MPC With Delayed Parties Over Star-Like Networks

Mariana Gama[1] , Emad Heydari Beni[1,2] , Emmanuela Orsini[3] , Nigel P. Smart[1,4] , and Oliver Zajonc[1]

[1] COSIC, KU Leuven, Leuven, Belgium.
[2] Nokia Bell Labs, Antwerp, Belgium.
[3] Bocconi University, Milano, Italy.
[4] Zama Inc., Paris, France.
mariana.botelhodagama@kuleuven.be, emad.heydari_beni@nokia-bell-labs.com,
emmanuela.orsini@unibocconi.it, nigel.smart@kuleuven.be,
oliver.zajonc@esat.kuleuven.be

Abstract. This paper examines multi-party computation protocols in the presence of two major constraints commonly encountered in deployed systems. Firstly, we consider the situation where the parties are connected not by direct point-to-point connections, but by a star-like topology with a few central post-office style relays. Secondly, we consider MPC protocols with a strong honest majority ($t \ll n/2$) in which we have stragglers (some parties are progressing slower than others). We model stragglers by allowing the adversary to delay messages to and from some parties for a given length of time.

We first prove that having only a single honest relay is enough to ensure consensus of the messages sent within a protocol; then, we show that special care must be taken to describe multiplication protocols in the case of relays and stragglers; finally, we present an efficient honest-majority MPC protocol which can be run ontop of the relays and which provides active-security with abort in the case of a strong honest majority, even when run with stragglers. We back up our protocol presentation with both experimental evaluations and simulations of the effect of the relays and delays on our protocol.

Table of Contents

1 Introduction

Multi-Party Computation (MPC) allows a set of mutually distrusting parties to compute a function of their joint private inputs, without revealing anything about the inputs bar what can be deduced from any output of the function. MPC is now practical for a number of use-cases, it is becoming increasingly deployed in special niche applications, and much research work is now focused on extending the application space beyond these specific use-cases.

While MPC has been studied in a variety of different settings, most of the protocols are based on strong assumptions, such as the existence of direct fast communication channels between each pair of computing parties, fully synchronous communication channels and a static set of parties which is progress through the protocol execution at the same speed.

Network topology. In almost all academic works, and almost all academic implementations of MPC, the computing parties $\{\mathcal{P}_1, \ldots, \mathcal{P}_n\}$ are all connected with each other by dedicated connections, thus if we have n parties, this requires $n \cdot (n-1)$ uni-directional channels, often realised using $n \cdot (n-1)/2$ bi-directional TLS connections. However, in commercial applications this is not practical, as it implies each commercial entity enables $n-1$ external connections per MPC calculation. This can be problematic, as a corporate network is often locked down to an extent that creating new connections on new ports is frowned upon by the IT department.

One solution to this problem is to route all messages via a relay, so that communication operates in a star-like pattern. Indeed, this was proposed by the company ZenGo in their white paper describing the White-City protocol [Zen20]. This protocol proposes a number of such relays (aimed to protect against adversarial behaviour) which maintain consistency via a consensus protocol between them. Each relay $\mathcal{R}$ does not (necessarily) need to be one of the computing parties, it simply acts as a message transmission conduit between $\mathcal{P}_i$ and $\mathcal{P}_j$, for each pair (i, j). This means that each party $\mathcal{P}_i$ only needs to maintain a single connection to the relay $\mathcal{R}$. Communication can be kept private from the relay by end-to-end encryption between $\mathcal{P}_i$ and $\mathcal{P}_j$, and communication on the links between $\mathcal{P}_i$ and $\mathcal{R}$ can be ensured to be authentic via the use of message authentication codes. Other than being more practical, another advantage of having one (or more) relay node is that this provides a business model for companies to supply MPC services to clients: the relay node is providing the MPC service, for clients to connect to, and it can also act as a broker in brokering relationships between parties who desire to compute some joint function on their input. By charging for the usage of the relay node the companies can obtain revenue for providing the service.

In this work, we propose a relay model similar to Amazon SQS (Simple Queue Service) where messages are sent to a server and receivers retrieve and delete them. However, in our model, multiple relays are utilized to ensure security guarantees, whereas Amazon uses multiple servers for quality-of-service purposes in its highly distributed system. In this work, we use a similar approach, but the relay model we envision is similar to the Amazon SQS (Simple Queue Service) provided by AWS. In SQS, messages are sent to the SQS server by a sender, and then receivers poll the SQS server to pick up their messages that they can delete afterwards. The SQS queues have retention policies, meaning that if a message is not processed for a number of days, it will be removed. Amazon SQS preserves the message even after the consumer processes it. Since SQS is a highly distributed system, we cannot be confident that the receiver picked up the message successfully. Therefore, it is the receiver itself that must tell SQS to remove the message from the queue after receiving and processing it. Moreover, Amazon SQS persists several replicas of each message across the SQS cluster on separate servers to offer redundancy and high availability. In our relay model, we will follow a similar approach, For example, it could be that one of the replica servers keeps a message replica that might be unreachable due to networking issues. Whilst this use of multiple servers by Amazon is for quality-of-service reasons, in our work we utilize multiple relays in order to provide security guarantees.

Dynamic Computation with Delays. In common with most secure protocols, one works (ideally) in a Dolev-Yao model [DY81] in which the adversary is able to control the messages which are sent between parties, for example by replacing, dropping or placing messages out-of-order. In a real network, this is relatively hard for an adversary to do, thus most (practical) MPC work assumes that if party $\mathcal{P}_i$ sends a message to party $\mathcal{P}_j$ then such a message will *eventually* get from the source to the destination, and messages will be delivered in order. In a real computer network, such as the internet, the latter is a valid assumption as the exact route taken by messages is often unknown before the message is sent, and underlying network protocols provide the guarantee that messages arrive in order. However, if the "network" has a single bottleneck of a relay $\mathcal{R}$, then an adversary who controls the relay can mount trivial attacks which break the assumption that a message will always get through. In addition, modelling the system as synchronous, with a publicly known upper bound (*time-out*) on message latency, in practice is either very difficult to achieve or can severely impact communication speed, for example, if a large time-out upper bound is set so to ensure that all the messages from all the parties are delivered.

1.1 Our Contributions

In this work, we consider both these aspects and define an MPC protocol in presence of relay nodes which ensures some kind of robustness against delays, without relying on large time-out bounds. Our goal is to design a concretely efficient protocol based on more realistic network assumptions.

Concretely, we can summarize our main results as follows: 1) We formalize our communication model in the UC framework by giving an ideal functionality, called $\mathcal{F}_{\mathsf{SecureRobustRelay}}$, which defines a network where all communication occurs through relay nodes in a star-like topology and in the presence of a *δ-delaying active adversary* which can arbitrarily delay a party's execution of a command for up to δ rounds. In addition, we provide a protocol implementing this ideal functionality. 2) We give a generic secret sharing based MPC protocol secure against a δ-delaying active adversary that can proceed at the speed of the fastest parties. Our protocol makes use of an ideal functionality $\mathcal{F}_{\mathsf{Mult}}$ to evaluate multiplication gates. 3) We instantiate $\mathcal{F}_{\mathsf{Mult}}$ and show that the efficiency of the resulting protocol is comparable to that of the most efficient protocols in the setting of an honest majority which assume point-to-point channels between each pair of parties and without delays. In particular, our protocol achieves $O(n|C|)$ communication and takes advantage of the star-like network topology which allows to implement broadcast communication essentially for free. In addition, we give a detailed description of related protocols and highlight the differences with our approach. 4) Finally, we implement the communication network with relay nodes in pure Rust and present experimental results comparing its performance to that of direct TLS connections between the parties and showing the practicality of our star-like topology. Additionally, we explore the behaviour of our network under different settings and the performance of the MPC protocol built on top of the relays.

We now describe our contributions and techniques in greater detail.

Relays Nodes/Star-Like Network Topology. We consider a star-like topology network and build a model for relay nodes. We demonstrate that it is possible to remove the problem of adversarial control of the relay node by providing a set of r relay nodes $\{\mathcal{R}_1, \ldots, \mathcal{R}_r\}$, instead of just one, such that quality of service is maintained, after an initial key agreement phase, as long as at *least one* relay node is honest. Compared to the White-City protocol, our setting requires

consensus only to set the communication channels (where pairwise keys are distributed), offering a significant advantage compared to other protocols, like White-City, which requires consensus at every interaction. Thus, once authenticated connections are established, and unlike the White-City protocol, we do not require an expensive consensus protocol to be run between the relays. However, this comes at the expense of requiring each party $\mathcal{P}_i$ to maintain authenticated connections to the r relay nodes $\mathcal{R}_j$. We provide more comparison with [Zen20] is given in Section 1.2.

In practice, the value of r can be much less than the number of MPC parties n. For example, one may allow for $r = 2$ and have the two relay nodes provided by two different companies or servers. In such a situation, the adversary can corrupt one out of the $r = 2$ relay nodes and we still maintain security.
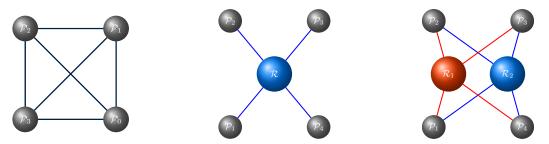

Fig. 1. Comparison between different topology networks with 4 computing parties $\mathcal{P}_1, \mathcal{P}_2, \mathcal{P}_3, \mathcal{P}_4$: a full network topology, a star-like network with 1 relay node and a replicated star-like topology with 2 relays.

More formally, we will model a relay node as an ideal functionality, and have the computing parties $\mathcal{P}_1, \ldots, \mathcal{P}_n$ connected with each other via a *replicated star-like network* with r relays $\mathcal{R}_1, \ldots, \mathcal{R}_r$, such that each $\mathcal{P}_i$ is connected only to the relays. The adversary, in practice, has full control of the communication, through point-to-point, secure authenticated channels, in that it can read (note not modify) messages sent between honest parties. In Figure 1, we graphically compare our model with a full-network topology and a more classical star-like topology. Notice the relays are not connected to each other and they maintain an internal state that is updated when interacting with parties.

The benefit of this network model is twofold: a more realistic communication model and low communication complexity in the MPC evaluation, since it allows us to reduce the communication between the computing parties, in that they only need to communicate with the central nodes. We will expand on this below.

Modelling Delays. We consider protocols where one could have a large number n of computing parties, some of which may be statically corrupt, over a network topology as described above with $r \ll n$. In such a situation it can be the case that even honest parties $\mathcal{P}_i$ may occasionally drop out of the computation and then come back or be suffering from some kind of delays in the communication. This could be for legitimate reasons, the need for something to be patched in the organization, a simple reboot, or it could be via adversarial behaviour, i.e. the adversary temporarily stops the given party from being part of the computation via a DoS attack, for example. We note that in these situations the adversary does not take control of the party, instead we are still in a static and not adaptive security model, but the adversary can actively make the party drop out of the computation for a while (a time interval we denote by δ).

We ensure that our relays, and the MPC protocol we run on top, can cope with a party dropping out of a computation and then returning to it. This is why we require that the relays must not simply act as a store-and-forward postal service, but must maintain some state in order to allow a party to rejoin and recover messages which they have not received.

More formally, we assume a synchronous network. This means that there is a publicly-known upper bound on message delays which allow the parties to follow the protocol specifications based on time. Therefore, the communication proceeds in rounds, each taking a fixed amount of time, and such that all the messages sent at the beginning of a certain round are delivered within the same round. However, we give the adversary the possibility of partially control the scheduling of the delivery of all messages. Concretely, we allow the adversary to choose whether a specific command is responded to, or not, by allowing a *delay*, i.e., the adversary can prevent the execution of a command for at most δ rounds. Clearly, without restriction, the adversary would be able to mount an indefinite denial of service attack but, since our usage of this ability is to model the situation where a party goes temporarily offline for a short period, we admit only *bounded delays*. In addition, the delays are *local*, i.e., it applies to a single party $\mathcal{P}_i$ and it applies to all the messages passed between $\mathcal{P}_i$ and all the relays $\mathcal{R}_1, \ldots, \mathcal{R}_r$.

We define an ideal functionality $\mathcal{F}_{\mathsf{SecureRobustRelay}}$ modelling such a network with relays and delays and describe a protocol implementing it.

Efficient MPC with Stragglers. A line of research [BJMS20, CGG+21, GHK+21, DEP21, FHM98, RS22, AHKP22, ANOS22], motivated by concrete applications, have proposed MPC protocols supporting a more dynamic form of participation, with parties that can join and/or leave the computation. Most of these protocols (see Section 1.2 for a short description of these works) rely on committees to carry on the computation.

We describe an MPC protocol which allows for parties to recover from dropped messages without the need for either the relays to maintain a list of *all* messages ever sent, or the parties restart the computation from scratch. However, we adopt a different approach compared to other works. Our protocol will proceed without waiting for all parties' message to be delivered in each round, but rather at the speed of the fastest parties. In particular, a party will progress through the MPC computation at its own pace, essentially stopping if a delay is activated. Thus, it can be the case that different parties progress through the MPC protocol at different rates, a bit like an asynchronous MPC protocol, even though the underlying communication model is synchronous.

Our main goal is to achieve efficiency by reducing the time-out bound of rounds and make the whole protocol proceed at the peace of the fastest parties. We note that a similar approach was also taken by Benhamouda et al. [BBG+21]. In this latter paper is defined the notion of *stragglers resilience*. Our security goal is similar, and we will give a detailed comparison both in techniques and efficiency between this and our work at the end of this section.

MPC Techniques. We describe an MPC protocol with computing parties $\mathcal{P}_1, \ldots, \mathcal{P}_n$ and relays $\mathcal{R}_1, \ldots, \mathcal{R}_r$, where a malicious static adversary can corrupt up to t_p computing parties and t_r relays and control the delays of an *arbitrary number* of computing parties, albeit for a limited number of rounds. We provide active *security with abort* in the case of a strong honest majority, i.e. $t_p/2 \ll n/2$.

Our protocol is based on a degree-t Shamir secret sharing scheme, with $t_p \le t < n/2$, and can proceed through a computation at the speed of the fastest $2 \cdot t + 1$ parties (which could include the t_p dishonest ones). This is possible as the fast parties can rely on the relays to act as a 'storage' mechanism to allow for the slower parties to catch up. We can also easily bound, and try to limit,

the state size which needs to be stored by the relays as a function of the multiplicative width and depth of the function being computed, and the number of computing parties.

We first present a generic secret-shared based protocol which makes use of a multiplication functionality, $\mathcal{F}_{\mathsf{Mult}}$, to evaluate multiplication gates. The functionality $\mathcal{F}_{\mathsf{Mult}}$ can be instantiated with a multiplication protocol that is secure up to additive attacks [GIP$^+$14]. As observed in prior work, some of the most efficient passively secure multiplication protocol [Mau06, DN07] are actually actively secure up to additive attacks in normal networks, or in networks without a super-honest majority. Roughly, this means that the only thing a fully malicious adversary can do is to add fixed values to the output of multiplication gates, and to the output of the computation. We show that our basic protocol is secure up to additive internal attacks (additive attacks mounted on the internal wires of the circuit) in the $\mathcal{F}_{\mathsf{SecureRobustRelay}}$-hybrid. Notice to allow input completeness, i.e. all honest parties' input being included in the computation, we assume that no delays occur in the input phase, or alternatively, we could make all the parties wait until they receive all the $n-1$ input messages from all the other parties.

We then compile this protocol to achieve active security with abort. A standard strategy to do this is to first run the basic passively secure protocol, then add a verification step aimed to check that all the multiplications were correctly done and finally reconstruct the output if the check passed. However, this would imply storing very large states on our relays. Hence, we use the same approach of Chida et al. [CGH$^+$18], also used in Fluid MPC [CGG$^+$21] and LeMans [RS22], and perform two computations of the circuit, one on secret shared values $\langle x \rangle$ and the other one with randomised versions $\langle \Delta \cdot x \rangle$ of the actual values. To avoid maintaining large states, we proceed like in FluidMPC and incrementally compute the checking equation during the computation.

We adopt the framework of circuit compilation [GIP$^+$14, CGG$^+$21], and compile the circuit C to be evaluated into a new circuit $\tilde{C}$, called a *robust circuit*, such that when $\tilde{C}$ is evaluated using a passively secure MPC protocol which is secure up to additive attacks, it results in a protocol for evaluating the original circuit C which is actively secure with abort. This view point allows us to bound the state which needs to be saved by the relays in our protocol.

We note that our notion of state is different from that in Fluid MPC, where the state needed to be transferred is the entire width of the circuit at any one layer, whereas in our situation the state is only the part of the width related to multiplication gates at that layer. Thus we adopt a slightly different notion of robust circuit, and associated compiler. This also implies that, although keeping small states is an important task, it is not as crucial as in committee-based protocols because we do not need to pass them from one committee to another. This gives rise to trade-offs between communication and state-complexity.

As a last remark, even if the general strategy we adopt in our MPC protocol is similar to that used in other works, adapting it to our setting and trying to maintain low complexity requires a careful design of the basic building blocks. For example, we show in Section 4.3 that if we instantiate $\mathcal{F}_{\mathsf{Mult}}$ with the widely used Maurer's multiplication protocol [Mau03], our general construction does not work anymore. Instead, we provide a simple and efficient instantiation of $\mathcal{F}_{\mathsf{Mult}}$ using a variant of the multiplicative protocol given by Damgård and Nielsen [DN07] (DN protocol), showing that the security of the multiplication and resulting MPC protocol relies on the robustness of $\mathcal{F}_{\mathsf{SecureRobustRelay}}$. In particular, our network allows a single party $\mathcal{P}_i$ to send a value to all the computing parties in a single command, by simply sending the value to all the relays. This means that we can remove the "king" from the DN protocol, and instead utilize a broadcast mechanism to allow all parties to essentially be the king. Since at least one relay is honest, this broadcast

mechanism comes "for free" and ensures that corrupt parties are forced to send the same value to all honest parties. This simple observation enables us to prevent the *double dipping attack* [GLS19, FL19] on the DN protocol.

Notice that the fact that it is simpler – and more efficient – for a party to communicate in a broadcast manner, as opposed to a point-to-point manner, via the relays means that traditional notions of communication complexity of protocols may not apply, since, traditionally a broadcast is considered more expensive than point-to-point communication.

Finally, we outline possible optimizations to our basic construction and give an estimation of the complexity of different strategies. In particular, we show that our main approach roughly match the communication complexity of [CGH+18], achieving linear communication and a concrete amortized communication cost of 12/13 field elements per multiplication gate per party. If we allow the use of PRGs this costs goes down to 8/9 field elements per party. We also sketch how to further reduce the communication costs to 6 (or 4 with PRGs) field elements per party using techniques from [GS20, GSZ20, BBC+19].

Implementation. While the use of relays results in a more realistic network, it might introduce additional communication. To evaluate the performance of our network topology, we implemented protocol $\Pi_{\mathsf{SecureRobustRelay}}$, presented in Figure 9, in pure Rust, and compared its performance to that of direct communication between two parties. The results in Section 7 show that using relays has no noticeable impact on the performance when sending up to 2^{19} 16-byte messages. Note that, a 16-byte message can correspond to a finite field element of around 128 bits in size, and thus can represent native data-type for any MPC computation layered ontop.

In addition, we analyse how to best configure the network in order to optimise the communication runtimes. First, we conclude that although erasing each message in the relays immediately after retrieval results in slower communication, erasing them in batches achieves similar performance to direct communication. On the other hand, never deleting messages not only means the relays might run out of memory, but also turns out to be slower since the relays must iterate through several messages when answering message request. Second, we show that a network with more relays has increased security while adding only a small overhead to the communication time.

A second set of experiments is dedicated to evaluate the performance of the MPC protocol. We measure the number of multiplications performed in one second both when parties all run at the same speed and when some of them are slower than others.

1.2 Other Related Work

Star-like Topology. Motivated both by practical issues, like the incompatibility of standard communication model used in MPC protocols with many real-world applications, and theoretical questions, like achievable security in networks with reduced interaction, some works have studied feasibility and efficiency of secure MPC in more general networks.

The paper closest to ours is the White-City protocol [Zen20], already mentioned before, which builds a star-like infrastructure with relay nodes. The underlying application is assumed to be a short MPC protocol, such as a threshold EC-DSA signing for use in blockchain applications. Thus, in the White-City protocol, the relay nodes do not need to worry about maintaining a large state, and thus the system does not enable mechanisms to keep the number of messages stored to a minimum. Unlike our work, the White-City protocol describes a methodology for interested parties, who wish to engage in an MPC computation to enrol in the system. For us, we assume

that such parties already come equipped with a certified public key. This distinction is, again, due to the use-case which White-City is focused on. Their registration system could be adapted to our work relatively easily. In common with our approach, White-City allows for parties to be offline and re-joining the computation, assuming a partially synchronous network where some of the sent messages may be lost. However, to ensure security, in White-City the relays are all connected with each other and constantly run a consensus protocol to maintain the state of the computation. In our model, the relays are not connected with each other and we prove that the consensus protocol is not necessary if at least one of the relays is honest.

Another recent work by Alon et al. [ANOS22] considers MPC over a star-like topology network with a single powerful server (Gulliver), playing the role of a single, central node, and n considerably less powerful users (Lilliputians). Thus, the model described in this work is different from ours in few aspects, mainly in the role played by the central node, that in [ANOS22], needs to do most of the heavy-lifting of the computation, and has the power to block messages between honest parties, while in our protocol this cannot happen if at least one relay is honest.

A different line of works [HLP11, HIJ$^+$16] studied the notion of security of MPC in different networks, and in particular in a star-like network. The goal of these works was mainly that of establishing feasility/infeseabilty results, and their model is different from the one considered in this paper since in our work we allow more than one single interaction between the computing parties and the central node.

Dynamic Participation. In Lazy MPC [BJMS20], as in our work, parties can leave the computation. These parties, called honest-but-lazy, are not assumed to be adversarial, but, unlike our work, they are not necessarily assumed to come back into the computation. Further, parties that leave the computation are not guaranteed to receive the output, and the protocol only achieves computational security. In our work we assume that a party who leaves, only leaves temporarily, and we wish to maintain enough state to enable them to return to the computation and 'catch up' with the others.

In You Only Speak Once (YOSO) [GHK$^+$21], the computation is split into "roles" which are distributed randomly among participating parties. Each of these roles consists of a single message. The assumption made here is that the adversary cannot target a party executing a role, if they do not know who it is, until after the role is completed. In this way the protocol ensure resilience against adaptive adversary for large scale MPC protocols. The adversary can, after a message is sent, take down the sending party for the rest of the computation. Parties removed from the computation in this way do not return, so the computation succeeding is defined as the collective receiving the output instead of any given party.

Motivated by the nice features of YOSO, in particular its resilience against adaptive adversaries, Acharya et al. in [AHKP22] recently described a YOSO-style large scale MPC protocol with much better performance compared to [GHK$^+$21], and also additional features.

Another relevant work is [CGG$^+$21] that introduces the *fluid setting*, where only a dynamically-changing subset of parties is involved in the protocol computation. In particular, a subset of parties is called to be part of the *computing committee* only for a specific number of rounds. This implies that, once a computing committee finishes its job, it has to transfer the entire state of the computation to the next committee. The protocol described in this work achieves active security with abort with honest majority and requires $O(n^2|C|)$ communication. Our protocol is not committee-based and achieves linear communication complexity taking advantages of the star-like topology with relays.

Similarly to our work, Phoenix [DEP21] deals with parties dropping out of the computation and rejoining. The adversary chooses some parties to be offline in each round of a computation.

However, unlike our scheme, Phoenix limits the number of parties that can be offline in a round. From such an unstable network, the authors build a standard stable network, assuming a certain threshold of honest parties is online in any given round. From this they can then build an MPC protocol ontop of the unstable network. In Phoenix, the protocol places a limit on the length of time a party may be kept offline by the adversary, whereas in our protocol this limit is not a protocol construct but a construct of the memory available in the relays.

There are works, for example [FHM98], which study the MPC setting in which the adversary, on top of being able to corrupt some number of parties, can cause some parties to fail and go offline. As in our model, parties which are online at one point in time do not know if another party is currently online or offline. However, parties that fail do not return to the computation in [FHM98], but they do in our work.

In [GPS19], a synchronous network is assumed in which, similar to our work and to Phoenix, parties may be offline and come back online later. The network is assumed to have the property that parties will receive all the messages sent to them while being offline. In our work, each party, once back online, can decide which messages sent to them retrieve. Unlike Phoenix, there are no assumptions made on the number of honest parties that are available in consecutive rounds. The protocol described in [GPS19] is a constant-round computationally secure protocol that requires honest-majority in each communication round.

Honest majority MPC There has been a huge amount of work that focuses on improving the efficiency of secure multi-party computation protocols in the honest-majority setting. Most of these works in the client-server model, where it is sufficient to only consider adversaries corrupting exactly t parties considering minimal honest majority with $n = 2t + 1$, and assume the existence of a private, synchronous point-to-point channel between every pair of parties. The main goal of these works is to reduce the *concrete* communication costs by allowing security with abort against active adversaries. While Genkin et al. [GIP+14] presented the first construction achieving the same *asymptotic* cost of state-of-the-art passively secure protocols ([DN07], DN protocol), the work of Chida et al. [CGH+18] and that of Nordholt and Veeningen [NV18] showed a concrete cost of 12 field elements per party for each multiplication gate, that is roughly twice the cost of the DN protocol. In [GS20, GSZ20, BBC+19], the authors presented a way to improve the communication efficiency of this type of protocols by using a new technique to check the correctness of multiplication gates achieving an amortized complexity of 6 field elements per multiplication gate, effectively matching the cost of DN. This result was further improved in [GLO+21], in which the communication of both the passive and active-secure with abort protocol is reduced to 4 elements per multiplication gate in the information-theoretical setting and to 2 in the computational setting.

The setting of a strong honest majority has not received much attention, with few exceptions [FL19, BBG+21]. If, on the one hand, the benefits of having more honest parties are obvious "in theory", this observation has not yet translated in concretely efficient constructions. For arbitrary number of parties, Furukawa and Lindell [FL19] described a 2/3-honest majority protocol which achieves the same efficiency of DN. We discuss about [BBG+21] extensively below.

MPC with Stragglers. The protocol of Benhamouda et al. [BBG+21] also considers security with abort in the strong honest majority setting and solves the stragglers problem by allowing the protocol to proceed at the speed of the $2 \cdot t + 1$ fastest parties. Compared to their work, our solution differs on some key aspects other than network setting, since we achieve stragglers resilience against node failures, while in [BBG+21] delays are caused by network channels.

First, we give a generic protocol where the functionality $\mathcal{F}_{\mathsf{Mult}}$ can be instantiated with any passively-secure multiplication protocol that is secure up to additive attack; whereas [BBG+21] gives a protocol based solely on the work of Damgård and Nielsen [DN07]. This choice seems indeed the most natural since Damgård and Nielsen is the most efficient MPC protocol in the presence of a minimal honest majority. However, in our setting, this choice comes with drawbacks. While the passively-secure Damgård and Nielsen protocol achieves privacy against a malicious adversary when $n = 2 \cdot t + 1$, this is no longer true if $n > 2 \cdot t + 1$. This was first observed in [GLS19, FL19]. The attack, also known as *double dipping attack*, exploits the redundancy of Shamir's secret sharing scheme in the presence of a strong honest majority and that can be used by corrupt parties when the masked result of a multiplication gate is open by the designated party (the king) to completely recover the input of the gate. To deal with this problem, [BBG+21] uses a solution based on a novel PRSS technique that however adds a certain overhead to the computation. Hence, this solution even if theoretical interesting, is not yet practical for a large number of parties.

The second issue rises as relying on a party playing the role of the king is not ideal in protocols with delays caused by node failures. If the king is delayed, then all parties are delayed because they cannot receive the shares necessary to continue the MPC evaluation.

We circumvent both these issues by removing the king from the protocol of Damgård and Nielsen, and instead utilize a broadcast mechanism to allow all parties to essentially be the king. This broadcast mechanism come "for free" (assuming one honest relay) in our network model, due to the `sendToAll` command of the robust relay functionality.

