{k\cdot\kappa} \to \{0,1\}^{\kappa}$ be a random oracle.
The protocol works with nodes $\mathcal{P}_1, \ldots, \mathcal{P}_k$ as follows:
The protocol:

1) Each node $\mathcal{P}_i$ initializes a list $\mathcal{L}_i$.
2) Until T_{req} units of time have passed: upon receiving the message (pk_{id}, σ), if $(pk_{id}, *) \in \mathcal{L}_i$, then node $\mathcal{P}_i$ ignores the message. Otherwise, it sets $\mathcal{L}_i \leftarrow \mathcal{L}_i \cup \{(pk_{id}, \sigma)\}$.
3) The nodes run Π_{cons} on $\mathcal{L}_1, \ldots, \mathcal{L}_k$ to obtain an output list $\mathcal{L}$, with validity condition:
$$(pk_{id}, \sigma) \in \mathcal{L} \iff \mathsf{VrfySign}_{pk_{id}}(pk_{id}, \sigma) = 1.$$
4) Each node $\mathcal{P}_i$ chooses a challenge $c_i \in \{0,1\}^{\kappa}$. Then, the nodes run Π_{cons} on $c_1, \ldots, c_k$ to obtain $c' = c_1 || \cdots || c_k$. Finally, the nodes call $G(c')$ to obtain and output a challenge $c \in \{0,1\}^{\kappa}$.
5) The nodes send c to each party with identity pk_{id} for which $(pk_{id}, *) \in \mathcal{L}$.
6) Each node $\mathcal{P}_i$ initializes an empty list $\mathcal{L}'_i$.
Until $T = \mathsf{time}_{\mathsf{proof}}/\pi + T_{sol}$ units of time have passed: upon receiving (pk_{id}, s, ρ), if $pk_{id} \notin \mathcal{L}$ then each node $\mathcal{P}_i$ ignores the message. Otherwise, there exists pk_{id} for which $pk_{id} \in \mathcal{L}_i$. Then, $\mathcal{P}_i$ sets $\mathcal{L}'_i \leftarrow \mathcal{L}'_i \cup \{(pk_{id}, s, \rho)\}$.
7) The nodes run Π_{cons} on $\mathcal{L}'_1, \ldots, \mathcal{L}'_k$ to obtain a final output list $\mathcal{L}'$, under the validity condition:
$$(pk_{id}, s, \rho) \in \mathcal{L}' \iff \left(\mathsf{VrfySign}_{pk_{id}}(s, \rho) = 1 \wedge \mathsf{VerifyPoW}(s, c||pk_{id}) = 1\right).$$
8) The nodes output all $pk_{id} \in \mathcal{L}'$.

Theorem 4.6: Assume that Π_{cons} is a consensus protocol that tolerates f authenticated byzantine failures, G is a random oracle, and the used PoW scheme is the Merkle-tree scheme from Section III-A with $\mathsf{time}_{\mathsf{proof}} = \kappa^2 + 2T_{sol} \cdot (\pi_A + \pi/2)$. Then, protocol 4.5 securely realizes $\mathcal{F}_{\text{coordinate}}$ in the presence of a malicious adversary with hashrate $\pi_A = t{\cdot}\pi$ (where $t \in \mathbb{N}$) controlling f nodes.

Proof: We present a simulator $\mathcal{S}$ who interacts with the real-world adversary $\mathcal{A}$ and show it is impossible to distinguish between the real and ideal world executions. Recall that by the definition of $\mathcal{F}_{\text{coordinate}}$, the simulator $\mathcal{S}$ receives all messages sent from the honest parties to $\mathcal{F}_{\text{coordinate}}$. Thus, it can perfectly simulate the honest parties sending registration requests and solutions for the challenge to the nodes controlled by $\mathcal{A}$. In addition, by the assumption that G is a random oracle, it follows that $\mathcal{S}$ can play the role of the random oracle, and hand the same challenge c chosen by $\mathcal{F}_{\text{coordinate}}$ to $\mathcal{A}$. This means in particular that the view of $\mathcal{A}$ in the simulation is identical to its view in a real execution. Finally, $\mathcal{S}$ plays the role of the honest nodes in the two executions of Π_{cons}. Then, it takes the identities added by $\mathcal{A}$, which are included in the final lists $\mathcal{L}$ and $\mathcal{L}'$ and sends these to $\mathcal{F}_{\text{coordinate}}$.

We thus only need to show that the output of the ideal world execution is the same as in a real execution. Note that the output consists of an initial list of parties which registered and a final list of parties who solved the challenge. By the unanimity property of Π_{cons}, it follows that all honest identities given to $\mathcal{S}$ from $\mathcal{F}_{\text{coordinate}}$ will be included in $\mathcal{L}$ and $\mathcal{L}'$ (since $\mathcal{S}$ plays the role of *all* honest parties).

It remains to prove that $\mathcal{A}$ will not be able to insert into the final output list more than $t = \frac{\pi_A}{\pi}$ identities. Let I be the set of identities for which $\mathcal{A}$ queried the oracle $\mathcal{H}$ more than $\xi \cdot \mathsf{time}_{\mathsf{proof}}$ times where $\xi = \frac{2\pi_A}{2\pi_A + \pi}$ (and so $0 < \xi < 1$). To complete the proof, we need to show that the following two claims holds: for any identity not in I, the probability that it is in the output list is negligible in κ and that $|I| = t = \frac{\pi_A}{\pi}$.

The latter holds since in time $T = \mathsf{time}_{\mathsf{proof}}/\pi + T_{sol}$, $\mathcal{A}$ can make $(\mathsf{time}_{\mathsf{proof}}/\pi + T_{sol}) \cdot \pi_A$ queries, which means that the number of identities in I is at most

$$\begin{aligned}
&(\mathsf{time}_{\mathsf{proof}}/\pi + T_{sol}) \cdot \pi_A \cdot \frac{1}{\xi \cdot \mathsf{time}_{\mathsf{proof}}} \\
&= \frac{\pi_A}{\pi} \cdot \frac{2\pi_A + \pi}{2\pi_A} + \frac{T_{sol} \cdot (\pi_A + \pi/2)}{\kappa^2 + 2T_{sol} \cdot (\pi_A + \pi/2)} \\
&< \frac{\pi_A}{\pi} + 1/2 + 1/2 = t + 1
\end{aligned}$$

and so since the number of identities must be an integer, we have that $|I| = t$ as required.

