On the Computational Overhead of MPC with Dishonest Majority

Jesper Buus Nielsen[1] and Samuel Ranellucci[2,3]
jbn@cs.au.dk, samuel@umd.edu

[1] Department of Computer Science, Aarhus University, Aarhus, Denmark
[2] Department of Computer Science, George Mason University, Virginia, USA
[3] Department of Computer Science, University of Maryland, Maryland, USA

Abstract. We consider the situation where a large number n of players want to securely compute a large function f with security against an adaptive, malicious adversary which might corrupt $t < cn$ of the parties for some given $c \in [0, 1)$. In other words, only some arbitrarily small constant fraction of the parties are assumed to be honest. For any fixed c, we consider the asymptotic complexity as n and the size of f grows. We are in particular interested in the computational overhead, defined as the total computational complexity of all parties divided by the size of f. We show that it is possible to achieve poly-logarithmic computational overhead for all $c < 1$. Prior to our result it was only known how to get poly-logarithmic overhead for $c < \frac{1}{2}$. We therefore significantly extend the area where we can do secure multiparty computation with poly-logarithmic overhead. Since we allow that more than half the parties are corrupted, we can only get security with abort, i.e., the adversary might make the protocol abort before all parties learn their outputs. We can, however, for all c make a protocol for which there exists $d > 0$ such that if at most dn parties are actually corrupted in a given execution, then the protocol will not abort. Our result is solely of theoretical interest. In its current form, it has not practical implications whatsoever.

1 Introduction

We consider the situation where a large number n of players want to securely compute a large function f with security against an adaptive, malicious adversary which might corrupt $t < cn$ of the parties for some given constant $c \in [0, 1)$. In other words, only some arbitrarily small constant fraction of parties are assumed to be honest. We also require that there exists $d > 0$ such that if at most dn parties are actually corrupted in a given execution, then the protocol will not abort. We call this the setting with constant honesty and constant termination guarantee.

For any fixed c, we consider the asymptotic complexity as n and the size of f grows. We are in particular interested in the computational overhead, defined by summing the total computational complexity of all parties and dividing by the size of f. We show that it is possible to achieve poly-logarithmic computational overhead for all $c < 1$. Prior to our result, it was only known how to get

poly-logarithmic overhead for settings with constant honesty and constant termination guarantee for $c < \frac{1}{2}$ (cf. [DIK+08,CDD+15,BSFO12,BCP15,CDI+13]). We therefore significantly extend the area where we can do secure multiparty computation with poly-logarithmic overhead. Let us state up front that our result is only meant as an asymptotic feasibility result. The constants hidden by the asymptotic analysis are so huge that the protocol has no practical implications.

Our protocol is based on standard assumptions. It can be built in a white-box manner from essentially any multiparty computation protocol secure against any number of corrupted parties and a secure multiparty computation protocol which has poly-logarithmic overhead and which is secure when at most a quarter of parties are corrupt. Both protocols should be secure against a malicious, adaptive adversary. We give an information-theoretic secure protocol in the hybrid model with oblivious transfer and a small number of initial broadcasts. We also give a computationally secure protocol in the hybrid model with a CRS and a PKI.

We note that approaches based on selecting a small committees and having the committee run the computation are doomed to failure in our model. This is because any small committee can be corrupted by the adaptive adversary. The protocol from [CPS14] is insecure in our model precisely for this reason. We also note that our protocol, in contrast to the low overhead protocols of [DPSZ12,DZ13,DKL+13], does not rely on pre-processing. The IPS compiler [IPS08] is also a generic protocol with low computational overhead, but it has a quadratic overhead in the number of players, so it does not have a low computational overhead in the sense that we consider here. Finally, notice that an approach based on fully homomorphic encryption, where the n parties send their encrypted inputs to one party and lets this party do the computation can have a poly-logarithmic computational overhead. However, it does not have constant termination guarantee. To ensure this, it seems one would still need some constant fraction of the parties to do all the computation, suffering a blow up in the overhead of a factor $\Theta(n)$.

2 Technical Overview

Our protocol follows the same high level approach as [DIK+08] which is based on the work of [Bra87]. Our protocol is also inspired by the IPS compiler from [IPS08] and the player virtualization technique from [HM00]. The main idea is that we will run an honest majority protocol with poly-logarithmic overhead. Following [IPS08], we call this the outer protocol. Each of the parties P_i in the outer protocol will be emulated by a constant number of the parties running a protocol with security against any number of corrupted parties. The set of parties that run P_i is called committee number i. The protocol that committees run is called the inner protocol.

We use an expander graph to set up the committees so that except with negligible probability, a vast majority of committees will contain at least one honest player as long as at most cn of the real parties are corrupted. We call a committee consisting of only honest parties an honest committee. We call a

committee containing at least one honest party and at least one corrupted party a crashable committee. We call a committee consisting of only corrupted parties a corrupted committee. Since the inner protocol is secure against any number of corrupted parties, an honest committee corresponds to an honest party in the outer protocol and a corrupted committee corresponds to a corrupted party in the outer protocol. Since the inner protocol only guarantees termination when all parties in the committee are honest, a crashable committee corresponds to a party in the outer protocol which is running correctly and which has a private state, but which might crash—if a corrupted committee member makes the inner protocol abort.

We need the outer protocol to tolerate a large constant fraction of malicious corruptions (one quarter) along with any number of fail-stop errors. At the same time, we need it to guarantee termination if there is a large enough fraction of honest parties. On top of that the protocol needs to have poly-logarithmic overhead. Prior to our work, there is no such protocol in the literature. We show how to build such a protocol in a white-box manner from off-the-shelf protocols with poly-logarithmic overhead.

There are many additional complications along the way. Most honest majority protocols rely on private and authenticated channels. Since an adversary can corrupt players so that all committees contain a corrupted member,[4] we need a way to allow the inner protocols emulated by different sets of parties to communicate securely while hiding the messages from the committee members. We should also prevent corrupted committee members from attacking the delivery or authentication of the transmitted messages. In addition, when a user sends his input to an emulated party of the outer protocol emulated by a committee that may only have a single honest party, we should still be able guarantee that he can securely send a message to the inner protocol. This is necessary to ensure that an honest party cannot be prevented from giving input. To prevent this, we employ a multitude of new techniques described in the following technical sections which includes player elimination and the use of a tamper-resilient secret-sharing scheme.