Another difference with our work is in the verification step necessary to achieve active security with abort. The paper [BBG+21] uses the technique described in [BBC+19] which only requires sublinear communication, but needs a large state to be stored. More importantly, to complete the check [BBG+21] assumes that all the messages delayed during the MPC evaluation arrived before the final check. This is not required in our protocol.

2 Preliminaries

For a set S, we denote by $a \leftarrow S$ the process of drawing a from S with a uniform distribution on the set S. If D is a probability distribution, we denote by $a \leftarrow D$ the process of drawing a with the given probability distribution. For a probabilistic algorithm A, we denote by $a \leftarrow A$ the process of assigning a the output of algorithm A, with the underlying probability distribution being determined by the random coins of A. We use $[n]$ to denote the set $\{1, \ldots, n\}$ and $\mathcal{P} = \{\mathcal{P}_1, \ldots, \mathcal{P}_n\}$ the set of parties.

2.1 Communication and Security Model

We summarize here our settings, as previously described. We assume $\mathcal{P} = \{\mathcal{P}_1, \ldots, \mathcal{P}_n\}$ computing parties and $\{\mathcal{R}_1, \ldots \mathcal{R}_r\}$ relay nodes. Parties are only connected with relays via authenticated, but not necessarily private, channels and not with each other and also the relays are not connected with each other. More formally, we prove security in the *authenticated-links model* (AM) [CK01], where the adversary can only deliver messages that were sent by parties and must deliver them unmodified. Relays maintain an internal state that is updated when interacting with parties.

We assume a synchronous network with a publicly-known upper bound (time-out) on message delays which allows the parties to follow protocol specifications based on time. Therefore, the communication proceeds in rounds, each taking a fixed amount of time.

We consider δ-*delaying malicious static adversaries* which are allowed to control up to $t_r \leq r-1$ relays and $t_p \leq t$ parties, where $n \geq 2t+1$. We also give the adversary the possibility of partially control the scheduling of the delivery of all messages by allowing it to arbitrarily delay parties by up to δ rounds. This means that each party can be delayed more than once during the computation, but each time only by δ rounds.

We consider security with *selective abort*, where the adversary receives the output and determines which honest parties will receive abort and which will receive their correct output.

2.2 Shamir Secret Sharing

Our protocols are built ontop of the specific secret sharing scheme due to Shamir [Sha79]. We will utilize a degree-t Shamir's secret sharing over a finite field $\mathbb{F}$ in most situations, and for our multiplication protocol a degree-$2 \cdot t$ sharing.

Shamir's sharings consists of two interactive algorithms. In the sharing phase a secret $x \in \mathbb{F}$ is shared amongst n parties $\mathcal{P}$ by the sharing party (the dealer) by defining a random degree-t polynomial $f_x(X) \in \mathbb{F}[X]$ whose constant term is x. Assuming $|\mathbb{F}| > n$ (this restriction can be remove by field extension in a standard manner), the integers $1, \ldots, n$ are mapped to distinct non-zero values $\alpha_1, \ldots, \alpha_n \in \mathbb{F}$, and each party $\mathcal{P}_i$ is given the share $x_i = f(\alpha_i) \in \mathbb{F}$. We denote such degree-$t$ sharing by $\langle x \rangle_t$. We will drop the index t when it is clear from the context. Let $\mathcal{M} \subseteq [n]$, we may write $\langle x \rangle_\mathcal{M}$ to denote the set of shares $\{x_i\}_{i \in \mathcal{M}}$.

The reconstruction algorithm rec takes as input a set of shares $\{x_i\}_{i \in \mathcal{M}}$, where $\mathcal{M} \subseteq [n]$ and outputs either the secret x or $\perp$, i.e. $\mathsf{rec}(\langle x \rangle_\mathcal{M}) = x$ or $\perp$. Given any subset $\mathcal{M} \subseteq [n]$ of size greater than t and valid sharings $x_i \in \mathbb{F}$, $i \in \mathcal{M}$, then there is a unique value x which can be reconstructed from $\langle x \rangle_\mathcal{M}$. We denote the recombination equation for the set $\mathcal{M}$ as

$$x = \sum_{i \in \mathcal{M}} r_i^\mathcal{M} \cdot x_i = \mathbf{r}^\mathcal{M} \cdot \mathbf{x}^\mathcal{M},$$

where $r_i^\mathcal{M}$ are constants which depend on the precise set $\mathcal{M}$, and $\mathbf{x}^\mathcal{M}$ is the share vector corresponding to the parties in the set $\mathcal{M}$. We say the vector $\mathbf{r}^\mathcal{M} \in \mathbb{F}^n$ is the *recombination vector* for the set $\mathcal{M}$. The existence of such a recombination vector follows from the existence of the Lagrange interpolation coefficients for polynomial interpolation.

Definition 2.1. *We say that a degree-t sharing $\langle x \rangle_t$ is* invalid *if exists a subset $\mathcal{M} \subseteq [n]$, $|\mathcal{M}| \geq t+1$, such that $\mathsf{rec}(\langle x \rangle_M) = \perp$; we say that $\langle x \rangle_t$ is* inconsistent *if exist two subsets $\mathcal{M}_1, \mathcal{M}_2$, with $t+1 \leq |\mathcal{M}_1|, |\mathcal{M}_2| < n$ such that $\mathsf{rec}(\langle x \rangle_{\mathcal{M}_1}) \neq \mathsf{rec}(\langle x \rangle_{\mathcal{M}_2})$.*

A degree-t Shamir secret sharing scheme can also be viewed as an $[n, t+1, n-t]$ Reed-Solomon code over $\mathbb{F}$, where $(f(\alpha_1), \ldots, f(\alpha_n))$ are codewords. This means that, even in presence of corrupt parties that lie about their share values, honest parties are still able to detect and (in same case) correct such errors if the number of corruptions are bounded. Concretely, let t_p be the number corrupted parties with $t_p \leq t$, then the code allows to detect the errors if $t_p < n - t$ and even correct them if $2t_p < n - t$. When $|\mathcal{M}| > 2 \cdot t$, and hence $t < n/2$, and the number t_p of dishonest parties is bounded by t, then we are guaranteed that the set $\mathcal{M}$ contains indexes of $t+1$ honest parties and they can either recover a *unique* value x, or will output $\perp$ indicating that the shares are invalid. This can be efficiently done using a parity check matrix $P^\mathcal{M}$ related to the code underlying the secret sharing scheme which, for a valid sharing, will satisfy

$$P^\mathcal{M} \cdot \mathbf{x}^\mathcal{M} = \mathbf{0},$$

12

where, again, $\mathbf{x}^{\mathcal{M}}$ is the share vector corresponding to the parties in the set $\mathcal{M}$.

When $|\mathcal{M}| > 3 \cdot t$, parties in $\mathcal{M}$ can robustly recover the secret x, and avoid the case of indicating the received shares are invalid; however this assumes that the honest parties have started with valid shares. We shall not use this latter property in this work.

Lemma 2.1. *[SW19] Given a set $\mathcal{M}$ of $d > 2 \cdot t$ parties, let $P^{\mathcal{M}}$ be the parity check matrix and $\mathbf{x}^{\mathcal{M}}$ be the share vector of a value x corresponding to the parties in $\mathcal{M}$. Let $\mathbf{e}$ be an error introduced to the shares of the parties controlled by the adversary. Then, one of the following holds:*

- $P^{\mathcal{M}} \cdot (\mathbf{x}^{\mathcal{M}} + \mathbf{e}) \neq 0$
- $(\mathbf{x}^{\mathcal{M}} + \mathbf{e})$ *is a sharing of x.*

Protocol Π_{Mult}

INPUT: $\langle x \rangle_t, \langle y \rangle_t$;   OUTPUT: $\langle z \rangle_t$

1. Parties locally compute $\langle x \cdot y \rangle_{2t} = \langle x \rangle_i \cdot \langle y \rangle_t$, i.e., each $\mathcal{P}_i$ computes $x_i \cdot y_i = v_i$.
2. Each party $\mathcal{P}_i$ acts as a dealer and produces a degree t sharing $\langle v_i \rangle_t$ of v_i. The shares are distributed to the other parties.
3. Each party $\mathcal{P}_i$ locally takes a subset $\mathcal{M}_i$ of size at least $2 \cdot t + 1$ of shares they have received in the previous step, including their own, and computes

$$v_i = \sum_{j \in \mathcal{M}_i} s_j^{\mathcal{M}_i} \cdot v_i,$$

where z_i is the ith shares of z.

Figure 2. The basic multiplication protocol for Shamir's secret sharing scheme

Properties of Shamir's secret sharing. Shamir's secret sharing scheme is a linear secret sharing scheme in that linear functions on share values can be locally applied, i.e., given $\langle x \rangle_t$ and $\langle y \rangle_t$ one can locally compute $\langle z \rangle_t = \alpha \cdot \langle x \rangle_t + \beta \cdot \langle y \rangle_t + \gamma$, for any constants $\alpha, \beta, \gamma \in \mathbb{F}$, by computing $z_i = \alpha \cdot x_i + \beta \cdot y_i + \gamma$. It is also what is called a *multiplicative* secret sharing scheme in the case when $t < n/2$. Given two sharings $\langle x \rangle$ and $\langle y \rangle$, with individual sharings $x_i = f_x(i)$ and $y_i = f_y(i)$ held by the parties $\mathcal{P}_i$, for two polynomials $f_x(X)$ and $f_y(X)$, and any set $\mathcal{M} \subset \{1, \ldots, n\}$ of size strictly greater than $2 \cdot t$ we have that

$$z = \sum_{i \in \mathcal{M}} s_i^{\mathcal{M}} \cdot x_i \cdot y_i.$$

We say that the vector $\mathbf{s}^{\mathcal{M}} \in \mathbb{F}^n$, where $s_i = 0$ if $i \notin \mathcal{M}$, is the *recombination vector for the Schur products*. By abuse of notation we also refer to $\mathbf{s}^{\mathcal{M}}$ as the vector length $|\mathcal{M}|$ where we drop all entries for which $i \notin \mathcal{M}$. The existence of such a recombination vector follows since $x_i \cdot y_i = f_z(i)$ for the polynomial $f_z(X) = f_x(X) \cdot f_y(X)$ of degree $2 \cdot t$, thus the values $x_i \cdot y_i$ are a Shamir sharing of the product z of degree $2 \cdot t$.

The fact it is linear and multiplicative means we have the (standard) protocol for multiplication [Mau06] described in Figure 2 in the case when $t < n/2$; often dubbed Maurer-multiplication. This

protocol, assuming the input sharings are valid, will produce a valid sharing of the product up to an additive error which adversarial parties may introduce. Note, that the protocol is one round (step 2) and that any subset $\mathcal{M}_i$ of size larger than $2 \cdot t + 1$ can be utilized in step 3. The total number of finite field elements transmitted in step 2 is $n \cdot (n-1)$.

Reducing communication using PRGs. Using a PRG the cost of step 2 per party is $(n-t-1)$ field elements [NV18]. Assuming that each pair of parties agreed on a common seed, that can be done by simply sending a value at the beginning of the protocol, then to distributed a Shamir's sharing of v_i, $\mathcal{P}_i$ first sends t random elements to the first t parties, which can be done with no interaction using a PRG, then it creates the remaining $n-t-1$ shares consistently with v_i and those t random values, and finally sends these $n-t-1$ values to the relevant parties.

2.3 Encryption

Our protocols will make use of both symmetric and public key encryption.

Symmetric key encryption algorithms will be denoted by a triple of probabilistic polynomial time algorithms $(\mathsf{KeyGen}, \mathsf{Enc_k}, \mathsf{Dec_k})$.

- $\mathsf{KeyGen}(1^\lambda)$: On input the security parameter λ, this algorithm outputs a key k drawn from a space $\mathcal{K}$.
- $\mathsf{Enc_k}(m, a)$: On input of a key k, a message m and associated data a this outputs a ciphertext ct.
- $\mathsf{Dec_k}(\mathsf{ct}, a)$: On input of a key k, a ciphertext ct and associated data a, this algorithm outputs a message m or the symbol $\perp$.

We adopt the standard definitions for correctness and security of such AEAD (Authenticated Encryption with Associated Data) schemes. In particular, we require the AEAD scheme to be IND-CCA and to be secure against forgeries, where AEAD-unforgeability is defined as follows.

Definition 2.2 (AEAD Unforgeability). *An* AEAD*-scheme (authenticated-encryption scheme with associated-data) $\mathcal{E}$ consists of three algorithms $(\mathsf{KeyGen}, \mathsf{Enc_k}, \mathsf{Dec_k})$ with the following properties. Let $\mathcal{A}$ be an adversary having access to an oracle $\mathcal{O}_{\mathsf{Enc}}$ for some key k, we say that $\mathcal{A}$ forges (for this key k) if it is able to output a ciphertext ct^* such that*

1. $\mathsf{Dec_k}(\mathsf{ct}^*) = m^* \neq \perp$;
2. $m^* \notin \mathcal{Q}$, where $\mathcal{Q}$ is the set of all queries that $\mathcal{A}$ asked its encryption oracle.

Public key schemes can be similarly defined using a triple of algorithms $(\mathsf{KeyGen}, \mathsf{Enc_{pk}}, \mathsf{Dec_{sk}})$, as follows:

- $\mathsf{KeyGen}(1^\lambda)$: On input the security parameter λ, this algorithm outputs a key pair $(\mathsf{pk}, \mathsf{sk})$.
- $\mathsf{Enc_{pk}}(m, a)$: On input of a public key pk and a message m, this outputs a ciphertext ct.
- $\mathsf{Dec_{sk}}(\mathsf{ct})$: On input of a private key sk and a ciphertext ct, this algorithm outputs a message m or the symbol $\perp$.

We assume the standard IND-CCA security definitions for public-key encryption. Whether a scheme is public key or secret key will be clear from the context, i.e., whether the associated key is written as k, pk or sk.

2.4 Internal Additive Attacks

Intuitively, an additive attack (Definition 2.3) is an attack which changes the value of a gate's output wire in the circuit by an additive value before the calculation is performed, i.e. blindly changing a wire from $f(x)$ to $f(x) + \delta_a$, where $f(x)$ is the function computed by the gate and δ_a is a value known by the adversary.

We recall the formal definition of an additive attack. One minor modification to the definition from [GIP$^+$14, CGG$^+$21] is due to the check in our protocol for opening a value, given in Figure 10, Section 4. In particular, since the output wires of a circuit will be checked via the underlying error-detection properties of the Shamir's secret sharing scheme, we restrict to an additive attack only related to the internal wires of a circuit and not to the output wires. We call such an attack "internal additive attack" in order to distinguish it from the standard notion of an additive attack (which also allows the adversary to add a known value to the output wires of the circuit).

Definition 2.3 (Internal Additive Attack). *Let $C : \mathbb{F}^n \to \mathbb{F}^k$ be a circuit. An additive attack A by an adversary on the evaluation of the circuit C assigns an element of $\mathbb{F}$ to each of the circuits' internal wires, i.e. a wire between two gates g^a and g^b. Let $A_{a,b}$ denote the value assigned by the attack to the internal wire between gates g^a and g^b. The additive attack changes the calculation of the circuit as follows: For each internal wire between gates g^a and g^b, the value $A_{a,b}$ is added to the wires value after the calculation of the output of gate g^a, but before the calculation of the gate g^b.*

Looking ahead, the protocol Π_{PMPC} we give in Section 5 will take the arithmetic circuit representation of the function to be evaluated and execute a standard secret-shared based MPC protocol, using some simple multiplication protocol, like the one described in Figure 2, to execute the multiplication gates. The result will be a protocol which is secure up to internal additive attacks.

We then compile Π_{PMPC} into a protocol which is actively secure with abort, i.e. which is secure even against internal additive attacks. This is done by applying the protocol which is secure up to internal additive attacks to a circuit which protects against such additive attacks. Thus our approach to proving security will be similar to that used by Genkin et al. [GIP$^+$14], Chida et al. [CGH$^+$18], and FluidMPC [CGG$^+$21].

However, in our situation, we also allow the adversary to delay messages, as explained in the next section, by a value of at most δ. Thus we need to execute the basic multiplication protocol in a manner which enables the fastest parties to complete the computation as soon as possible and, in addition, we need to bound the size of the state which needs to be stored by the relays.

2.5 Depth and Width of Randomized Arithmetic Circuits

We recall some useful definitions that we will need to describe our MPC protocol. To bound the size of the state which needs to be stored, we look in detail at the structure of the circuit being evaluated. We define our basic (randomized) circuit as follows.

Definition 2.4 ((Randomized) Arithmetic Circuit). *An arithmetic circuit C over a finite field $\mathbb{F}$ is a directed graph consisting of nodes made of input, linear[5], multiplication and output gates, and edges called wires. The nodes are such that*

- *Input gates have no incoming edges and one outgoing edge, they are labeled with a party number which indicates which party will provide that input.*

---

[5] Linear gates are those implementing addition, addition-by-constant and multiplication-by-constant operations

- *Linear gates may have arbitrary fan-in, and one output edge.*
- *Multiplication gates have two input and one output edge.*
- *Output gates have one input and one output edge, they are labelled with a party number which indicates which party will obtain that output.*

A randomized arithmetic circuit *allows an additional gate in its construction (called a random-input gate) with no input and a single outgoing edge, which emulates a truly random source. Thus randomized circuits can represent probabilistic functions, whereas non-randomized circuits represent deterministic functions.*

The FluidMPC protocol [CGG$^+$21] also needs to consider circuit structure in order to bound state, but this is the state which is passed from one committee to another at each stage. This state transfer leads them to the idea of a layered circuit. We do not have such a need to transfer state, thus our terminology differs in a number of ways. For example, we do not restrict from which depth an input wire for the gates at depth d comes. Due to the way our state is measured we make small changes to the way depth and width are calculated, for examoke in FluidMPC each gate (and not just multiplication gates) is important for the size of the state passed between committees. Thus multiplicative depth and width (as defined below) are the only values relevant to us. Given a standard circuits, as defined by Definition 2.4, we define the circuit depth and width as follows.

Definition 2.5 (Circuit Depth and Width). *Let C denote a randomized arithmetic circuit. We assign a depth to each wire and gate as follows:*

- *All input and random-input gates have depth zero.*
- *Each wire has depth the depth of the gate which produces that wire.*
- *Each linear gate has depth the maximum depth of the associated input wires.*
- *Each multiplication gate has depth one more than the maximum depth of the associated input wires.*
- *Output gates are all placed at the depth of their associated input wire.*

The depth of a circuit *is the maximum depth of all output gates in the circuit. The* width *at depth d of a circuit, w_d, is the number of multiplication and output gates of the given depth d. The* width of a circuit *is the maximum value of w_d across all possible depths d.*

3   Relays and Delays

In this section, we present the protocol $\Pi_{\mathsf{SecureRobustRelay}}$, which formally describes how parties and relays securely communicate in presence of bounded delays. The protocol is given in Figure 9. It implements the ideal functionality $\mathcal{F}_{\mathsf{SecureRobustRelay}}$, given in Figure 7 and Figure 8.

The protocol $\Pi_{\mathsf{SecureRobustRelay}}$ uses two main distinct building blocks, namely the functionalities $\mathcal{F}_{\mathsf{SingleRelay}}$ and $\mathcal{F}_{\mathsf{Delay}}$, that we briefly describe before introducing our main protocol. This approach allows to first introduce the topology of our network with relays and then add the possibility of adversarial delays.

3.1   A Single Relay

The functionality $\mathcal{F}_{\mathsf{SingleRelay}}$, described in Figure 3, captures all the interactions between parties $\mathcal{P}_1, \ldots, \mathcal{P}_n$ and a single relay $\mathcal{R}$. The functionality is described by six commands, other than the initialization command `Init`, as explained below.

<div style="border:1px solid">

Functionality $\mathcal{F}_{\mathsf{SingleRelay}}(\mathcal{R}, \mathcal{P}_1, \ldots, \mathcal{P}_n)$

This functionality runs with an adversary $\mathcal{S}$, a special party (denoted by $\mathcal{R}$), which is the relay, and n parties (denoted by $\mathcal{P}_1, \ldots, \mathcal{P}_n$). The functionality maintains pairwise counters $\tau_{i,j}$, for all $i \neq j$, and global counters $\tau_{i,\mathcal{P}}$ for all $i \in \mathcal{P}$, and variables $\epsilon_{\tau_{i,j}} \in \{\bot, 0, 1\}$, and $\epsilon_{\tau_{i,\mathcal{P},j}} \in \{\bot, 0, 1\}$.
Upon activation the functionality receives either $(-, \mathcal{P}_I)$ or $(\mathcal{R}, \mathcal{P}_I)$ from the adversary, where $\mathcal{P}_I \subset \{\mathcal{P}_1, \ldots, \mathcal{P}_n\}$ is the set of corrupt parties, indicating in the first case that the relay is honest, and the in second case that the relay is dishonest. We assume $|\mathcal{P}_I| \leq t_p$.

Init: On input (init) from all parties the functionality sets all the counters $\tau \leftarrow 0$ and $\epsilon \leftarrow \bot$.
Send: On input $(\mathtt{send}, \mathcal{R}, \mathcal{P}_i, \mathcal{P}_j, m)$ from $\mathcal{P}_i$,
    1. Send $(\mathtt{sent}, i, j, m)$ to $\mathcal{R}$ and increment $\tau_{i,j}$ by one.
    2. Set $m_{\tau_{i,j}} \leftarrow m$ and store it. Set $\epsilon_{\tau_{i,j}} \leftarrow 0$.
    3. Delete (i, j, m) from the store.
SendToAll: On input $(\mathtt{sendToAll}, \mathcal{R}, \mathcal{P}_i, \mathcal{P}, m)$ from $\mathcal{P}_i$,
    1. Send $(\mathtt{sendToAll}, i, \mathcal{P}, m)$ to $\mathcal{R}$ and increment $\tau_{i,\mathcal{P}}$ by one.
    2. Set $m_{\tau_{i,\mathcal{P}}} \leftarrow m$ and store it. Set $\epsilon_{\tau_{i,\mathcal{P},j}} \leftarrow 0, \forall j \in \mathcal{P} \setminus i$.
    3. Delete $(i, \mathcal{P}, m)$ from the store.
Erase: On input $(\mathtt{erase}, \mathcal{R}, \mathcal{P}_i, \mathcal{P}_j, \tau_{i,j})$ from $\mathcal{P}_j$.
    1. Send $(\mathtt{erase}, \mathcal{P}_i, \mathcal{P}_j, \tau)$ to $\mathcal{R}$.
    2. Set $\epsilon_{\tau_{i,j}} \leftarrow 1$, and delete $m_{\tau_{i,j}}$ for all $\tau_{i,j} \leq \tau$.
EraseAll: On input $(\mathtt{eraseAll}, \mathcal{R}, \mathcal{P}, \mathcal{P}_j, \boldsymbol{\tau})$ from $\mathcal{P}_j$, where $\boldsymbol{\tau} = \{\tau_{i,\mathcal{P}}\}_{i \neq j}$
    1. Send $(\mathtt{eraseAll}, \mathcal{P}, \mathcal{P}_j, \{\tau_{i,\mathcal{P}}\}_{i \neq j})$ to $\mathcal{R}$.
    2. Set $\epsilon_{\tau_{i,\mathcal{P},j}} \leftarrow 1, \forall i$
    3. For each i, if $\epsilon_{\tau_{i,\mathcal{P},j}} = 1$ for all $j \in \mathcal{P}$, delete $m_{\tau_{i,\mathcal{P}}}$ for all $\tau_{i,\mathcal{P}} \leq \tau$.
Request: On input $(\mathtt{request}, \mathcal{R}, \mathcal{P}_i, \mathcal{P}_j, \tau_{i,j})$ from $\mathcal{P}_j$,
    1. Send $(\mathtt{request}, \mathcal{P}_i, \mathcal{P}_j, \tau_{i,j})$ to $\mathcal{R}$.
    2. If $\mathcal{R}$ is corrupt, wait for $(\mathtt{Deliver}, \tilde{m})$ from $\mathcal{S}$, send $\tilde{m}$ to $\mathcal{P}_j$.
    Else, if $\epsilon_{\tau_{i,j}} \neq 0$ then return $\bot$ to $\mathcal{P}_j$, else retrieve $m_{\tau_{i,j}}$ and send it to $\mathcal{P}_j$.
RequestFromAll: On input $(\mathtt{requestFromAll}, \mathcal{R}, \mathcal{P}, \mathcal{P}_j, \boldsymbol{\tau})$ from $\mathcal{P}_j$, where $\boldsymbol{\tau} = \{\tau_{i,\mathcal{P}}\}_{i \neq j}$,
    1. Send $(\mathtt{requestFromAll}, \mathcal{P}, \mathcal{P}_j, \{\tau_{i,\mathcal{P}}^{\mathcal{R}}\}_{i \neq j})$ to $\mathcal{R}$.
    2. If $\mathcal{R}$ is corrupt, wait for $(\mathtt{Deliver}, \tilde{\mathbf{m}})$ from $\mathcal{S}$, send $\tilde{\mathbf{m}}$ to $\mathcal{P}_j$.
    Else, for all $i \neq j$, if $\epsilon_{\tau_{i,\mathcal{P},j}} \neq 0$ then set the ith-coordinate of $\mathbf{m}$ to be equal to $\bot$, otherwise retrieve the ith-message corresponding to $\tau_{i,\mathcal{P}}$. Send the vector $\mathbf{m}$ to $\mathcal{P}_j$.

</div>

Figure 3. Functionality modelling a single relay

Send. In the `send` command we let the adversary see the message being sent even between honest parties and an honest relay. This captures the fact that our connections are only authentic. To manage the `send` commands the functionality maintains pairwise counters, $\tau_{i,j}, \forall(i,j)$, that are used to store, retrieve and erase messages sent from party $\mathcal{P}_i$ to $\mathcal{P}_j$. The relay uses the variable $\epsilon_{\tau_{i,j}}$ to indicate whether the $\tau_{i,j}$-th message from $\mathcal{P}_i$ to $\mathcal{P}_j$ needs to be stored for future possible retrievals by party $\mathcal{P}_j$. We have $\epsilon_{\tau_{i,j}} = \bot$ if the specific message associated to counter $\tau_{i,j}$ has not been sent, $\epsilon_{\tau_{i,j}} = 0$ if the associated message is stored for future use, and $\epsilon_{\tau_{i,j}} = 1$ if the associated message is never going to be retrieved.

The variable $\epsilon_{\tau_{i,j}}$ is used to avoid messages being stored indefinitely by the relay. We allow a receiving party to indicate that the network can erase messages, and these will never be requested in the future. To indicate which messages are not going to be retrieved, a receiving party uses the `erase` command.

Request. Parties use the `request` command to retrieve a message. The request is of the form $(i, j, \tau_{i,j})$, where party $\mathcal{P}_j$ is requesting the $\tau_{i,j}$-th message sent to it by party $\mathcal{P}_i$. The adversary is

allowed to replace any sent message, which has not been erased, to any value it wants, including $\perp$, as long as the relay is corrupt, via the `request` query. A $\perp$ value is returned by an honest relay if the message has been erased, or it has not yet been received by the relay. Note, the `request` command allows an adversarial relay to send different messages for the same $(i, j, \tau_{i,j})$ tuples for different `request` queries.

SendToAll. Similarly, a party $\mathcal{P}_i$ can also send a message to all the other parties (or even a subset of $\mathcal{P}$), by just sending a single message to the relay. This is captured by the `sendToAll` command. To manage these, the functionality maintains 'global' counters, $\tau_{i,\mathcal{P}}$ and $\epsilon_{\tau_{i,\mathcal{P},j}}$, $\forall i, j \in \mathcal{P}$, used to store, retrieve and erase messages sent from $\mathcal{P}_i$ to all parties in $\mathcal{P} \setminus \mathcal{P}_i$. The `sendToAll` command is paired with `requestFromAll` and `eraseAll` commands in order to manage these sent messages.