The former holds since by Lemma 3.2, the probability of any identity for which less than $\xi \cdot \mathsf{time}_{\mathsf{proof}}$ queries were made to $\mathcal{H}$, to have a valid solution is

$$\begin{aligned}
&\xi \cdot \mathsf{time}_{\mathsf{proof}} \cdot \left(\frac{\xi \cdot \mathsf{time}_{\mathsf{proof}} + 1}{\mathsf{time}_{\mathsf{proof}}}\right)^{\kappa} + \eta(\kappa) \\
&= \xi \cdot \mathsf{time}_{\mathsf{proof}} \cdot \left(\xi + \frac{1}{\mathsf{time}_{\mathsf{proof}}}\right)^{\kappa} + \eta(\kappa) \\
&= \xi \cdot (\kappa^2 + const) \cdot \left(\xi + \frac{1}{\kappa^2 + const}\right)^{\kappa} + \eta(\kappa)
\end{aligned}$$

where $const = 2T_{sol} \cdot (\pi_A + \pi/2)$, which since $0 < \xi < 1$ is a constant, is negligible in κ as $\kappa \to \infty$. This completes the proof. ■

a) Preventing a denial-of-service attack.: The above protocol bounds the number of fake identities in the final list. However, it does not prevent the adversary from generating many identities at the first step, and then making $\mathcal{F}_{\text{coordinate}}$ work for infinite time in verifying their proofs (even though these proofs will be eventually rejected), thus mounting a DoS (denial of service) attack. This can be prevented by using a PoW mechanism yet again. Specifically, each party will have to provide some proof for making a computational effort before sending a registration request. Note that here we do not need to bound the exact the number of initial identities, but only to make sure that the number is "reasonable" to handle. Thus, it suffices to use a simple Bitcoin-like PoW scheme.

B. Party-to-Party Handshake

After the registration step, a list of parties was established. These parties will participate in the execution of the multi-party computation. However, protocols for secure multiparty computation may require parties to send private messages to other parties. Thus, before starting the computation, the parties need to establish pairwise secure channels for these private messages. That is, each pair of parties need to agree on private keys, known only to both of them, which will be used to encrypt and authenticate peer-to-peer messages. To achieve this, each pair of parties will run a key agreement protocol, which will end with the two parties holding the same private key.

Intuitively, the two main security requirements from this protocol are: (1) *Authenticity*: if the protocol outputs a key $e_{i,j}$ in the execution between party P_i and party P_j, then P_i is convinced that it talked with P_j and vice versa; (2) *Secrecy*: from an adversary's point of view, the obtained key is indistinguishable from a random string. Note that the first security requirement is easily satisfied in our setting, since following the registration step, each party holds a private signing key with a corresponding public verification key known to the other parties. Thus, by signing each message during the key agreement protocol, any attack that involves forging parties' identities is avoided. In order to achieve the second requirement of secrecy, we first need to define the exact power of the adversary. This is in fact rather tricky; there are several models and levels of security that are typically considered in the literature. This stems for the fact that we want that the key for a particular session remains secret even if the adversary sees the session keys from other user instances. A strong assumption (which result in a weaker security definition) is that the adversary never compromises the long-term secret key of any honest user. A stronger security notion, called "perfect forward secrecy" aim to prevent the adversary from learning a particular session key, even though it is given at some point the long-term key of an honest user. Other strong potential security requirements are key-compromise impersonation (KCI) resistance and post-compromise security [24]; We refer the reader to [18], [36] for a more detailed discussion.

In this work, we have no intention to design a new key agreement protocol. Our aim is to prove that any Key Agreement (KA) protocol that is secure according to some definition, will remain secure in our setting as well. We show in this in two steps. First, we define an ideal functionality $\mathcal{F}_{\text{ComChannel}}$ (defined in Fuctionality 4.7) which serves a trusted communication Chanel between the parties. Specifically, it receives data sent from party P_i to P_j and stores it in a state. This is done via the instruction WriteKA which consists of the indexes of the involved parties, the round number r and the send data. To receive messages from other parties, each party can send the instruction ReadKA to $\mathcal{F}_{\text{ComChannel}}$. Note that in our formal description, we omitted for simplicity the fact that each party signs each message using its long term signing key received from the registration step. This is however crucial to prevent forgery.

FUNCTIONALITY 4.7 (The ideal functionality $\mathcal{F}_{\text{ComChannel}}$):

The functionality works with two parties P_i and P_j.
The functionality is given an execution identifier pid as an input.
Upon invocation, $\mathcal{F}_{\text{ComChannel}}$ initializes an empty list state. Then:

- Upon receiving a message $(pid, \mathsf{WriteKA}, i, r, j, data)$ from party P_i, if pid or r are invalid, or if $(pid, \mathsf{WriteKA}, i, r, j, *) \in \mathsf{state}$ then $\mathcal{F}_{\text{ComChannel}}$ ignores the message. Otherwise, it adds the message to state.
- Upon receiving a message $(pid, \mathsf{WriteKA}, j, r, i, data)$ from party P_j, if pid or r are invalid, or if $(pid, \mathsf{WriteKA}, j, r, i, *) \in \mathsf{state}$ then $\mathcal{F}_{\text{ComChannel}}$ ignores the message. Otherwise, it adds the message to state.
- Upon receiving the message $(\overline{pid}, \mathsf{ReadKA}, \overline{r})$ from party P_i (or P_j), functionality $\mathcal{F}_{\text{ComChannel}}$ sends P_i (or P_j) all messages in state where $pid = \overline{pid}$ and $r = \overline{r}$.

