

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

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.

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

© International Association for Cryptologic Research 2023
J. Guo and R. Steinfeld (Eds.): ASIACRYPT 2023, LNCS 14438, pp. 172–203, 2023.
https://doi.org/10.1007/978-981-99-8721-4_6

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 [23]. 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.

Dynamic Computation with Delays. In common with most secure protocols, one works (ideally) in a Dolev-Yao model [11] 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. We provide more comparison with [23] is given in the full version.

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.

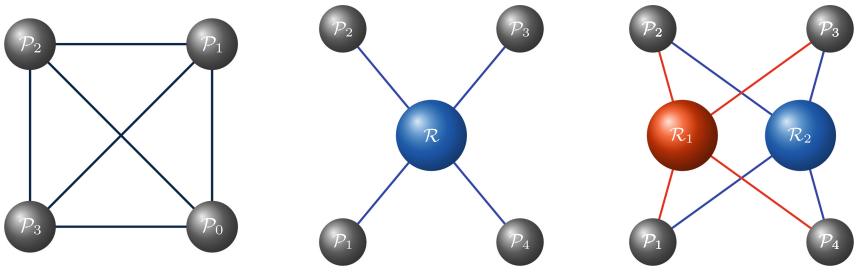

Fig. 1. Comparison between different topology networks with 4 computing parties $\mathcal{P}_1, \mathcal{P}_2, \mathcal{P}_3, \mathcal{P}_4$: a full network topology, a star-like network with 1 relay node and a replicated star-like topology with 2 relays.

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 Fig. 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 [1–3,8,9,12,15,22], 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 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. [4]. 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 in the full version.

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 $t_p \leq t < n/2$, 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 [14]. As observed in prior work, some of the most efficient passively secure multiplication protocol [10,21] 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. [7], also used in Fluid MPC [8] and LeMans [22], 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 [8,14], 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 the full version that if we instantiate $\mathcal{F}_{\mathsf{Mult}}$ with the widely used Maurer's multiplication protocol [20], 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 [10] (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* [13,17] 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, in the full version, 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 [7], 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 [5,18,19].

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 Fig. 4, in pure Rust, and compared its performance to that of direct communication between two parties. The results in Sect. 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.

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.

In the full version we cover the basic definitions of Shamir's secret sharing, encryption and circuit depth/width that we will require.

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) [6], 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 (timeout) 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 Internal Additive Attacks

Intuitively, an additive attack (Definition 1) 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 [8,14] is due to the check in our protocol for opening a value, given in Fig. 5, Sect. 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 1 (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.*

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 Fig. 4. It implements the ideal functionality $\mathcal{F}_{\mathsf{SecureRobustRelay}}$, given in the full version.

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.

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 $\epsilon_{\tau_{i,j}} \in \{\perp, 0, 1\}$, and $\epsilon_{\tau_{i,\mathcal{P},j}} \in \{\perp, 0, 1\}$.
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 $(\texttt{init})$ from all parties the functionality sets all the counters $\tau \leftarrow 0$ and $\epsilon \leftarrow \perp$.