RequestToAll. The command `requestFromAll` allows a single party $\mathcal{P}_j$ to retrieve messages from all parties. To ease the exposition, we only allow this command on global messages with counters $\tau_{i,\mathcal{P}}$, for all i. It can be used by $\mathcal{P}_j$ to obtain all the $n-1$ messages $m_{\tau_{i,\mathcal{P}}}$, for $i \neq j$. This command will retrieve a vector of messages, one from each sending party. If a specific message has not yet been received by the relay, then $\perp$ is returned in this location.

Erase EraseFromAll. The relaying party $\mathcal{R}$ only stores messages for which $\epsilon_{\tau_{i,j}}$ (resp. $\epsilon_{\tau_{i,\mathcal{P},j}}$) is not equal to one. Notice that, the relay $\mathcal{R}$ does not delete the message on retrieval (by setting $\epsilon_{\tau_{i,j}} = 1$) since the receiving party may wish to request it again (in the case of it failing for some reason during the execution of the `request` command). Relays delete messages only when they are instructed to, which happens when either `erase` or `eraseAll` are called.

An implementation of this underlying functionality $\mathcal{F}_{\mathsf{SingleRelay}}$ is immediate, in that the relaying party $\mathcal{R}$ just needs to maintain a list of messages sent, which may be requested in future, and it needs to maintain authenticated links with all parties.

3.2   Key Agreement

In order to describe our main robust form of relay, which allows an adversary to delay messages, we require that the parties have exchanged secret keys; this is to enable the communication between parties to be both secure and authenticated. Now we could assume that such keys are pre-assigned, however we show that, assuming set of relays of which we have an honest majority and pre-authenticated public keys, we can run a simple key agreement process across the network provided by the said relays.

In Figure 4, we give a key agreement functionality $\mathcal{F}_{\mathsf{KE}}$ and then, in Figure 5, we present the corresponding protocol Π_{KE}. The protocol is run between $r+2$ parties, r of the parties are the relays $\mathcal{R}_1, \ldots, \mathcal{R}_r$ and the other two parties, $\mathcal{P}_i$ and $\mathcal{P}_j$, are the two parties who wish to agree on a secret key, and assumes that the number of corrupt relays t_r is less than $r/2$, i.e. the majority of the relays are honest.

More concretely, $\mathcal{F}_{\mathsf{KE}}$ captures the security requirements from a single key-exchange between two parties: if both parties $\mathcal{P}_i$ and $\mathcal{P}_j$ are honest, then they receive the same uniformly-distributed key $\mathsf{k} \in \mathcal{K}$, where $\mathcal{K}$ is the key space for a symmetric encryption scheme $\mathcal{E}$, while the adversary learns nothing except that a secret key was generated; if either $\mathcal{P}_i$ or $\mathcal{P}_j$ are corrupt, then the adversary is given the power to generate the secret key, since it will know that key in any case. The protocol Π_{KE}, implementing $\mathcal{F}_{\mathsf{KE}}$, is in the $\mathcal{F}_{\mathsf{SingleRelay}}$-hybrid model, and assumes an already existing PKI, in that the parties $\mathcal{P}_i$ have certified public keys pk_i.

Using these public keys and the functionality $\mathcal{F}_{\mathsf{SingleRelay}}$ from Figure 3, the parties can establish pairwise secret keys k for every pair $\mathcal{P}_i$ and $\mathcal{P}_j$. Concretely, this can be done via a protocol such as the Station-to-Station protocol [DvW92] or via TLS [Res18]. However, since we are in a static corruption model, we can actually deploy a non-forward secure key agreement scheme in order to establish the keys k. In Figure 9, we therefore use a simplistic key agreement scheme based upon key transport. This can obviously be replaced by a more complex key agreement scheme if desired.

Functionality $\mathcal{F}_{\mathsf{KE}}(\mathcal{R}_1, \ldots, \mathcal{R}_k, \mathcal{P}_i, \mathcal{P}_j)$

This functionality runs with an adversary $\mathcal{S}$ and $r + 2$ parties, r special parties (denoted by $\mathcal{R}_k$) which are the relays and 2 parties (denoted by $\mathcal{P}_i, \mathcal{P}_j$).
It is parametrized by a domain D.

1. Upon receiving ($\mathtt{establish\text{-}key}, sid, \mathcal{P}_i, \mathcal{P}_j, sender$) from $\mathcal{P}_i$ and ($\mathtt{establish\text{-}key}, sid, \mathcal{P}_i, \mathcal{P}_j, receiver$) from $\mathcal{P}_j$, record the tuple $(sid, \mathcal{P}_i, \mathcal{P}_j)$ and send this tuple to $\mathcal{S}$ and $\mathcal{R}_1, \ldots, \mathcal{R}_k$.
2. If both $\mathcal{P}_i$ and $\mathcal{P}_j$ are honest, sample $\mathsf{k} \leftarrow D$, send ($\mathtt{key}, sid, \mathsf{k}$) to $\mathcal{P}_i$ and $\mathcal{P}_j$ and a message ($\mathtt{key}, sid, \mathcal{P}_i, \mathcal{P}_j$) to $\mathcal{S}$, and halt.
3. If $\mathcal{P}_i$ is corrupt (or $\mathcal{P}_j$ is corrupt), then send a message ($\mathtt{chooseKey}, sid, P_i, P_j$) to the adversary. If $\mathcal{S}$ sends $\mathtt{abort}$, forward $\mathtt{abort}$ to the honest parties and halt; otherwise, receive a value k from $\mathcal{S}$ and send ($\mathtt{key}, sid, \mathsf{k}$) to $\mathcal{P}_j$ (resp. $\mathcal{P}_i$), then halt.
4. If both $\mathcal{P}_i$ and $\mathcal{P}_j$ are corrupt, then send a message ($\mathtt{chooseKey}, sid, \mathcal{P}_i, \mathcal{P}_j$) to the adversary. If $\mathcal{S}$ sends $\mathtt{abort}$, forward $\mathtt{abort}$ to the honest parties.

Abort: $\mathcal{S}$ can at any point send the message $\mathtt{abort}$, upon which the functionality sends ($\mathtt{abort}$) to all honest parties and halt.

Figure 4. Functionality for key exchange

To define these keys we assume a cryptographic hash function $\mathcal{H} : \{0, 1\}^* \longrightarrow \mathcal{K}$ which we model as a random oracle and we capture the fact that the parties own certified key by exploiting the presence of an ideal functionality $\mathcal{F}_{\mathsf{CKeyGen}}$ which generates certifies keys $(\mathsf{pk}_i, \mathsf{sk}_i)$ for each party $\mathcal{P}_i, i \in [n]$.

The problem is that to perform this key agreement we rely on the $\mathtt{request}$ commands from the functionality $\mathcal{F}_{\mathsf{SingleRelay}}$. At this point we need consensus which is guaranteed if $t_r < r/2$, or we abort. Thus in the key-agreement protocol protocol, if the adversary does not mount a Denial-of-Service attack, (i.e. all parties will complete the pair-wise key agreement protocol) each pair of parties will end up with a unique uni-directional key k (for some AEAD symmetric encryption scheme) with which to securely encrypt messages from $\mathcal{P}_i$ to $\mathcal{P}_j$. We hence obtain the following theorem.

Theorem 3.1. *Assuming $t_r < r/2$, a random oracle, a public-key IND-CCA encryption scheme $\mathcal{E} = (\mathsf{Enc}_{\mathsf{pk}}, \mathsf{Dec}_{\mathsf{sk}})$, an ideal functionality $\mathcal{F}_{\mathsf{CKeyGen}}^C$ for the key-generation of $\mathcal{E}$ which returns certified keys, the protocol Π_{KE}, described in Figure 5, securely implements the functionality $\mathcal{F}_{\mathsf{KE}}$ in the $\{\mathcal{F}_{\mathsf{SingleRelay}}, \mathcal{F}_{\mathsf{KeyGen}}^C\}$-hybrid model.*

Proof. Let $\mathcal{A}$ be the real world adversary, we construct an ideal-world adversary $\mathcal{S}$ such that no environment can distinguish between an ideal and real execution. Throughout the execution, $\mathcal{S}$ emulates the random oracle $\mathcal{H}$ by answering every new query with a random value from the relevant set and maintaining a list of past queries to answer repeated queries consistently.

- Upon receiving $(sid, \mathcal{P}_i, \mathcal{P}_j)$ from the functionality, $\mathcal{S}$ emulates $\mathcal{F}_{\mathsf{CKeyGen}}$ obtaining $(\mathsf{pk}_i, \mathsf{sk}_i)$ and $(\mathsf{pk}_j, \mathsf{sk}_j)$. Send pk_i and pk_j to the adversary. If either $\mathcal{P}_i$ or $\mathcal{P}_j$ is corrupt, send the corresponding secret key to $\mathcal{A}$.
- HONEST P_i AND HONEST P_j. Send random ciphertexts $\mathsf{ct}_{i,j}$ and $\mathsf{ct}_{j,i}$ to the adversary. Then emulate $\mathcal{F}_{\mathsf{SingleRelay}}$ and receive values $\widetilde{\mathsf{ct}_{i,j}}^k$ and $\widetilde{\mathsf{ct}_{j,i}}^k$, for $k \in \mathcal{R}_I$, from $\mathcal{A}$. If some $\widetilde{\mathsf{ct}}^k \neq \mathsf{ct}$, set the flag $\mathsf{corrupt}\mathcal{R}_k = \mathsf{true}$. Since we assume than more than $r/2$ relays are honest, halt.
- CORRUPT P_i AND CORRUPT P_j. Receive $(\mathsf{chooseKey}, sid, P_i, P_j)$ from the functionality. Simulates the honest relays sending the received ciphertexts to $\mathcal{A}$ in the emulation of $\mathcal{F}_{\mathsf{SingleRelay}}$. If $\mathcal{A}$ sends abort, forward it to the functionality and halt.
- CORRUPT P_i AND HONEST P_j. Receive $(\mathsf{chooseKey}, sid, \mathcal{P}_i, \mathcal{P}_j)$ from the functionality. Receive $(\mathsf{send}, \mathcal{R}_k, \mathcal{P}_i, \mathcal{P}_j, \mathsf{ct}_{i,j}^k)$ from $\mathcal{A}$, emulating $\mathcal{F}_{\mathsf{SingleRelay}}$. Then emulate again the same functionality on the command $\mathsf{request}$ and receive from the adversary $\widetilde{\mathsf{ct}}_{ij}^k$, for $k \in \mathcal{R}_I$. Decrypt and check whether there are more than $r/2$ values that are consistent. If this is the case, let $\mathsf{k}_{i,j}$ be such a value; otherwise send abort to the functionality and halt. $\mathcal{S}$ samples a random k' and encrypt it sending $\mathsf{ct}_{ji} = \mathsf{Enc}_{\mathsf{pk}_i}(\mathsf{k}')$ to the adversary. If $\mathcal{A}$ sends abort, then forward it to the functionality. In case of no abort, compute $\mathsf{k} = \mathcal{H}(\mathsf{k}_{i,j}, \mathsf{k}')$, forward this value to the functionality and halt.
- HONEST P_i AND CORRUPT P_j. This is symmetric to the previous case.

Indistinguishability. When both $\mathcal{P}_i$ and $\mathcal{P}_j$ are honest, the outputs of the two executions are indistinguishable as $\mathcal{H}$ is modelled as a truly random function. The transcript is also indistinguishable due to the semantic security of the encryption scheme. The only point where the simulation could fail is if the adversary managed to change the output of the real execution; however this is not possible because honest relays will always send the correct ciphertexts and of the assumption $t_r > r/2$.

When $\mathcal{P}_i$ is corrupt in both executions the honest relays will send the ciphertexts obtained by $\mathcal{A}$ which fix the output. All other cases are similar. □

3.3 Modelling Bounded Delays

We now turn to modelling the bounded-delay communication setting for our main protocol. This is captured by the functionality $\mathcal{F}_{\mathsf{Delay}}$ given in Figure 6. It is parametrized by a constant δ and it works by querying the adversary, who can then impose a delay bounded by δ. This means that, as soon as a party has been delayed for δ rounds, the next command will proceed. More in details, once the functionality is called with $(\mathsf{delay}, \mathcal{P}_i, \mathsf{command})$, where $\mathsf{command}$ represent a specific action that $\mathcal{P}_i$ is trying to execute, we distinguish two different cases. If $\delta_i = \bot$, it means that $\mathcal{P}_i$ is not currently delayed. Hence, the functionality sends a message (delay, i) to the ideal adversary $\mathcal{S}$. If $\mathcal{S}$ replies with (delay, D_i), then the functionality sets $\delta = D_i$, meaning that $\mathcal{P}_i$ will be delayed for D_i rounds, otherwise returns ok to $\mathcal{P}_i$. If otherwise $\delta_i \neq \bot$, $\mathcal{P}_i$ has been already delayed, so the functionality sets $\delta_i \leftarrow \delta_i - 1$, and if the resulting δ_i is 0, it sends ok to $\mathcal{P}_i$.

In a real world implementation, as communication is essentially synchronous, if a party does not receive a valid response (e.g. a message or an ok signal) after a request, then they interpret this as a delay. Note, that the delay is *local* for a party $\mathcal{P}_i$, it does not depend on the specific corresponding party $\mathcal{P}_j$. However, it does apply for all messages passed between $\mathcal{P}_i$ and *all* of the relays $\mathcal{R}_1, \dots, \mathcal{R}_r$.

In addition, we require that parties are delayed only for a limited number of rounds; if we removed this condition, and set $\delta = \infty$, then the resulting functionality would produce something

---

Protocol $\Pi_{\mathsf{KE}}(\mathcal{R}_1, \ldots, \mathcal{R}_r, \mathcal{P}_i, \mathcal{P}_j)$

The protocol assumes $t_r > r/2$. It uses a public-key encryption scheme with certified keys. Let $\mathcal{H}$ be an hash function modelled as a random oracle.

OUTPUT: A unidirectional secret key $\mathsf{k}_{i,j}$ for an AEAD encryption scheme.

1. Upon activation, parties $\mathcal{P}_i, \mathcal{P}_j$ send input $\mathtt{init}$ to $\mathcal{F}_{\mathsf{SingleRelay}}$ and set pairwise counter $\tau_{i,j} \leftarrow 0$ and $\epsilon_{i,j} \leftarrow \bot$.
2. Both parties runs $\mathcal{F}_{\mathsf{CKeyGen}}$ obtaining $(\mathsf{pk}_i, \mathsf{sk}_i)$ and $(\mathsf{pk}_j, \mathsf{sk}_j)$, respectively.
3. Parties $\mathcal{P}_i$ and $\mathcal{P}_j$ respectively generate keys $\mathsf{k}'_{i,j} \leftarrow \mathcal{K}$ and $\mathsf{k}'_{j,i} \leftarrow \mathcal{K}$.
4. $\mathcal{P}_i$ computes $\mathsf{ct}_{i,j} \leftarrow \mathsf{Enc}_{\mathsf{pk}_j}(\mathsf{k}'_{i,j})$ and $\mathcal{P}_j$ computes $\mathsf{ct}_{j,i}$ similarly.
5. For $k \in [r]$, $\mathcal{P}_i$ calls the command $(\mathtt{send}, \mathcal{R}_k, \mathcal{P}_i, \mathcal{P}_j, \mathsf{ct}_{i,j})$ on the functionality $\mathcal{F}_{\mathsf{SingleRelay}}$. $\mathcal{P}_j$ does the same with $\mathsf{ct}_{j,i}$.
6. For $k \in [r]$, party $\mathcal{P}_j$ sends $(\mathtt{request}, \mathcal{R}_k, \mathcal{P}_i, \mathcal{P}_j, 1)$ to $\mathcal{F}_{\mathsf{SingleRelay}}$ which forwards the request to $\mathcal{R}_k$, so that $\mathcal{P}_j$ obtains the message $\mathsf{ct}_{i,j}$; for $k \in [r]$, party $\mathcal{P}_i$ sends $(\mathtt{request}, \mathcal{R}_k, \mathcal{P}_j, \mathcal{P}_i, 1)$ to $\mathcal{F}_{\mathsf{SingleRelay}}$ which forwards the request to $\mathcal{R}_k$, so that $\mathcal{P}_i$ obtains the message $\mathsf{ct}_{j,i}$
7. Each party decrypts all the received ciphertexts: $\mathcal{P}_j$ computes $\mathsf{k}'_{i,j} = \mathsf{Dec}_{\mathsf{sk}_j}(\mathsf{ct}_{i,j})$, for each $r \in [k]$, and $\mathcal{P}_i$ computes $\mathsf{k}'_{j,i} = \mathsf{Dec}_{\mathsf{sk}_i}(\mathsf{ct}_{j,i})$, for each $r \in [k]$. Both P_i and P_j does as follows:
   - If there is no value that is repeated more that $r/2$ times, then $\mathtt{abort}$
   - Otherwise, if the value repeated more than $r/2$ is $\mathsf{k}'_{i,j} = \bot$ (similarly for P_i if the decrypted value is $\mathsf{k}'_{j,i} = \bot$), then $\mathtt{abort}$; else, parties $\mathcal{P}_i$ and $\mathcal{P}_j$ define $\mathsf{k}_{i,j} = \mathcal{H}(\mathsf{k}'_{i,j}, \mathsf{k}'_{j,i})$.
8. For $k \in [r]$, party $\mathcal{P}_j$ calls the command $(\mathtt{erase}, \mathcal{R}_k, \mathcal{P}_i, \mathcal{P}_j, 1)$ on the functionality $\mathcal{F}_{\mathsf{SingleRelay}}$.

Figure 5. Protocol instantiating the key exchange functionality

---

Functionality $\mathcal{F}_{\mathsf{Delay}}(\mathcal{P}_1, \ldots, \mathcal{P}_n, \mathcal{R}_1, \ldots, \mathcal{R}_r, \delta)$

This functionality runs with an adversary $\mathcal{S}$, n parties, denoted by $\mathcal{P}_1, \ldots, \mathcal{P}_n$, and r relays, denoted by $\mathcal{R}_1, \ldots, \mathcal{R}_r$. In addition as input it takes a parameter $\delta \in \mathbb{N}$. Further, it stores $\delta_i \leq \delta$ for each $i \in [n]$.

Init: Set $\delta_i \leftarrow \bot$ for all $i \in [n]$.
Delay: On input $(\mathtt{delay}, \mathcal{P}_i, \mathtt{command})$:
  - If $\delta_i = \bot$: Send $(\mathtt{delay}, i)$ to $\mathcal{S}$ and wait for input.
    - If $\mathcal{S}$ returns $(\mathtt{delay}, D_i)$ and $0 < D_i \leq \delta$, then
      1. Send $(\mathtt{delayed}, D_i)$ to $\mathcal{S}$.
      2. Set $\delta_i \leftarrow D_i$.
    - If $\mathcal{S}$ returns $(\mathtt{delay}, D_i)$ and $D_i \geq \delta$, then
      1. Send $(\mathtt{delayed}, \delta)$ to $\mathcal{S}$.
      2. Set $\delta_i \leftarrow \delta$.
    - Else, send $\mathtt{ok}$ to $\mathcal{P}_i$
  - If $\delta_i \neq \bot$:
    1. Set $\delta_i \leftarrow \delta_i - 1$.
    2. If $\delta_i = 0$: send $\mathtt{ok}$ to $\mathcal{P}_i$

Figure 6. Functionality modelling delays

akin to an asynchronous network, which would result in some changes needed to the resulting MPC protocol which we run on top of our relays.

As remarked, the delays model the fact that parties can execute the protocol at different speeds, and can reboot themselves or go offline for a short period. We give the adversary the ability to control this operation.

Functionality $\mathcal{F}_{\mathsf{SecureRobustRelay}}(\mathcal{R}_1, \ldots, \mathcal{R}_r, \mathcal{P}_1, \ldots, \mathcal{P}_n, \delta)$ **- Part 1**

This functionality runs with an adversary $\mathcal{S}$ and $n + r$ parties, such that parties $\mathcal{R}_1, \ldots, \mathcal{R}_r$ act as relays, whilst parties $\mathcal{P}_1, \ldots, \mathcal{P}_n$ wish to use the set of relays as a means of relaying messages between themselves. In addition as input it takes a parameter $\delta \in \mathbb{N}$. The functionality maintains pairwise counters $\tau_{i,j}$ for all $i \neq j$ and global counters $\tau_{i,\mathcal{P}}$ for all $i \in \mathcal{P}$, variables $\epsilon_{\tau_{i,j}}^{\mathcal{R}} \in \{\perp, 0, 1\}$, $\epsilon_{\tau_{i,\mathcal{P},j}}^{\mathcal{R}} \in \{\perp, 0, 1\}$, $\delta_n \in \{\perp\} \cup \mathbb{N}$, $i \in [n]$, and a variable `operation`.

Upon activation, the functionality receives $(\mathcal{R}_I, \mathcal{P}_I)$ from the adversary, where $\mathcal{R}_I$ is the set of corrupt relays, and similarly $\mathcal{P}_I$ is the set of corrupt parties, where $|\mathcal{R}_I| \leq t_r$ and $|\mathcal{P}_I| \leq t_p$. The functionality stores these two sets in its variable `state`.

Abort: $\mathcal{S}$ can at any point send the message (`abort`), upon which the functionality sends (`abort`) to all honest parties and halts.

Init: On input (`init`) from all parties, the functionality sets all counters $\tau \leftarrow 0$ and variables $\epsilon_\tau \leftarrow \perp$, $k \in [r]$, and sets $\delta_i \leftarrow \perp$, for all $i \in \mathcal{P}$. In addition, it runs as a copy of $\mathcal{F}_{\mathsf{KE}}$ and sends to the adversary all the relevant keys; the keys obtained in this stage are available for later use.

Send: On input (`send`, $\mathcal{P}_i, \mathcal{P}_j$) from $\mathcal{P}_i$
1. Run the `DelayMacro` on input (`send`, $i, j, \perp$).
2. We distinguish two cases:
    - If $\mathcal{P}_i$ is honest, receive m from $\mathcal{P}_i$. Compute $\mathsf{ct} \leftarrow \mathsf{Enc}_{\mathsf{k}_{i,j}}(m)$ and send it to the adversary and honest relays. Set $m \leftarrow m_k$, for each $k \in [r]$.
    - If $\mathcal{P}_i$ is corrupt, receive $m_k, k \in [r]$, from the adversary. Compute $\mathsf{ct}_k \leftarrow \mathsf{Enc}_{\mathsf{k}_{i,j}}(m_k)$ and send it to $\mathcal{R}_k$, for all $k \notin \mathcal{R}_I$.
3. Increment $\tau_{i,j}$ by one.
4. For $k \in [r]$, set $m_{\tau_{i,j}}^k \leftarrow m_k$ and store it.
5. For $k \in [r]$, set $\epsilon_{\tau_{i,j}} \leftarrow 0$.
Note the messages sent in a `send` command **are not** *sent to the adversary.*

SendToAll: On input (`sendToAll`, $\mathcal{P}_i, \mathcal{P}$) from $\mathcal{P}_i$
1. Run the `DelayMacro` on input (`send`, $i, \mathcal{P}, \perp$).
2. We distinguish two cases:
    - If $\mathcal{P}_i$ is honest, receive m from $\mathcal{P}_i$. Send it to the adversary and honest relays. Set $m \leftarrow m_k$, for each $k \in [r]$.
    - If $\mathcal{P}_i$ is corrupt, receive $m_k, k \in [r]$ from the adversary. Send it to $\mathcal{R}_k$, for all $k \notin \mathcal{R}_I$.
3. For all $k \in [r]$,
    (a) Increment $\tau_{i,\mathcal{P}}$ by one.
    (b) Set $m_{\tau_{i,\mathcal{P}}}^k \leftarrow m^k$ and store it.
    (c) Set $\epsilon_{\tau_{i,\mathcal{P},j}} \leftarrow 0$.
Note the messages sent in a `sendToAll` command **are** *sent to the adversary.*

Figure 7. Functionality $\mathcal{F}_{\mathsf{SecureRobustRelay}}$ - Part 1

3.4 Implementing a Secure Robust Relay using Multiple Single Relay's

We can now formally describe our star-like topology where n parties, instead of relying on a single relay, rely on a set of relays, assuming at least one of them is honest. We show that, once each party has agreed pair-wise keys for an AEAD encryption scheme with each other party, then a robust relay protocol can be implemented, assuming only one honest relay and without expensive consensus procedures. These keys can either be pre-distributed or they can be agreed using the honest majority protocol presented in Section 3.2.

We stress that AEAD encryption is only used on point-to-point channels, i.e. with the `send` command. Looking ahead, our MPC protocol only relies on `sendToAll`, which works like a public broadcast in a normal MPC protocol and does not require encryption.

We model $\mathcal{F}_{\mathsf{SecureRobustRelay}}$ in such a way that it does not manipulate the received messages, unless this is strictly necessary. In this way we keep the description of the functionality as simple as

---

Functionality $\mathcal{F}_{\mathsf{SecureRobustRelay}}(\mathcal{R}_1, \ldots, \mathcal{R}_r, \mathcal{P}_1, \ldots, \mathcal{P}_n, \delta)$ **- Part 2**

Request: On input $(\mathtt{request}, \mathcal{P}_i, \mathcal{P}_j)$ from party $\mathcal{P}_j$,
  1. Run the $\mathtt{DelayMacro}$ on the input $(\mathtt{request}, i, j, \perp)$.
  2. Receive $\tau, k \in [r]$, from $\mathcal{P}_i$ (or the adversary if P_i is corrupt), where τ is of type $\tau_{i,j}$, and send $(\mathtt{request}, i, j, \tau)$ to $\mathcal{R}_k$,
  3. There are three different cases:
      - If the adversary sends $\mathtt{abort}$, forward $\mathtt{abort}$ to the honest parties and halt.
      - Else, if $\mathcal{S}$ sends $(\mathtt{request}, \perp)$, retrieve one of the m_τ for k at choice and set this value to be the output of $\mathcal{P}_j$
      - Else, if $\mathcal{S}$ sends a value $\widetilde{m}$, retrieve m_τ^k for all $k \notin \mathcal{R}_I$. If either $m_\tau^k = \widetilde{m}$ for all $k \notin \mathcal{R}_I$ or $m_\tau^k = \perp$ for all $k \notin \mathcal{R}_I$, then set this value $\widetilde{m}$ to be the output of $\mathcal{P}_j$; else output $\mathtt{abort}$.
RequestFromAll: On input $(\mathtt{requestFromAll}, \mathcal{P}, \mathcal{P}_j)$ from party $\mathcal{P}_j$,
  1. Run the $\mathtt{DelayMacro}$ on the input $(\mathtt{requestFromAll}, \mathcal{P}, j, \perp)$.
  2. Receive $\boldsymbol{\tau}, k \in [r]$, from $\mathcal{P}_i$ (or the adversary if P_i is corrupt), and send $(\mathtt{requestFromAll}, \mathcal{P}, j, \boldsymbol{\tau})$ to $\mathcal{R}_k$.
      Note, $\boldsymbol{\tau}$ is an $(n-1)$-dimensional vector of counters of type $\tau_{i,\mathcal{P}}, i \neq j$.
  3. There are three different cases:
      - If the adversary sends $\mathtt{abort}$, forward $\mathtt{abort}$ to the honest parties and halt.
      - Otherwise, for each $\tau_{i,\mathcal{P}}$, $i \neq j$:
          - If $\mathcal{S}$ sends $(\mathtt{requestFromAll}, \perp)$, retrieve one of the $m_{\tau_{i,\mathcal{P}}}^k$, for k at choice, and set this value to be the ith value of the output of $\mathcal{P}_j$
          - Else, if $\mathcal{S}$ sends a value $\widetilde{m}_i$, retrieve $m_{\tau_{i,\mathcal{P}}}^k$ for all $k \notin \mathcal{R}_I$. If $m_{\tau_{i,\mathcal{P}}}^k = \widetilde{m}_i$ for all $k \notin \mathcal{R}_I$, then set this value $\widetilde{m}_i$ to be the value of the output of $\mathcal{P}_j$ corresponding to $\tau_{i,\mathcal{P}}$; else output $\mathtt{abort}$.
  4. If no $\mathtt{abort}$ has occurred, output the vector of messages $\mathbf{m}_{\boldsymbol{\tau}}$ to honest $\mathcal{P}_j$
Erase: On input $(\mathtt{erase}, \mathcal{P}_i, \mathcal{P}_j)$ from $\mathcal{P}_j$
  1. Run the $\mathtt{DelayMacro}$ on the input $(\mathtt{erase}, i, j, \perp)$.
  2. Receive $\tau_{i,j}, k \in [r]$, from $\mathcal{P}_i$ (or the adversary if P_i is corrupt), and send $(\mathtt{erase}, i, j, \tau_{i,j})$ to $\mathcal{R}_k$.
  3. Set $\epsilon_{\tau_{i,j}} \leftarrow 1$, and delete $m_{\bar{\tau}_{i,j}}^k$ for all $\bar{\tau}_{i,j} \leq \tau_{i,j}$.
EraseAll: On input $(\mathtt{eraseAll}, \mathcal{P})$ from $\mathcal{P}_j$
  1. Run the $\mathtt{DelayMacro}$ on the input $(\mathtt{eraseAll}, j)$.
  2. Receive $\tau_{i,\mathcal{P}}$, from $\mathcal{P}_i$ (or the adversary if P_i is corrupt), and send $(\mathtt{erase}, i, j, \tau_{i,\mathcal{P}})$ to $\mathcal{R}_k, k \in [r]$.
  3. Set $\epsilon_{\tau_{i,\mathcal{P},j}} \leftarrow 1$.
  4. If $\epsilon_{\tau_{i,\mathcal{P},j}} = 1$ for all $j \in \mathcal{P}$, delete $m_{\bar{\tau}_{i,\mathcal{P}}}^k$ for all $\bar{\tau}_{i,\mathcal{P}} \leq \tau_{i,\mathcal{P}}$

$\mathtt{DelayMacro}$: On input $(\mathtt{command}, i, j, \mathrm{arg})$:
  1. Store $(\mathtt{command}, i, j, \mathrm{arg})$ in the variable $\mathtt{operation}$.
  2. Send $(\mathtt{command}, i, j)$ to $\mathcal{S}$ and wait for an input.
  3. If $\mathcal{S}$ sends $(\mathtt{continue})$, send $\mathtt{ok}$ to $\mathcal{P}_i$; else, if $\mathcal{S}$ sends $(\mathtt{continue}^*)$, move to the next step.
  4. Retrieve $(\mathtt{command}, i, j, \mathrm{arg})$ from the variable $\mathtt{operation}$.