Although the basic approach is the same, there are important technical differences between this work and [DIK+08]. In the following, we describe the most important ones. The work of [DIK+08] employs Verifiable Secret Sharing (VSS) to solve the problem of secure message transmission between parties. It also uses VSS to allow real parties to provide their inputs to the emulated parties of the outer protocol. The work of [DIK+08] can employ VSS because it can set up committees so that it is guaranteed that most committees have an honest majority. In contrast, since it could be that a majority of players are corrupt, we cannot ensure that any committee has an honest majority and therefore we cannot employ VSS. Another difference is that we need an explicit bipartite expander graph with constant left degree and constant right degree. Since we

[4] If we for instance start out with a setting where 99 out of every 100 parties are corrupted and we start forming random committees, of course we should expect all or essentially all committees to get a corrupted member.

could not find such a construction in the literature, we constructed such an expander using standard techniques.

3 Setting the Stage

We use λ to denote the *security parameter*. We consider a setting with n *players* $\mathsf{P}_1, \ldots, \mathsf{P}_n$. Here P_i is just a distinct name for each of the participating players. We use $\mathcal{P} = \{\mathsf{P}_1, \ldots, \mathsf{P}_n\}$ to denote the *set of players*. We assume that all players agree on $\mathcal{P}$. We assume that n is a function of λ and that $n(\lambda) \geq \lambda$. We often write n instead of $n(\lambda)$.

We also assume that the parties agree on a circuit C to be computed. We assume that all parties have a fixed size input in C. We use $s = \text{size}_{\text{BOOL}}(C)$ to denote the size of C.

By a protocol π, we mean a generic protocol which has an instantiation $\pi(C, \lambda, n)$ for each circuit C, value of the security parameter λ and number n of parties. We assume that there exists a uniform poly-time Turing machine which produces circuits computing $\pi(f, \lambda, n)$ given input $(C, 1^\lambda, 1^n)$. We do not consider the production of $\pi(C, \lambda, n)$ as part of the complexity of π. We use $\text{comp}(\pi(C, n, \lambda))$ to denote the expected total work done by all parties in $\pi(C, n, \lambda)$, where the work is measured as local *computation*, counting the sending and receiving of one bit as 1 towards the work. Note that $\text{comp}(\pi(C, n, \lambda))$ in particular is an upper bound on the *communication* of the protocol.

We are interested in the complexity of MPC as the size of C and the number of parties grow. We are in particular interested in the *overhead* of the computation, defined as the complexity of the protocol divided by the size of C. As usual, we are also interested in how the complexity grows with the security parameter λ and the number of parties n. In defining the computational overhead, we follow [DIK10]. Let OH be a function $\text{OH} : \mathbb{N} \times \mathbb{N} \times \mathbb{N} \to \mathbb{R}$. We say that π has computational overhead OH if there exists a polynomial $p : \mathbb{N} \times \mathbb{N} \times \mathbb{N} \to \mathbb{N}$ such that for all C, n and λ it holds that

$$\text{comp}(\pi(C, n, \lambda)) \leq \text{size}(C) \cdot \text{OH}(n, \lambda, \text{size}(f)) + p(n, \lambda, \log \text{size}(f)) \ .$$

Let NC be Nick's class, i.e., the set of functions that can be computed by circuits of poly-logarithmic depth. We want to securely evaluate f in a distributed manner without much overhead. Current techniques even for honest majority only achieve this if the computation of f can be parallelised. This is why we consider NC. Previous protocols essentially have the same restriction. For instance, the protocol in [DIK10] has a complexity of the form $s \log(s) + d^2 \cdot \text{poly}(n, \log(s))$, where s is the size of the circuit computing f and d is the depth of the circuit. That means that if d is not $\text{polylog}(s)$, then the overhead will not be $\text{polylog}(s)$.

We prove security in the UC model assuming a synchronous model, point-to-point channels and broadcast. The UC model is originally best geared towards modeling asynchronous computation, but it can be adapted to model synchronous computation.

3.1 UC and Synchronous Computation

Our study is cast in the synchronous model. Still, we would like to prove security in the UC model which by design considers asynchronous computation. The reason why we would like to use the UC model is to give modular proofs. We need to consider reactive secure computations for which the UC model is the *de facto* standard. One can cast synchronous computation in the UC model by using the techniques of [KMTZ13]. The model from [KMTZ13] is, however, much more refined and detailed than what we need, so we have decided to go for a simpler model that we present below.

We are going to assume that synchrony is ensured by the environment giving the parties P_i special inputs TICK modeling that the time has increased by one tick for P_i. The parties can then simply count what time it is. We then simply require that the environment keeps the clocks of two honest parties at most one tick apart. To make sure that all parts of a composed protocol and all ideal functionalities know what time it is, we require that all parties which receive an input TICK passes it on to all its sub-protocols and ideal functionalities.

In a bit more detail, synchrony is defined via a *synchrony contract* that all entities must follow for as long as all other entities do so. We describe the contract now for the different entities of the UC framework. In doing so, we describe the behaviour that the entity must show, assuming that all other parties followed the contract until that point. If an entity A observes another entity B breaking the contract, then A is allowed to exhibit arbitrary behaviour after that point.

Synchronous Environment A round is defined by all parties having received the input TICK from the environment. The environment might in each round give additional input x_i to a party P_i by inputting (TICK, x_i). In most of our We use r_i to denote the round in which P_i is. We say that party P_i is in round r_i, if it has received the input TICK exactly r_i times from the environment. The environment must ensure that $r_i \leq r_j + 1$ for all honest parties P_i and P_j. Furthermore, when the environment sends TICK to an honest party, it cannot send another TICK to that party until it has received an output from P_i.

Synchronous Parties If a party P_i gets input TICK from its environment it must after this input, send TICK exactly once to each of its ideal functionalities. Note that the caller might be a super-protocol instead of an environment and that P_i might be calling a sub-protocol instead of an ideal functionality. This is transparent to P_i and we will use *environment* to denote the entity calling P_i and *ideal functionality* to denote the entity being called by P_i. When an honest party received back an output from all the sub-entities to which it input TICK, it must deliver an output to its environment as the next thing.