Given the ideal functionality $\mathcal{F}_{\text{ComChannel}}$, it is immediate that every secure key-agreement protocol in the point-to-point model, where each pair of parties communicate directly, remains secure where communication is carried-out via a trusted third party. This is formalized in the next theorem.

Theorem 4.8: Let Π_{KA} be a secure key-agreement protocol in the point-to-point model, which ends with each pair of parties P_i and P_j holding session keys $e_{i,j}$ and $e_{j,i}$. Let Π'_{KA} be an identical protocol to Π_{KA}, where each message sent by P_i to P_j in Π_{KA} is replaced by a WriteKA message to $\mathcal{F}_{\text{ComChannel}}$. Then, Π'_{KA} is a secure key-agreement protocol in the $\mathcal{F}_{\text{ComChannel}}$-hybrid model.

Next, we show how to compute $\mathcal{F}_{\text{ComChannel}}$ by using a cluster of nodes. This is done by having each node $\mathcal{P}_k$ updating its local state each time it receives a write message, and having the nodes running periodically a consensus protocol to agree on the current global state. Each read message is responded by sending the corresponding blocks from the agreed-upon global state. This is formalized in Protocol 4.10.

Theorem 4.9: Assume that Π_{cons} is a concensus protocol that tolerates f authenticated byzantine failures. Then, Protocol 4.10 securely computes $\mathcal{F}_{\text{ComChannel}}$ in the presence of malicious adversaries controlling f nodes.

Proof: We construct a simulator $\mathcal{S}$ for the ideal world execution as follows. The simulator $\mathcal{S}$ will play the role of the honest nodes in the execution interacting with an adversary $\mathcal{A}$ controlling f nodes. Note that the simulator can receive all honest parties messages stored in the state, by simply sending $\mathcal{F}_{\text{ComChannel}}$ a read request. Then, it can add these messages to the block held by each honest node and send these to the malicious nodes. In addition, it can send each message sent by $\mathcal{A}$ to the honest nodes to $\mathcal{F}_{\text{ComChannel}}$. By the properties of Π_{cons}, it follows that messages issued by the honest parties and received from the trusted party computing $\mathcal{F}_{\text{ComChannel}}$ will be part of the block that is added to the stored state. Thus, the state maintained by the honest nodes in the simulation will be identical to the state held by $\mathcal{F}_{\text{ComChannel}}$ and so is $\mathcal{A}$'s view. ■

PROTOCOL 4.10 (Securely Computing $\mathcal{F}_{\text{ComChannel}}$):
Let $\mathcal{P}_1, \ldots, P_k$ be the nodes participating in the protocol.
Let Π_{cons} be a consensus protocol as defined in Defintion 2.1.
The nodes receive pid as an input.
Upon invocation with an execution identifier pid, each node $\mathcal{P}_j$ initializes empty lists state and block_j. Then:

- Upon receiving a message $(pid, \mathsf{WriteKA}, i, r, t, data)$, if pid, i, t or r are invalid, or if $(pid, \mathsf{WriteKA}, i, r, t, *) \in \mathsf{state}$ then each node $\mathcal{P}_j$ ignores the message. Otherwise, it adds the message to block_j.
- Upon receiving the message $(\overline{pid}, \mathsf{ReadKA}, \overline{r})$, each node $\mathcal{P}_j$ outputs all messages in state where $pid = \overline{pid}$ and $r = \overline{r}$.
- Repeat infinitely: The nodes run Π_{cons} on $\mathsf{block}_1, \ldots, \mathsf{block}_k$ to obtain an output block, conditioned on pid, r, i and t being all valid.
  Then, each node $\mathcal{P}_j$ sets $\mathsf{state} \leftarrow \mathsf{state} \cup \mathsf{block}$ and initializes block_j as an empty list.

V. Running the MPC Protocol

In the previous section, we showed how the parties can generate a list of identities and establish secure channels between each pair of parties. In this section, we show how the parties carry-out the joint computation over their private inputs. At the beginning of the computation, each party P_i knows a list of public keys $(pk_1, \ldots, pk_{n'})$ and holds a pair-wise private session keys $e_{i,j}$ for each $j \in [n']$. Let F be the functionality which the parties wish to compute, and let Π_{mpc} be a multi-party protocol that securely computes F in the presence of t corrupted parties (recall that t is the bound on the number of malicious identities created by the adversary).

MPC protocols usually proceed by rounds. In each round, each party sends or receives a message or both. A standard MPC protocol Π_{mpc} has two types of sent messages: (1) broadcast messages intended to all the other parties; (2) point-to-point messages (P2P) from some party P_i to another party P_j with private data that cannot be revealed to any other party. The next message that each party sends is a function of its incoming/sent messages up to this point, its private input and its randomness.

Based on this intuition, we define state_i to be the set of all P_i's incoming and sent messages, x_i to be its private input and s_i to be its private randomness. Let $\mathsf{nxtMsg}()$ be a next-message function that takes state_i,x_i and s_i as its inputs and outputs one of the three next tuples:

- $(\mathsf{Broadcast}, round, data)$
- $(\mathsf{P2P}, round, P_j, data)$
- $(\mathsf{Read}, round)$

where $round$ is the round number and $data$ is the transmitted information. The first two tuples are messages that P_i needs to send, whereas the latest indicates that P_i waits for a messages to arrive.

The idea behind our protocol is simple. Each message is sent to an ideal functionality $\mathcal{F}_{\text{ComChannel}}$ who stores it. When a party waits for a message to arrive, it sends a read request to the functionality, who returns back the current state of the computation. In this way, the parties can work and advance at a different pace. Moreover, the functionality allows easy recovery for parties which crashed and want to rejoin the computation. The ideal functionality is realized using our cluster of nodes. To maintain the state of the computation, the nodes run a consensus protocol over and over again.

Throughout the section, we omit for simplicity the fact that each message sent by some party is signed by its private signing key and verified by all other nodes and parties using its known public key.