Send: On input $(\texttt{send}, \mathcal{R}, \mathcal{P}_i, \mathcal{P}_j, m)$ from $\mathcal{P}_i$,
1. Send $(\texttt{sent}, i, j, m)$ 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 $(\texttt{sendToAll}, \mathcal{R}, \mathcal{P}_i, \mathcal{P}, m)$ from $\mathcal{P}_i$,
1. Send $(\texttt{sendToAll}, i, \mathcal{P}, m)$ 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 $(\texttt{erase}, \mathcal{R}, \mathcal{P}_i, \mathcal{P}_j, \tau_{i,j})$ from $\mathcal{P}_j$.
1. Send $(\texttt{erase}, \mathcal{P}_i, \mathcal{P}_j, \tau)$ 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 $(\texttt{eraseAll}, \mathcal{R}, \mathcal{P}, \mathcal{P}_j, \boldsymbol{\tau})$ from $\mathcal{P}_j$, where $\boldsymbol{\tau} = \{\tau_{i,\mathcal{P}}\}_{i \neq j}$
1. Send $(\texttt{eraseAll}, \mathcal{P}, \mathcal{P}_j, \{\tau_{i,\mathcal{P}}\}_{i \neq j})$ 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 $(\texttt{request}, \mathcal{R}, \mathcal{P}_i, \mathcal{P}_j, \tau_{i,j})$ from $\mathcal{P}_j$,
1. Send $(\texttt{request}, \mathcal{P}_i, \mathcal{P}_j, \tau_{i,j})$ to $\mathcal{R}$.
2. If $\mathcal{R}$ is corrupt, wait for $(\texttt{Deliver}, \tilde{m})$ from $\mathcal{S}$, send $\tilde{m}$ to $\mathcal{P}_j$.
   Else, if $\epsilon_{\tau_{i,j}} \neq 0$ then return $\perp$ to $\mathcal{P}_j$, else retrieve $m_{\tau_{i,j}}$ and send it to $\mathcal{P}_j$.

RequestFromAll: On input $(\texttt{requestFromAll}, \mathcal{R}, \mathcal{P}, \mathcal{P}_j, \boldsymbol{\tau})$ from $\mathcal{P}_j$, where $\boldsymbol{\tau} = \{\tau_{i,\mathcal{P}}\}_{i \neq j}$,
1. Send $(\texttt{requestFromAll}, \mathcal{P}, \mathcal{P}_j, \{\tau^{\mathcal{R}}_{i,\mathcal{P}}\}_{i \neq j})$ to $\mathcal{R}$.
2. If $\mathcal{R}$ is corrupt, wait for $(\texttt{Deliver}, \widetilde{\mathbf{m}})$ from $\mathcal{S}$, send $\widetilde{\mathbf{m}}$ 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 $\perp$, otherwise retrieve the ith-message corresponding to $\tau_{i,\mathcal{P}}$. Send the vector $\mathbf{m}$ to $\mathcal{P}_j$.

Figure 2. Functionality modelling a single relay

3.1 A Single Relay

The functionality $\mathcal{F}_{\mathsf{SingleRelay}}$, described in Fig. 2, 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.

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 $\epsilon_{\tau_{i,j}} = \perp$ 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.

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 $\delta_i \leftarrow \perp$ for all $i \in [n]$.
Delay: On input $(\texttt{delay}, \mathcal{P}_i, \texttt{command})$:

- If $\delta_i = \perp$: Send $(\texttt{delay}, i)$ to $\mathcal{S}$ and wait for input.
  - If $\mathcal{S}$ returns $(\texttt{delay}, D_i)$ and $0 < D_i \leq \delta$, then
    1. Send $(\texttt{delayed}, D_i)$ to $\mathcal{S}$.
    2. Set $\delta_i \leftarrow D_i$.
  - If $\mathcal{S}$ returns $(\texttt{delay}, D_i)$ and $D_i \geq \delta$, then
    1. Send $(\texttt{delayed}, \delta)$ to $\mathcal{S}$.
    2. Set $\delta_i \leftarrow \delta$.
  - Else, send ok to $\mathcal{P}_i$
- If $\delta_i \neq \perp$:
  1. Set $\delta_i \leftarrow \delta_i - 1$.
  2. If $\delta_i = 0$: send ok to $\mathcal{P}_i$

Figure 3. Functionality modelling delays

3.2 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 Fig. 3. 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 $(\texttt{delay}, \mathcal{P}_i, \texttt{command})$, where `command` represent a specific action that $\mathcal{P}_i$ is trying to execute, we distinguish two different cases. If $\delta_i = \perp$, it means that $\mathcal{P}_i$ is not currently delayed. Hence, the functionality sends a message $(\texttt{delay}, i)$ to the ideal adversary $\mathcal{S}$. If $\mathcal{S}$ replies with $(\texttt{delay}, D_i)$, 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 $\delta_i \neq \perp$, $\mathcal{P}_i$ 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 $\mathcal{R}_1, \ldots, \mathcal{R}_r$.