Notice that if we compose a synchronous environment with a synchronous protocol to get a new environment, then we again have a synchronous environment, which is the main observation needed to lift the UC composition theorem to the synchronous setting.

In the following we will ignore the inputs TICK as they are only used to define which round we are in. We will say that P_i gets input x_i in round r_i if it gets input (TICK, x_i) in that round. We will say that P_i gets no input in round r_i if it gets input (TICK) in that round.

A *synchronous ideal functionality* is given by a transition function Tr which in each evaluation takes the state from the previous evaluation, an input from each other party and computes a new state and one output for each of the other parties. Each evaluation is started by the honest parties, each giving an input. For simplicity we require that these inputs are all given in the same round. We also assume that each evaluation has a fixed round complexity, given by a round function R. Evaluation number e will take $\mathsf{R}(e)$ rounds. If a corrupted party does not give an input, a default value is used. For a given transition function Tr and round function R the corresponding synchronous ideal functionality $\mathcal{F}_{\mathsf{Tr},\mathsf{R}}^{\text{SYNC}}$ is given in Fig. 1.

Initialize Let $e = 0$; This is a counter of how many evaluations were done so far. Throughout, let C denote the current set of corrupted parties and let H denote the current set of honest parties. Let $\sigma = 1^\lambda$; This is the initial internal state. Let $\mathsf{State} \leftarrow$ INPUTTING.

Honest Input If in some round all parties $\mathsf{P}_i \in H$ give an input x_i and $\mathsf{State} =$ INPUTTING, then set $x_j = \perp$ for $\mathsf{P}_j \in C$, store $(x_1, \ldots, x_n)$, let $e \leftarrow e+1$ and let $\mathsf{State} \leftarrow$ COMPUTING. (If in some round some honest party P_i gives an input x_i and some honest party P_j does not give an input or $\mathsf{State} \neq$ INPUTTING, then do a complete breakdown.)

Corrupt Input On input (P_i, x) for $\mathsf{P}_i \in C$ while $\mathsf{State} =$ COMPUTING, update $x_i \leftarrow x$. Then turn over the activation to the adversary.

Compute During the next $\mathsf{R}(e)$ rounds after setting $\mathsf{State} \leftarrow$ COMPUTING all honest parties just output TICK.

Eval If the adversary inputs (EVAL) and $\mathsf{State} =$ COMPUTING, then compute $(\sigma_e, y_1, \ldots, y_n) \leftarrow \mathsf{Tr}(\sigma_{e-1}, x_1, \ldots, x_n)$. Set $\mathsf{State} \leftarrow$ EVALUATED. Output $\{(i, y_i)\}_{\mathsf{P}_i \in C}$ to the adversary.

Output In round $\mathsf{R}(e)+1$, after setting $\mathsf{State} \leftarrow$ COMPUTING, output y_i to P_i for $\mathsf{P}_i \in H$ and let the adversary decide the order of delivery.

Abort The ideal functionality can be parametrized by an abort threshold a. If a is not specified, it is assumed that $a = n$. If the adversary inputs (ABORT) and $|C| > a$, then output ABORT to all honest parties and terminate.

Total Breakdown Doing a total breakdown in a given round means that the ideal functionality outputs the current and all previous σ_i to the adversary along with all previous inputs and then switches to a mode where it is the adversary that determines which messages are sent by the ideal functionality.

Fig. 1. Synchronous Ideal Functionality $\mathcal{F}_{\mathsf{Tr}}^{\text{SYNC}}$ for Transition Function Tr and Round Function R

We will be using an ideal functionality for synchronous communication. In each evaluation, party P_i has input $(x_{i,1}, \ldots, x_{i,n})$ and receives the output $(x_{1,i}, \ldots, x_{n,i})$, i.e., in each round each party can send a message to each other

party. We will not write this ideal functionality explicitly in our formal statements. We consider it the ground model of communication, i.e., it is present in all our (hybrid) models. The round complexity of each evaluation is 1.

We will be using an ideal functionality for broadcast between a set of parties $\mathsf{P}_1, \ldots, \mathsf{P}_n$. In each evaluation each party has input x_i and receives the output $(x_1, \ldots, x_n)$. The round complexity is the same in all rounds but might depend on the number of parties. For n parties, we use $\mathsf{R}_{\text{BROADCAST}}(n)$ to denote the round complexity of each round of broadcast among n parties.

3.2 Broadcast.

For our protocols, we require a synchronous broadcast channel. A broadcast channel is a primitive that allows a player to broadcast a message to a subset of the players. When a player receives a broadcasted message, he is assured that each other player received the same message.

The ideal functionality is for one sender S and r receivers $\mathsf{R}_1, \ldots, \mathsf{R}_r$.

Broadcast On input m from S and input BEGIN from all honest receivers R_i in the same round, wait for $\mathsf{R}_{\text{BROADCAST}}$ rounds and then output m to all receivers, letting the adversary determine the order of delivery.

Corrupt sender If S is corrupt and does not provide an input, then let $m = \perp$. Furthermore, if S is corrupt and the adversary inputs (REPLACE INPUT, m') before an output was delivered to the first honest party, then let $m \leftarrow m'$.

Fig. 2. The broadcast functionality $\mathcal{F}_{\text{BROADCAST}}$

4 The Outer Protocol

The *outer protocol* π_{OUT} involves n users $\mathsf{U}_1, \ldots, \mathsf{U}_n$ and m servers $\mathsf{S}_1, \ldots, \mathsf{S}_m$. Only the users have inputs and outputs. The protocol computes an n-party function $f : D^n \to E^n$ given by circuit C. We assume that $D = \{0,1\}^k$ and that $E = D \cup \{\perp\}$, but the protocol obviously generalises to differently structured inputs and outputs. We use $\perp$ to signal that a user did not get an output.

We assume that f is fixed and that the protocol runs in some fixed number of rounds, which we denote by R_{OUT}.

We assume that the only interactions involving users is in the first round where all users send a message to all servers (we call these the input messages) and in some given round R_{OUT} all servers send a message to all users (we call these the output messages). We use In_{OUT} to denote the randomized function used to compute input messages from an input and we use $\mathsf{Out}_{\text{OUT}}$ to denote the deterministic function used to compute the output from output messages.