A. The Updated Ideal Functionality $\mathcal{F}_{\text{ComChannel}}$

In this section, we present an updated description of the ideal functionality that allows the parties to securely send and received messages in the protocol's execution at their own pace. The ideal functionality $\mathcal{F}_{\text{ComChannel}}$ receives two types of requests: write and read. A writing request is added to the execution state maintained by the functionality. Upon receiving a read request, $\mathcal{F}_{\text{ComChannel}}$ returns back a set of messages retrieved from the state, which satisfy the parameters in the request (e.g., all messages from a specific round r). The formal description is in Functionality 5.1.

FUNCTIONALITY 5.1 (The updated ideal functionality $\mathcal{F}_{\text{ComChannel}}$):
The functionality works with n' parties and an ideal world adversary $\mathcal{S}$ controlling $t < n'$ parties.
The functionality is given an execution identifier pid as an input.
Upon invocation, $\mathcal{F}_{\text{ComChannel}}$ initializes an empty list state. Then:

- Upon receiving a message $(pid, \mathsf{Write}, i, \mathsf{Broadcast}, r, data)$ from party P_i, if pid, i, j or r are invalid, or $(pid, \mathsf{Write}, i, \mathsf{Broadcast}, r, *) \in \mathsf{state}$ then $\mathcal{F}_{\text{ComChannel}}$ ignores the message. Otherwise, it adds the message to state.
- Upon receiving a message $(pid, \mathsf{Write}, i, \mathsf{P2P}, r, j, data)$ from party P_i, if pid, i, j or r are invalid, or if $(pid, \mathsf{Write}, i, \mathsf{P2P}, r, j, *) \in \mathsf{state}$ then $\mathcal{F}_{\text{ComChannel}}$ ignores the message. Otherwise, it adds the message to state.
- Upon receiving the message $(\overline{pid}, \mathsf{Read}, \overline{r})$ from party P_i, functionality $\mathcal{F}_{\text{ComChannel}}$ sends P_i all messages in state where $pid = \overline{pid}$ and $r = \overline{r}$.

Note that the functionality enforces a minimal level of integrity, by verifying that each party is not sending more than a single broadcast or point-to-point message in each round.

B. From Round-Based to State-Based Computation

We now present a protocol to compute any functionality F, using the ideal functionality $\mathcal{F}_{\text{ComChannel}}$. Given any multi-party protocol Π_{mpc} to compute F, that provides security in the presence of t corrupted parties, our protocol runs Π_{mpc} such that all messages are transmitted via $\mathcal{F}_{\text{ComChannel}}$. This is formalized in Protocol 5.2. Observe that our protocol can be seen as a "compiler" that takes a synchronous protocol that works round-by round, and translate it to a protocol that works in a partial synchronicity environment, with progress that is triggered by the state of the computation.

PROTOCOL 5.2 (Computing any F in the $\mathcal{F}_{\text{ComChannel}}$-Hybrid Model):
Let $(\mathsf{KeyGen}, \mathsf{Enc}, \mathsf{Dec})$ be a symmetric encryption scheme.
Let $P_1, \ldots, P_{n'}$ be the parties participating in the protocol.
Let Π_{mpc} be a protocol to compute F.

- **Input:** Each party P_i holds an input x_i and a randomness s_i. In addition, each pair of parties P_i, P_j hold a private symmetric encryption key $e_{i,j}$
- **Public input:** An execution identifier pid.
- **The protocol:**
  - At the beginning of the execution, each party P_i initializes an empty list $\mathsf{state}_{pid,i}$.
  - Until output is obtained, each party P_i run $str \leftarrow \mathsf{nxtMsg}(\mathsf{state}_i, x_i, s_i)$. Then:
    - If $str := (\mathsf{Broadcast}, r, data)$, then P_i sends $(pid, \mathsf{Write}, i, \mathsf{Broadcast}, r, data)$ to $\mathcal{F}_{\text{ComChannel}}$.
    - If $str := (\mathsf{P2P}, r, P_j, data)$, then P_i computes $c_{i,j} = \mathsf{Enc}_{e_{i,j}}(data)$ and sends $(pid, \mathsf{Write}, i, \mathsf{P2P}, r, j, c_{i,j})$ to $\mathcal{F}_{\text{ComChannel}}$.
    - If $str := (\mathsf{Read}, r)$, then P_i sends (pid, Read, r) to $\mathcal{F}_{\text{ComChannel}}$.
  - Upon receiving a set of messages $m_1, \ldots, m_\ell$ from $\mathcal{F}_{\text{ComChannel}}$: for each $k \in [\ell]$:
    - m_k is a broadcast message: party P_i parse it as $(\mathsf{Broadcast}, j, r, data)$. If $(\mathsf{Broadcast}, j, r, *) \notin \mathsf{state}_{pid,i}$, then P_i sets $\mathsf{state}_{pid,i} \leftarrow \mathsf{state}_{pid,i} \cup (\mathsf{Broadcast}, j, r, data)$
    - m_k is a p2p message sent or intended to P_i: party P_i parse it as $(\mathsf{P2P}, j, r, c)$. If $(\mathsf{P2P}, j, r, *) \notin \mathsf{state}_{pid,i}$, then P_i computes $data = \mathsf{Dec}_{e_{i,j}}(c)$ and sets $\mathsf{state}_{pid,i} \leftarrow \mathsf{state}_{pid,i} \cup (\mathsf{P2P}, j, r, data)$

Theorem 5.3: Let F be a n'-ary functionality. If Π_{mpc} is a protocol that securely computes F in the presence of a malicious adversary controlling up to t parties, then so is Protocol 5.2 in the $\mathcal{F}_{\text{ComChannel}}$-hybrid model.