Figure 8. Functionality $\mathcal{F}_{\mathsf{SecureRobustRelay}}$ - Part 2

possible. Moreover, the functionality does not control the delays, but messages can be arbitrarily delayed by the adversary (up to δ rounds).

Our main goal is to present a relay functionality in which the adversary has no access to the underlying messages sent, via the $\mathtt{send}$ command, between $\mathcal{P}_i$ and $\mathcal{P}_j$, unless the adversary controls party $\mathcal{P}_i$ and/or $\mathcal{P}_j$. However, the messages sent from a party $\mathcal{P}_i$ to all other parties in $\mathcal{P}$, via a $\mathtt{sendToAll}$ command, are public. Also, we require a $\mathtt{request}$ (resp. $\mathtt{requestFromAll}$) which is guaranteed to be correct, and which does not enable a trivial Denial-of-Service attack upon this command.

In terms of the underlying messages sent over the underlying synchronous network, every command sent from the party $\mathcal{P}_i$ to the relays has an "implicit" or "explicit" response. Thus a $\mathtt{send}/\mathtt{sendToAll}$ or $\mathtt{erase}/\mathtt{eraseAll}$ command will result in the relays responding with an im-

---

Protocol $\Pi_{\mathsf{SecureRobustRelay}}(\mathcal{R}_1, \ldots, \mathcal{R}_r, \mathcal{P}_1, \ldots, \mathcal{P}_n, \delta)$

Let $\mathcal{H}$ be a hash function modelled as a random oracle and $\mathcal{E} = (\mathsf{KeyGen}, \mathsf{Enc}, \mathsf{Dec})$ an AEAD encryption scheme that uses $\mathcal{F}_{\mathsf{KE}}$ as key-exchange functionality.

init: Each pair of parties (i, j) call the functionality $\mathcal{F}_{\mathsf{KE}}$ obtaining $\mathsf{k}_{i,j}$.

send: When $\mathcal{P}_i$ wishes to send a message m to $\mathcal{P}_j$:
1. $\mathcal{P}_i$ computes $\mathsf{ct} \leftarrow \mathsf{Enc}_{\mathsf{k}_{i,j}}(m)$ and sends $(\mathtt{delay}, \mathcal{P}_i, (\mathtt{send}, \mathcal{P}_j))$ to $\mathcal{F}_{\mathsf{Delay}}$ until it receives an ok message from the functionality.
2. When $\mathcal{P}_i$ receives ok, it calls $(\mathtt{send}, \mathcal{R}_k, \mathcal{P}_i, \mathcal{P}_j, \mathsf{ct})$ on $\mathcal{F}_{\mathsf{SingleRelay}}$, $\forall k \in [r]$.

sendToAll: When $\mathcal{P}_i$ wishes to send a message m to all other parties:
1. $\mathcal{P}_i$ sends the command $(\mathtt{delay}, \mathcal{P}_i, (\mathtt{sendToAll}, \mathcal{P}))$ to $\mathcal{F}_{\mathsf{Delay}}$ until it receives an ok message from the functionality.
2. When $\mathcal{P}_i$ receives ok, it calls $(\mathtt{send}, \mathcal{R}_k, \mathcal{P}_i, \mathcal{P}, m)$ on $\mathcal{F}_{\mathsf{SingleRelay}}$ $\forall k \in [r]$.

request: When party $\mathcal{P}_j$ wishes to get the message from $\mathcal{P}_i$ with index $\tau_{i,j}$, $\mathcal{P}_j$ sends $(\mathtt{delay}, \mathcal{P}_j, (\mathtt{request}, \mathcal{P}_i))$ to $\mathcal{F}_{\mathsf{Delay}}$ until it receives an ok message from the functionality.
1. When $\mathcal{P}_j$ receives ok, it calls $(\mathtt{request}, \mathcal{R}_k, \mathcal{P}_i, \mathcal{P}_j, \tau_{i,j})$ on $\mathcal{F}_{\mathsf{SingleRelay}}$, for each $k \in [r]$, obtaining ct_k for $k \in [r]$.
2. **Check:** Party $\mathcal{P}_j$ performs the following check
    - If, for all values of k, $\mathsf{ct}_k = \bot$ then return $\bot$ //it might be that the message has not yet been sent by party $\mathcal{P}_i$.
    - Else, for each k such that $\mathsf{ct}_k \neq \bot$, compute $m_k \leftarrow \mathsf{Dec}_{\mathsf{k}_{i,j}}(\mathsf{ct}_k)$.
        - If there is a *unique* $m_k \neq \bot$ then accept this value.
        - Else, if there is more than one value $m_k \neq \bot$, then $\mathtt{abort}$.

requestFromAll: When party $\mathcal{P}_j$ wishes to get $n-1$ messages from $\mathcal{P} \setminus \mathcal{P}_j$ with index $\tau_{i,\mathcal{P}}$, $i \neq j$, $\mathcal{P}_j$ sends $(\mathtt{delay}, \mathcal{P}_j, (\mathtt{requestFromAll}, \mathcal{P}))$ to $\mathcal{F}_{\mathsf{Delay}}$ until it receives an ok from the functionality.
1. When $\mathcal{P}_j$ receives ok, it calls $(\mathtt{requestFromAll}, \mathcal{R}_k, \mathcal{P}, \mathcal{P}_j, \boldsymbol{\tau})$, where $\boldsymbol{\tau} = \{\tau_{i,\mathcal{P}}\}_{i \neq j}$, on $\mathcal{F}_{\mathsf{SingleRelay}}$, for each $k \in [r]$, obtaining $\mathbf{m}_k$, $k \in [r]$.
2. **Check:** Party $\mathcal{P}_j$ performs the following check for each coordinate of the received vectors $\mathbf{m}_k = (m_{i,k})_{i \neq j}$.
    - If, for some values of k, $m_{i,k} = \bot$, then set $m_{\tau_{i,\mathcal{P}}} = \bot$.
    - Otherwise, if there is a *unique* $m_{i,k} \neq \bot$, for all k, then accept this value; else, if there is more than one value $m_{i,k}$, for different k, such that $m_{i,k} \neq \bot$, then $\mathtt{abort}$.

erase: When $\mathcal{P}_j$ wishes to erase messages from $\mathcal{P}_i$:
1. $\mathcal{P}_j$ sends $(\mathtt{delay}, \mathcal{P}_j, (\mathtt{erase}, \mathcal{P}_i))$ to $\mathcal{F}_{\mathsf{Delay}}$ until it receives an ok message from the functionality.
2. When $\mathcal{P}_j$ receives ok, it calls $(\mathtt{erase}, \mathcal{R}_k, \mathcal{P}_i, \mathcal{P}_j, \tau_{i,j})$ on $\mathcal{F}_{\mathsf{SingleRelay}}$ $\forall k \in [r]$.

eraseAll: When $\mathcal{P}_j$ wishes to erase messages from $\mathcal{P}$:
1. $\mathcal{P}_j$ sends $(\mathtt{delay}, \mathcal{P}_j, (\mathtt{eraseAll}, \mathcal{P}))$ to $\mathcal{F}_{\mathsf{Delay}}$ until it receives an ok message from the functionality.
2. When $\mathcal{P}_j$ receives ok, it calls $(\mathtt{eraseAll}, \mathcal{R}_k, \mathcal{P}, \mathcal{P}_j, \boldsymbol{\tau})$ on $\mathcal{F}_{\mathsf{SingleRelay}}$ for each $k \in [r]$, where $\boldsymbol{\tau}$ is a vector of counters $(\tau_{i,\mathcal{P}})_{i \neq j}$.

Figure 9. Protocol $\Pi_{\mathsf{SecureRobustRelay}}$

plicit acknowledgement. The `request`/`requestFromAll` command has an obvious explicit response which comes from the relays. The delay operations will apply to the *commands* sent from the party, and not separately to the responses from the relays. Thus we assume that as soon as a command is sent to the relays, and not delayed, then it completes at the relays and a response is sent back.

Notice, that the `request` command is both secure and robust: the adversary does not see the content of a message sent from an honest party, and that if the `request` command fails then the receiving party knows this is due to adversarial behaviour on the part of the sending party and not on the part of the relays.

When a party $\mathcal{P}_i$ sends a message to all members of $\mathcal{P}$, via `sendToAll`, this message is not encrypted with an AEAD scheme, but just sent over authenticated channels. Also in this case, we are able to prove that the `requestFromAll` command is robust

The protocol implementing `request`/`requestFromAll` is very lightweight, and will form the basis of the requesting of delivery of messages when we run in our final MPC protocol. The final correct messages will be identified using an underlying symmetric key AEAD encryption scheme in case of 'private' messages sent by a party $\mathcal{P}_i$ to another party $\mathcal{P}_j$. To allow the use of an AEAD algorithm is the reason we will need to create pairwise keys $\mathsf{k}_{i,j}$, agreed between $\mathcal{P}_i$ and $\mathcal{P}_j$, and to agree such keys we will need a key agreement protocol to be run across the network.

Protocol intuition. The protocol $\Pi_{\mathsf{SecureRobustRelay}}$ has an initialization phase where parties call the key-exchange functionality $\mathcal{F}_{\mathsf{KE}}$. To `send` a message $m_{i,j}$ to $\mathcal{P}_j$, a party $\mathcal{P}_i$ first encrypts the message and then waits for an `ok` message from the network. This model the fact that $\mathcal{P}_i$ might be either temporarily offline, for example for a reboot, or delayed by the adversary. When `ok` is received, it sends the ciphertext $\mathsf{ct}_{i,j} \leftarrow \mathsf{Enc}_{\mathsf{k}_{i,j}}(m_{i,j})$ to all the relays. Each relay stores the ciphertext, and makes it available for a later `request` from $\mathcal{P}_j$. Note that if $\mathcal{P}_i$ is honest, then all the relays have the same ciphertext $\mathsf{ct}_{i,j}$ stored. When $\mathcal{P}_j$ wants to get this message, again it waits for an `ok` message from the network, and then requests these ciphertexts to all the relays. Intuitively, security is guaranteed by the following argument.

- If both $\mathcal{P}_i$ and $\mathcal{P}_j$ are honest, and $\mathcal{P}_j$ requests a message sent by $\mathcal{P}_i$, it receives $\mathsf{ct}_{i,j}$ from the honest relays, which will decrypt to the input message $m_{i,j}$ sent by $\mathcal{P}_i$. Note that corrupt relays can send arbitrary messages/ciphertexts, however, since they do not know the secret key $\mathsf{k}_{i,j}$, these messages are either invalid ciphertexts or $\perp$. For this reason, $\mathcal{P}_j$ will always receive the correct message.
- If $\mathcal{P}_i$ is honest and $\mathcal{P}_j$ is corrupt, then $\mathcal{P}_j$ can output whatever they want. It can also `abort` after it decrypts the message $m_{i,j}$ sent by $\mathcal{P}_i$.
- The case of $\mathcal{P}_i$, $\mathcal{P}_j$ both corrupt is similar to the previous one.
- If only $\mathcal{P}_i$ is corrupt, then it can send arbitrary values to the relays during the send command. However, if $\mathcal{P}_i$ is colluding with some of the corrupt relays, then it can make an honest $\mathcal{P}_j$ accept a value that was not previously stored. This can happen for example when the honest relays reply $\perp$ on a request command from $\mathcal{P}_j$, while the corrupt ones send valid ciphertexts corresponding to a unique message m. This is possible since in this case the key $\mathsf{k}_{i,j}$ is known to the relays in $\mathcal{R}_I$, i.e., the set of corrupt relays. This means that a corrupt sender cannot change a value that was previously stored by the honest relays, but can input a new value if no previous value was stored in $\mathcal{R}_k$, $k \notin \mathcal{R}_I$.

When a party $\mathcal{P}_i$ wants to send a common message to all parties, via `sendToAll`, it does not encrypt the message but simply sends it to all the relays via authenticated links. Similar to the previous case, we show that, since we assume at least one honest relay, the output of the request step, if the receiving party $\mathcal{P}_j$ is honest, either is the value actually sent and stored in the relays or $\mathcal{P}_j$ outputs `abort`. Note that this time we do not allow the adversary to send values that are not stored in all the relays, so a message is not accepted unless it is the only valid message stored in all the relays. Similarly, we extend `request` to `requestFromAll` allowing a party $\mathcal{P}_j$ to request messages from all other parties. The security of it can be proven by applying the same arguments given for `request` to each of the messages that $\mathcal{P}_j$ is retrieving. More formally, we prove the following theorem.

Theorem 3.2. *The protocol $\Pi_{\mathsf{SecureRobustRelay}}$ (Figure 9) securely with abort realizes $\mathcal{F}_{\mathsf{SecureRobustRelay}}$ (Figure 7 and Figure 8) in the $\{\mathcal{F}_{\mathsf{SingleRelay}}, \mathcal{F}_{\mathsf{Delay}}, \mathcal{F}_{\mathsf{KE}}\}$-hybrid model.*

Proof. We first describe a simulator $\mathcal{S}$, which runs a simulated copy of $\mathcal{A}$ and mimics an interaction with parties executing the protocol. For each party, it maintains a counter δ_i. We denote by $\mathcal{R}_I$ the set of corrupt relays.

Description of the simulation.

INIT. At the beginning of the protocol $\mathcal{S}$ receives the keys $\mathsf{k}_{i,j}$ for each pair (i,j) of parties such that at least one between $\mathcal{P}_i$ and $\mathcal{P}_j$ is corrupt. It sends these keys to the adversary, when the functionality $\mathcal{F}_{\mathsf{KE}}$ is queried, and stores them for later use. The simulator also maintains all the pairwise tags $\tau_{i,j}, \forall(i,j)$, the global tags $\tau_{i,\mathcal{P}}, \forall i \in \mathcal{P}$, and corresponding variables ϵ.

SEND: We start from the case of an honest sender $\mathcal{P}_i$.
  - When $\mathcal{P}_i$ calls $\mathcal{F}_{\mathsf{Delay}}$ in the protocol, the simulator sends $(\mathtt{delay}, i)$ to the adversary receiving either a message to delay or to continue.
    - If $\mathcal{S}$ receives $D_i > 0$, it sends a message $(\mathtt{delayed}, D_i)$ to the adversary and sets $\delta_i \leftarrow D_i$. It emulates $\mathcal{F}_{\mathsf{Delay}}$, when $\delta_i = 0$, $\mathcal{S}$ sends $\mathtt{continue}$ to $\mathcal{F}_{\mathsf{SecureRobustRelay}}$.
    - Else, if there is no delay, it sends $\mathtt{continue}$ to the functionality $\mathcal{F}_{\mathsf{SecureRobustRelay}}$.
  The simulator then emulates $\mathcal{F}_{\mathsf{SingleRelay}}$: it receives ct from $\mathcal{F}_{\mathsf{SecureRobustRelay}}$ and sends this value to $\mathcal{A}$, for all $k \in \mathcal{R}_I$. $\mathcal{S}$ stores ct for later use.

  We now consider the case of corrupt sender $\mathcal{P}_i$.
  - $\mathcal{S}$ emulates $\mathcal{F}_{\mathsf{Delay}}$ as before. When $\delta_i = 0$ or when there is no delay, $\mathcal{S}$ sends $\mathtt{continue}$ to the functionality. It then receives ct_k from $\mathcal{A}$ emulating $\mathcal{F}_{\mathsf{SingleRelay}}$, for each $k \in [r]$, and sends $m_k \leftarrow \mathsf{Dec}_{\mathsf{k}_{i,j}}(\mathsf{ct}_k)$, for each $k \in [r]$, to $\mathcal{F}_{\mathsf{SecureRobustRelay}}$.
  If P_i calls $\mathcal{F}_{\mathsf{SingleRelay}}$ on input ct_k for some k before receiving an ok message , $\mathcal{S}$ sends $\mathtt{continue}^*$ to the functionality together with $m_k \leftarrow \mathsf{Dec}_{\mathsf{k}_{i,j}}(\mathsf{ct}_k)$. Emulating $\mathcal{F}_{\mathsf{SingleRelay}}$, $\mathcal{S}$ stores ct_k for each k for later use.

SENDTOALL: When a party $\mathcal{P}_i$ sends a message m to all other parties, the simulation is similar to the previous case, except that $\mathcal{S}$ receives m from the functionality if $\mathcal{P}_i$ is honest, and from $\mathcal{A}$ when $\mathcal{P}_i$ is corrupt.

REQUEST: $\mathcal{S}$ emulates $\mathcal{F}_{\mathsf{Delay}}$ as above. We recall that when a corrupt $\mathcal{P}_j$ calls the functionality $\mathcal{F}_{\mathsf{SingleRelay}}$ before an ok message, $\mathcal{S}$ sends $\mathtt{continue}^*$ to $\mathcal{F}_{\mathsf{SecureRobustRelay}}$. We distinguish different cases:
  - $\mathcal{P}_i, \mathcal{P}_j$ *honest*: The simulator just sends $(\mathtt{request}, \mathcal{R}_k, \mathcal{P}_i, \mathcal{P}_j)$ to the adversary simulating an honest $\mathcal{P}_j$. The simulator ignores the ciphertexts received from $\mathcal{A}$ and sends the message $(\mathtt{request}, \perp)$ to the ideal functionality.
  - $\mathcal{P}_i$ *honest and* $\mathcal{P}_j$ *corrupt*: $\mathcal{S}$ retrieves the value ct previously stored and sends it to the adversary on behalf of honest relays emulating $\mathcal{F}_{\mathsf{SingleRelay}}$. Output whatever the adversary outputs.
  - $\mathcal{P}_i$ *corrupt and* $\mathcal{P}_j$ *honest*: The simulator receives ct_k from the adversary, for each $k \in \mathcal{R}_I$, when emulating $\mathcal{F}_{\mathsf{SingleRelay}}$ and retrieves the one previously stored on behalf of honest relays. If all those ciphertexts decrypt to the same value $\widetilde{m}$, $\mathcal{S}$ forwards the value to the functionality, otherwise it sends $\mathtt{abort}$.
  - $\mathcal{P}_i, \mathcal{P}_j$ *corrupt*: Emulating $\mathcal{F}_{\mathsf{SingleRelay}}$, it retrieves the values $\mathsf{ct}_k, k \notin \mathcal{R}_I$, previously stored and sends these values to $\mathcal{A}$. It outputs as the adversary does.

REQUESTFROMALL: When a party $\mathcal{P}_j$ requests a message sent to all parties in $\mathcal{P}$ by $\mathcal{P}_i$, again we need to distinguish different cases as before. When $\mathcal{P}_i$ is honest, $\mathcal{S}$ retrieves the message m previously stored with the tag corresponding to the one requested by $\mathcal{P}_j$ and also receives messages from $\mathcal{A}$.

Note this is different from what the simulator does in the pairwise send command. Since the messages are not encrypted and we only assume authenticated links, a corrupt relay can actually change the message here. However, since we have at least one honest relays, in the real protocol the receiving party $\mathcal{P}_j$ can detect this malicious behaviour if it receives different messages from different relays or some $\perp$, and hence it can abort the computation. When $\mathcal{P}_i$ is corrupt, $\mathcal{S}$ receives the messages $m_k, k \in \mathcal{R}_I$ from $\mathcal{A}$ and retrieves the message received by $\mathcal{A}$ in the sending phase. If all those messages are the same, $\mathcal{S}$ forwards m to the ideal functionality, otherwise it aborts.

Indistinguishability. We need to argue that the transcripts of corrupt parties and all the outputs are indistinguishable in the real and ideal executions. More formally, given the simulator $\mathcal{S}$ described above, we prove that no environment $\mathcal{Z}$ can distinguish between the ideal execution of $\mathcal{F}_{\mathsf{SecureRobustRelay}}$ and $\mathcal{S}$ and the real execution of Π and $\mathcal{A}$ in the $\{\mathcal{F}_{\mathsf{SingleRelay}}, \mathcal{F}_{\mathsf{Delay}}, \mathcal{F}_{\mathsf{KE}}\}$-hybrid model. We denote by $\mathsf{REAL}_{\Pi,\mathcal{A},\mathcal{Z}}(1^\lambda, z)$ and $\mathsf{IDEAL}_{\mathcal{F},\mathcal{S}}(1^\lambda, z)$ the distributions of $\mathcal{Z}$'s view in the protocol and ideal execution, respectively.

Lemma 3.1. *In the authenticated-link model, if $\mathcal{E}$ is an AEAD encryption scheme, then*

$$\mathsf{REAL}_{\Pi,\mathcal{A},\mathcal{Z}}(1^\lambda, z) \approx \mathsf{IDEAL}_{\mathcal{F},\mathcal{S}}(1^\lambda, z).$$

Proof. First, note that since in the real protocol all the communications are authenticated, this ensures the receiver, i.e. the relays in SEND and the requesting party in REQUEST, to obtain a non-transferable and non-repudiable proof of communication. We start with the SEND command. Consider the case of an honest $\mathcal{P}_i$ first. $\mathcal{Z}$ knows all the inputs, in particular $\mathcal{P}_i$ sends these inputs to $\mathcal{F}_{\mathsf{SecureRobustRelay}}$. When $\mathcal{P}_i$ calls $\mathcal{F}_{\mathsf{Delay}}$, $\mathcal{S}$ is given D_i by $\mathcal{A}$. When $\mathcal{P}_i$ receives ok in the real execution, it calls $\mathcal{F}_{\mathsf{SingleRelay}}$ that sends the messages to the relays; in the ideal execution, when $\delta_i = 0$, $\mathcal{S}$ sends `continue` to $\mathcal{F}_{\mathsf{SecureRobustRelay}}$. The simulator receives the ciphertexts from the ideal functionality and sends them to the adversary, for each of the corrupt relays in $\mathcal{R}_I$. The indistinguishability of the two executions directly follows from standard security of the encryption scheme.

The case of a corrupt $\mathcal{P}_i$ is similar. In the real execution, $\mathcal{P}_i$ receives ok from $\mathcal{F}_{\mathsf{Delay}}$, in the ideal execution, when $\delta_i = 0$, $\mathcal{S}$ sends `continue` to $\mathcal{F}_{\mathsf{SecureRobustRelay}}$ which sends ok to $\mathcal{P}_i$ and an encryption of the messages received from $\mathcal{S}$ to the relays. However, in the real protocol $\mathcal{P}_i$ can send at any time a request to $\mathcal{F}_{\mathsf{SingleRelay}}$, even before receiving an ok message. In this case $\mathcal{F}_{\mathsf{SingleRelay}}$ immediately sends the message to the relays. The same happens in the ideal execution, when the adversary calls $\mathcal{F}_{\mathsf{SingleRelay}}$, $\mathcal{S}$ sends `continue`* to $\mathcal{F}_{\mathsf{SecureRobustRelay}}$ that does not send ok to $\mathcal{P}_i$, but instead immediately sends the messages received from the simulator to the relays. In this case the simulation is perfect.

In the REQUEST step, we distinguish different cases. First we consider a request on private messages between two parties $\mathcal{P}_i$ and $\mathcal{P}_j$. Intuitively, if both $\mathcal{P}_i$ *and* $\mathcal{P}_j$ *are honest*, even if the adversary $\mathcal{A}$ sends arbitrary messages to $\mathcal{S}$, these cannot be valid ciphertexts except with some negligible probability; on the other hand, in the real world, the honest relays will send the same

previously stored ciphertext. The same happens in the ideal execution where the functionality retrieves the value $m_{\tau_{i,j}}$ and sets this value to be the output of $\mathcal{P}_j$. Note that in this case there is no abort, except when $\mathcal{A}$ calls abort, which can happen at any point in both the executions. More formally, we see that the simulation fails on behalf of the honest receiver $\mathcal{P}_j$, if it receives a ciphertext $\widetilde{\mathsf{ct}}$ from a corrupt relay which correctly decrypts to a message $\widetilde{m}$ that was not previously stored. Assume that there exists an environment $\mathcal{Z}$ that distinguishes with non-negligible probability between the two executions. We can describe an adversary $\mathcal{B}$ for the AEAD encryption scheme $\mathcal{E}$ according to Definition 2.2. $\mathcal{B}$ emulates the execution of Π in the $\{\mathcal{F}_{\mathsf{SingleRelay}}, \mathcal{F}_{\mathsf{Delay}}, \mathcal{F}_{\mathsf{KE}}\}$-hybrid model and mimics for $\mathcal{Z}$ the role of $\mathcal{A}$ and honest parties with the exception that the required encryptions are obtained using $\mathcal{O}_{\mathsf{Enc}}$. Given that $\mathcal{Z}$ can distinguish between the two executions with some noticeable probability, it can also recognize when some forgery is received during the emulation of the protocol with respect to $\mathcal{A}$. This forgery will also be the output of $\mathcal{B}$. Thus, unless the simulation fails, the two executions are identical.

When $\mathcal{P}_i$ is honest and $\mathcal{P}_j$ corrupt, the indistinguishability again follows from the security of the encryption scheme since the transcript only consists of the ciphertexts that the simulator previously stored.

If $\mathcal{P}_i$ is corrupt and $\mathcal{P}_j$ is honest, the transcripts are trivially indistinguishable. We need to ensure that the ideal execution outputs abort exactly when the real protocol does. In more detail, in the real protocol an honest $\mathcal{P}_j$ will abort if the ciphertexts it receives are not consistent, i.e., they decrypt to different values. This can happen either when the corrupt relays send encryptions of different messages (in the ideal execution these are checked by $\mathcal{S}$) or if their message is not the same as the message stored by the honest relays (in the ideal execution this check is performed by the ideal functionality). Moreover, in both worlds, if a corrupt sender does not send any message to the relays, since it knows the secret keys, it can always decide to send a (delayed) message $\widetilde{m}$ to $\mathcal{P}_i$ through the corrupt relays, if they collude. In these cases, since the ciphertexts are directly provided by $\mathcal{A}$, the output of the honest parties are indistinguishable.