We assume that in each round r, each server S_j sends one message $y^r_{j,k}$ to each of the other servers S_k. We use $y^r_{j,j}$ to denote the state of S_i after round r

and at the start of round $r+1$. We use Tr_{OUT} to denote the transition function of the servers: the function applied in each round to compute a new state and the messages to be sent in the given round.

Inputs
For $i = 1, \ldots, n$ user U_i has input x_i and has random tape t_i. For $j = 1, \ldots, m$ server S_j has no input and has random tape r_j.
Server initialization
For $j, k = 1, \ldots, m$ server S_j lets $y^0_{k,j}$ be the empty string.
Generation of input shares
For $i = 1, \ldots, n$ user U_i samples $(x_{i,1}, \ldots, x_{i,m}) \leftarrow \mathsf{In}_{\text{OUT}}(x_i; t_i)$.
Distribution of input shares
For $i = 1, \ldots, n$ and $j = 1, \ldots, m$ user U_i sends $x_{i,j}$ to server S_j.
Embedding of input shares
For $j = 1, \ldots, m$ server S_j sets $y^0_{j,j} \leftarrow (x_{1,j}, \ldots, x_{n,j}, r_j)$.
Evaluation rounds
For $r = 1, \ldots, \mathsf{R}_{\text{OUT}}$ round r runs as follows:
Transition
For $j = 1, \ldots, m$ server S_j computes $(y^r_{j,1}, \ldots, y^r_{j,m}) \leftarrow \mathsf{Tr}_{\text{OUT}}(r, y^{r-1}_{1,j}, \ldots, y^{r-1}_{m,j})$.
Communication
For $j, k = 1, \ldots, m$ server S_j sends $y^r_{j,k}$ to server S_k.
Generation of output shares
For $j = 1, \ldots, m$ server S_j computes $(z_{j,1}, \ldots, z_{j,n}) \leftarrow \mathsf{Tr}_{\text{OUT}}(\mathsf{R}_{\text{OUT}} + 1, y^{\mathsf{R}_{\text{OUT}}}_{1,j}, \ldots, y^{\mathsf{R}_{\text{OUT}}}_{m,j})$
Distribution of output shares
For $j = 1, \ldots, m$ and $i = 1, \ldots, n$ server S_j sends $z_{j,i}$ to user U_i.
Output reconstruction
For $i = 1, \ldots, n$ user U_i computes $z_i \leftarrow \mathsf{Out}_{\text{OUT}}(z_{1,i}, \ldots, z_{n,i})$.

Fig. 3. Running an outer protocol $\pi_{\text{OUT}} = (\mathsf{R}_{\text{OUT}}, \mathsf{In}_{\text{OUT}}, \mathsf{Tr}_{\text{OUT}}, \mathsf{Out}_{\text{OUT}})$ for f

We assume that users can be actively corrupted. To actively corrupt U_i the adversary will input (ACTIVE-CORRUPT) to U_i. In response to this U_i sends its internal state to the adversary, will forward all incoming messages to the adversary, and from now on, it is the adversary that determines what U_i sends. After an active corruption, a user is called malicious. A user is called correct if it is not malicious. We assume that a server S_j can be actively corrupted or crash-stop corrupted. Active corruption is handled as usual. To crash-stop corrupt S_j the adversary will input (CRASH-STOP-CORRUPT) to S_j. In response to this S_j, sends CRASHED to all other servers and stops giving any outputs and stops sending any messages. After this we say that S_j is crashed. The adversary might actively corrupt a crashed server. A server is called correct if it is not malicious nor crashed. We work with two thresholds $\mathsf{t}^{\text{MAL}}_{\text{OUT}}$ $\mathsf{t}^{\text{TERM}}_{\text{OUT}}$ which are values between and 0 and 1 that represent proportions of servers. We assume that at most a $\mathsf{t}^{\text{MAL}}_{\text{OUT}}$ proportion of servers are actively corrupted. We will allow any number of malicious users and we will allow any number of crashed servers. However, we

will only guarantee termination if less than a $\mathsf{t}_{\text{OUT}}^{\text{TERM}}$ proportion of servers are incorrect.

The ideal functionality is for n users $\mathsf{U}_1, \ldots, \mathsf{U}_n$ and m servers $\mathsf{S}_1, \ldots, \mathsf{S}_m$ and a function f.

Input If in some round all correct users U_i give an input x_i and in the same round all correct servers get an input `begin`, then set $x_i = \perp$ for all actively corrupted users. In the first round where a correct user or correct server gets an input not of the above form, do a total break down.

Compute During the next R_{OUT} rounds, output nothing. We call this the *computation period*.

Eval If during the computation period the adversary inputs `eval` or if R_{OUT} rounds have passed without such an input from the adversary, then compute $(z_1, \ldots, z_n) \leftarrow f(x_1, \ldots, x_n)$. After this we say that the *evaluation has taken place*. Now for all U_i which are passively or actively corrupted, output (i, z_i) to the adversary.

Replace inputs If the evaluation has not yet taken place and the adversary inputs $(\texttt{replace input}, i, x_i')$ and U_i is actively corrupted, then set $x_i \leftarrow x_i'$.

Replace outputs If the evaluation has taken place and outputs have not yet been delivered and the adversary inputs $(\texttt{replace output}, i)$ and there are more than $\mathsf{t}_{\text{OUT}}^{\text{TERM}}$ incorrect servers, then set $z_i = \perp$.

Output After $\mathsf{R}_{\text{OUT}} + 1$ rounds have passed output z_i to U_i. After this we say that *outputs have been delivered*.

Fig. 4. The ideal functionality $\mathcal{F}_{\text{OUT}}^{\mathsf{t}_{\text{OUT}}^{\text{TERM}}}$ corresponding to an outer protocol for f

Definition 1. *We say that* π_{OUT} *is a* $(\mathsf{t}_{\text{OUT}}^{\text{MAL}}, \mathsf{t}_{\text{OUT}}^{\text{TERM}})$-suitable outer protocol *if it UC realises the corresponding* $\mathcal{F}_{\text{OUT}}^{\mathsf{t}_{\text{OUT}}^{\text{TERM}}}$ *against a proportion* $\mathsf{t}_{\text{OUT}}^{\text{MAL}}$ *of adaptive, active corruptions and any number of adaptive crash-stop corruptions.*

Theorem 1. *There exists a suitable outer protocol* π *for all* $C \in \mathsf{NC}$ *with* $\text{OH} = \text{polylog}(n) \cdot \log(\text{size}(C))$.