Proof: By the assumption that Π_{mpc} is a secure protocol that computes F, it follows that there exists an ideal world simulator $\mathcal{S}$ for Π_{mpc}, which can interact with the real world adversary $\mathcal{A}$ and produce an execution where the joint distribution over $\mathcal{A}$'s view and the honest parties' outputs, are indistinguishable from the view and the output in a real execution. Thus, we can construct a simulator $\bar{\mathcal{S}}$ for our protocol who follows the instructions of $\mathcal{S}$ and plays the role of $\mathcal{F}_{\text{ComChannel}}$.

A crucial key for the success of the simulation is that $\bar{\mathcal{S}}$ will be able to decrypt the messages sent from corrupted to honest parties. Here we rely on the fact that every execution of Protocol 5.2 is preceded by an execution of the setup protocol, where $\bar{\mathcal{S}}$ playes the role of the honest parties, and so learns the pairwise encryption keys. Specifically, each message generated by $\mathcal{S}$ in its simulation, it translated by $\bar{\mathcal{S}}$ to a writing message that is stored in the state of $\mathcal{F}_{\text{ComChannel}}$, which is then emulated by $\bar{\mathcal{S}}$ for sending the messages to $\mathcal{A}$. Thus, $\mathcal{A}$' view the simulation with $\bar{\mathcal{S}}$ is identical to its view in the simulation with $\mathcal{S}$. Finally, $\bar{\mathcal{S}}$ extracts the corrupted parties' inputs as $\mathcal{S}$ would do, and sends them to the trusted party computing F, to receive back the outputs. It then simulates the opening of the outputs towards each corrupted party as $\mathcal{S}$ would do, and outputs whatever $\mathcal{A}$ outputs.

We thus conclude that the joint distribution of $\mathcal{A}$'s view and the honest parties' output is identical in both the simulation with $\mathcal{S}$ and with $\bar{\mathcal{S}}$, and so by a simple transitivity argument, it follows that the distribution in the simulation with $\bar{\mathcal{S}}$ is indistinguishable from the joint distribution of $\mathcal{A}$'s view and the honest parties' output in the real execution as required. ■

a) Crach-recoverability.: An interesting feature of our protocol, which is highly valuable in real world applications, is that it allows parties whose device crashed to easily recover and rejoin the computation. This is achieved by reading the state from $\mathcal{F}_{\text{ComChannel}}$ and continue the computation from the point where the crash occurred. This feature is not easily supported by standard MPC protocols, which work in point-to-point model.

C. Realizing $\mathcal{F}_{\text{ComChannel}}$ using BFT-SMR

In this section, we show how to compute $\mathcal{F}_{\text{ComChannel}}$ using a cluster of nodes. The idea is that each node will store the requests it receives in a block. Then, the nodes will run a consensus protocol on their blocks to agree on an update to the joint state. For each read request, the nodes will return only messages for which an agreement has been reached, i.e., messages that are stored in the state. This is formalized in Protocol 5.4. We remind the reader that we omitted here the fact that each message sent to the nodes mush be signed, and so the consensus protocol must accept only messages with valid signature.

PROTOCOL 5.4 (Securely Computing $\mathcal{F}_{\text{ComChannel}}$):
Let $\mathcal{P}_1, \ldots, P_k$ be the nodes participating in the protocol.
Let Π_{cons} be a consensus protocol as defined in Defintion 2.1.
The nodes receive pid as an input.
Upon invocation with an execution identifier pid, each node $\mathcal{P}_j$ initializes empty lists state and block_j. Then:

- Upon receiving a message $(pid, \mathsf{Write}, i, \mathsf{Broadcast}, r, data)$, if pid, i or r are invalid, or $(pid, \mathsf{Write}, i, \mathsf{Broadcast}, r, *) \in \mathsf{state}$ then each node $\mathcal{P}_j$ ignores the message. Otherwise, it adds the message to block_j.
- Upon receiving a message $(pid, \mathsf{Write}, i, \mathsf{P2P}, r, t, data)$, if pid, i, t or r are invalid, or if $(pid, \mathsf{Write}, i, \mathsf{P2P}, r, t, *) \in \mathsf{state}$ then each node $\mathcal{P}_j$ ignores the message. Otherwise, it adds the message to block_j.
- Upon receiving the message $(\overline{pid}, \mathsf{Read}, \overline{r})$, each node $\mathcal{P}_j$ outputs all messages in state where $pid = \overline{pid}$ and $r = \overline{r}$.
- Repeat infinitely: The nodes run Π_{cons} on $\mathsf{block}_1, \ldots, \mathsf{block}_k$ to obtain an output block, conditioned on pid, r, i and t being all valid.
  Then, each node $\mathcal{P}_j$ sets $\mathsf{state} \leftarrow \mathsf{state} \cup \mathsf{block}$ and initializes block_j as an empty list.

Theorem 5.5: Assume that Π_{cons} is a concensus protocol that tolerates f authenticated byzantine failures. Then, Protocol 5.4 securely computes $\mathcal{F}_{\text{ComChannel}}$ in the presence of malicious adversaries controlling f nodes.

The proof is almost identical to the proof of Theorem 4.9 and so we omit the details.

Note that our protocol implicitly assumes that each party is in contact with $2f + 1$ nodes (since f malicious nodes may provide a false state). In Section V-E, we discuss how to optimize this by allowing the parties to be in contact with $f + 1$ nodes only. Observe that this seems to be optimal, as it looks inevitable that a party will be in contact with at least one honest node.

D. Communication Complexity

Assume without loss of generality that in each round of the underlying MPC protocol, each party sends one message. This is translated in our protocol to one write and one read message that is sent to the nodes. Then, the nodes need to reach an agreement about the current state. Recall that there are n parties and k nodes. Thus, sending/receiving a message by the parties costs $O(n \cdot k)$, while the consensus protocol adds cost of $O(k^2)$. Overall, the communication complexity is thus $O(n \cdot k) + O(k^2)$.