When a party $\mathcal{P}_j$ sends or requests a 'global' non-private message from $\mathcal{P}_i$, via the sendToAll and requestFromAll commands, a similar argument can be applied. For the case of requestFromAll we again need to distinguish different cases. If both $\mathcal{P}_i$ and $\mathcal{P}_j$ are honest, a correct message m is stored by the honest relay because we are in the AM and an adversary cannot change the value sent by an honest party towards another honest party. Since the adversary is able to send arbitrary messages for corrupt relays in $\mathcal{R}_I$, both executions terminate with abort if this happens. When $\mathcal{P}_i$ is corrupt, the messages for honest relays are provided by $\mathcal{A}$; when only $\mathcal{P}_i$ is honest, then $\mathcal{S}$ uses the message provided by the ideal functionality to simulate the communication between honest relays and $\mathcal{P}_j$. In both cases transcript and output of the two executions are identical.

This also concludes the proof of the theorem. □

4 MPC Building Blocks

We describe our MPC protocol via a set of standard MPC functionalities and sub-protocols which utilize $\mathcal{F}_{\mathsf{SecureRobustRelay}}$ to implement the communication between the parties. We let $\mathcal{P}_I$ denote the set of computing parties which are adversarially controlled, i.e. $\mathcal{P}_I \subset \{\mathcal{P}_1, \ldots, \mathcal{P}_n\}$.

We recall that in our protocols, both parties $\mathcal{P}$ and relays $\mathcal{R}$ maintain pairwise and global counters and variables, as described in the previous section. To ease the exposition, we describe our protocols implicitly assuming that each message is associated with its counter.

In describing our protocols in the $\mathcal{F}_{\text{SecureRobustRelay}}$-hybrid model, we present each command as separate `send`, `sendToAll`, `request` and `requestFromAll` commands. However, evaluating each layer of the circuit (bar those at depth zero) consists in each party executing a set of `send`/`sendToAll` commands followed, by a set of `request`/`requestFromAll` commands. In terms of the underlying synchronous communication model upon which the relays are built, a `send`/`sendToAll` command passed to the r-relays from party $\mathcal{P}_i$ is executed in one round. This means that, if P_i wants to send a message to P_j and there are no delays, then the `send` from $\mathcal{P}_i$ to all of the $\mathcal{R}_k$'s terminates within the same communication round. However, two consecutive `send` commands by party $\mathcal{P}_i$ destined for $\mathcal{P}_j$ and $\mathcal{P}_k$ will take up two rounds in the underlying synchronous communication model.

We will make use of the following sub-protocols and functionalities.

Sub-protocol $\mathsf{Open}(j, \langle x \rangle)$

INPUT: Each party $\mathcal{P}_i$ holds a share x_i of the unknown value x. We denote by ct_{x_i} the ciphertext corresponding to x_i according to an AEAD encryption scheme.
OUTPUT: $\mathcal{P}_j$ obtains x

$\mathsf{Open}(j, \langle x \rangle_t)$:
  1. For all $i \in [n]$, $j \neq i$ $\mathcal{P}_i$ calls $(\mathsf{send}, \mathcal{P}_i, \mathcal{P}_j)$ on $\mathcal{F}_{\text{SecureRobustRelay}}$, inputting a vector $\mathbf{x}$ after receiving ok, where $\mathbf{x}$ is the vector consisting of r-values ct_{x_i} and r is the number of relays. Let the associated τ value for these messages be $\tau_{i,j}$.
  2. $\mathcal{P}_j$ runs the sub-protocol $\mathbf{y} \leftarrow \mathsf{Receive}(j, 2t)$ below.
  3. $\mathcal{P}_j$ forms the set $\mathcal{M}$ of all indices for which $y_i \neq \bot$, and the vector $\mathbf{x}^{\mathcal{M}}$ of values $y_i \neq \bot$.
  4. $\mathcal{P}_j$ computes $P^{\mathcal{M}} \cdot \mathbf{x}^{\mathcal{M}}$ and outputs abort if the result is not equal to $\mathbf{0}$, where $P^{\mathcal{M}}$ is the parity check matrix, from Lemma 2.1, restricted to the set of parties $\mathcal{M}$.
  5. Party $\mathcal{P}_j$ computes $x \leftarrow \mathbf{r}^{\mathcal{M}} \cdot \mathbf{x}^{\mathcal{M}}$, where $\mathbf{r}^{\mathcal{M}}$ is the recombination vector restricted to the parties in $\mathcal{M}$, and returns this as the output of the procedure.

Figure 10. Procedure to open a sharing $\langle x \rangle$ towards a single party $\mathcal{P}_j$ or to $\mathcal{P}$

Sub-protocol $\mathsf{Open}(i, \langle x \rangle_t)$. Described in Figure 10, it takes a shared value $\langle x \rangle_t$ and opens it to $\mathcal{P}_i$.

Sub-protocol $\mathsf{Open}(\langle x \rangle_t)$. It takes a shared value $\langle x \rangle_t$ and opens it to all parties $\mathcal{P}$. The cost of $\mathsf{Open}(\langle x \rangle_t)$ and $\mathsf{Open}(i, \langle x \rangle_t)$ is r field elements per party, where r is the number of relays.

Sub-protocol $\mathsf{Receive}(i, \iota)$. It is described in Figure 11 and allows party $\mathcal{P}_i$ to receive a vector $\mathbf{y}$ of shares/values via a number of `request` to $\mathcal{F}_{\text{SecureRobustRelay}}$. The second input parameter ι, indicates the minimum number of shares $\mathcal{P}_i$ needs to receive to complete the command. Notice, in $\mathsf{Receive}(i, \iota)$ after executing enough `request` commands to complete the recovery of the secret shared value, we then execute `erase` commands for all other parties. Thus data which has been received is deleted on the relays, and data which is not received for this round is not stored by the relays when they do eventually receive it. These `erase` commands could be executed every so often, and not every execution of $\mathsf{Receive}(i, \iota)$, as they increase the number of underlying rounds needed. However, the less one executes them, the more data needs to be stored by the relays. Thus there is a trade-off, and we settle on executing the `erase` commands for every $\mathsf{Receive}(i, \iota)$ for expository purposes. It can be modified to obtain the sub-procedure $\mathsf{ReceiveFromAll}(i, \iota)$ by using the `requestFromAll` to

---

Sub-protocols Receive$(j, \langle x \rangle)$ and ReceiveFromAll(j, ι)

Receive(j, ι):
1. Party $\mathcal{P}_j$ initializes a vector $\mathbf{y}$ of length n containing $\bot$ in each location, bar location j where it places the value x_j.
2. $\mathcal{P}_j$ repeats the following step until the vector $\mathbf{y}$ contains at least ι non-$\bot$ values:
   (a) For $i \in [n]$, if $y_i = \bot$ then call $(\texttt{request}, \mathcal{P}_i, \mathcal{P}_j)$ on $\mathcal{F}_{\mathsf{SecureRobustRelay}}$, inputting $\tau_{i,j}$ after receiving ok, to obtain the value x_i. If the functionality return abort, then abort, otherwise place x_i in position i in the vector $\mathbf{y}$.
   Note, this may take many iterations since party $\mathcal{P}_j$ may possibly not yet have sent its message yet, or the adversary may be delaying messages.
3. Party $\mathcal{P}_j$ calls $(\texttt{erase}, \mathcal{P}_i, \mathcal{P}_j)$ on $\mathcal{F}_{\mathsf{SecureRobustRelay}}$, inputting $\tau_{i,j}$ on receiving ok, for all parties $\mathcal{P}_i$.
   This ensures that data on relays is either deleted, or not stored if it is not going to be called for.

ReceiveFromAll(j, ι):
1. Party $\mathcal{P}_j$ initializes a vector $\mathbf{y}$ of length n containing $\bot$ in each location, bar location j where it places the value x_j.
2. $\mathcal{P}_j$ repeats the following step until it receives at least ι non-$\bot$ values:
   (a) Call $(\texttt{requestFromAll}, \mathcal{P}, \mathcal{P}_j)$ on $\mathcal{F}_{\mathsf{SecureRobustRelay}}$, inputting $\boldsymbol{\tau} = (\tau_{i,\mathcal{P}})_i$ after receiving ok, to obtain the values $(x_i)_{i \neq j}$. If the functionality return abort, then abort, otherwise place $(x_i)_i$ in position i in the vector $\mathbf{y}$, for each i.
   Note, this may take many iterations since party $\mathcal{P}_j$ may possibly not yet have sent its message yet, or the adversary may be delaying messages.
3. Party $\mathcal{P}_j$ calls $(\texttt{eraseAll}, \mathcal{P}, \mathcal{P}_j)$ on $\mathcal{F}_{\mathsf{SecureRobustRelay}}$, inputting $\boldsymbol{\tau} = (\tau_{i,\mathcal{P}})_i$ on receiving ok, for all parties $\mathcal{P}_i$.
   This ensures that data on relays is either deleted, or not stored if it is not going to be called for.

Figure 11. Sub-Procedures to open a sharing $\langle x \rangle$

$\mathcal{F}_{\mathsf{SecureRobustRelay}}$. Notice that both variants allow the receiving parties to proceed as soon as they have received $2 \cdot t + 1$ shares, hence they do not need to wait for the full set of n shares.

Sub-protocol Input(i)

INPUT: Party $\mathcal{P}_i$ holds a secret value x
OUTPUT: Parties in $\mathcal{P}$ hold $\langle x \rangle_t$
This protocol assumes $\delta = 0$.

1. All the parties call $\mathcal{F}_{\mathsf{Rand}}$ for a new counter value cnt and obtains a sharing $\langle v \rangle$.
2. $\mathcal{P}_i$ runs Open$(\mathcal{P}_i, \langle v \rangle)$, to obtain the random value v. If the output is abort, then $\mathcal{P}_i$ outputs abort.
3. $\mathcal{P}_i$ computes $w \leftarrow x - v$ and calls $\mathcal{F}^{\delta=0}_{\mathsf{SecureRobustRelay}}$ on $(\texttt{sendToAll}, \mathcal{P}_i, \mathcal{P})$, inputting a vector $\mathbf{w}$ consisting of r-ciphertexts ct_w.
4. Each $\mathcal{P}_j$, for $\mathcal{P}_j \neq \mathcal{P}_i$, calls $(\texttt{request}, \mathcal{P}_i, \mathcal{P}_j)$, inputting τ on receiving ok, for the requisite value of τ.
   - If this returns $\bot$ then party $\mathcal{P}_j$ aborts.
   - If a value w is returned then call $(\texttt{erase}, \mathcal{P}_i, \mathcal{P}_j)$, inputting τ on receiving ok.
5. The parties set $\langle x \rangle \leftarrow \langle v \rangle + w$.

Figure 12. Sub-protocol Input(i) to enable party $\mathcal{P}_i$ to enter a secret value x into the computation.

Sub-Protocol $\Pi_{\mathsf{Input(i)}}$. It is a data-input sub-protocol given in Figure 12. It requires that all parties are in consensus about the value broadcast by party $\mathcal{P}_i$, thus we need all parties to terminate Input(i) before proceeding. This is unlike other parts of our MPC scheme, which allow faster parties to continue with the computation, i.e. they do not need to wait for all other parties to terminate

the protocol. The communication cost of the protocol is $\approx 2 \cdot r$ field elements per party for each input, where r is the number of relays, plus a call to $\mathcal{F}_{\mathsf{Rand}}$.

4.1 Functionalities $\mathcal{F}_{\mathsf{Rand}}$ and $\mathcal{F}_{\mathsf{Coin}}$

<div style="border:1px solid black; padding:1em;">

Functionality $\mathcal{F}_{\mathsf{Rand}}^{\mathsf{NI}}$

$\mathsf{init}()$: Wait for input from the adversary. If the adversary inputs abort, send the message abort to all parties.
$\mathsf{Rand}(\mathsf{cnt}, i, t)$: On input of $\mathsf{Rand}(\mathsf{cnt}, i, t)$ from party $\mathcal{P}_i$
    - If the counter value has not been used before, sample a Shamir sharing $\langle r \rangle_t$ of a random value r, store $(\mathsf{cnt}, \langle r \rangle_t)$ and send r_i to party $\mathcal{P}_i$.
    - Otherwise, retrieve r_i and send the value to $\mathcal{P}_i$.

</div>

Figure 13. The non-interactive functionality $\mathcal{F}_{\mathsf{Rand}}$

In our protocols, we use the relatively standard functionality $\mathcal{F}_{\mathsf{Rand}}^{\mathsf{NI}}$, which generates a random degree-t Shamir sharing $\langle r \rangle_t$. This is given in Figure 13, and it assumes a non-interactive method of generating the share values. For "reasonable" values of n and t_p this can be realised using a PRSS (pseudo-random secret sharing) [CDI05]. In [BBG+21], the authors propose a new PRSS scheme that greatly improve the efficiency of [CDI05] in some cases. This new construction can also be applied in our protocol. Notice, that our functionality only delivers the shares to the parties when asked, this is to enable our protocol to be described in a manner in which parties are not fully in sync with each other.

For larger values of n and t, interactive versions are possible, which would require few changes in the functionality and the usage we make of it. An interactive functionality $\mathcal{F}_{\mathsf{Rand}}$ is described in Figure 14. Before showing a protocol Π_{Rand} implementing it, we observe that a coin functionality, $\mathcal{F}_{\mathsf{Coin}}$, can also be implemented by using $\mathcal{F}_{\mathsf{Rand}}$ to generate a random $\langle x \rangle_t$ and calling $\mathsf{Open}(\langle x \rangle)$ to publicly reconstruct the value.

The interactive protocol Π_{Rand} [DN07] (Figure 15) uses a fixed Vandermonde matrix of size $(n-t) \times n$ as randomness extractor to generate $n-t$ random degree-t sharings $\langle r^1 \rangle, \ldots, \langle r^{n-t} \rangle$. Notice parties wait the input from all other parties, in particular each party $\mathcal{P}_i$ run the procedure $\mathsf{Receive}$ with $\iota = n - 1$, equivalently, we can assume the calls to $\mathcal{F}_{\mathsf{Rand}}$ only happen in the input stage of the MPC protocol where no delays occur. The amortized cost of Π_{Rand} is $\frac{n-1}{n-t}$ or, assuming a PRG, $\frac{n-t-1}{n-t}$ field elements per party.

We stress that this protocol does not securely realize $\mathcal{F}_{\mathsf{Rand}}$, as malicious parties may cheat and cause the resulting sharing to be incorrect. Roughly, when they send honest parties' shares these might correspond to a sharing that is either invalid or incorrect. This is true even in a setting with no delays.

To correctly implement $\mathcal{F}_{\mathsf{Rand}}$ against malicious parties, we need to add a check to verify the correctness of the shares [LN17]. This is given in Figure 16. If we denote by Π_{Rand}^+ the protocol Π_{Rand} augmented with the check $\Pi_{\mathsf{CheckShares}}$, then we have the following lemma. The amortized extra cost of this check is that of 2 openings, i.e. $2 \cdot r$ field elements per party.

Lemma 4.1. *The protocol Π_{Rand}^+ securely implements $\mathcal{F}_{\mathsf{Rand}}$ with abort against malicious adversaries corrupting up to $t < n/2$ computing parties and $r - 1$ relays in the $(\mathcal{F}_{\mathsf{Coin}}, \mathcal{F}_{\mathsf{SecureRobustRelay}})$-hybrid model with statistical error $\frac{1}{|\mathbb{F}|-1}$.*

Functionality $\mathcal{F}_{\mathsf{Rand}}$

The functionality runs with an adversary $\mathcal{S}$, n parties $\mathcal{P}_1, \ldots, \mathcal{P}_n$, and r relays $\mathcal{R}_1, \ldots, \mathcal{R}_r$.

$\mathsf{init}()$: The functionality is activated. If the adversary inputs abort, send the message abort to all parties.
$\mathsf{Rand}(\mathsf{cnt}, i, t)$: On input of $\mathsf{Rand}(\mathsf{cnt}, i, t)$ from $\mathcal{P}_i$:
  1. If the counter value has not been used before, wait to receive an input from the adversary. If it sends abort, halt; otherwise, when the adversary sends $(\mathtt{continue}, \{r_i\}_{i \in \mathcal{P}_I})$, then sample a random r and generate $\langle r \rangle_t$. Store $(\mathsf{cnt}, \langle r \rangle_t)$.
  2. Otherwise, retrieve r_i and send the value to $\mathcal{P}_i$.

Figure 14. The standard functionality $\mathcal{F}_{\mathsf{Rand}}$

Protocol Π_{Rand}

Let M be a $(n-t) \times n$ a fixed Vandermonde matrix.

  1. Each party $\mathcal{P}_i$ randomly creates $\langle s^i \rangle_t$ and call $(\mathtt{send}, \mathcal{P}_i, \mathcal{P}_j)$ on $\mathcal{F}_{\mathsf{SecureRobustRelay}}$. After receiving ok from the functionality, $\mathcal{P}_i$ inputs s^i_j. Let $\tau_{i,j}$ be the counters associated with these messages that will be stored by both parties and relays.
  2. Each $\mathcal{P}_i$ run $\mathbf{s} \leftarrow \mathsf{Receive}(i, n-1)$
  3. All parties locally compute $(\langle r^1 \rangle_t, \ldots, \langle r^{n-t} \rangle_t)^T = M \cdot (\langle s^1 \rangle_t, \ldots, \langle s^n \rangle_t)^T$

Figure 15. Protocol for random degree$-t$ sharing

Sub-protocol $\Pi_{\mathsf{CheckShares}}$

INPUT: Parties hold ℓ shares $\langle x^1 \rangle_t, \ldots, \langle x^\ell \rangle_t$

CheckShares:
  1. Parties call $\mathcal{F}_{\mathsf{Coin}}$, receiving random $\gamma_1, \ldots, \gamma_\ell$
  2. Parties run Π_{Rand}, obtaining $\langle r \rangle_t$
  3. Parties locally compute $\langle w \rangle_t = \sum_{i=1}^{\ell} \gamma_i \cdot \langle x^i \rangle_t + \langle r \rangle_t$
  4. Parties run $\mathsf{Open}(\langle w \rangle_t)$ and output either accept or abort accordingly to the output of this sub-protocol.

Figure 16. Procedure to check that ℓ shares are correct, i.e. that are neither invalid nor inconsistent.

Proof. We first describe the simulator $\mathcal{S}$. Recall that $\mathcal{P}_I$ and $\mathcal{R}_I$ denote the set of corrupt parties and corrupt relays, respectively.

Description of the simulation. When an honest party, $\mathcal{P}_i$, needs to distribute a random sharing $\langle s^i \rangle_t$, $\mathcal{S}$ samples t random shares of corrupt parties and emulates the command $(\mathtt{send}, \mathcal{P}_i, \mathcal{P}_j)$ of $\mathcal{F}_{\mathsf{SecureRobustRelay}}$ for all $j \neq i$. If $\delta = 0$, then it sends the corresponding ciphertexts and counters $\tau_{i,j}$ to the $\mathcal{R}_I$, otherwise receives an input $D \leq \delta$ from $\mathcal{A}$ and wait to send those values accordingly. When the command request on $\mathcal{F}_{\mathsf{SecureRobustRelay}}$ is called, forwards the ciphertexts on behalf of the honest relays. If $\mathcal{A}$ sends abort, $\mathcal{S}$ forwards it to $\mathcal{F}_{\mathsf{Rand}}$. For each corrupt relay, $\mathcal{S}$ calls the command request on behalf of the honest parties.

For each corrupt party $\mathcal{P}_i \in \mathcal{P}_I$, $\mathcal{S}$ receives the shares of $\langle s^i \rangle_t$ held by honest parties. This is done by emulating request, hence $\mathcal{S}$ forward abort to the functionality if something goes wrong. When all the shares are received, $\mathcal{S}$ learns $n-t$ shares for each $\langle s^i \rangle$, $i \in \mathcal{P}_I$. If any of these shares are either invalid or inconsistent, $\mathcal{S}$ forward abort to the functionality. Otherwise, it computes the shares $\langle r^i \rangle_t$ held by $\mathcal{P}_I$ and passes these shares to $\mathcal{F}_{\mathsf{Rand}}$.

Indistinguishability. Showing that $\mathcal{S}$ correctly simulated the behaviours of honest parties is quite standard and one can adapt the proof given for example in [GS20]. The only difference is that in our setting, we have $t < n/2$, hence the shares obtained by honest parties can be either invalid or inconsistent. In the real protocol, this is checked by $\Pi_{\mathsf{CheckShares}}$, while in the simulation, $\mathcal{S}$ can directly check the correctness of the received shares. Therefore, indistinguishability follows from the following lemma, whose proof can be found in [LN17].

Lemma 4.2. *If there exists a $j \in [m]$ such that $\langle x^j \rangle_t$ is not correct, then honest parties output* accept *in $\Pi_{\mathsf{CheckShares}}$ with probability at most $\frac{1}{|\mathbb{F}|-1}$.*

<div align="right">□</div>

Intuitively, it is possible to prove that if some of the $\langle x_i \rangle, i \in [\ell]$, are incorrect, then for any (possibly incorrect) sharing $\langle r \rangle$, $\langle w \rangle$ is also not correct except with negligible probability. The complete proof is given in [LN17].

We can define in a similar manner a functionality $\mathcal{F}_{\mathsf{DoubleRand}}$ that generates a random *double sharing* $(\langle r \rangle_t, \langle r \rangle_{2t})$. The interactive protocol is exactly the same as Π_{Rand}, except that in the first step parties create n double-sharings $(\langle s^i \rangle_t, \langle s^i \rangle_{2t})$ that are randomized using a Vandermonde matrix. For the non-interactive version one uses a PRSS to obtain $\langle r \rangle_t$, a PRZS to obtain $\langle 0 \rangle_{2t}$, from which one can obtain $\langle r \rangle_{2t}$ by locally computing $\langle r \rangle_t + \langle 0 \rangle_{2t}$.

4.2 Multiplication Protocols

Functionality $\mathcal{F}_{\mathsf{Mult}}$

Let $\mathcal{P}_I$ be the set of corrupt parties and $\mathcal{P}_H = \mathcal{P} \setminus \mathcal{P}_I$ the set of honest parties.
On input $(\texttt{Multiply}, \mathcal{P}', \mathsf{id}_x, \mathsf{id}_y, \mathsf{id}_z)$, where $\mathcal{P}' \subseteq \mathcal{P}$, and $\mathsf{id}_x, \mathsf{id}_y$ are present in memory. Retrieve the values x, y. Send $(\texttt{Multiply}, \mathcal{P}', \mathsf{id}_x, \mathsf{id}_y)$ to $\mathcal{S}$ along with corrupt shares, and wait for a reply. We can have the following cases:

  - If $\mathcal{S}$ sends $(\mathsf{abort}, \hat{H})$, forward abort to $\hat{H} \subseteq \mathcal{P}_H$
  - If $\mathcal{S}$ sends $(\texttt{Done}, \hat{\mathcal{P}}, \delta_a, \{z_i\}_{i \in \mathcal{P}_I})$, compute $x \cdot y + \delta_a$, construct a full sharing $\langle z \rangle_t$ using $x \cdot y + \delta_a$ and $\{z_i\}_{i \in \mathcal{P}_I}$ and distributes it to the honest parties in $\hat{\mathcal{P}}$. Moreover, store $(\mathsf{id}_z, \langle z \rangle_t)$.
  - If the adversary sends $(\texttt{Done}, \hat{H})$, retrieve the $(\mathsf{id}_z, \langle z \rangle)$ and sends $\{z_i\}_{\hat{H}}$ to parties in $\hat{H}$.

Figure 17. The functionality $\mathcal{F}_{\mathsf{Mult}}$ secure up to additive attacks

In our main MPC protocol we will use the the ideal functionality $\mathcal{F}_{\mathsf{Mult}}$ (given in Figure 17) to evaluate multiplication gates. $\mathcal{F}_{\mathsf{Mult}}$ takes two sharing $\langle x \rangle_t$ and $\langle y \rangle_t$ present in memory and outputs a sharing $\langle z \rangle_t = \langle x \cdot y + \delta_a \rangle_t$, where δ_a is a value chosen by the adversary.

$\mathcal{F}_{\mathsf{Mult}}$ is a delayed functionality and it works as follows. It takes as input $\langle x \rangle_t$ and $\langle y \rangle_t$. When a subset $\mathcal{P}' \subseteq \mathcal{P}$ of parties calls the functionality, $\mathcal{F}_{\mathsf{Mult}}$ sends a message to the ideal adversary $\mathcal{S}$. The ideal adversary $\mathcal{S}$ emulates $\mathcal{F}_{\mathsf{SecureRobustRelay}}$ and controls all the communications between the computing parties and relays receiving delay messages from the adversary $\mathcal{A}$. In the simulation, every time a party $\mathcal{P}_i$, or a subset of parties $\hat{\mathcal{P}}$, is able to request enough shares to compute the shared output of the multiplication gate, $\mathcal{S}$ checks if these shares are valid. If this is the case, it extracts the error δ_a that could have been introduced by $\mathcal{A}$ and forward the value δ_a, along with corrupt shares $\{z_i\}_{i \in \mathcal{P}_I}$ to the functionality which constructs a valid sharing of $x \cdot y + \delta_a$ consistent

with the corrupt shares obtained by $\mathcal{S}$. In addition, the functionality stores $\langle z \rangle_t$. If $\mathcal{S}$ detects same inconsistency in the output shares, then it sends an abort message to the functionality, together with the set of honest parties that in the simulation received those shares. Finally, when slower parties successfully conclude the evaluation of the multiplication gate, $\mathcal{S}$ just communicates to the functionality the indices of those parties, that will receive their consistent shares from $\mathcal{F}_{\mathsf{Mult}}$.

Crucially, since the $2t + 1$ or more shares used by the "fastest parties" to evaluate the multiplication gate are stored in the relays and used also by the slower parties later in the protocol, these shares fix any (potential) malicious behaviour. Indeed, if the first parties output abort when they reconstruct their shares, also slower parties that are going to use the same shares will output abort; on the other hand, if the fastest parties successfully evaluate the gate, then the shares used by those fix the (potential) additive error, so that the next set of parties concluding the gate evaluation either output the same value (and additive error) or abort.

Protocol Π_{Mult1}

INPUT: $\langle x \rangle_t$ and $\langle y \rangle_t$. Also we need random $\langle r \rangle_t$ and $\langle r \rangle_{2t}$
OUTPUT: $\langle z \rangle_t$, s.t. $z = x \cdot y$

Init: Parties call $\mathcal{F}_{\mathsf{DRand}}$ to obtain $\langle r \rangle_t$ and $\langle r \rangle_{2t}$.
Mult: 1. Each party locally computes $\langle v \rangle_{2t} = \langle x \rangle_t \cdot \langle y \rangle_t + \langle r \rangle_{2t}$
   2. Each $\mathcal{P}_i$ calls $\mathcal{F}_{\mathsf{SecureRobustRelay}}$ on $(\mathtt{sendToAll}, \mathcal{P}_i, \mathcal{P})$.
      After receiving ok, $\mathcal{P}_i$ inputs its share value v_i.
   3. Each party $\mathcal{P}_i$ runs the sub-protocol $\mathsf{ReceiveFromAll}(i, 2t + 1)$ to obtain the shares $v_j, j \neq i$.
      Note, each party has to receive $2 \cdot t + 1$ shares, $2 \cdot t$ are not enough to ensure the simulation is correct.
   4. Parties reconstruct $v = x \cdot y + r$ and locally compute $\langle z \rangle_t = v - \langle r \rangle_t$.
      Note, on reconstruction the parties may notice that the sharing is not a degree $2 \cdot t$ sharing, in which case they abort.