Proof (sketch). We only sketch a proof of the theorem as the desired protocol can be built in a fairly straightforward manner from off-the-shelf techniques.

Starting from [DIK10] we get a protocol π for m servers and a circuit C which is perfectly secure against $m/4$ adaptive, active corruptions. We can extend this to the client-server setting by having each U_i secret share its input among $\mathsf{S}_1, \ldots, \mathsf{S}_m$ and then computing the function f' which first reconstructs the secret sharings, computes f, and outputs secret sharings of the results. We denote the resulting protocol by π_f'. It runs the protocol $\pi_{f'}$, i.e., the protocol π from [DIK10] for the function f'.

The secret-sharing scheme used for the inputs and outputs should have the following properties. First, that given m shares of which at most $\frac{1}{4}m$ are incorrect, one can efficiently reconstruct the message. Furthermore, the secret-sharing scheme should also have the property that given at most $m/4$ shares,

one gets no information on the secret. Finally, when secret sharing a message x, the secret-sharing scheme should produce a secret sharing of size $O(|x| + m)$ and it should be possible to share and reconstruct in time $O(|x| + m)$. The secret sharing scheme from [CDD+15] meets these criteria.

We now do a generic, white-box transformation of the protocol π into a protocol π'_f which can tolerate crash errors. Each server S_j will run exactly as in π except that it keeps a counter c_j which is initialized to 0 and which is increased whenever S_j sees a party sent CRASHED. There is a threshold $t = m/8$ and when $c_j \geq t$, server S_j will crash itself, i.e., it sends CRASHED to all parties and stops sending messages. If at the end of the computation of f', a server is not crashed, it sends its share of the output of U_i to U_i.

The intuition behind π'_f is that we try to keep the number malicious servers plus the number of crashed servers within the threshold $m/4$ of π. We will use $m/8$ of the budget for crashes and have $m/8$ left for tolerating some additional malicious corruptions. If we see too many crashed servers, then all servers will shut down the execution of π by self-crashes. We say that π was *shut down* if all correct servers did a self-crash.

We are going to reduce the security of π'_f to that of π by considering all parties which deviate from π'_f as actively corrupted. Notice that in π'_f there are three types of parties which deviate from the underlying protocol $\pi_{f'}$. 1) The servers S_j which are actively corrupted in π'_f. 2) The servers S_j which are crash-stop corrupted in π'_f. 3) The correct servers S_j which did a self-crash and hence stopped running $\pi_{f'}$. At any point in the execution, let d_i denote the number of servers which deviated from π and are of type i and let $d = d_1 + d_2 + d_3$. We are going to show that at any point before the shut-down point, it holds that $d < m/4$. This means that up to the shut-down point, we can perfectly emulate an attack on π'_f by an attack on $\pi_{f'}$ using $< m/4$ active corruption. This also holds after the shut-down point since all honest parties have self-crashed and therefore there is no more communication from honest parties to simulate.

What remains is to show that if $d \geq m/4$, then the shut-down point has been reached. Assume that $d \geq m/4$. If in a given round there are (d_1, d_2, d_3) deviators of the various types, then at the beginning of the next round all correct servers have seen $d_2 + d_3$ messages CRASHED as both crashed and self-crashed parties sent out CRASHED to all parties. Hence before the next round begins it will hold for all correct S_j that $c_j \geq d_2 + d_3 = d - d_1 \geq m/4 - d_1 \geq m/4 - m/8 = m/8 = t$. Hence the shut-down point has been reached.

We then show that if any party gets an output then all honest users have their inputs considered correctly and all honest parties who get an output get the correct output. If the shut-down point is reached, then clearly no party gets an output, so assume that the shut-down point was not reached. Then the attack can be emulated given $m/4$ active corruptions. This means that at most $m/4$ of the shares of the honest parties are missing or modified. Therefore each honest U_i will correctly reconstruct z_i.

This ends the proof that the protocol is secure. We now address when the protocol is guaranteed to terminate.

It is clear that as long as $d_1 + d_2 < t$, we will have that $d_3 = 0$ as all $c_j \leq d_1 + d_2$ until the first self-crash. This shows that as long as $d_1 + d_2 < t$ we will have $d < t = m/8$ and therefore we will have guaranteed termination of π'_f. Furthermore, if $d_1 + d_2 < m/8$ then at least $m - d \geq \frac{7}{8}m$ shares of the secret shared inputs are correct and at most $d_1 + d_2 \leq m/8$ are incorrect. Hence, all honest parties will have their inputs x_i reconstructed inside f'. It will similarly be guaranteed that each U_j receives at least $m - d \geq \frac{7}{8}m$ shares of the secret shared output and that at most $m/8$ of these are incorrect. Hence U_i can compute the correct z_i.

We then address the complexity of the functions. By the assumptions on the secret sharing scheme and on the size of f, we have that $|f'| = O(|f| + n \cdot m)$. Assuming that $m = O(n)$, this is of the form $|f'| = O(|f|) + \text{poly}(n)$, so for the sake of computing the overhead, we can assume that $|f'| = O(|f|)$. When $f \in \mathsf{NC}$, then the protocol from [DIK10] has $\text{OH} = \text{polylog}(n) \cdot \log(|f'|)$.

5 The Inner Protocol

The *inner protocol* π_{OUT} involves c parties $\mathsf{U}_1, \ldots, \mathsf{U}_c$. It must securely realize reactive secure computation, i.e., there are several stages of inputs and outputs and a secure state is kept between the stages. Each stage is computed via a transition function Tr_{IN}. We need that the round complexity of each stage is known before the protocol is run. The round complexity of stage Stage is denoted by $\mathsf{R}_{\text{IN}}(\mathsf{Stage})$.