In the case where $k << n$, we can view k as a constant, and so obtain *linear communication complexity*. Interestingly, in all peer-to-peer MPC protocol in the dishonest majority setting known to date, the communication complexity is quadratic. Thus, from an asymptotic point of view, our protocol also gives a way to go around this barrier and reduce the communication cost of protocols in the dishonest majority setting. Clearly, in practice, the advantages of our protocol will become visible only when the number of parties is very large compared to the number of nodes.

E. Optimizations and Advanced Features

We next describe several optimizations and extensions that can be incorporated into our framework.

a) Termination.: Knowing when a protocol is terminated is important since some MPC require 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. Thus, without clear termination-signal, erasure is not safe. Note that a solution where there is a pre-defined timeout, after which the nodes clear the state, is not safe in the partial synchronicity setting, as it is not clear how to set such a time-out.

A possible solution is to have the nodes be aware of the number of rounds in the executed MPC protocol. In the case that the state consists of messages of write for each round from all parties, and all parties have read the final state, the nodes will be able to erase the history of the execution. A better solution is adding another instruction to our protocol, where the parties can send $(pid, \mathsf{Terminate})$ to the nodes. If at least $t+1$ parties sent this message, and all parties read the state with these requests, then the nodes can delete the state of this protocol's execution. Note that the same applies when the MPC protocol is only secure with abort (this is inherent in the dishonest majority setting), which means that the parties may send at any time a broadcast message with $data = \mathsf{abort}$ to the nodes. In this case, the party who sent this message aborts the protocol, and so if any party sent such a message and all parties read the state containing these messages, the nodes can erase the state.

b) Reducing communication via Threshold signatures.: Our protocol requires each party to send a read request to at least $2f+1$ nodes, in order to be able to identify the correct state. This is due to the fact that f nodes may be malicious and may provide a false state (note that a malicious node cannot forge honest parties' messages, but it can "ignore" them, causing endless delays in the execution), and so by viewing $2f+1$ states, an honest party can choose the correct state by a simple majority rule.

This can be improved by using threshold signatures (see Section II). Specifically, we can ask the nodes to jointly sign each block that is added to the state. In this case, it is sufficient for the parties to contact one honest node in order to retrieve the correct current state. We note that since we assume an honest majority among the nodes, this can be implemented in an efficient way. For example, by implementing with BLS signatures [12] the process of threshold signing can be extremely efficient in comparison to SMR complexity.

c) Message Ack.: Currently, a party in the compiled protocol has no way to tell if its messages was received by a node. In case that the sent message was not added to the read state, this should trigger the party to resend. We remedy this by adjusting our protocol, such that for every Read message, the nodes will return also the messages sent previously from the sender party in that round.

d) Hybrid designs.: Having a BFT SMR in place allows us to consider advanced options that use the nodes for other purposes. For example, the nodes can be used to backup secret data by the parties. Here we can use the assumption that less than third of the nodes can be corrupted to secret share any data in a robust way that allows recovery.

VI. Implementation and Experimental Results

We have implemented our system using Tendermint for BFT state machine replication. Each node comprises a Tendermint node [14] 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 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 transaction is deemed to be valid by the application, it is propagated among the nodes. When consensus is reached, the block of 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 an open source for the benefit of the community [2]. As a proof of concept for an MPC protcol execution, we have implemented a client performing a threshold Schnorr signature with dishonest majority using curve25519. The protocol is separated into a distributed key generation protocol (DKG), and a distributed signing protocol.

[2]Reference to implementation omitted for anonymity

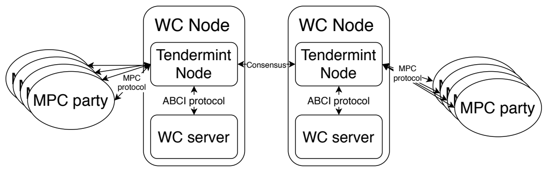

Fig. 1: Overview of White-City nodes and connected parties

All parties participate in both protocols. 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.

a) Experiment setting.: 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.

b) Experimental Results: Figure 2 shows the running time to complete the threshold signature on a known message by a varying number of parties, with 4 and 16 nodes. As can be seen, the total running time grows quadratically as a function of the number of parties. Figure 3 shows the linear growth of the execution time, when it is normalized by the number of participating parties.

We expect running time to behave as $A \cdot k^2 + B \cdot k \cdot n$ 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 and message sizes. This means that at some point the growth should become closer to linear in the number of parties. However, note that the size of the read messages may grow with the number of parties. Thus, although the communication seems linear in n when k is small, it may actually grow in a super-linear rate.

We also tested the effect of the number of White-City nodes on computation time, when the number of parties is constant. The results are presented in Appendix A. As expected, when $k << n$, adding WC node have a smaller effect on performance, while making the system more robust.

Finally, figure 4 presents computation times for both DKG and co-signing a message by all parties. In this protocol, DKG is complete within a single round, while it takes 4 rounds to complete signing.

At the moment, the largest documented MPC implementations are 128 parties computing AES encryption [43], 256 parties computing ECDSA signatures [25] and 500 parties computing mean and variance over secret inputs [6]. All demonstrated in academic literature only. With our system, one can take any of these MPC protocols and run it in an model that better simulates real world scenarios. We believe that using our scheme, Massive MPC will become possible for real world applications based on the above examples.

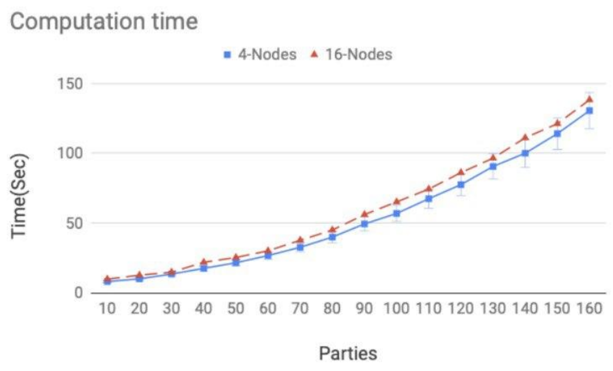

Fig. 2: Running time for varying number of parties

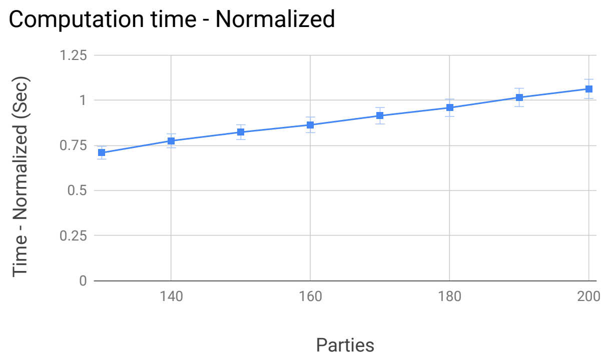

Fig. 3: Running time, normalized by the number of parties. The number of nodes is 16.