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 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.

3.3 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. We could assume these keys are pre-distributed, however for completeness in the full version we present how the parties can execute a key agreement protocol over a set of relays, as just defined. However, to do so requires that a majority of the relays is honest.

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.

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

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, $\mathsf{ct}_k = \perp$ then return $\perp$ //it might be that the message has not yet been sent by party $\mathcal{P}_i$.
   - Else, for each k such that $\mathsf{ct}_k \neq \perp$, compute $m_k \leftarrow \mathsf{Dec}_{\mathsf{k}_{i,j}}(\mathsf{ct}_k)$.
     - If there is a *unique* $m_k \neq \perp$ then accept this value.
     - Else, if there is more than one value $m_k \neq \perp$, then 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 message 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, $m_{i,k} = \perp$, then set $m_{\tau_{i,\mathcal{P}}} = \perp$.
   - Otherwise, if there is a *unique* $m_{i,k} \neq \perp$, for all k, then accept this value; else, if there is more than one value $m_{i,k}$, for different k, such that $m_{i,k} \neq \perp$, then 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 4. Protocol $\Pi_{\mathsf{SecureRobustRelay}}$

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. For the proof see the full version.

Theorem 1. *The protocol* $\Pi_{\mathsf{SecureRobustRelay}}$ *securely with abort realizes* $\mathcal{F}_{\mathsf{SecureRobustRelay}}$ *in the* $\{\mathcal{F}_{\mathsf{SingleRelay}}, \mathcal{F}_{\mathsf{Delay}}, \mathcal{F}_{\mathsf{KE}}\}$*-hybrid model.*

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 $\mathcal{F}_{\mathsf{SecureRobustRelay}}$-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.

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 $\mathtt{ct}_{x_i}$ 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 $(\mathtt{send}, \mathcal{P}_i, \mathcal{P}_j)$ on $\mathcal{F}_{\mathsf{SecureRobustRelay}}$, inputting a vector $\mathbf{x}$ after receiving ok, where $\mathbf{x}$ is the vector consisting of r-values $\mathtt{ct}_{x_i}$ 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 $y_i \neq \perp$, and the vector $\mathbf{x}^{\mathcal{M}}$ of values $y_i \neq \perp$.
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, 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 5. Procedure to open a sharing $\langle x \rangle$ towards a single party $\mathcal{P}_j$ or to $\mathcal{P}$

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

Sub-protocol $\mathsf{Open}(i, \langle x \rangle_t)$**.** Described in Fig. 5, 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 Fig. 6 and allows party $\mathcal{P}_i$ to receive a vector $\mathbf{y}$ of shares/values via a number of `request` to $\mathcal{F}_{\mathsf{SecureRobustRelay}}$. 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 $\mathcal{F}_{\mathsf{SecureRobustRelay}}$. 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-protocols $\mathsf{Receive}(j, \langle x \rangle)$ **and** $\mathsf{ReceiveFromAll}(j, \iota)$

$\mathsf{Receive}(j, \iota)$:

1. Party $\mathcal{P}_j$ initializes a vector $\mathbf{y}$ of length n containing $\perp$ 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-$\perp$ values:
   (a) For $i \in [n]$, if $y_i = \perp$ 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.

$\mathsf{ReceiveFromAll}(j, \iota)$:

1. Party $\mathcal{P}_j$ initializes a vector $\mathbf{y}$ of length n containing $\perp$ 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-$\perp$ 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 6. Sub-Procedures to open a sharing $\langle x \rangle$

Sub-Protocol $\Pi_{\mathsf{Input(i)}}$. It is a data-input sub-protocol given in Fig. 7. It requires that all parties are in consensus about the value broadcast by party $\mathcal{P}_i$, thus we need all parties to terminate $\mathsf{Input}(i)$ 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 $\mathcal{F}_{\mathsf{Rand}}$.