Definition 2. *We say that π_{IN} is a* suitable inner protocol *for $(\mathsf{Tr}_{\text{IN}}, \mathsf{R}_{\text{IN}})$ if it UC realises $\mathcal{F}_{\text{IN}}^{\mathsf{Tr}_{\text{IN}}, \mathsf{R}_{\text{IN}}}$ against adaptive, active corruption of any number of parties.*

Theorem 2. *For for all c and poly-sized Tr_{IN} there exists R_{IN} and π_{IN} such that π_{IN} is a suitable inner protocol for $(\mathsf{Tr}_{\text{IN}}, \mathsf{R}_{\text{IN}})$ in the OT-hybrid model with statistical security and complexity $O(\text{poly}(c)|\mathsf{Tr}_{\text{IN}}|)$, where in the complexity the calls to the OT functionality are counted as the size of the inputs and outputs.*

Proof. One can use the protocol from [IPS08]. One can in particular note that once the circuit to be computed is fixed, [IPS08] has a fixed round complexity.

Theorem 3. *For all c and poly-sized Tr_{IN} there exists R_{IN} and π_{IN} such that π_{IN} is a suitable inner protocol for $(\mathsf{Tr}_{\text{IN}}, \mathsf{R}_{\text{IN}})$ in the CRS model with computational complexity $O(\text{poly}(c)|\mathsf{Tr}_{\text{IN}}|\lambda)$.*

Proof. Replace the ideal OTs in Thm. 2 by the adaptive secure OT from [GWZ09].

6 Combining the Inner Protocol and the Outer Protocol

In this section, we describe how to combine the inner and outer protocol into the protocol that we call the combined protocol. This is a new instance of a black-box protocol transformation defined by [IKP+16]. First, we will describe tools that

The ideal functionality is for c parties $\mathsf{P}_1, \ldots, \mathsf{P}_c$, transition function Tr_{IN} and round complexity function R_{IN}.

Init Initialize a stage counter $\mathsf{Stage} \leftarrow 1$ and initialize a state variable $\mathsf{State} \leftarrow$ INPUTTING.

Input If in some round all correct parties U_i give an input x_i and $\mathsf{State} =$ INPUTTING, then set $x_i = \perp$ for all actively corrupted parties. Set $\mathsf{State} \leftarrow$ COMPUTING. If in some round some honest party gives an input and ($\mathsf{State} \neq$ INPUTTING or some honest party does not give an input), then do a complete breakdown.

Compute During the $\mathsf{R}_{\text{IN}}(\mathsf{Stage})$ rounds which follow State being set to COMPUTING, output TICK to all parties.

Replace inputs If $\mathsf{State} =$ COMPUTING and the adversary inputs (REPLACE INPUT, i, x_i') and P_i is actively corrupted, then set $x_i \leftarrow x_i'$.

Eval If $\mathsf{State} =$ COMPUTING and the adversary inputs EVAL or if $\mathsf{R}_{\text{IN}}(\mathsf{Stage})$ rounds have passed since State was set to COMPUTING without such an input from the adversary, then compute $(\sigma_{\mathsf{Stage}+1}, y_1, \ldots, y_c) \leftarrow \mathsf{Tr}_{\text{IN}}(\sigma_{\mathsf{Stage}}, x_1, \ldots, x_c)$ and for all corrupted P_i, output (i, y_i) to the adversary. Set $\mathsf{State} \leftarrow$ EVALUATED.

Replace outputs If $\mathsf{State} =$ EVALUATED and the adversary inputs (REPLACE OUTPUT, i) and P_i is honest, then set $y_i = \perp$.

Output Exactly $\mathsf{R}_{\text{IN}}(\mathsf{Stage})$ rounds after State was set to COMPUTING, output y_i to P_i and let the adversary specify the order of delivery. Set $\mathsf{State} \leftarrow$ INPUTTING. If any honest P_i receives $\perp$, then do a crash (see below).

Crash Set $\mathsf{State} \leftarrow$ CRASH, output CRASH to all parties, ignore all future input, and in all future rounds output TICK to all parties.

Crashing If all honest parties input CRASH in the same round, then do a crash as above. If some corrupted party inputs CRASH then do a crash as above. If in some round, some honest party inputs CRASH and some honest party does not input CRASH, then do a complete breakdown.

Fig. 5. The ideal functionality $\mathcal{F}_{\text{IN}}$ for the inner protocol

we will need. The first tool is called an expander graph. The second tool called authentic secret sharing is a secret sharing scheme that allows an honest party that receives shares to detect tampering. Our third tool is called Authenticated One-Time Encryption which is an information-theoretic authenticated encryption scheme. It is analogous to the one-time pad. Finally, we will describe how to run the combined protocol. We will describe what to do when an emulated server crashes, how emulated servers can exchange keys with other parties even when its committee only has a single honest party and then how the servers can then use those keys to securely communicate. We will then describe our final protocol and prove that it has poly-log overhead and some termination guarantees.

6.1 More Tools

Threshold bipartite expander graph. A threshold bipartite expander graph is a bipartite graph with n left nodes and m right nodes which guarantees that that for any set of left nodes that has size greater or equal to αn, the

size of the neighborhood of that set is greater or equal to βm. Recall that given a graph $G = (V, E)$ and some subset $S \subseteq V$, the neighbourhood of S denoted by $N(S)$ is the set of nodes that are adjacent to a node in S, i.e., $N(S) := \{v \in V \mid \exists\, u \in S : (u, v) \in E\}$. As usual a bipartite graph is a graph $G = (L \cup R, E)$ where $L \cap R = \emptyset$, $N(L) \subseteq R$ and $N(R) \subseteq L$. The left (right) degree is the maximal degree of a node in L (R).

Definition 3. *A (n, m, α, β)-threshold bipartite expander is a bipartite graph $G = (L \cup R, E)$ with $|L| = n$, $|R| = m$ such that if $S \subseteq L$ and $|S| \geq \alpha n$ then $N(S) \geq \beta m$.*

We show that for all constant $0 < \alpha < 1$ and $0 < \beta < 1$ there exists $m = O(n)$ and an (n, m, α, β)-threshold bipartite expander where the left degree is $O(1)$ and the right degree is $O(1)$.

We describe a simple construction of a bipartite threshold expander graph. It is inspired by [SS96]. We will show that for any $\alpha > 0$, that there exists a degree d such that for every n, β there exists an explicit way of constructing graphs such that the resulting graph is (n, n, α, β)-threshold bipartite expander graph except with probability negligible in n. In addition, the degree of the graph is at most d and each right node has at least one edge. We denote the binary entropy function as $\mathbb{H}$.