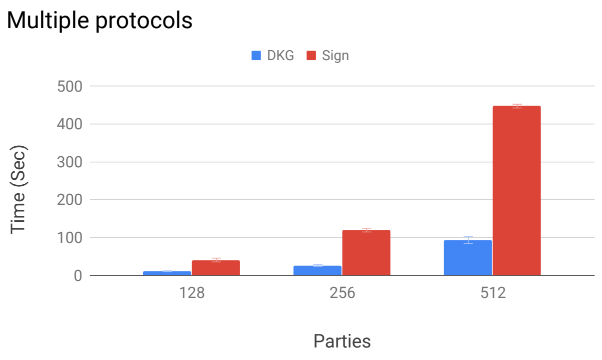

Fig. 4: Comparing DKG and signing protocols using White-City. The number of nodes is 16.

VII. Towards an Economic Model for the Nodes

From a game theoretic prospective, while it is clear what utility the parties gain with our scheme, it is unclear what incentivizes the nodes to prefer the strategy of participating and running the network in the most efficient way. Clearly, this issue does not occur when parties also play the role of nodes. However, when the two layers are separated, the parties should compensate the nodes, provided the nodes can produce a proof of work and in a way which will be profitable given the costs of running a node. 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 an 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.

A first and partial solution would be to use a public ledger to store and release the payments to the nodes. As BFT SMR reduces to an application on top of the public ledger, e.g., a smart contact on Ethereum as in [42], we can leverage the same infrastructure to deploy the incentive mechanism.

Towards achieving this, we introduce a Judge Functionality $\mathcal{F}_J$ in Functionality 7.1. $\mathcal{F}_J$ 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. To simplify the solution we make the following 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. The second assumption removes the potential complexity of parties that are not interested in the output or fair execution. We deffer the treatment of different client strategies to future work. Finally, the third assumption is coherent with 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.

FUNCTIONALITY 7.1 (the Judge Functionality):

Functionality $\mathcal{F}_J$ works with parties $P_1, \ldots, P_n$ and nodes $\mathcal{P}_1, \ldots, \mathcal{P}_k$. The functionality maintains a state $S^{\{i\}}$ for each $i \in [k]$. $S_r^{\{i\}}$ is the state at round r as received from node $\mathcal{P}_i$, which is initialized with an empty string. $S^{\{i\}} = \cup_r S_r^{\{i\}}$

Auxiliary Input: A mapping η between operations (send, store etc.) and costs. A 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 $\mathcal{P}_1, \ldots, \mathcal{P}_k$ balance.
- Upon receiving $\mathsf{UpdateState}(pid, state, r)$ from some node $\mathcal{P}_i$, if $state$ is signed by the nodes, $r \leq \rho$ and $S_r^{\{i\}} \subset state$, then update $S_r^{\{i\}} \leftarrow state$.
- Upon Receiving $\mathsf{Blame}(pid, transcript, \mathcal{P}_i)$ from some party P_j, compare $transcript$ to $S^{\{i\}}$. If $transcript \subseteq S^{\{i\}}$, send $S^{\{i\}}$ to P_j. Otherwise, if $transcript$ is signed by the nodes, $\eta(S^{\{i\}}) < \beta$ and $transcript \nsubseteq S^{\{i\}}$, call **Punish** with input $\mathcal{P}_i$.

Based on the third assumption we chose to operate in a *Gas* model [45] 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 which will use it to pay the nodes for computation. In functionality 7.1 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. If no dispute has occurred, then 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 in our construction). 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 has two options: If the party is right, then the Judge will punish the accused node using a predicate called **Punish** we keep abstract. If the party is wrong, then the Judge will provide the most updated state. To receive eventually the payment, it is in the nodes best interest to update the Judge constantly when state is updated. They can do it via the interface $\mathsf{UpdateState}$. 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 a node $\mathcal{P}_i$ under several conditions: (i) The input transcript is authenticated. (ii) The input transcript includes the Judge's internal state $S^{\{i\}}$. (iii) 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. 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 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 describe 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 punishes using a reputation penalty for registered nodes. A full analysis is out of scope for this paper. Note that this solution is only partial and still has some of the shortcomings discussed before. Most notably, network fees are not related to the actual resource consumption of the nodes and sharing the ledger with other applications may increase the cost of doing computation on the ledger. Nevertheless, it incentives the nodes to update the state and provide the correct state to the parties.

REFERENCES

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

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

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

[4] 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.

[5] Marcin Andrychowicz and Stefan Dziembowski. Pow-based distributed cryptography with no trusted setup. In *Advances in Cryptology - CRYPTO 2015 - 35th Annual Cryptology Conference, Santa Barbara, CA, USA, August 16-20, 2015, Proceedings, Part II*, pages 379–399, 2015.

[6] 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.

[7] Donald Beaver, Silvio Micali, and Phillip Rogaway. The round complexity of secure protocols (extended abstract). In *Proceedings of the 22nd Annual ACM Symposium on Theory of Computing, May 13-17, 1990, Baltimore, Maryland, USA*, pages 503–513, 1990.