Functionalities $\mathcal{F}_{\mathsf{Rand}}$ **and** $\mathcal{F}_{\mathsf{Coin}}$**.** When the number of parties is small, we will use the relatively standard functionality $\mathcal{F}_{\mathsf{Rand}}^{\mathsf{NI}}$, described in the full version. It generates a random degree-t Shamir sharing $\langle r \rangle_t$ in a non-interactive manner. In the full version, we also discuss how we proceed for larger values of n and t_p, and describe a protocol Π_{Rand} that implements the interactive functionality $\mathcal{F}_{\mathsf{Rand}}$ in the $\mathcal{F}_{\mathsf{SecureRobustRelay}}$-hybrid model. The amortized communication cost

Sub-protocol $\mathsf{Input}(i)$

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 $\mathsf{Open}(\mathcal{P}_i, \langle v \rangle)$, 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 $(\mathtt{sendToAll}, \mathcal{P}_i, \mathcal{P})$, 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 $(\mathtt{request}, \mathcal{P}_i, \mathcal{P}_j)$, inputting τ on receiving ok, for the requisite value of τ.
   - If this returns $\perp$ then party $\mathcal{P}_j$ aborts.
   - If a value w is returned then call $(\mathtt{erase}, \mathcal{P}_i, \mathcal{P}_j)$, inputting τ on receiving ok.
5. The parties set $\langle x \rangle \leftarrow \langle v \rangle + w$.

Figure 7. Sub-protocol $\mathsf{Input}(i)$ to enable party $\mathcal{P}_i$ to enter a secret value x into the computation.

of Π_{Rand} is $\frac{n-1}{n-t-1}$. In the full version, we also explain how to produce a protocol implementing a functionality $\mathcal{F}_{\mathsf{DoubleRand}}$, which produces a random double sharing $(\langle r \rangle_t, \langle r \rangle_{2t})$, and a protocol to implement a functionality $\mathcal{F}_{\mathsf{Coin}}$, which generates a common random value.

4.1 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 $(\mathtt{Multiply}, \mathcal{P}', \mathsf{id}_x, \mathsf{id}_y, \mathsf{id}_z)$, where $\mathcal{P}' \subseteq \mathcal{P}$, and $\mathsf{id}_x, \mathsf{id}_y$ are present in memory. Retrieve the values x, y. Send $(\mathtt{Multiply}, \mathcal{P}', \mathsf{id}_x, \mathsf{id}_y)$ 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 $(\mathtt{Done}, \hat{\mathcal{P}}, \delta_a, \{z_i\}_{i \in \mathcal{P}_I})$, 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 $(\mathtt{Done}, \hat{H})$, retrieve the $(\mathsf{id}_z, \langle z \rangle)$ and sends $\{z_i\}_{\hat{H}}$ to parties in $\hat{H}$.

Figure 8. The functionality $\mathcal{F}_{\mathsf{Mult}}$ secure up to additive attacks

In our main MPC protocol we will use the ideal functionality $\mathcal{F}_{\mathsf{Mult}}$ (given in Fig. 8) 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.

The protocol Π_{Mult1} (Fig. 9), which we use to implement $\mathcal{F}_{\mathsf{Mult}}$, is an adaptation of [10,16] 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. For the proof see the full version.

Lemma 1. *The protocol Π_{Mult1} described in Fig. 9 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.*

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 $(\texttt{sendToAll}, \mathcal{P}_i, \mathcal{P})$. 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 9. Protocol Π_{Mult1} to compute $\langle z\rangle_t$.

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[1], 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 the full version. At a high level the protocol proceeds in three stages: an *input stage*, that is instantiated with the sub-protocol Π_{Input}, described in Fig. 7, and with calls to the ideal functionalities $\mathcal{F}_{\mathsf{Rand}}$ and $\mathcal{F}_{\mathsf{DRand}}$, an *evaluation stage*, consisting in the evaluation of linear and multiplicative gates, and an *output stage*, where

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

parties call the sub-protocol Open, given in Fig. 5. We start with a protocol Π_{PMPC} that evaluates a (randomized) circuit C in the $\mathcal{F}_{\mathsf{SecureRobustRelay}}$-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 1.