The construction is rather simple. First, we sample at random a set of d permutations.

$$\Pi \leftarrow \{\pi_1, \ldots, \pi_d : [n] \to [n]\}$$

We denote $L = \{1, \ldots, n\}$ as the set of left edges and $R = \{n+1, \ldots, 2n\}$ as the set of right edges. We select the graph as follows:

$$E \leftarrow \bigcup_{\pi \in \Pi} \{(1, n + \pi(1)), \ldots, (n, n + \pi(n))\} \tag{1}$$

$$G \leftarrow (L \cup R, E) \tag{2}$$

Theorem 4. *For any $0 < \alpha, \beta < 1$, let $d = \left\lceil -\frac{\mathbb{H}(\alpha)+\mathbb{H}(\beta)}{\alpha \log \beta} \right\rceil + 1$ then for any $n \in N$ the previous construction results in a bipartite (n, n, α, β)-threshold expander except with probability smaller than $2^{\alpha n \log \beta}$*

We note that the number of left sets of size αn is equal to $\binom{n}{\alpha n}$. We note that the number of right sets of size $(1-\beta)n$ is equal to $\binom{n}{\beta n} = \binom{n}{(1-\beta)n}$.

We will now upper bound the probability that the neighborhood of αn left nodes does not intersect a set of $(1-\beta)n$ right nodes. We can see that for each permutation, for each element in the left set, the probability that the element is not mapped to an element in the right set is less than or equal to β. Therefore we have that the probability is upper bounded $\beta^{\alpha n d}$.

By the union bound, we know that the probability that there exists such sets is less than $\beta^{\alpha n d}\binom{n}{\alpha n}\binom{n}{\beta n}$. By applying Stirling's approximation, we get that this

probability is upper bounded by

$$2^{n\mathbb{H}(\alpha)+n\mathbb{H}(\beta)+\alpha nd\log(\beta)} = 2^{n(\mathbb{H}(\alpha)+\mathbb{H}(\beta)+\alpha d\log(\beta))}$$

Finally, by setting $d = \left\lceil -\frac{\mathbb{H}(\alpha)+\mathbb{H}(\beta)}{\alpha\log\beta} \right\rceil + 1$

$$2^{n(\mathbb{H}(\alpha)+\mathbb{H}(\beta)+\alpha d\log(\beta))} \leq 2^{\alpha n\log\beta}$$

Lemma 1. *For the construction above, the degree of the graph is at most d.*

This follows since there are d permutations in Π and each node gains at most one edge per permutation.

Lemma 2. *For the construction above, each right node has at least one edge.*

This follows since each permutation assigns each right node to a left node.

Authentic Secret Sharing. Let $\mathbb{F}$ be a finite field and n be an integer. A secret sharing scheme consists of two algorithms share and rec. For every $s \in \mathbb{F}$, $\mathsf{share}(s)$ outputs a randomized set of shares $(s_1, \ldots, s_n)$. We use $\mathsf{share}(s)$ to denote the distribution on $(s_1, \ldots, s_n)$ when the input is s. The algorithm rec takes as input $(s'_1, \ldots, s'_n)$ and gives an output in $\mathbb{F} \cup \{\perp\}$ where $\perp$ signals error.

For any $i \in [n]$ we let $(s_1, \ldots, s_n)_{-i} = (s_1, \ldots, s_{i-1}, s_{i+1}, \ldots, s_n)$. For any $(s_1, \ldots, s_n)_{-i}$ and any s we let $((s_1, \ldots, s_n)_{-i}, s) = (s_1, \ldots, s_{i-1}, s, s_{i+1}, \ldots, s_n)$. For all i and all $s \in \mathbb{F}$ and all unbounded adversaries A taking as input $(s_1, \ldots, s_n)_{-i}$ and giving outputs $(s'_1, \ldots, s'_n)_{-i}$ consider the game where we sample $(s_1, \ldots, s_n) \leftarrow \mathsf{share}(s)$ and compute $s' = \mathsf{rec}(A((s_1, \ldots, s_n)_{-i}), s_i)$. Note that it might be the case that $s' = \perp$. We use $A_{-i}(s)$ to denote the distribution of s', i.e., the result of reconstructing with the $n-1$ possibly modified shares. Let $\delta(\perp) = \perp$ and $\delta(x) = \top$ for $x \neq \perp$. We use $\hat{A}_{-i}(s)$ to denote $(\delta(s'), (s_1, \ldots, s_n)_{-i})$, i.e., the shares seen by the adversary plus the information whether reconstructing with the wrong shares gave an error or not.

Definition 4 (authentic secret sharing). *Let $(\mathsf{share}, \mathsf{rec})$ be a secret sharing scheme. We call $(\mathsf{share}, \mathsf{rec})$ an authentic secret sharing scheme if the following conditions hold.*

Reconstruction *For all $s \in \mathbb{F}$ it holds that $\Pr[\mathsf{rec}(\mathsf{share}(s)) = s] = 1$.*
Sound *For all $s \in \mathbb{F}$ and all $i \in [n]$ and all unbounded adversaries A it holds that $\Pr[A_{-i}(s) \in \{s, \perp\}] = 1$.*
Privacy *For all $s, \bar{s} \in \mathbb{F}$ and all $i \in [n]$ and all unbounded adversaries A it holds that $\hat{A}_{-i}(s)$ and $\hat{A}_{-i}(\bar{s})$ are statistically close.*

Authenticated One-Time Encryption. An Authenticated One-Time Encryption scheme is given by a key space, encryption algorithm and decryption algorithm $(\mathcal{K}, \mathsf{Enc}, \mathsf{Dec})$. For each message length m and value λ of the security parameter we have a key space $\mathcal{K}_{m,\lambda}$. Given $K \in \mathcal{K}_{m,\lambda}$, λ and message $x \in \{0,1\}^m$ the encryption algorithm outputs a ciphertext $A = \mathsf{Enc}_{K,\lambda}(x)$. Given $K \in \mathcal{K}_{m,\lambda}$, λ, m and ciphertext A the decryption algorithm outputs message $x = \mathsf{Dec}_{K,\lambda,m}(A)$.

Correctness For all m and all $x \in \{0,1\}^m$ it holds with probability 1 for a random key $K \leftarrow \mathcal{K}_{m,\lambda}$ that $\mathsf{Dec}_{K,\lambda,m}(\mathsf{Enc}_{K,\lambda}(x)) = x$.