[8] Michael Ben-Or, Shafi Goldwasser, and Avi Wigderson. Completeness theorems for non-cryptographic fault-tolerant distributed computation (extended abstract). In *Proceedings of the 20th Annual ACM Symposium on Theory of Computing, May 2-4, 1988, Chicago, Illinois, USA*, pages 1–10, 1988.

[9] 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.

[10] 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.

[11] 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.

[12] 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.

[13] 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.

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

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

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

[17] Ran Canetti. Universally composable security: A new paradigm for cryptographic protocols. In *42nd Annual Symposium on Foundations of Computer Science, FOCS 2001, 14-17 October 2001, Las Vegas, Nevada, USA*, pages 136–145, 2001.

[18] Ran Canetti and Hugo Krawczyk. Universally composable notions of key exchange and secure channels. In *Advances in Cryptology - EUROCRYPT 2002, International Conference on the Theory and Applications of Cryptographic Techniques, Amsterdam, The Netherlands, April 28 - May 2, 2002, Proceedings*, pages 337–351, 2002.

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

[20] David Chaum, Claude Crépeau, and Ivan Damgård. Multiparty unconditionally secure protocols (extended abstract). In *Proceedings of the 20th Annual ACM Symposium on Theory of Computing, May 2-4, 1988, Chicago, Illinois, USA*, pages 11–19, 1988.

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

[22] Megan Chen, Carmit Hazay, Yuval Ishai, Yuriy Kashnikov, Daniele Micciancio, Tarik Riviere, Abhi Shelat, Muthuramakrishnan Venkitasubramaniam, and Ruihan Wang. Diogenes: Lightweight scalable RSA modulus generation with a dishonest majority. *IACR Cryptol. ePrint Arch.*, 2020:374, 2020.

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

[24] Katriel Cohn-Gordon, Cas J. F. Cremers, and Luke Garratt. On post-compromise security. In *IEEE 29th Computer Security Foundations Symposium, CSF 2016, Lisbon, Portugal, June 27 - July 1, 2016*, pages 164–178, 2016.

[25] 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.

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

[27] Cynthia Dwork and Moni Naor. Pricing via processing or combatting junk mail. In *Advances in Cryptology - CRYPTO '92, 12th Annual International Cryptology Conference, Santa Barbara, California, USA, August 16-20, 1992, Proceedings*, pages 139–147, 1992.

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

[29] Hongmin Gao, Zhaofeng Ma, Shoushan Luo, and Zhen Wang. Bfr-mpc: A blockchain-based fair and robust multi-party computation scheme. *IEEE Access*, 7:110439–110450, 2019.

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

[31] Oded Goldreich, Silvio Micali, and Avi Wigderson. How to play any mental game or A completeness theorem for protocols with honest majority. In *Proceedings of the 19th Annual ACM Symposium on Theory of Computing, 1987, New York, New York, USA*, pages 218–229, 1987.

[32] 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.

[33] 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.

[34] 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.

[35] Jonathan Katz, Andrew Miller, and Elaine Shi. Pseudonymous secure computation from time-lock puzzles. *IACR Cryptology ePrint Archive*, 2014:857, 2014.

[36] Brian A. LaMacchia, Kristin E. Lauter, and Anton Mityagin. Stronger security of authenticated key exchange. In *Provable Security, First International Conference, ProvSec 2007, Wollongong, Australia, November 1-2, 2007, Proceedings*, pages 1–16, 2007.

[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 multi-party 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] Ralph C. Merkle. A digital signature based on a conventional encryption function. In *Advances in Cryptology - CRYPTO '87, A Conference on the Theory and Applications of Cryptographic Techniques, Santa Barbara, California, USA, August 16-20, 1987, Proceedings*, pages 369–378, 1987.

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

[43] 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.

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

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

[46] Andrew Chi-Chih Yao. How to generate and exchange secrets (extended abstract). In *27th Annual Symposium on Foundations of Computer Science, Toronto, Canada, 27-29 October 1986*, pages 162–167, 1986.

[47] 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.

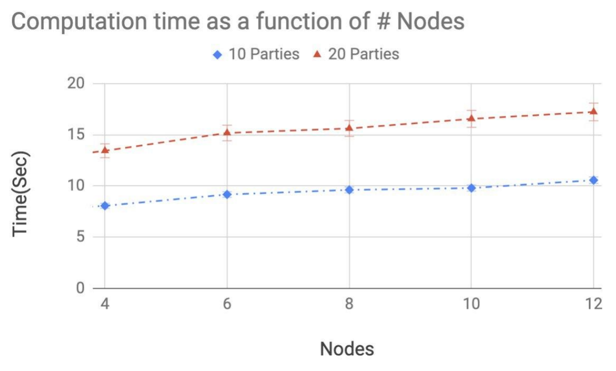

Fig. 5: Computation time for varying number of nodes

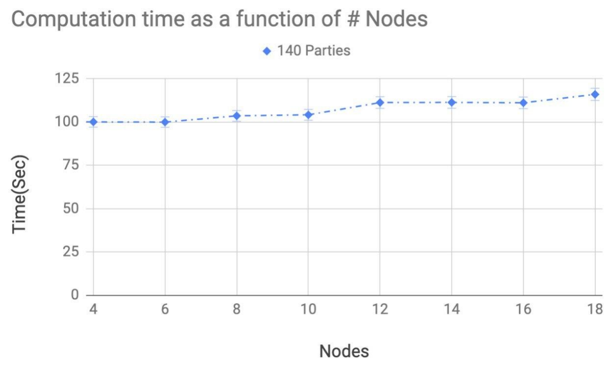

Fig. 6: Computation time for varying number of nodes and 140 parties

Appendix

A. More Experimental Results

In Figure 5 we show the total running time for signing a message as a function of the number of nodes, for 10 and 20 MPC parties. In Figure 6 we show the same, but for 100 MPC parties. The main observation is that increasing the number of nodes has an effect on performance. However, the slope is steeper when the number of parties is small. This implies that, as one would expect, when $k << n$, adding WC node have a smaller effect on performance, while making the system more secure and robust.