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 in the full version the following theorem.

Theorem 2. *Given a randomized circuit C, the protocol Π_{PMPC} for computing C is secure against any δ-delaying passive adversary controlling up to $t < n/2$ parties and $r-1$ relays in the $\{\mathcal{F}_{\mathsf{DoubleRand}}, \mathcal{F}_{\mathsf{Rand}}, \mathcal{F}_{\mathsf{Mult}}, \mathcal{F}_{\mathsf{SecureRobustRelay}}\}$-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.*

In the full version we also examine the effect of delays on the state size needed to be held by a relay in the best case of no delays, the worst case and the case of random delays. We present a simulation for the case of random delays which models the state size of the relays and the round in which the MPC protocol terminates, when evaluating a circuit of a given multiplicative depth.

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

In Fig. 10, we give our functionality $\mathcal{F}_{\mathsf{DelayedMPC}}$, with the associated implementing protocol $\Pi_{\mathsf{DelayedMPC}}$, being given in Fig. 11. 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 [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.

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 $(\texttt{init}, \mathbb{F})$ from all parties, if $(\texttt{init})$ 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 $(\mathsf{abort}, \hat{H})$, send abort to honest parties $\hat{H} \subseteq \mathcal{P}_H$; when $\mathcal{S}$ sends $(\texttt{Done}, \hat{\mathcal{P}})$, send the output to honest parties in $\hat{\mathcal{P}}$.

Figure 10. 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 abort then output abort.
2. If $T \neq 0$ then output abort, otherwise return $\mathbf{z}$.

Figure 11. The protocol $\Pi_{\mathsf{DelayedMPC}}$ for secure delayed MPC

Unlike [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 [14] and it is perhaps closest to the definition given in [8] 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 [8].

Definition 2 (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:*

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 2. *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 2 of depth $d + 4$ and maximum width $4 \cdot w$, assuming $|\mathbf{x}| \leq w$.*

The proof of this lemma and a more intuitive description of how a robust circuit is evaluated is given in the full version. 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, which is proved in the full version. In the full version, we also give an informal intuition of why the final check can be performed even in presence of delays and without requiring additional randomness for slower parties.

Theorem 3. *Let $\tilde{C}$ be the robust circuit over $\mathbb{F}$ corresponding to C (according to Definition 2). The protocol $\Pi_{\mathsf{DelayedMPC}}$ in Fig. 11, 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.*

A complete proof can be found in the full version. Intuitively, however, the security of the protocol follows from the security of Π_{PMPC} and the zero-check

provided by the robust variant of C. Note that, while corrupt parties can always output abort, if an honest party $\mathcal{P}_i$ outputs abort in the final stage of $\Pi_{\mathsf{DelayedMPC}}$, either they received inconsistent shares in one of the openings or the check did not pass. Since, in both cases, the shares causing abort are stored in the relays and cannot be changed, this means that all honest parties concluding the protocol after $\mathcal{P}_i$ will also output abort. If, on the contrary, the check passed, then finishing parties $\hat{\mathcal{P}}$ open the correct value $\mathbf{y}$ except with negligible probability. An important observation is that, when more parties will finish the circuit evaluation, they are going to use the same shares and randomness used by a previous set of parties in the zero-check, rather than additional shares, which in general could be problematic. However, as noticed in the previous section, a malicious behaviour of $\mathcal{A}$ is somehow fixed by the first $2t + 1$ shares stored in the relays, because these shares cannot be changed any more if at least one of the relays is honest. Therefore, if there is an additive attack on the output of a multiplication gate, this must be reflected in the first $2t + 1$ shares revealed, otherwise when more shares of the output value are sent to the relays, and later to the parties, inconsistencies will be noticed by honest parties. More concretely, this implies that if the first subset of honest parties $\hat{\mathcal{P}}_H$ successfully finish the protocol, then slower honest parties either output the same correct value or they abort the computation. In the full version we also give an estimation of the communication complexity of our protocol and discuss further optimizations.