Figure 18. Protocol Π_{Mult1} to compute $\langle z \rangle_t$.

The protocol Π_{Mult1} (Figure 18), which we use to implement $\mathcal{F}_{\mathsf{Mult}}$, is an adaptation of [DN07, GLO$^+$21] and requires a communication of only r field elements per party, plus the cost of generating random $\langle r \rangle_t$ and $\langle r \rangle_{2t}$, that can be amortized as we have seen before.

Recall, the DN was proven to be *insecure* (i.e., not private) in the case of MPC protocols with strong honest majority via the double-dipping attack. However, in our setting, the double-dipping attack does not work anymore because corrupt parties have to send the same share (that can be incorrect) to all honest parties and, moreover, we do not have a special computing party playing the role of the king. We will prove this in the next lemma, showing that the view of the adversary controlling up to $t_p \leq t$ parties is independent from honest parties' shares.

Lemma 4.3. *The protocol Π_{Mult1} described in Figure 18 securely with abort implements the functionality $\mathcal{F}_{\mathsf{Mult}}$ in the $\{\mathcal{F}_{\mathsf{SecureRobustRelay}}, \mathcal{F}_{\mathsf{Rand}}\}$-hybrid model against a malicious adversary corrupting up to $t < n/2$ computing parties and $r - 1$ relays.*

Proof. We start by describing the simulator $\mathcal{S}$. As before, $\mathcal{P}_I$ denotes the set of corrupt parties and let $\mathcal{P}_H = \mathcal{P} \setminus \mathcal{P}_I$ denote the set of honest parties.

Description of the simulation. We recall that $\mathcal{S}$ keeps track of all the counters $\tau_{i,j}$ when it emulates the functionality $\mathcal{F}_{\mathsf{SecureRobustRelay}}$.

- $\mathcal{S}$ emulates $\mathcal{F}_{\mathsf{Rand}}$. Since by the privacy property of Shamir secret sharing with privacy threshold t or $2t$, the distribution of messages received by the adversary does not depend on the value secret shared, $\mathcal{S}$ just creates two random sharing $\langle r \rangle_t$ and $\langle r \rangle_{2t}$. It sends to $\mathcal{A}$ the shares held by corrupt parties.
- It receives the shares $\langle x \rangle_t, \langle y \rangle_t$ held by corrupt parties from $\mathcal{F}_{\mathsf{Mult}}$
- Then it starts to simulate the protocol. It sets the honest shares of v to be equal to honest shares $\langle r \rangle_{2t}$ plus some random value[6]. It emulates $\mathcal{F}_{\mathsf{SecureRobustRelay}}$ waiting for an input $D_i \leq \delta, i \in \mathcal{P}_H$, and sending to $\mathcal{A}$ the honest shares $\{v_i\}_{i \in \mathcal{P}_H}$ according to these D_i. $\mathcal{S}$ also receives corrupt shares $v_i, i \in \mathcal{P}_I$ from $\mathcal{A}$, sends request messages to corrupt relays (i.e. to $\mathcal{A}$) on behalf of honest parties accordingly to the delays schedule, and replies consistently to $\mathcal{A}$'s request messages. It sends abort to $\mathcal{F}_{\mathsf{Mult}}$ if something goes wrong during the emulation of $\mathcal{F}_{\mathsf{SecureRobustRelay}}$.
- $\mathcal{S}$ knows all the shares of $\langle v \rangle$, it reconstructs this value, simulating honest behaviour for corrupt parties.
- As soon as one or more parties $\hat{\mathcal{P}} \subseteq \mathcal{P}$ get enough shares to reconstruct the value $\tilde{v}$ (note each party needs to receive $2t + 1$ shares, so that the value $\tilde{v}$ is uniquely identified. If we wait only $2t$ shares, and say P_1 adds its own share and then say P_2 adds its own share in the same round, they might reconstruct to 2 different values. Note, in case $n = 2t+1$ this is not possible because if a party receives $2t$ shares from $2t$ other parties, there is no other party that can reconstruct to a different value. $\mathcal{S}$ checks if these shares are correct. If they are not, then it sends $(\mathsf{abort}, \hat{H})$ to the functionality, where $\hat{H} = \mathcal{P}_H \cap \hat{\mathcal{P}}$. Otherwise, if $\mathcal{A}$ does not send any abort, it checks the shares delivered to $\hat{\mathcal{P}}$. From these, reconstructs $\tilde{v}$ and computes $\langle \delta_a \rangle_{2t} = \langle \tilde{v} \rangle_{2t} - \langle v \rangle_{2t}$. Note this value does not depend on the honest shares, i.e. honest shares of $\langle \delta_a \rangle_{2t}$ are equal to 0. $\mathcal{S}$ also computes the shares $\{z_i\}_{i \in \mathcal{P}_I}$ held by corrupt parties and forwards $(\mathsf{Done}, \hat{\mathcal{P}}, \delta_a, \{z_i\}_{i \in \mathcal{P}_I})$ to the functionality.
- If the error δ_a was already communicated to the functionality, $\mathcal{S}$ just forward to $\mathcal{F}_{\mathsf{Mult}}$ the set of honest parties that can receive the output of the gate.

Indistinguishability. Consider the following hybrid experiments.

$\mathcal{H}_0$: This hybrid corresponds to the real execution.
$\mathcal{H}_1$: This hybrid is described as the previous one, except that here $\mathcal{S}$ computes the difference δ_a and the shares $\{z_i\}_{i \in \mathcal{P}_I}$ held by corrupt parties as described above. $\mathcal{S}$ sends δ_a and the corrupt shares to the ideal functionality; honest parties that successfully complete the computation output the shares received by $\mathcal{F}_{\mathsf{Mult}}$ instead of the real ones.
  Note that, when at least $2t + 1$ shares are delivered to parties in $\hat{\mathcal{P}}$, these shares uniquely determine a value $\tilde{v}$. Recall that all parties in $\hat{\mathcal{P}}$ receive the same set of shares. If these are inconsistent with some shares held by $\hat{\mathcal{P}}$, the corresponding parties output abort; otherwise they accept the value $\tilde{v}$, that at this point is fixed. Indeed, when more shares are delivered, and more parties complete the evaluation of the multiplication, there are only two possible options: either the new shares are consistent with the first $2t + 1$, so they obtain the same value $\tilde{v}$, or they output abort. Using $\tilde{v}$ and v, $\mathcal{S}$ computes $\langle \delta_a \rangle_{2t}$ and reconstructs δ_a. Hence, it computes all the corrupt shares $\{z_i\}_{i \in \mathcal{P}_I}$. From these and $x \cdot y + \delta_a$ the honest shares are uniquely determined. The distributions $\mathcal{H}_0$ and $\mathcal{H}_1$ are indistinguishable.

---

[6] Usually, when we have $n = 2 \cdot t + 1$, the honest shares are set to be just random values, because a corrupt party, i.e. the adversary, will receive $2 \cdot t + 1$ shares and these uniquely fix a secret. In our case, a corrupt party can receive in theory more that $2 \cdot t + 1$ shares, and if in the real execution these are consistent, they should be consistent in the ideal one as well. So we need that the adversary is able to receive a correct sharing to ensure this.

$\mathcal{H}_2$: Here $\mathcal{S}$ simulate the honest parties in the entire multiplication protocol. This hybrid is the same as the previous one, except that in the previous one $\mathcal{S}$ uses the real shares of $\langle v \rangle_{2t}$ and now $\mathcal{S}$ uses consistent random values instead. Since the shares of $\langle r \rangle_{2t}$ are uniformly random, the distributions of the two hybrids are indistinguishable.

By observing that $\mathcal{H}_2$ corresponds to the ideal execution, we conclude the proof.

□

4.3 Instantiating $\mathcal{F}_{\mathsf{Mult}}$ with Maurer's protocol

We could be tempted to instantiate the ideal functionality $\mathcal{F}_{\mathsf{Mult}}$ with Maurer's multiplication protocol Π_{Mult} (Figure 2). Surprisingly, this approach does not work and in order to use Π_{Mult} we should change the way we model additive attacks significantly. This is because with this protocol, the additive attack A is not uniquely determined for all parties $\mathcal{P}$. More concretely, when a corrupt party $\mathcal{P}_i$ acts as a dealer and produces a sharing of $\langle v_i \rangle_t$, then it can send inconsistent shares to honest parties, so that different sets of parties reconstruct different values without detecting any malicious behaviour. The crucial difference is that in Π_{Mult} parties use the pairwise `send` command of $\mathcal{F}_{\mathsf{SecureRobustRelay}}$, while in Π_{Mult1}, parties use the "global" command `sendToAll`. It is possible to show that again a simulator would be able to reconstruct the additive attack A, but this will not consists any more of a single value A^c for each multiplication-output wire, but rather of different values $A^c_{\hat{\mathcal{P}}}, A^c_{\bar{\mathcal{P}}}, \dots$ for different subsets $\hat{\mathcal{P}}, \bar{\mathcal{P}}, \dots \subset \mathcal{P}$. We remark that instantiating $\mathcal{F}_{\mathsf{Mult}}$ with Π_{Mult} has the advantage of not requiring random secret shared values from $\mathcal{F}_{\mathsf{Rand}}$ and $\mathcal{F}_{\mathsf{DRand}}$, but the disadvantage of more communication since each party needs to send $n-1$ field elements, while in Π_{Mult1} each party only sends r elements plus the communication required by instantiations of $\mathcal{F}_{\mathsf{Rand}}$ and $\mathcal{F}_{\mathsf{DRand}}$ that can be amortized as shown in the previous section.

5 MPC Secure up to an (Internal) Additive Attack Using Secure Robust Relays

In this section, we show how to run an MPC protocol with an honest majority using the network model described in the previous sections, i.e., according to the functionality $\mathcal{F}_{\mathsf{SecureRobustRelay}}$, which is secure up to a form of additive attack. Assuming only adversaries which do not delay messages, this is relatively simple[7], thus our main challenge is to efficiently deal with delays. Our protocol will allow the parties not being delayed, (or being delayed less) to proceed without waiting messages from the slowest parties and as soon as they have the required $2 \cdot t + 1$ values from other parties. Any messages that are sent to the delayed parties will of course be stored in the relays until they are needed. As an extreme example, assuming a large enough number of participating parties, a party could simply send no messages, wait for the other parties to finish the computation, and then request all of the messages from other parties to compute the output for themselves.

5.1 The δ-iaa MPC protocol in the $\mathcal{F}_{\mathsf{SecureRobustRelays}}$-hybrid model

The protocol Π_{PMPC} to securely (up to internal additive attacks) evaluate a randomized arithmetic circuit C over a finite field $\mathbb{F}$ is given in Figure 19. At a high level the protocol proceeds in three

---

[7] No delays means that the relays function like a regular point to point network on which one can run general MPC protocols.

stages: an *input stage*, that is instantiated with the sub-protocol Π_{Input}, described in Figure 12, and with calls to the ideal functionalities $\mathcal{F}_{\text{Rand}}$ and $\mathcal{F}_{\text{DRand}}$, an *evaluation stage*, consisting in the evaluation of linear and multiplicative gates, and an *output stage*, where parties call the sub-protocol Open, given in Figure 10. We start with a protocol Π_{PMPC} that evaluates a (randomized) circuit C in the $\mathcal{F}_{\text{SecureRobustRelay}}$-hybrid model with security against δ-delaying passive adversaries, except for an actively secure input step without delays, i.e. with $\delta = 0$. We show that, when Π_{PMPC} is executed in the presence of an active adversary, it computes a circuit C' that is the same as C up to some internal additive attacks. The key point is that, if the actively secure input protocol completes, then we know that the share values in the input gates are correct and the output gates are self-authenticating in that Π_{Open} will output abort if the input shared value is not a valid Shamir sharing. Thus the only place where the adversary can introduce errors, and avoid an abort, is by transmitting the sharing of the wrong value in the multiplication protocol. This wrong value will equate to the adversary introducing a known error, as per Definition 2.3.

Note that, whilst the underlying network is synchronous and the underlying MPC protocol proceeds in what looks like "rounds" of interaction, due to the delays, the parties can actually be at different rounds of the MPC protocol at the same point in time.

More formally, we prove the following theorem.

Theorem 5.1. *Given a randomized circuit C, the protocol Π_{PMPC} (Figure 19) for computing C is secure against any δ-delaying passive adversary controlling up to $t < n/2$ parties and $r - 1$ relays in the $\{\mathcal{F}_{\text{DoubleRand}}, \mathcal{F}_{\text{Rand}}, \mathcal{F}_{\text{Mult}}, \mathcal{F}_{\text{SecureRobustRelay}}\}$-hybrid model. In addition, Π_{PMPC} securely evaluates a circuit C' with abort against δ-delaying active adversaries controlling up to $t < n/2$ parties and $r - 1$ relays, where C' is a corruptible version of C that additionally takes an input A, which specifies an additive attack on each internal wire of C from the adversary, and outputs the result of the additively corrupted C as specified by A to a subset $\hat{\mathcal{P}} \subseteq \mathcal{P}$ of parties.*

Protocol Π_{PMPC}

The protocol takes as input a randomized arithmetic circuit C. The protocol executes all the gates at a given depth in parallel. We proceed from depth zero to depth d. The depth zero gates consist of input, random-input and linear gates. At depth greater than zero there are linear, multiplication and output gates.

Setup*: Parties compute the randomness needed for evaluating multiplication gates, calling $\mathcal{F}_{\text{DoubleRand}}$.
  This step depends on the actual instantiation of $\mathcal{F}_{\text{Mult}}$, so it may not be necessary.
Input: Input gates with label i, corresponding to party $\mathcal{P}_i$, are executed by running the sub-protocol Input(i). This produces a secret shared value $\langle x \rangle_t$.
Random-Input: Random input gates are executed by calling the Rand command on the functionality $\mathcal{F}_{\text{Rand}}$ for a value cnt uniquely associated with the gate. This produces a secret shared random value $\langle r \rangle_t$ (or $\langle r \rangle_{2t}$).
Linear Gates: A linear gate with fan-in f and input $x^1, \ldots, x^f$ is of the form $y \leftarrow v_0 + \sum_{j=1}^{f} v^j \cdot x^j$, for given constants $v^j \in \mathbb{F}$. To execute these gates, parties utilize the fact that Shamir sharing is a linear secret sharing scheme in order to compute $\langle y \rangle_t \leftarrow \langle v^0 \rangle_t + \sum_{j=1}^{f} v^j \cdot \langle x^j \rangle_t$.
Multiplication Gates: All the multiplication gates at depth l are executed in parallel. Let w_l be the number of such gates with inputs $\langle x^k \rangle_t$, $\langle y^k \rangle_t$ producing output $\langle z^k \rangle_t$, for $k \in [w_l]$. Each party call $\mathcal{F}_{\text{Mult}}$ on input (Multiply, $\mathcal{P}'$, $\text{id}_{x_i^k}$, $\text{id}_{y_i^k}$).
Output Gates: An output gate is executed by running the sub-protocol Open($\langle y \rangle_t$).

Figure 19. The protocol Π_{PMPC} for δ-iaa secure MPC

Proof. Privacy against semi-honest adversary comes directly from the privacy of the underlying building blocks and properties of Shamir's secret sharing scheme. To prove the second part of the theorem, we proceed as follows. We prove that the protocol can be simulated against a malicious adversary up to the output step. In this way we describe what in Genkin et al. is called simulator for weak privacy against a malicious adversary. Simultaneously, we construct the simulator up to additive attacks extracting the additive errors during the evaluation (which does not change the view of the adversary).

Description of the simulation. Through the simulation $\mathcal{S}$ emulates $\mathcal{F}_{\mathsf{SecureRobustRelay}}$ to send honest shares to $\mathcal{A}$ and receive corrupt shares from $\mathcal{A}$, following the delays schedule dictated by $\mathcal{A}$. It keeps track of all the tags $\tau_{i,j}$, for all $i,j \in \mathcal{P}$, and $\tau_{i,\mathcal{P}}, \forall i \in \mathcal{P}$. The simulator initializes an internal additive attack A.

- $\mathcal{S}$ emulates $\mathcal{F}_{\mathsf{Rand}}$ and $\mathcal{F}_{\mathsf{DRand}}$ according to the protocol and sends corrupt shares to $\mathcal{A}$.
- When it simulates the opening procedure in the input stage, $\mathcal{S}$ reconstructs all corrupt parties' input. If, during these openings, some inconsistency is detected, the $\mathcal{S}$ also outputs abort to the ideal functionality and halts.
- During the circuit evaluation, when a subset of parties $\mathcal{P}'$ evaluate a multiplication gate g^a, $\mathcal{S}$ emulates the functionality $\mathcal{F}_{\mathsf{Mult}}$. Note that it can compute both the "honest" messages the adversary should have sent based on its internal state, i.e., based on the evaluation of previous gates, and both the actual messages sent by $\mathcal{A}$. As proved in Lemma 4.3, the difference δ_a between the actual messages and the honest messages is an additive term that corresponds to the additive attack on the output of the multiplication gate. Since this value is the same for all the parties that successfully conclude the evaluation of the gate, it uniquely determines the additive attack corresponding to the output wire of the gate g^a. More formally, the simulator determines the entry for the additive attack corresponding to the output wire of g^a: let g^b be any non-output gate connected to the output wire of the multiplicative gate g^a, it holds that $A_{a,b} = \delta_a$.
- At the end of the evaluation phase, for each output gate g^{Out}, each corrupt parties $\hat{\mathcal{P}}_I$, holds a share of the supposed output. $\mathcal{S}$ sets to 0 all the coordinates of A that were not previously set. Then sends to the ideal functionality all the inputs of the corrupt parties $\mathcal{P}_I$ and the additive attack A. The functionality replies with the output $\mathbf{y}$. For each output gate g^{Out} of C that is connected to an output of some gate g^a, $\mathcal{S}$ chooses honest shares of y_{Out} that are compatible with corrupt shares. Then, it simulates the opening of the shares in $\hat{\mathcal{P}}$, i.e., among the parties actually running this step according to the delays schedule. If this step terminates correctly, it sends $(\mathtt{Done}, \hat{H})$ to the corresponding ideal functionality, otherwise it returns abort.

Indistinguishability. First we show that the view of $\mathcal{A}$ during circuit evaluation is distributed identically regardless of the input held by honest parties. We can observe that the view of $\mathcal{A}$ excluding the output step during a real execution of the protocol consists of the input of corrupt parties and all the messages received to $\mathcal{P}_I$ and $\mathcal{R}_I$ during the input step and evaluation of the multiplication gate. When parties call the ideal functionalities $\mathcal{F}_{\mathsf{Rand}}$ and $\mathcal{F}_{\mathsf{DRand}}$, there is no communication so the two executions are indistinguishable. We recall that during the input step, we set $\delta = 0$ and that all the communications are simulated by calling the ideal functionality $\mathcal{F}_{\mathsf{SecureRobustRelay}}$. The only difference between the two executions during the input stage is that in the ideal execution $\mathcal{S}$ uses random shares as input of the honest parties. Since $|\mathcal{P}_I| \leq t$, by the privacy of Shamir's secret sharing with privacy threshold t or $2t$, the distribution of the messages received by $\mathcal{A}$ does

not depend on the secret shared value. The distributions of shares of corrupt parties in both executions are the same. In addition, in both executions, the opening sub-protocol will output abort independently of honest shares (see Lemma 2.1). During the evaluation of multiplication gates, $\mathcal{S}$ emulates $\mathcal{F}_{\mathsf{Mult}}$. Again, here the two executions are indistinguishable. This proves that the view of the adversary up to the point where the output value is opened has exactly the same distribution in the real and simulated case. Now, if the ideal and simulated protocol proceed to the last step, it means that enough parties $\hat{\mathcal{P}}$ concluded the circuit evaluation. Note that since the delay schedule is dictated by $\mathcal{A}$, both the executions start this last step with the same set of parties. The only new messages that $\mathcal{A}$ can see in the output step is the output value $\mathbf{y}$ and same shares of honest parties in $\hat{\mathcal{P}}$. These are random shares that are consistent with the output $\mathbf{y}$ and corrupt shares. If during the opening step, $\mathcal{A}$ sends incorrect values, then $\mathcal{S}$ outputs abort. Indistinguishability is again guaranteed by Lemma 2.1. When more parties complete the evaluation of the circuit, then either new shares from parties in $\mathcal{P}_I$ are consistent with the sharing stored by $\mathcal{S}$ and the ideal functionality, or ideal execution sends abort to the functionality. The same also happens in the real protocol, indeed indistinguishability of the output as well as of $\mathcal{A}$'view in the last round depends on whether or not the additive errors A were correctly computed by $\mathcal{S}$. The correctness of A is guaranteed by $\mathcal{F}_{\mathsf{Mult}}$ and from the fact that such an attack, also in a real execution, is fixed by the first $2t + 1$ shares used by the parties to complete the evaluation of multiplication gates. Therefore, in the simulation the output value $\mathbf{y}$ is the correct evaluation of C on the inputs matching the shares extracted by $\mathcal{S}$ and the additive attack A computed by $\mathcal{S}$.

5.2 Modeling the State Size of the Relays

The size of state that each relay must store for our MPC protocol can be bounded and modelled under various scenarios. In this section we examine this in the worst case, best case and (a specific form of) average case.

The storage is dominated, for most circuits, by the number of multiplications (i.e. we can ignore the input and output gates in general). The number of multiplication gates can in turn be bounded by the multiplicative width w per layer of the circuit, and there are d such layers. In the multiplication protocol each message to a relay consists of just a single field element, and so can be represented by $\log_2(p)$ bits.

Worst Case Analysis: In the worst case, assuming a δ-delaying active adversary, the way the adversary can force the relays to store the most state is to choose a minimal number (i.e. $2 \cdot t + 1$) of parties to complete the computation and delay the remaining parties indefinitely. Each party not selected to complete the protocol will repeatedly get delayed δ messages for each message they wish to send, i.e. they will be allowed to send a single command to the relays, then delayed for a further δ rounds, and so on. As the commands that are delayed include all the necessary erase commands, the maximum state the relays will need to store may contain all messages. So the state size each relay needs to keep in the worst case scenario is dominated by the term

$$n \cdot w \cdot d \cdot \log_2 p.$$

Best Case Analysis: In the best case the adversary introduces no delays. Considering only the multiplication gates (again for simplicity) we see that each gate requires the relay to store n values.

Simulation for average case analysis of the MPC protocol

INPUT PARAMETERS: n, t, d, w, δ and a probability p.

Simulator:
1. $\mathsf{rnd} \leftarrow 0$, $\quad \mathsf{state} \leftarrow 0$.
2. For $i \in \{1, \ldots, n\}$ do
   (a) $\mathsf{depth}_i \leftarrow 0$, $\quad \mathsf{comm}_i \leftarrow \mathsf{send}$, $\quad \delta_i \leftarrow 0$.
3. For $i \in \{0, \ldots, d-1\}$ do
   (a) $\mathbf{s}_i \leftarrow \mathbf{0} \in \{0,1\}^n$. *Stores in $\mathbf{s}_i[j]$ whether party j has sent it's messages at depth i.*
   (b) $\mathbf{e}_i \leftarrow \mathbf{0} \in \{0,1\}^n$. *Stores in $\mathbf{e}_i[j]$ whether party j has requested an erase at depth i.*
4. While $\exists i$ such that $\mathsf{depth}_i \neq d$ do
   (a) For all $i \in [1, \ldots, n]$ such that $\mathsf{depth}_i \neq d$ do
      i. $\pi \leftarrow [0,1]$. *Pull a value to decide whether to delay this command for party $\mathcal{P}_i$.*
      ii. If $\pi < p$ and $\delta_i < \delta$ then
         A. $\delta_i \leftarrow \delta_i + 1$.
      iii. Else
         A. $\delta_i \leftarrow 0$.
         B. If $\mathsf{comm}_i = \mathsf{send}$ then execute sub-procedure `SendToAll` below.
         C. Else if $\mathsf{comm}_i = \mathsf{request}$ then execute sub-procedure `RequestFromAll` below.
         D. Else execute sub-procedure `EraseAll` below.
      iv. If $\mathsf{depth}_i = d$ then output "party i finished in round rnd".
   (b) Output "State size in round rnd is state".
   (c) $\mathsf{rnd} \leftarrow \mathsf{rnd} + 1$.
5. Output "End of simulation".

`SendToAll` :
1. $\mathbf{s}_{\mathsf{depth}_i}[i] \leftarrow 1$.
2. $\mathsf{state} \leftarrow \mathsf{state} + w$. *Increase state as we are sending data.*

`RequestFromAll` :
1. If $\mathbf{s}_{\mathsf{depth}_i}$ has Hamming weight at least $2t + 1$ then
   (a) $\mathsf{comm}_i \leftarrow \mathsf{erase}$. *Complete the multiplication for party $\mathcal{P}_i$.*

`EraseAll` :
1. $\mathbf{e}_{\mathsf{depth}_i}[i] \leftarrow 1$.
2. If $\mathbf{e}_{\mathsf{depth}_i}$ equals the all one vector then
   (a) $\mathsf{state} \leftarrow \mathsf{state} - n \cdot w$. *Decrease state as everyone has read it.*
3. $\mathsf{comm}_i \leftarrow \mathsf{send}$, $\mathsf{depth}_i \leftarrow \mathsf{depth}_i + 1$. *Pass to the next level for party $\mathcal{P}_i$.*

Figure 20. Simulator for average case analysis of the MPC protocol

However, as we pass from depth d to depth $d + 1$ the parties execute the `erase` commands needed to remove this state. Thus the maximum state needed to be stored in the best case is

$$n \cdot w \cdot \log_2 p.$$

In this case we can also bound the number of rounds of communication in the underlying synchronous network on which the relays are built. Each MPC multiplication requires each party to send (sequentially) one `sendToAll` command, one `requestFromAll` commands and one `eraseAll` commands; each of which requires one round of communication. The parties operate in parallel, thus across all parties the total number of rounds needed, when there are no delays, is

$$3 \cdot d.$$

Average Case Analysis: To understand what happens in the intermediate cases we examine in this sub-section the situation where a delay is applied to a message with a given probability p. We

then simulate (for a given set of parties n and threshold t) the number of rounds and the relay state size to execute a circuit of multiplicative depth d and multiplicative width w, assuming a maximum delay of δ. Again we concentrate just on the multiplication sub-protocol as this will dominate for most circuits. Our simulation is given in Figure 20, and it assumes each level of the circuit consists of w multiplication gates.

From the simulation we can compute the average (over time) and worst case state-size, as well as the number of rounds needed for *some* parties to complete the execution, as well as *all* parties. We ran two sets of experiments to demonstrate the effect of the delays. Both sets were ran with $w = 1$, as adding more would simply scale the state size with w, with the value of δ was set to 5, and the number of rounds of the computation was 3. Regardless of the number of parties, t is set to 1.