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. Precise numerical values for the main results presented in this section are given in the full version.

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 128 GB RAM. The parties communicating through the relays run machines with an Intel i7-770 CPU and a 32 GB 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 Fig. 12 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 16 kB 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.

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 with-

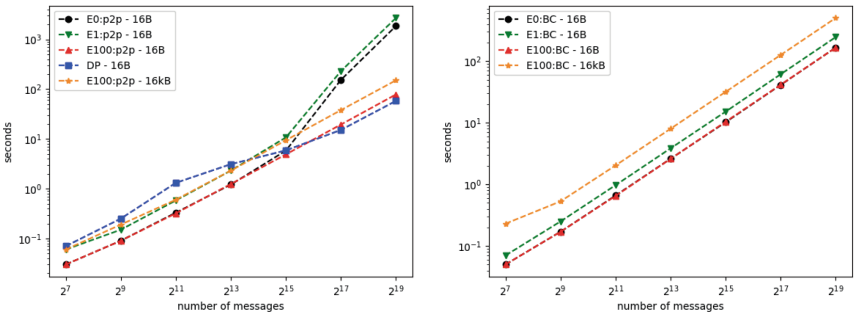

Fig. 12. Runtimes (with logarithmic scale on the y-axis) for experiments *DP*, *E0*, *E1* and *Ek* with $k = 100$ and 16-byte messages, and for *Ek* with $k = 100$ and 16 kB messages, showing: (left) p2p messages; (right) broadcast messages.

out 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 Subsect. 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 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 Fig. 13.

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.

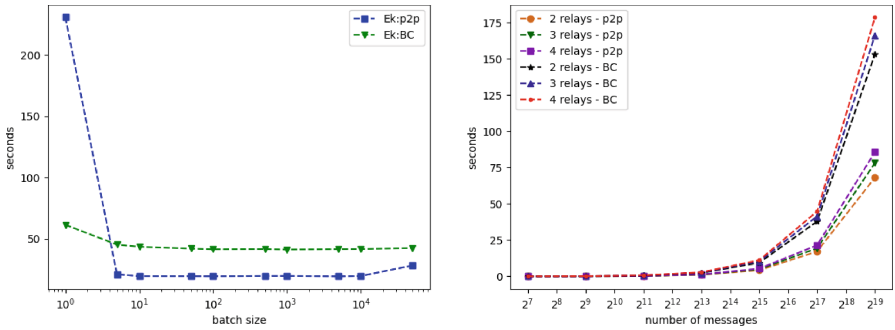

Fig. 13. Communication runtimes, showing for both the p2p and broadcast (BC) communications: (left) experiment *Ek* with 2^{17} 16-byte messages and increasing batch size k; (right) experiment *Ek* with $k = 100$ and 16-byte messages with 2, 3 and 4 relays.

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 Fig. 13 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 128 GB RAM) and the other on one of the slower machines (Intel i7-770 CPU and 32 GB 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 Fig. 14, 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 Fig. 14, for the average of the three fast and the three slow parties.

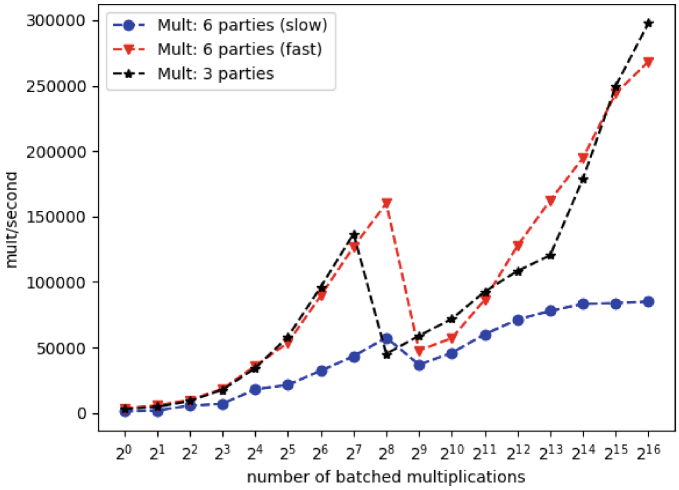

Fig. 14. Number of multiplications per second for increasing size of multiplication batches and erasing messages every 100 multiplication rounds, for 3 parties and 3 relays and for 6 parties and 2 relays.

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. This work was supported by CyberSecurity Research Flanders with reference number VR20192203, by the FWO under an Odysseus project GOH9718N, and by the Flemish Government through FWO SBO project SNIPPET S007619N.

The work of the second and third authors was primarily carried out while these authors were affiliated with COSIC.

References

1. Acharya, A., Hazay, C., Kolesnikov, V., Prabhakaran, M.: SCALES - MPC with small clients and larger ephemeral servers. In: Kiltz, E., Vaikuntanathan, V. (eds.) TCC 2022: 20th Theory of Cryptography Conference, Part II. LNCS, vol. 13748, pp. 502–531. Springer, Heidelberg (2022). https://doi.org/10.1007/978-3-031-22365-5_18
2. Alon, B., Naor, M., Omri, E., Stemmer, U.: 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
3. Badrinarayanan, S., Jain, A., Manohar, N., Sahai, A.: Secure MPC: laziness leads to GOD. In: Moriai, S., Wang, H. (eds.) ASIACRYPT 2020. LNCS, vol. 12493, pp. 120–150. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-64840-4_5
4. Benhamouda, F., Boyle, E., Gilboa, N., Halevi, S., Ishai, Y., Nof, A.: Generalized pseudorandom secret sharing and efficient straggler-resilient secure computation. In: Nissim, K., Waters, B. (eds.) TCC 2021. LNCS, vol. 13043, pp. 129–161. Springer, Cham (2021). https://doi.org/10.1007/978-3-030-90453-1_5
5. Boneh, D., Boyle, E., Corrigan-Gibbs, H., Gilboa, N., Ishai, Y.: Zero-knowledge proofs on secret-shared data via fully linear PCPs. In: Boldyreva, A., Micciancio, D. (eds.) CRYPTO 2019. LNCS, vol. 11694, pp. 67–97. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-26954-8_3
6. Canetti, R., Krawczyk, H.: Analysis of key-exchange protocols and their use for building secure channels. In: Pfitzmann, B. (ed.) EUROCRYPT 2001. LNCS, vol. 2045, pp. 453–474. Springer, Heidelberg (2001). https://doi.org/10.1007/3-540-44987-6_28
7. Chida, K., et al.: Fast large-scale honest-majority MPC for malicious adversaries. In: Shacham, H., Boldyreva, A. (eds.) CRYPTO 2018. LNCS, vol. 10993, pp. 34–64. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-96878-0_2
8. Choudhuri, A.R., Goel, A., Green, M., Jain, A., Kaptchuk, G.: Fluid MPC: secure multiparty computation with dynamic participants. In: Malkin, T., Peikert, C. (eds.) CRYPTO 2021. LNCS, vol. 12826, pp. 94–123. Springer, Cham (2021). https://doi.org/10.1007/978-3-030-84245-1_4
9. Damgård, I., Escudero, D., Polychroniadou, A.: Phoenix: secure computation in an unstable network with dropouts and comebacks. Cryptology ePrint Archive, Report 2021/1376 (2021). https://eprint.iacr.org/2021/1376
10. Damgård, I., Nielsen, J.B.: Scalable and unconditionally secure multiparty computation. In: Menezes, A. (ed.) CRYPTO 2007. LNCS, vol. 4622, pp. 572–590. Springer, Heidelberg (2007). https://doi.org/10.1007/978-3-540-74143-5_32