In Figure 21, the size of state through six experiments is shown. Each combination of between $n \in \{4, 5, 6\}$ parties and a 25% chance to get delayed or no delays are represented here. The simulation without delays will always be the same (i.e. it proceeds deterministically). When there are no delays the state rises to n, stays there while everyone reads and asks for the removal of all the messages. Then the state size drops to zero for a round until everyone sends messages again. Thus the simulation for no-delays is regular, and we represent it in the figure by a dotted line. The non-dotted line represents the simulation when we have a 25% chance of delaying each specific command. Here we see that the drop in the state size is less regular, and the maximum state size can increase past n if parties are delayed.
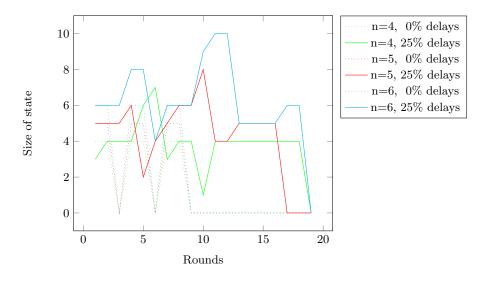

Fig. 21. Size of state for sample runs of the simulation

In Figure 22, for each value of n in $\{4, 5, 6\}$ we a run $10,000$ simulations, each with a 25% chance of delay. In the graph we show the proportion of parties on the y-axis and the round in which they finished on the x-axis. The random distribution of the delays in the simulation causes the finish position to become a bell curve as the finish position is dependent on the minimum number of rounds added to the number of rounds that party is delayed and the number of rounds they need to wait on enough messages being sent from other parties. We notice that for $n = 4$ the distribution of finishing rounds is skewed slightly to the right compared to $n = 5$ and $n = 6$. This is because

with such a small number of parties (for the same value of $t = 1$) a delay in two parties will slow down everyone. Whereas for $n = 5$ if two parties are delayed, the other three can proceed without waiting for the slow two. Thus on average parties will terminate faster.
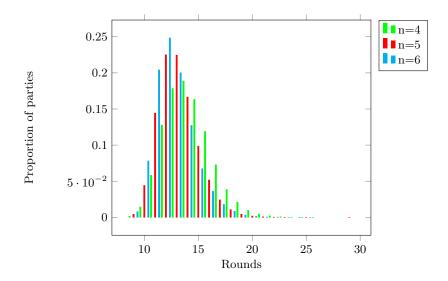

Fig. 22. The distribution of the round in which a party terminates, in our simulation of our MPC protocol applied to a circuit with multiplicative depth three with 25% chance of delay on each parties commands to the relay.

6 Actively Secure MPC-with-Abort Using Secure Robust Relays

In Figure 23, we give our functionality $\mathcal{F}_{\mathsf{DelayedMPC}}$, with the associated implementing protocol $\Pi_{\mathsf{DelayedMPC}}$, being given in Figure 24. Intuitively, the functionality presents a variant of the standard active-with-abort MPC security definition, modified to the situation where we have delays, so that some subset of parties can conclude the evaluation before others. In the previous section, we presented a protocol Π_{PMPC} which achieves security up to δ-internal additive attacks. In this section, we compile the prior protocol into one which does not allow internal additive attacks. This is done by following the same approach described in [GIP+14], in which it was proved that every circuit C can be compiled to a *robust* circuit $\tilde{C}$, i.e., the circuit itself protects the protocol against internal additive attacks. Therefore, we simply apply Π_{PMPC} to a robust circuit and prove that this is enough to achieve our goal.

Unlike [GIP+14], we do not need to compile into the robust circuit an error correcting code, since this comes "for free" with Shamir's secret sharing. Thus our definition of a robust circuit differs slightly from the one in [GIP+14] and it is perhaps closest to the definition given in [CGG+21] even if we do not need to preserve each wire value in the state across committees, therefore our definition becomes simpler than that considered in [CGG+21].

Definition 6.1 (Robust Circuit). *Given an arithmetic circuit C for a functionality f of depth d and width $w = \max\{w_1, \ldots, w_d\}$, a robust circuit $\tilde{C}$ corresponding to C is a circuit that realizes the functionality $\tilde{f}$ that computes:*

Functionality $\mathcal{F}_{\mathsf{DelayedMPC}}(\mathcal{P}_1, \ldots, \mathcal{P}_n)$

The functionality runs with parties $\mathcal{P}_1, \ldots, \mathcal{P}_n$ and an adversary $\mathcal{S}$. Let C be a (randomized) arithmetic circuit

Initialise: On input $(\mathtt{init}, \mathbb{F})$ from all parties, if $(\mathtt{init})$ was received before then ignore the command, otherwise store $\mathbb{F}$.

Evaluation: The functionality receives input from parties and adversary. Evaluate the circuit C and compute the output. Send the output to $\mathcal{S}$.

Output: The functionality wait an input from the adversary. When $\mathcal{S}$ sends $(\mathtt{abort}, \hat{H})$, send $\mathtt{abort}$ to honest parties $\hat{H} \subseteq \mathcal{P}_H$; when $\mathcal{S}$ sends $(\mathtt{Done}, \hat{\mathcal{P}})$, send the output to honest parties in $\hat{\mathcal{P}}$.

Figure 23. The delayed MPC functionality

Protocol $\Pi_{\mathsf{DelayedMPC}}$

The protocol takes as input a randomized arithmetic circuit C and the corresponding robust circuit $\tilde{C}$. The protocol executes all the gates at a given depth in parallel. We proceed from depth zero to depth d. The depth zero gates consist of input, random-input and linear gates. At depth greater than zero there are linear, multiplication and output gates.

Evaluation: Parties evaluates the robust circuit $\tilde{C}$ on their private input using Shamir's secret sharing scheme to run the protocol Π_{PMPC}.

Output: This is the final stage of the protocol and parties $\hat{\mathcal{P}} \subseteq \mathcal{P}$ reaching this point hold the output of $\tilde{C}$, i.e. $\langle \mathbf{z} \rangle_t = \langle \mathbf{y} + \mathbf{c} \cdot Z \rangle_t$ and $\langle T \rangle_t = \langle \beta \cdot Z \rangle_t$.
   1. Each $\mathcal{P}_i \in \hat{\mathcal{P}}$ runs $\mathsf{Open}(\langle T \rangle)$ and $\mathsf{Open}(\langle \mathbf{z} \rangle)$ to obtain the values T and $\mathbf{z}$. If either of these protocols outputs $\mathtt{abort}$ then output $\mathtt{abort}$.
   2. If $T \neq 0$ then output $\mathtt{abort}$, otherwise return $\mathbf{z}$.

Figure 24. The protocol $\Pi_{\mathsf{DelayedMPC}}$ for secure delayed MPC

ORIGINAL OUTPUT: *Compute* $\mathbf{y} = C(\mathbf{x})$ *for some inputs* $\mathbf{x}$.
RANDOM VALUES: *Sample random values* $\Delta, \beta, \alpha_{k,l} \in \mathbb{F}$, *where each* $\alpha_{k,l}$ *is associated to the* k-*th multiplication gate at depth* l, *and* $\mathbf{c} \in \mathbb{F}^{|\mathbf{y}|}$.
LINEAR COMBINATIONS: *Computes the following linear combinations*

$$u = \sum_{l=1}^{d} \left(\sum_{k=1}^{w_l} \alpha_{k,l} \cdot z_{k,l} \right),$$

$$v = \sum_{l=1}^{d} \left(\sum_{k=1}^{w_l} \alpha_{k,l} \cdot (\Delta \cdot z_{k,l}) \right),$$

where $z_{k,l}$ *corresponds to the output of the* k-*th multiplication at depth* l.
ZERO CHECK VALUE: *Compute* $Z = \Delta \cdot u - v$.
FINAL OUTPUT: *Output* $(\mathbf{z}, T) = (\mathbf{y} + \mathbf{c} \cdot Z, \ \beta \cdot Z)$.

A robust circuit can be computed from a standard circuit with a small increase in depth and a linear increase in the width, as the following lemma demonstrates.

Lemma 6.1. *Any arithmetic circuit C for functionality f, with input $\mathbf{x}$, output $\mathbf{y}$, depth d and width w, can be transformed into a robust randomized circuit $\tilde{C}$ for functionality $\tilde{f}$ of Definition 6.1 of depth $d + 4$ and maximum width $4 \cdot w$, assuming $|\mathbf{x}| \leq w$.*

43

Evaluating the robust circuit $\tilde{C}$ instead of C implies that all the secret shared values in the circuit are randomized using a random secret-shared value Δ. In particular, during the input stage the robust circuit requires additional calls to $\mathcal{F}_{\mathsf{Rand}}$ to receive $\langle\Delta\rangle_t$, $\langle\beta\rangle_t$, $\langle\mathbf{c}\rangle_t$ $\langle\alpha_{k,l}\rangle$, $k \in [w], l \in [d]$, and computation of $\langle x_i \cdot \Delta\rangle$ for each input x_i. The evaluation of the circuit is required also on these randomized values: for linear gates with inputs $y_1, \ldots, y_s$ it is enough to evaluate the gates also on inputs $\langle y_1 \cdot \Delta\rangle, \ldots, \langle y_s \cdot \Delta\rangle$; for multiplication gates with inputs $x_{k,l}, y_{k,l}$, it is necessary to compute $\langle z_{k,l} \cdot \Delta\rangle = \langle\Delta \cdot x_{k,l} \cdot y_{k,l}\rangle$ other than $\langle z_{k,l}\rangle = \langle x_{k,l} \cdot y_{k,l}\rangle$, which doubles the number of multiplications for each round. Finally, the output of the randomized evaluation is used to check the correctness of the non-randomized one. To this end, the circuit $\tilde{C}$ requires the computation of the two values u and v as in Definition 6.1. Instead of computing these two values at the end of the computation, and hence needing to maintain a huge state for all randomized and non-randomized values, we incrementally compute random linear combinations of the output of multiplication gates at each round. More precisely, at depth l we have the values

$$\langle u_{l-2}\rangle = \sum_{l=1}^{l-2}\left(\sum_{k=1}^{w_l}\langle\alpha_{k,l}\rangle \cdot \langle z_{k,l}\rangle\right)$$

$$\langle v_{l-2}\rangle = \sum_{l=1}^{l-2}\left(\sum_{k=1}^{w_l}\langle\alpha_{k,l}\rangle \cdot \langle\Delta \cdot z_{k,l}\rangle\right)$$

and we compute

$$\langle u_{l-1}\rangle = \langle u_{l-2}\rangle + \langle\alpha_{k,l-1}\rangle \cdot \langle z_{k,l-1}\rangle)$$
$$\langle v_{l-1}\rangle = \langle v_{l-2}\rangle + \langle\alpha_{k,l-1}\rangle \cdot \langle\Delta \cdot z_{k,l-1}\rangle).$$

Notice that at level l we compute the random linear combination related to multiplications at depth up to $l-1$. In this way, these operations do not increase the number of rounds.

At the end, when the evaluation of C is concluded, so that some subset of parties $\hat{\mathcal{P}}$ holds $\langle\mathbf{y}\rangle_t = \langle C(\mathbf{x})\rangle$, $\langle u\rangle_t, \langle v\rangle_t$, $\tilde{C}$ requires the computation of $\langle Z\rangle_t = \langle\Delta \cdot u - v\rangle$. Parties in $\hat{\mathcal{P}}$ then open both $\langle T\rangle = \langle\beta \cdot Z\rangle$ and $\langle\mathbf{z}\rangle = \langle\mathbf{y} + \mathbf{c} \cdot Z\rangle$. The value of T will be zero if and only if all the values, in particular all the multiplication gates, are correctly computed, except with some negligible probability. If T is zero then $\mathbf{z}$ will equal the correct output of the circuit $\mathbf{y}$, otherwise it will equal a uniformly random value in $\mathbb{F}^{|\mathbf{y}|}$.

Proof. The transformation proceeds as follows:

1. ADDING RANDOM-INPUT GATES: Add (at most) $d \cdot w + |\mathbf{z}| + 2$ random-input gates at depth $l = 0$ for $\Delta, \beta, \alpha_{k,l} \in \mathbb{F}$ and $\mathbf{c}$.
2. Add $|\mathbf{x}|$ multiplication gates of depth one with inputs Δ and x_i.
   This does not increase the width as we assume $|\mathbf{x}| \leq w$.
   All the gates at depth $l \in \{1, \ldots, d\}$ in the original circuit C are now placed at depth $l+1$
3. Now for each depth $l \in \{2, \ldots, d+1\}$ do the following:
   - FOR EACH LINEAR GATE IN C: Let the gate take as input $y_1, \ldots, y_s$, add an additional linear gate which takes as input $\Delta \cdot y_1, \ldots, \Delta \cdot y_s$.
   - FOR THE k-TH MULTIPLICATION GATE AT DEPTH l: Let $x_{k,l}$ and $y_{k,l}$ be the input of the gate and $z_{k,l}$ its output, add a new multiplication gate at the same depth, which takes as input $X_{k,l}$ and $\Delta \cdot y_{k,l}$ and produces $r_{k,l} = \Delta \cdot z_{k,l}$.
   This step doubles the width at this depth from w_l to $2 \cdot w_l$.

5. Now for each depth $l \in \{3, \ldots, d + 2\}$ do the following:
   - Add w_{l-1} multiplication gates at depth l with input $\alpha_{k,l-1}$ and $z_{k,l-1}$, producing $s_{k,l-1} = \alpha_{k,l-1} \cdot z_{k,l-1}$.

     This increases the width at this depth to $w_{l-1} + 2 \cdot w_l$.
   - Add w_{l-1} multiplication gates at depth l with input $\alpha_{k,l-1}$ and $r_{k,l-1} = \Delta \cdot z_{k,l-1}$ producing $t_{k,l-1}$.

     This increases the width at this depth to $2 \cdot w_{l-1} + 2 \cdot w_l$.
   - Assuming $u_1 = v_1 = 0$, one adds an additional two linear gates at depth l in order to compute the sum, u_l, of u_{l-1} and the set $\{s_{k,l-1}\}_{k \in [w_{l-1}]} = \{\alpha_{k,l-1} \cdot z_{k,l-1}\}_{k \in [w_{l-1}]}$, and the sum, v_l, of v_{l-1} and the set $\{t_{k,l-1}\}_{k \in [w_{l-1}]} = \{\alpha_{k,l-1} \cdot (\Delta \cdot z_{k,l-1})\}_{k \in [w_{l-1}]}$.
4. At depth $d + 3$, set $u = u_{d+1}$ and $v = v_{d+1}$. Add a single additional multiplication gate which computes the multiplication of $\Delta \cdot u$, followed by a single linear gate which computes $T = \Delta \cdot u_{d+1} - v_{d+1}$.
5. At depth $d + 3$, add $|\mathbf{y}| + 1$ additional multiplication gates, and $|\mathbf{y}|$ additional linear gates to compute $\mathbf{z}$ and t.
6. At the end the circuit the output gates for $\mathbf{y}$, are replaced by those for $\mathbf{z}$ and an additional output gate for t is added, all of which are at depth $d + 3$.

Since $w_i \leq w$ the total (multiplicative) width increases to at most $4 \cdot w$.

Given this lemma, we can now prove that if we evaluate a robust circuit $\tilde{C}$ corresponding to a (randomized) circuit C using the protocol Π_{PMPC} described in the previous section, we obtain an actively secure protocol with abort implementing the ideal functionality $\mathcal{F}_{\mathsf{DelayMPC}}$ for C, where the communication can be adversarially delayed by the adversary and modelled using relays.

This discussion is formalised in the following theorem.

Theorem 6.1. *Let $\tilde{C}$ be the robust circuit over $\mathbb{F}$ corresponding to C (according to Definition 6.1). The protocol $\Pi_{\mathsf{DelayedMPC}}$ in Figure 24, computing $\tilde{C}$, securely implements $\mathcal{F}_{\mathsf{DelayedMPC}}$ with abort against a δ-delaying active adversary corrupting up to $t < n/2$ parties and $r - 1$ relays, except with probability $3/|\mathbb{F}|$ in the $\{\mathcal{F}_{\mathsf{Rand}}, \mathcal{F}_{\mathsf{SecureRobustRelay}}, \mathcal{F}_{\mathsf{Mult}}\}$-hybrid model.*

Our main delayed MPC functionality $\mathcal{F}_{\mathsf{DelayedMPC}}(\mathcal{P}_1, \ldots, \mathcal{P}_n)$ is given in Figure 23, with the protocol to implement this $\Pi_{\mathsf{DelayedMPC}}$ being given in Figure 24.

Proof. Let $\mathcal{A}$ be the real adversary which controls the set of corrupt parties and relays, the simulator $\mathcal{S}$ works as follows.

- $\mathcal{S}$ emulates the ideal functionality $\mathcal{F}_{\mathsf{SecureRobustRelay}}$ and simulates the communication among parties and relays according to the delay messages delivered by $\mathcal{A}$. It keeps track of all the tags and variables associated to the relays.
- $\mathcal{S}$ invokes $\mathcal{S}_{\mathsf{PMPC}}$, the simulator for Π_{PMPC}, as described in the previous section, and obtains a simulated view up to the output step.
- $\mathcal{S}$ simulates the honest parties sending their shares in the Open sub-protocol. This is done by using the view obtained by $\mathcal{S}_{\mathsf{PMPC}}$; and it receives the shares from $\mathcal{A}$ sent to the honest parties. If any honest party would abort, $\mathcal{S}$ sends abort to the ideal functionality, otherwise it checks if $A = \mathbf{0}$. If this is not the case, it sends abort to $\mathcal{F}_{\mathsf{DelayedMPC}}$.
- If no abort occurred, $\mathcal{S}$ sends the input of the corrupted parties to the ideal functionality and simulates the output step as shown in the previous section.

Indistinguishability. The only major difference between the real and the ideal execution is that the former checks whether the output $T = \beta \cdot Z$ of the robust circuit is zero and aborts if this is not the case, while in the latter the simulator aborts the computation if the extracted additive error A is not zero. The following lemma shows that for the first set of parties concluding the protocol, the difference between these two executions is statistically small.

Lemma 6.2. *If $\mathcal{A}$ inputs an internal additive attack A to the evaluation of C, the value $\langle \beta \cdot Z \rangle_t$ open by the first $\hat{\mathcal{P}} \subseteq \mathcal{P}$ parties equals zero with probability less than $3/|\mathbb{F}|$.*

Proof. The proof of the lemma follows the same arguments in [CGH+18] and [CGG+21], to which we remand for a more formal discussion. At a high level, the upper bound on the probability of $3/|\mathbb{F}|$ can be seen as follows. For an adversary to avoid detection, the value of T should be zero on computation of the robust circuit. This can happen either because $\beta = 0$, which happens with probability $1/|\mathbb{F}|$, or because $Z = 0$, since, due to the random linear combinations, the value of u can be assumed to be uniformly random and outside the control of the adversary. Thus to obtain $Z = 0$ either the adversary needs to be lucky in fixing the unknown Δ value to be v/u, in the case of $u \neq 0$, an event which happens with probability $1/|\mathbb{F}|$, or the adversary needs to be lucky in obtaining a random linear combination which gives $u = 0$, and hence $v = 0$. Again an event which happens with probability $1/|\mathbb{F}|$. Since the adversary can only win (i.e. avoid detection after introducing an additive error) if one of these three conditions is satisfied, we have that the probability the adversary cheats and avoids detection is at most $3/|\mathbb{F}|$. □

When more shares are delivered, we need to distinguish different cases. If $A = 0$ and the check correctly passed, then the shares used by the previous set of parties $\hat{\mathcal{P}}$ are also used by the slower parties $\bar{\mathcal{P}}$. So either the new shares are consistent with the shares used by $\hat{\mathcal{P}}$, in which case also parties in $\bar{\mathcal{P}}$ obtain the correct output, or the new shares are inconsistent with the previously stored ones and parties in $\bar{\mathcal{P}}$ output abort. In the simulation, $\mathcal{S}$ checks the consistency of the new shares and detect any malicious behaviour, in which case it will forward abort to the ideal functionality. The two executions are indistinguishable. If $A \neq 0$, $\mathcal{S}$ always aborts. We proved that the same happens in the real protocol with the fastest set of parties $\hat{\mathcal{P}}$ concluding the computation, except with some negligible probability. When more parties reach the output step, the shares stored in the relays, using the `sendToAll` command in $\mathcal{F}_{\mathsf{SecureRobustRelay}}$, are the same shares already used by $\hat{\mathcal{P}}$ and $\mathcal{A}$ cannot change them anymore. Therefore, even if the zero-check is performed using the same randomness already used by $\mathcal{P}$, this does not help $\mathcal{A}$ to pass the test. □

6.1 Efficiency and Optimizations

We can now estimate the efficiency of our scheme in term of communication and round complexity, and sketch different possible instantiations of the general construction given in the previous section. As done in previous works, to estimate the communication costs of our protocol we consider the number of field elements required for the evaluation of multiplication gates.

Naïve implementation of the protocol $\Pi_{\mathsf{DelayedMPC}}$ with Π_{Mult1}. The first, simpler instantiation is given by implementing $\mathcal{F}_{\mathsf{Mult}}$ with Π_{Mult1} to evaluate both randomized and non-randomized multiplication gates. We recall that for each multiplication, Π_{Mult1} needs to call the functionality $\mathcal{F}_{\mathsf{DRand}}$ plus it requires the communication of r field elements per party. Setting $r = 2$, the passively secure version of our protocol, Π_{PMPC}, achieves the same amortized communication complexity of

DN, i.e. 6 field elements per party, or 4 assuming PRGs. We recall that these figures have been improved in [GLO$^+$21] to 4 and 2, respectively, using t-wise independence. We leave to future work the possibility to further improve the efficiency of our scheme with similar techniques.

In the active security setting with abort, considering that the evaluation of $\tilde{C}$ requires the computation of $\langle\alpha_{k,l}\rangle$, $\langle z_{k,l}\rangle$, $\langle\Delta \cdot z_{k,l}\rangle$ for each multiplication gate, and values $\langle u_l\rangle$, $\langle v_l\rangle$ for each layer of the circuit, our simpler instantiation has an amortized cost of $13 + 12\frac{d}{m}$ field elements per party, where m is the total number of multiplication gates and d is the depth of the circuit. Using PRGs, this goes down to $9 + 8\frac{d}{m}$ fields element per party. Note that the communication can be further reduced if instead of calling the interactive functionalities $\mathcal{F}_{\mathsf{Rand}}$ and $\mathcal{F}_{\mathsf{DRand}}$, the protocol calls the non-interactive functionality $\mathcal{F}_{\mathsf{Rand^{NI}}}$.

We already mentioned that in order to avoid maintaining large states, we incrementally compute the checking equation during the computation. This means that we need to compute the values $\langle u_l\rangle$ and $\langle v_l\rangle$ at each level. If we compute the random linear combinations $\langle u\rangle$ and $\langle v\rangle$ at the end of the circuit evaluation, then the communication of our protocol is essentially the same as in [CGH$^+$18, NV18], i.e. 13 (or 9 with PRGs) field elements per party.

Use the trick of Beaver-friendly shares from [GLO$^+$21] to reduce the round complexity. Following [GLO$^+$21], we can use the special form of the output of Π_{Mult1} either to evaluate all the multiplications in 2 consecutive levels in parallel, so to reduce the round complexity by a factor of 2, or evaluate both randomized and non-randomized multiplication gates in parallel, as in the naïve instantiation described above, but with slightly better communication.

First, we briefly recall the technique from [GLO$^+$21]. If the inputs $\langle x\rangle_t$ and $\langle y\rangle_t$ of a multiplication gate are expressed in a so called *Beaver friendly form* (BF-form), i.e. as $\langle x\rangle_t = v - \langle r\rangle_t$ and $\langle y\rangle_t = u - \langle s\rangle_t$, where v, u are publicly known and random sharings $\langle r\rangle_t$, $\langle s\rangle_t$ and $\langle r \cdot s\rangle_t$ are known before the gate being evaluated, then the output of the multiplication can be computed without interaction as

$$\langle x \cdot y\rangle_t = (v - \langle r\rangle_t) \cdot (u - \langle s\rangle_t) = u \cdot v - u \cdot \langle s\rangle_t - v \cdot \langle r\rangle_t + \langle r \cdot s\rangle_t.$$

Therefore, to evaluate two consecutive levels in the same round, first we transform the input sharings $\langle x\rangle_t$ of all the multiplications in the second level in a BF-form, i.e., $\langle x\rangle = u - \langle r\rangle$. Concretely, if this sharing is an output of a multiplication gate in the first level, we know it is already in BF-form (see Π_{Mult1} Figure 18); if, otherwise, it is either an input sharing of the circuit or an output sharing of an additive gate, it is enough to set $u = 0$ and $\langle r\rangle = -\langle x\rangle$.

Since the evaluation of the multiplications in the second level does not require any interaction, the security and the formal proof of this approach follows directly from the security of Π_{Mult1}, assuming an ideal functionality $\mathcal{F}_{\mathsf{RTriple}}$ that outputs a random triples $\langle r\rangle_t$, $\langle s\rangle_t$ and $\langle r \cdot s\rangle_t$.

Reducing the round complexity in our setting is extremely important, as it allows to mitigate the effects of delays introduced by $\mathcal{A}$. However, this approach has the disadvantage of requiring more randomness, in particular it needs pre-computed random triples of the form $\langle r\rangle_t$, $\langle s\rangle_t$ and $\langle r \cdot s\rangle_t$ in case the second-level multiplications we compute in this way are the multiplications in the next layer of the circuit. Overall, the communication complexity is roughly the same as in the previous approach, but the number of rounds is reduced by an half.

If we use this approach to compute in parallel $\langle z\rangle = \langle x \cdot y\rangle$ and $\langle z \cdot \Delta\rangle$, the technique sketched above only requires a pre-computed value $\langle r \cdot \Delta\rangle_t$, for a random $\langle r\rangle$. Given this value, it is indeed possible to compute $\langle z \cdot \Delta\rangle_t$ as follows:

$$\langle z \cdot \Delta\rangle_t = v \cdot \langle\Delta\rangle_t - \langle r \cdot \Delta\rangle_t = (z + r) \cdot \langle\Delta\rangle_t - \langle r \cdot \Delta\rangle_t = \langle z \cdot \Delta\rangle_t.$$

We leave to future work to investigate how to efficiently implement the functionality $\mathcal{F}_{\mathsf{RTriple}}$, especially in the case of correlated random triples, i.e. in the case the triples have the special form $\langle r^i \rangle, \langle \Delta \rangle, \langle r^i \cdot \Delta \rangle$, for $i \in [m]$.

Changing the definition of robust circuit. In [GS20, GSZ20, BBC$^+$19], it was described how to reduce the communication complexity of the protocol of [CGH$^+$18] by using a new technique for checking the correctness of multiplication gates with communication complexity independent of the number of multiplication gates in the circuit. This allowed to match the communication complexity of the passively secure DN protocol. In order to apply the same technique, we need to change the definition of robust circuit given in Definition 6.1, in such a way that, after the original output $\mathbf{y} = C(\mathbf{x})$, the robust circuit consists of the circuit $\hat{C}$ evaluating the verification procedure of [GS20]. This would require an extra $\log_2(m)$ rounds of communication, but overall an amortized communication of 6 (4 with PRGs) field elements per party.

The security of this instantiation can be argued similarly to what we did in Section 6. Note however that, even if this approach permits to reduce the communication compared to what we did in previous sections, it does not immediately allow to keep small states, because parties need to keep the sharings of $\langle x_{k,l} \rangle$, $\langle y_{k,l} \rangle$ and $\langle z_{k,l} \rangle$ in order to perform the evaluation of the verification circuit $\hat{C}$ after the evaluation of C.

7 Experiments

We present two forms of experiments. The first evaluates the networking performance of the relays. We investigate the difference multiple relays have on performance, as well as the communication slowdown induced by the relays' existence. The second set of experiments evaluates the performance of the MPC protocol built on top of the relays. Here we measure performance by the number of multiplications per second that can be performed. In Table 1, Table 2 and Table 3, we give the precise numerical values for the main results presented in this section.

Experiment (p2p)	Number of messages						
	2^7	2^9	2^{11}	2^{13}	2^{15}	2^{17}	2^{19}
DP - 16B	0.07	0.25	1.31	3.11	5.88	15.16	58.05
Ek - 16B 2 relays	0.03	0.08	0.29	1.08	4.25	17.21	68.24
Ek - 16B 3 relays	0.03	0.09	0.32	1.24	4.89	19.44	78.24
Ek - 16kB 3 relays	0.06	0.17	0.61	2.34	9.43	37.58	149.94
Ek - 16B 4 relays	0.03	0.10	0.35	1.35	5.36	21.51	85.82

Table 1. Runtimes in seconds for experiment *DP* with 16-byte messages, p2p experiment *Rk* with $k = 100$ and 16-byte messages with 2,3 and 4 relays, as well as p2p experiment *Rk* with $k = 100$ and 16kB messages with 3 relays.