11. Dolev, D., Yao, A.C.C.: On the security of public key protocols (extended abstract). In: 22nd Annual Symposium on Foundations of Computer Science, pp. 350–357. IEEE Computer Society Press, Nashville, TN, USA, 28–30 October 1981. https://doi.org/10.1109/SFCS.1981.32
12. Fitzi, M., Hirt, M., Maurer, U.: Trading correctness for privacy in unconditional multi-party computation. In: Krawczyk, H. (ed.) CRYPTO 1998. LNCS, vol. 1462, pp. 121–136. Springer, Heidelberg (1998). https://doi.org/10.1007/BFb0055724
13. Furukawa, J., Lindell, Y.: Two-thirds honest-majority MPC for malicious adversaries at almost the cost of semi-honest. In: Cavallaro, L., Kinder, J., Wang, X., Katz, J. (eds.) ACM CCS 2019: 26th Conference on Computer and Communications Security, pp. 1557–1571. ACM Press, London, UK, 11–15 November 2019. https://doi.org/10.1145/3319535.3339811
14. Genkin, D., Ishai, Y., Prabhakaran, M., Sahai, A., Tromer, E.: Circuits resilient to additive attacks with applications to secure computation. In: Shmoys, D.B. (ed.) 46th Annual ACM Symposium on Theory of Computing, pp. 495–504. ACM Press, New York, NY, USA, 31 May–3 June 2014. https://doi.org/10.1145/2591796.2591861
15. Gentry, C., et al.: YOSO: you only speak once. In: Malkin, T., Peikert, C. (eds.) CRYPTO 2021. LNCS, vol. 12826, pp. 64–93. Springer, Cham (2021). https://doi.org/10.1007/978-3-030-84245-1_3
16. Goyal, V., Li, H., Ostrovsky, R., Polychroniadou, A., Song, Y.: ATLAS: efficient and scalable MPC in the honest majority setting. In: Malkin, T., Peikert, C. (eds.) CRYPTO 2021. LNCS, vol. 12826, pp. 244–274. Springer, Cham (2021). https://doi.org/10.1007/978-3-030-84245-1_9
17. Goyal, V., Liu, Y., Song, Y.: Communication-efficient unconditional MPC with guaranteed output delivery. In: Boldyreva, A., Micciancio, D. (eds.) CRYPTO 2019. LNCS, vol. 11693, pp. 85–114. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-26951-7_4
18. Goyal, V., Song, Y.: Malicious security comes free in honest-majority MPC. Cryptology ePrint Archive, Report 2020/134 (2020). https://eprint.iacr.org/2020/134
19. Goyal, V., Song, Y., Zhu, C.: Guaranteed output delivery comes free in honest majority MPC. In: Micciancio, D., Ristenpart, T. (eds.) CRYPTO 2020. LNCS, vol. 12171, pp. 618–646. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-56880-1_22
20. Maurer, U.: Secure multi-party computation made simple. In: Cimato, S., Persiano, G., Galdi, C. (eds.) SCN 2002. LNCS, vol. 2576, pp. 14–28. Springer, Heidelberg (2003). https://doi.org/10.1007/3-540-36413-7_2
21. Maurer, U.M.: Secure multi-party computation made simple. Discret. Appl. Math. **154**(2), 370–381 (2006). https://doi.org/10.1016/j.dam.2005.03.020
22. Rachuri, R., Scholl, P.: Le mans: dynamic and fluid MPC for dishonest majority. In: Dodis, Y., Shrimpton, T. (eds.) Advances in Cryptology - CRYPTO 2022, Part I. LNCS, vol. 13507, pp. 719–749. Springer, Heidelberg (2022). https://doi.org/10.1007/978-3-031-15802-5_25
23. ZenGo: White-city: a framework for massive MPC with partial synchrony and partially authenticated channels (2020). https://github.com/ZenGo-X/white-city/blob/master/White-City-Report/whitecity_new.pdf