48

Experiment (BC)	Number of messages						
	2^7	2^9	2^{11}	2^{13}	2^{15}	2^{17}	2^{19}
Ek - 16B 2 relays	0.04	0.15	0.57	2.37	9.44	37.92	153.41
Ek - 16B 3 relays	0.05	0.17	0.65	2.60	10.40	41.47	166.40
Ek - 16kB 3 relays	0.23	0.53	2.03	8.01	31.77	126.20	503.65
Ek - 16B 4 relays	0.05	0.19	0.73	2.81	11.20	44.84	178.77

Table 2. Runtimes in seconds for BC experiment Rk with $k = 100$ and 16-byte messages with 2,3 and 4 relays, as well as BC experiment Rk with $k = 100$ and 16kB messages with 3 relays.

Experiment (Multiplication)	Number of multiplications per batch												
	2^0	2^1	2^6	2^7	2^8	2^9	2^{10}	2^{11}	2^{12}	2^{13}	2^{14}	2^{15}	2^{16}
3 Parties, 3 Relays	2917	4644	96235	136818	45104	59002	71808	92594	108449	120366	178802	249692	297783
6 Parties, 2 Relays (fast parties)	3521	5772	89550	126885	160055	47736	57125	86412	127602	162238	194710	244092	268381
6 Parties, 2 Relays (slow parties)	1165	1835	32223	43356	57467	36934	45912	59992	71384	77884	83298	83955	85051

Table 3. Number of multiplications per second for increasing size of multiplication batches and erasing messages in batches of 100, for 3 parties and 3 relays and for 6 parties and 2 relays.

7.1 Networking Experiments

We now provide a more detailed explanation of our implementation of the protocol $\Pi_{\mathsf{SecureRobustRelay}}$, as well as the results of our experimental evaluations. To maximise the degree of concurrency, asynchronicity and parallelism, we used the `tokio` framework. For high-performance and manageable communications between the parties and relays, we used the `tonic` framework to employ `gRPC`, which is a remote-procedure-call framework that uses Protocol Buffers (known as `protobuf`) for data serialisation. We used the RustCrypto crate for standard cryptographic primitives, such as the AES256-GCM-SIV authenticated encryption scheme and the CMAC-AES256 message authentication code (MAC) algorithm. We ensure that our deployment takes advantage of AES hardware acceleration, namely AES-NI.

In the following experiments, the relays are run on identical machines with an Intel i9-9900 CPU and a 128GB RAM. The parties communicating through the relays run machines with an Intel i7-770 CPU and a 32GB RAM. The ping time between all of the machines is 1.003 ms.

We examine the case of both the `send` and `request` commands and the `sendToAll` and `requestFromAll` commands; we refer to the former as the "p2p experiments" whereas the latter we refer to as the "broadcast experiments". The `send` and `request` commands have potentially more overhead, since the sending party needs to encrypt and the receiving party needs to decrypt.

Data Structures. The data structures used to implement the different message stores in the relays are different depending on whether we are considering p2p or broadcast communication.

For the p2p messages, the relays use multi-value maps to store the messages exchanged between party $\mathcal{P}_i$ and party $\mathcal{P}_j$. At runtime, there are $n \cdot (n-1)$ entry lists in the map for all uni-directional channels. For each entry $(\mathcal{P}_i, \mathcal{P}_j)$, the map stores a list of messages sent by $\mathcal{P}_i$ to $\mathcal{P}_j$. Each message is

composed of a round number, an encrypted message payload and a MAC for authentication between parties and relays. Since the relays should be capable of handling many requests concurrently in a multi-threaded setting, the multi-value map should be resilient against the problems caused by concurrent accesses. We used the `evmap` crate for this purpose. The crate `evmap` offers lock-free, eventually consistent, and concurrent read handles. Our protocol requires many reads since parties continuously invoke the `request` command until a new message is retrieved. However, the write handles in a multi-writer setting require a mutex for thread safety. As a result, writing operations will be considerably more expensive than reads.

For the broadcast messages, multi-value maps are again used through the `evmap` crate, and the messages contain the same information. However, since each message is now meant to be received by all the other parties, messages are no longer stored according to the corresponding $(\mathcal{P}_i, \mathcal{P}_j)$ pair, but to their counter (or round number). Thus, for each counter, the map stores a list of messages sent by the different parties during that same round. Note that every time we erase broadcast messages, we always erase all the messages associated with the same counter. Hence, erasing messages will correspond to deleting entries from the map.

Experiments. We identified four key experimental setups that we wanted to investigate, which we label as *DP*, *E0*, *E1* and *Ek*.

- *DP*: This experiment used no relays between the parties. Parties communicate directly, and all communications are protected by TLS.
- *E0*: This experiment used relays to establish indirect communications between the parties. Parties request messages without removing them from the relays.
- *E1*: This experiment had the same setting as *E0*, but parties immediately remove each message after retrieval, i.e. the removal batch size is one.
- *Ek*: This experiment is similar to *E1*, but parties issue `erase` commands after retrieval of $k > 1$ messages, i.e., the removal batch size is k.

For the p2p experiments we considered the case of only two communicating parties: a sender and a receiver. The sender sends a predetermined number of messages one by one, either directly to the receiver (in *DP*) or to all the relays (in *E0*, *E1* and *Ek*). In *DP*, the receiver passively waits until all the expected messages were received and the runtime is measured on the sender's side as the time taken to send all the messages. In the experiments with relays (*E0*, *E1* and *Ek*), the receiver continuously queries the relays for each message. Once message i is successfully retrieved, queries for message $i + 1$ are sent, and so on, until all of the expected messages were sent and received. At the end of the experiment, an acknowledgement of receipt is sent to the sender through the relays (the sender starts querying the relays for the receipt once all messages have been sent). The runtime is measured on the sender's side as the time between starting sending the messages and receiving the acknowledgement of receipt.

For the broadcast experiments we considered the case of three communicating parties, where each party simultaneously acts as a sender and receiver. Parties alternate between sending and receiving messages: after sending message i, they continuously query the relays until they receive message i from the other parties. They then send and request message $i + 1$, and so on, until all the expected messages were received. When this happens, each party broadcasts an acknowledgement of receipt and then waits for the acknowledgement of receipt from the other parties. The runtime is measured by each party as the time between starting sending the messages and receiving the acknowledgement of receipt from every other party.

Relays vs direct comm. We seek first to understand the overhead caused by relays compared to a deployment topology wherein parties communicate directly. The graphs in Figure 25 and Figure 26 show the runtimes of the experiments above for an increasing number of 16-byte messages, with three relays and *Ek* with the removal batch size $k = 100$ for both the p2p and broadcast messages. Additionally, runtimes for *Ek* with batch size $k = 100$ and 16kB messages are also presented. First, it is clear that the results for all experiments are very similar when sending up to 2^{15} messages. For 2^{17} and 2^{19} messages, *E1* has the slowest runtimes, which is due to the receiver invoking the `erase` command after each received message and before requesting the next message. However, erasing messages is necessary to guarantee the relays do not run out of memory. Furthermore, for p2p communications, even though *E0* is faster than *E1*, it is still much slower than *Ek*. This happens because when answering message requests, the relays need to iterate through all of their stored messages until the desired one is found. As the number of stored messages grows, this will substantially affect the performance.

Fig. 25. Runtimes (with linear scale on the *y*-axis) for experiments *DP*, *E0*, *E1* and *Ek* with $k = 100$ and 16-byte messages, and for *Ek* with $k = 100$ and 16kB messages, showing: (left) p2p messages; (right) broadcast messages.

Erasing batches of messages as in the *Ek* experiment prevents running into a memory limit while keeping the runtimes very close to the ones for *DP*. Indeed, batch erasure avoids accumulating large numbers of messages in the relays without the high cost of repeatedly invoking the `erase` command. Sending larger messages will naturally increase the communication time, but the overhead is insignificant.

Note that the broadcast experiments are always slower than the p2p experiments, which results from the following difference: in the p2p experiments, there is one sender and one receiver; in the broadcast experiments, there are three parties, and all of them send and receive (additionally, each of them needs to receive the messages from the other parties before sending the next message).

One can also see that E0 and E1 behave differently in the broadcast experiments compared to the p2p experiments. This is due to the difference in the way messages are organised in the relays, as was mentioned in Sub-section 7.1. In the p2p experiments, the relays first find the list of messages between two parties and then iterate through the list to obtain the message with the requested counter. In the broadcast experiments, the relays find the list of messages for the requested counter (which will contain at most a message by each party), and return all of them. Therefore, requesting
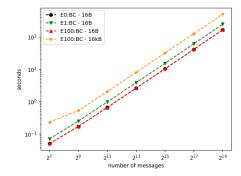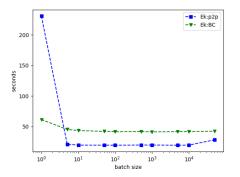
Fig. 26. Runtimes (with logarithmic scale on the y-axis) for experiments *DP*, *E0*, *E1* and *Ek* with $k = 100$ and 16-byte messages, and for *Ek* with $k = 100$ and 16kB messages, showing: (left) p2p messages; (right) broadcast messages.
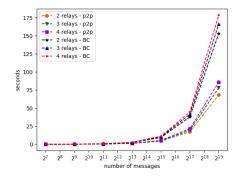

Fig. 27. Communication runtimes, showing for both the p2p and broadcast (BC) communications: (left) experiment *Ek* with 2^{17} 16-byte messages and increasing batch size k; (right) experiment *Ek* with $k = 100$ and 16-byte messages with 2, 3 and 4 relays.

broadcast messages does not become slower even when we never erase them. In both cases erasing in batches is to be preferred anyway, as it prevents an explosion in the memory requirements.

Removal batch size. We also wished to determine the influence of the size k of the removal batch in the *Ek* experiment. To do so, we perform this experiment with a fixed number of sent messages (2^{17}) and an increasing batch size k for both the p2p and the broadcast settings. The results are presented in the left graph of Figure 27.

As seen in the previous experiment, erasing each message after retrieval is considerably slower than batch erasure. However, the batch size $k > 1$ has little influence on the total runtime up until $k = 50000$. For this batch size, because the relays will store up to $k-1$ messages before deleting, the time required for the relays to iterate through all of the stored messages and retrieve the correct one becomes noticeable in the total communication time. This is, however, only a small increase when compared to lower batch sizes, and still much faster than erasing every message after retrieval.

In the p2p experiments with $k = 1$, we observe a considerable slowdown because of numerous concurrent write queries (requests and erasures) sent to the relays. The relays employ a pessimistic concurrency control by locking the data items to mitigate conflicting updates. Therefore, such controls offer data safety and integrity at the cost of latency due to resource contention. In our

implementation, as explained earlier, the p2p and the broadcast experiments have slightly different data structures impacting the locking mechanisms. Another critical factor for concurrent systems is the resource access pattern. The p2p experiment ($k = 1$) and the broadcast experiments are inherently different. In the latter, the parties send messages to the relays simultaneously and wait until all messages for that round are available within the relays. Afterwards, they issue delete requests concurrently. They do not need to wait for all deletes; the faster parties can continue to the next broadcast round. However, in the former, a sending and a receiving party exchange messages simultaneously via relays (causing contention within the relays). The write queries fight to acquire locks, which is one of the reasons that the larger `erase` batches reduce the number of concurrent write queries, resulting in faster overall runtimes. These experiments are quite different in terms of deployment topology and the order of executions.

Note that when choosing the optimal batch size we must consider our specific setting, e.g., the number of players and the memory of the relays.

Number of relays. Finally, we analysed the influence of the number of relays on the network performance. In the right graph of Figure 27 we present results for experiment *Ek* with $k = 100$ when using 2,3 and 4 relays (as opposed to the previous experiments which used a fixed number of three relays). On one hand, more relays mean the sender will send each message more times (one for each relay). On the other hand the receiver accepts a message as soon as it decrypts correctly (in the p2p experiments) and hence does not need to wait until all relays reply to the message request. However, requesting and erasing messages from more relays will also increase the communication. We therefore obtain slower communication times when using more relays, even though the overhead of adding each new relay is relatively small.

Recall also that because we only require one honest relay to ensure the overall communication is secure, having more relays will allow a higher corruption percentage within the relays themselves. Thus, the exact number of relays used should depend on the desired performance security trade-off.

Concluding remarks. We have shown that communicating through relay nodes introduces only a small overhead when compared to direct communication between parties, especially when erasing messages in batches. Our implementation can be further optimized by, e.g., using a lock-free data structure for both reading and writing. For a real-world deployment of this network topology, we do recommend doing application-specific benchmarking to find a (sub)optimal batch size and relay number for the desired settings since there are unlimited possible deployment plans and hardware profiles.

7.2 Multiplication

We now turn to benchmarking our MPC protocol running on top of the relay enabled network. We benchmark the protocol by examining the number of multiplications which can be performed per second by the MPC protocol. Recall that the main communication required in the multiplication protocol is that of each party sending a single broadcast message via the `sendToAll` command, and then each party executing the requisite `requestFromAll` commands. The computation cost on top of this is then the execution of the relevant PRSS and some associated simple arithmetic operations.

We first targeted an MPC protocol with three parties, with at most one corruption; thus the non-interactive version of the PRSS could be utilized. We examined an implementation based on a finite field of 128 bits in size, which fits into the 16 bytes of our earlier experiments. We performed

experiments similar to the broadcast experiments mentioned above. In particular, we examined the effect on the throughput (measured in multiplications per second) of varying the number of multiplications which are batched in each execution of the protocol. One can think of the batch size as the number of multiplications at a given depth in the evaluated circuit, since all such multiplications can be batched together.

Then, we examined an MPC protocol with six parties, again with at most one corruption. The setup is similar to the experiment with three parties, except now instead of three relays we have only two, one of them run on one of the faster machines (Intel i9-9900 CPU and 128GB RAM) and the other on one of the slower machines (Intel i7-770 CPU and 32GB RAM) - note that in all previous experiments, the relays were always run on the faster machines. Regarding the parties, instead of running all of them on the slower machines as before, three of them are now run on the faster machines. This allows us to simulate a situation where some of the parties participating the protocol are faster and progress in the computation ahead of the others. Indeed, since we assume at most one corruption, it is possible to process multiplications as soon as messages from any three of the parties are available.

The experiments are presented in Figure 28, where we averaged the execution time over a total of 2^{11} multiplication rounds. For the six party setting, we averaged the run time over the three slow parties, and the three fast parties, separately. We see that for the three party multiplication, our implementation can cope with up to around 300 thousand multiplications per second when the number of parallel multiplications exceeds $2^{16} = 65536$. When considering six parties, the faster parties achieve close to 270 thousand multiplications per second for the same batch size. The effect of having both fast and slow parties is observable in the throughput of multiplications per second shown in Figure 28, for the average of the three fast and the three slow parties.
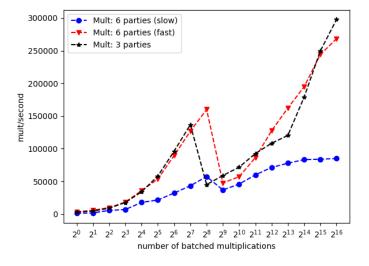

Fig. 28. Number of multiplications per second for increasing size of multiplication batches and erasing messages every 100 multiplication rounds, for 3 parties and 3 relays and for 6 parties and 2 relays.

For sequential multiplication (i.e., batch size 2^0) the difference is less obvious (1165 multiplications per second for slow parties vs 3483 for fast parties) since the slow parties can catch up after every single multiplication. As the batch size increases, the computation and communication to be performed in each multiplication round also increase, resulting in the fast parties computing on average 180 thousand more multiplications per second than the slow ones. This illustrates how the use of relays allows faster parties to progress in the computation, while slower ones are still able to retrieve all the necessary messages at their own pace. We note that one of the slow parties was on average 4 times slower than the other two, which impacts the presented average multiplication rate. There is a small kink in the graph at a batch size of around 2^7 (for the three party case) or around 2^8 (for the six party case) which we could not explain, we think this is an effect of the underlying Rust switching between two different algorithms for data access.

Our code can be significantly improved, as this is just a first implementation, with the multiplications being performed in a single threaded manner. We expect throughputs of around one million multiplications per second could be easily achieved with a fully optimized implementation.

Acknowledgements

References

AHKP22. Anasuya Acharya, Carmit Hazay, Vladimir Kolesnikov, and Manoj Prabhakaran. SCALES - MPC with small clients and larger ephemeral servers. In Eike Kiltz and Vinod Vaikuntanathan, editors, *TCC 2022: 20th Theory of Cryptography Conference, Part II*, volume 13748 of *Lecture Notes in Computer Science*, pages 502–531, Chicago, IL, USA, November 7–10, 2022. Springer, Heidelberg, Germany.

ANOS22. Bar Alon, Moni Naor, Eran Omri, and Uri Stemmer. MPC for tech giants (GMPC): Enabling gulliver and the lilliputians to cooperate amicably. Cryptology ePrint Archive, Report 2022/902, 2022. https://eprint.iacr.org/2022/902.

BBC+19. Dan Boneh, Elette Boyle, Henry Corrigan-Gibbs, Niv Gilboa, and Yuval Ishai. Zero-knowledge proofs on secret-shared data via fully linear PCPs. In Alexandra Boldyreva and Daniele Micciancio, editors, *Advances in Cryptology – CRYPTO 2019, Part III*, volume 11694 of *Lecture Notes in Computer Science*, pages 67–97, Santa Barbara, CA, USA, August 18–22, 2019. Springer, Heidelberg, Germany.

BBG+21. Fabrice Benhamouda, Elette Boyle, Niv Gilboa, Shai Halevi, Yuval Ishai, and Ariel Nof. Generalized pseudorandom secret sharing and efficient straggler-resilient secure computation. In Kobbi Nissim and Brent Waters, editors, *TCC 2021: 19th Theory of Cryptography Conference, Part II*, volume 13043 of *Lecture Notes in Computer Science*, pages 129–161, Raleigh, NC, USA, November 8–11, 2021. Springer, Heidelberg, Germany.

BJMS20. Saikrishna Badrinarayanan, Aayush Jain, Nathan Manohar, and Amit Sahai. Secure MPC: Laziness leads to GOD. In Shiho Moriai and Huaxiong Wang, editors, *Advances in Cryptology – ASIACRYPT 2020, Part III*, volume 12493 of *Lecture Notes in Computer Science*, pages 120–150, Daejeon, South Korea, December 7–11, 2020. Springer, Heidelberg, Germany.

CDI05. Ronald Cramer, Ivan Damgård, and Yuval Ishai. Share conversion, pseudorandom secret-sharing and applications to secure computation. In Joe Kilian, editor, *TCC 2005: 2nd Theory of Cryptography Conference*, volume 3378 of *Lecture Notes in Computer Science*, pages 342–362, Cambridge, MA, USA, February 10–12, 2005. Springer, Heidelberg, Germany.

CGG+21.   Arka Rai Choudhuri, Aarushi Goel, Matthew Green, Abhishek Jain, and Gabriel Kaptchuk. Fluid MPC: Secure multiparty computation with dynamic participants. In Tal Malkin and Chris Peikert, editors, *Advances in Cryptology – CRYPTO 2021, Part II*, volume 12826 of *Lecture Notes in Computer Science*, pages 94–123, Virtual Event, August 16–20, 2021. Springer, Heidelberg, Germany.

CGH+18.   Koji Chida, Daniel Genkin, Koki Hamada, Dai Ikarashi, Ryo Kikuchi, Yehuda Lindell, and Ariel Nof. Fast large-scale honest-majority MPC for malicious adversaries. In Hovav Shacham and Alexandra Boldyreva, editors, *Advances in Cryptology – CRYPTO 2018, Part III*, volume 10993 of *Lecture Notes in Computer Science*, pages 34–64, Santa Barbara, CA, USA, August 19–23, 2018. Springer, Heidelberg, Germany.

CK01.   Ran Canetti and Hugo Krawczyk. Analysis of key-exchange protocols and their use for building secure channels. In Birgit Pfitzmann, editor, *Advances in Cryptology – EUROCRYPT 2001*, volume 2045 of *Lecture Notes in Computer Science*, pages 453–474, Innsbruck, Austria, May 6–10, 2001. Springer, Heidelberg, Germany.

DEP21.   Ivan Damgård, Daniel Escudero, and Antigoni Polychroniadou. Phoenix: Secure computation in an unstable network with dropouts and comebacks. Cryptology ePrint Archive, Report 2021/1376, 2021. https://eprint.iacr.org/2021/1376.

DN07.   Ivan Damgård and Jesper Buus Nielsen. Scalable and unconditionally secure multiparty computation. In Alfred Menezes, editor, *Advances in Cryptology – CRYPTO 2007*, volume 4622 of *Lecture Notes in Computer Science*, pages 572–590, Santa Barbara, CA, USA, August 19–23, 2007. Springer, Heidelberg, Germany.

DvW92.   Whitfield Diffie, Paul C. van Oorschot, and Michael J. Wiener. Authentication and authenticated key exchanges. *Designs, Codes and Cryptography*, 2(2):107–125, June 1992.

DY81.   Danny Dolev and Andrew Chi-Chih Yao. On the security of public key protocols (extended abstract). In *22nd Annual Symposium on Foundations of Computer Science*, pages 350–357, Nashville, TN, USA, October 28–30, 1981. IEEE Computer Society Press.

FHM98.   Matthias Fitzi, Martin Hirt, and Ueli M. Maurer. Trading correctness for privacy in unconditional multiparty computation (extended abstract). In Hugo Krawczyk, editor, *Advances in Cryptology – CRYPTO'98*, volume 1462 of *Lecture Notes in Computer Science*, pages 121–136, Santa Barbara, CA, USA, August 23–27, 1998. Springer, Heidelberg, Germany.

FL19.   Jun Furukawa and Yehuda Lindell. Two-thirds honest-majority MPC for malicious adversaries at almost the cost of semi-honest. In Lorenzo Cavallaro, Johannes Kinder, XiaoFeng Wang, and Jonathan Katz, editors, *ACM CCS 2019: 26th Conference on Computer and Communications Security*, pages 1557–1571, London, UK, November 11–15, 2019. ACM Press.

GHK+21.   Craig Gentry, Shai Halevi, Hugo Krawczyk, Bernardo Magri, Jesper Buus Nielsen, Tal Rabin, and Sophia Yakoubov. YOSO: You only speak once - secure MPC with stateless ephemeral roles. In Tal Malkin and Chris Peikert, editors, *Advances in Cryptology – CRYPTO 2021, Part II*, volume 12826 of *Lecture Notes in Computer Science*, pages 64–93, Virtual Event, August 16–20, 2021. Springer, Heidelberg, Germany.

GIP+14.   Daniel Genkin, Yuval Ishai, Manoj Prabhakaran, Amit Sahai, and Eran Tromer. Circuits resilient to additive attacks with applications to secure computation. In David B. Shmoys, editor, *46th Annual ACM Symposium on Theory of Computing*, pages 495–504, New York, NY, USA, May 31 – June 3, 2014. ACM Press.

GLO+21.   Vipul Goyal, Hanjun Li, Rafail Ostrovsky, Antigoni Polychroniadou, and Yifan Song. ATLAS: Efficient and scalable MPC in the honest majority setting. In Tal Malkin and Chris Peikert, editors, *Advances in Cryptology – CRYPTO 2021, Part II*, volume 12826 of *Lecture Notes in Computer Science*, pages 244–274, Virtual Event, August 16–20, 2021. Springer, Heidelberg, Germany.

GLS19.   Vipul Goyal, Yanyi Liu, and Yifan Song. Communication-efficient unconditional MPC with guaranteed output delivery. In Alexandra Boldyreva and Daniele Micciancio, editors, *Advances in Cryptology – CRYPTO 2019, Part II*, volume 11693 of *Lecture Notes in Computer Science*, pages 85–114, Santa Barbara, CA, USA, August 18–22, 2019. Springer, Heidelberg, Germany.

GPS19.   Yue Guo, Rafael Pass, and Elaine Shi. Synchronous, with a chance of partition tolerance. In Alexandra Boldyreva and Daniele Micciancio, editors, *Advances in Cryptology – CRYPTO 2019, Part I*, volume 11692 of *Lecture Notes in Computer Science*, pages 499–529, Santa Barbara, CA, USA, August 18–22, 2019. Springer, Heidelberg, Germany.

GS20.   Vipul Goyal and Yifan Song. Malicious security comes free in honest-majority MPC. Cryptology ePrint Archive, Report 2020/134, 2020. https://eprint.iacr.org/2020/134.

GSZ20.   Vipul Goyal, Yifan Song, and Chenzhi Zhu. Guaranteed output delivery comes free in honest majority MPC. In Daniele Micciancio and Thomas Ristenpart, editors, *Advances in Cryptology – CRYPTO 2020,*

Part II, volume 12171 of *Lecture Notes in Computer Science*, pages 618–646, Santa Barbara, CA, USA, August 17–21, 2020. Springer, Heidelberg, Germany.

HIJ$^+$16.　Shai Halevi, Yuval Ishai, Abhishek Jain, Eyal Kushilevitz, and Tal Rabin. Secure multiparty computation with general interaction patterns. In Madhu Sudan, editor, *ITCS 2016: 7th Conference on Innovations in Theoretical Computer Science*, pages 157–168, Cambridge, MA, USA, January 14–16, 2016. Association for Computing Machinery.

HLP11.　Shai Halevi, Yehuda Lindell, and Benny Pinkas. Secure computation on the web: Computing without simultaneous interaction. In Phillip Rogaway, editor, *Advances in Cryptology – CRYPTO 2011*, volume 6841 of *Lecture Notes in Computer Science*, pages 132–150, Santa Barbara, CA, USA, August 14–18, 2011. Springer, Heidelberg, Germany.

LN17.　Yehuda Lindell and Ariel Nof. A framework for constructing fast MPC over arithmetic circuits with malicious adversaries and an honest-majority. In Bhavani M. Thuraisingham, David Evans, Tal Malkin, and Dongyan Xu, editors, *ACM CCS 2017: 24th Conference on Computer and Communications Security*, pages 259–276, Dallas, TX, USA, October 31 – November 2, 2017. ACM Press.

Mau03.　Ueli M. Maurer. Secure multi-party computation made simple (invited talk). In Stelvio Cimato, Clemente Galdi, and Giuseppe Persiano, editors, *SCN 02: 3rd International Conference on Security in Communication Networks*, volume 2576 of *Lecture Notes in Computer Science*, pages 14–28, Amalfi, Italy, September 12–13, 2003. Springer, Heidelberg, Germany.

Mau06.　Ueli M. Maurer. Secure multi-party computation made simple. *Discrete Applied Mathematics*, 154(2):370–381, 2006.

NV18.　Peter Sebastian Nordholt and Meilof Veeningen. Minimising communication in honest-majority MPC by batchwise multiplication verification. In Bart Preneel and Frederik Vercauteren, editors, *ACNS 18: 16th International Conference on Applied Cryptography and Network Security*, volume 10892 of *Lecture Notes in Computer Science*, pages 321–339, Leuven, Belgium, July 2–4, 2018. Springer, Heidelberg, Germany.

Res18.　Eric Rescorla. The Transport Layer Security (TLS) Protocol Version 1.3. IETF, RFC 8446, 2018.

RS22.　Rahul Rachuri and Peter Scholl. Le mans: Dynamic and fluid MPC for dishonest majority. In Yevgeniy Dodis and Thomas Shrimpton, editors, *Advances in Cryptology – CRYPTO 2022, Part I*, volume 13507 of *Lecture Notes in Computer Science*, pages 719–749, Santa Barbara, CA, USA, August 15–18, 2022. Springer, Heidelberg, Germany.

Sha79.　Adi Shamir. How to share a secret. *Communications of the Association for Computing Machinery*, 22(11):612–613, November 1979.

SW19.　Nigel P. Smart and Tim Wood. Error detection in monotone span programs with application to communication-efficient multi-party computation. In Mitsuru Matsui, editor, *Topics in Cryptology – CT-RSA 2019*, volume 11405 of *Lecture Notes in Computer Science*, pages 210–229, San Francisco, CA, USA, March 4–8, 2019. Springer, Heidelberg, Germany.

Zen20.　ZenGo. White-City: A framework for massive mpc with partial synchrony and partially authenticated channels, 2020.