Security Let $\mathcal{A}$ be a computationally unbounded algorithm. Input λ to $\mathcal{A}$ and run it to get m and $x_0, x_1 \in \{0,1\}^m$. Sample a uniformly random bit $b \leftarrow \{0,1\}$. Sample $K \leftarrow \mathcal{K}_{m,\lambda}$ and $A \leftarrow \mathsf{Enc}_{K,\lambda}(x_b)$. Let $\mathcal{O}_{m,K,A}(B)$ be the oracle which on input $B \neq A$ returns $\mathsf{Dec}_{K,\lambda,m}(B)$. Compute $g \leftarrow \mathcal{A}^{\mathcal{O}_{m,K,A}(B)}(A)$ for $g \in \{0,1\}$. The advantage of $\mathcal{A}$ is given by $\mathsf{Adv}_{\mathcal{A}}(\lambda) = |\Pr[g = b] - \frac{1}{2}|$. We say that $(\mathcal{K}, \mathsf{Enc}, \mathsf{Dec})$ is secure if $\mathsf{Adv}_{\mathcal{A}}(\lambda) \in \mathsf{negl}(\lambda)$ for all $\mathcal{A}$ which makes at most a polynomial number of queries to its oracle.

Authenticity Let $\mathcal{A}$ be a computationally unbounded algorithm. Input λ to $\mathcal{A}$ and run it to get m. Sample $K \leftarrow \mathcal{K}_{m,\lambda}$. Let $\mathcal{O}_{m,K}(B)$ be the oracle which on input B returns $\mathsf{Dec}_{K,\lambda,m}(B)$. Compute $c \leftarrow \mathcal{A}^{\mathcal{O}_{m,K,A}(B)}()$. We say that $(\mathcal{K}, \mathsf{Enc}, \mathsf{Dec})$ has authenticity if $\Pr[\mathsf{Dec}_{K,\lambda,m}(c) \neq \perp] \in \mathsf{negl}(\lambda)$ for all $\mathcal{A}$ which makes at most a polynomial number of queries to its oracle.

We say that an Authenticated One-Time Encryption scheme has overhead $O(1)$ if it holds for all messages x and all $K \in \mathcal{K}_{|x|,\lambda}$ that $|K| + |\mathsf{Enc}_K(x)| \in O(|x| + \mathrm{poly}(\lambda))$ for a polynomial independent of $|x|$ and if we can encrypt and decrypt in time $O(|K| + |\mathsf{Enc}_K(x)| + |x|)$. This means that for large enough x we have that $|K| + |\mathsf{Enc}_K(x)| \in O(|x|)$ and that we can encrypt and decrypt in time $O(|x|)$. We can construct such a scheme off-the-shelf. Let MAC be an information-theoretic MAC which can handle message of length $O(\lambda)$ using keys of length $O(\lambda)$ and which can be computed in time $\mathrm{poly}(\lambda)$. Such a scheme is described for instance in [WC81]. Let $\mathcal{H}$ be a family of almost universal hash-functions which can be computed in linear time, see for instance [IKOS08]. For messages of length m, the key for the encryption scheme will consist of (L, H, P), where L is a random key for the MAC, $H \leftarrow \mathcal{H}$ is a random hash function from the family and P is uniformly random in $\{0,1\}^m$. To encrypt, compute $C = x \oplus P$, $M = H(C)$ and $A = \mathsf{MAC}_L(M)$ and send (C, A). To decrypt, if $|C| \neq m$, output $\perp$. Otherwise, compute $M = H(C)$ and $A' = \mathsf{MAC}_L(M)$. If $A' \neq A$, output $\perp$. Otherwise, output $C \oplus P$. The complexity is as claimed and the security follows from [WC81].

6.2 The Combined Protocol

For the combined protocol we have n parties $\mathsf{P}_1, \ldots, \mathsf{P}_n$ of which at most $n \cdot \mathsf{t}_{\mathrm{MAL,COMB}}$ are corrupted. We want to compute a function f. The parties are going to run one execution of the outer protocol to compute f. In the outer protocol we have n users $\mathsf{U}_1, \ldots, \mathsf{U}_n$ and m servers $\mathsf{S}_1, \ldots, \mathsf{S}_m$ of which we need that at most $\mathsf{t}_{\mathrm{MAL,OUT}} \cdot m$ are corrupted. Party P_i is going to run the code of U_i. Each server S_j is going to be emulated by a small subset of the parties. The inner protocol will be used to emulate servers. We set $\alpha = 1 - \mathsf{t}_{\mathrm{MAL,COMB}}$ and set $\beta = 1 - \mathsf{t}_{\mathrm{MAL,OUT}}$. We use a (n, m, α, β)-threshold expander graph $G = (V, E)$ to form the committees. For $j = 1, \ldots, m$ we let

$$\mathcal{C}_j = \{\mathsf{P}_i \mid (i, j) \in V\}$$

We call $\mathcal{C}_j$ *committee* j. Using the graph from Section 6.1, the size of committees is constant and except with negligible probability all sets of αn parties have members in at least βm committees.

We will present our result in a hybrid model with ideal functionalities for the inner protocol. We call this the *inner-hybrid model*. For each $i = 1, \ldots, m$, we are going to have an ideal functionality $\mathcal{F}_j$ of the form given in Fig. 5 with $c = |\mathcal{C}_j|$ and the parties being $\mathcal{C}_j$. We call $\mathcal{F}_j$ virtual server j and we specify later the behaviour of $\mathcal{F}_j$.

We set up some notation. Let $\mathcal{P} = \{\mathsf{P}_1, \ldots, \mathsf{P}_n\}$. At any point in the execution, $\mathcal{P}_{\text{HONEST}} \subset \mathcal{P}$ denotes the set of parties which are honest in the combined protocol and we let $\mathcal{P}_{\text{MAL}}$ be the set of maliciously corrupted parties..

We use $\mathcal{S} = \{1, \ldots, m\}$ to denote the identities of the virtual servers. We define three disjoint subsets as follows

$$\begin{aligned}
\mathcal{S}_{\text{HONEST}} &= \{j \in \mathcal{S} \mid \mathcal{C}_j \subseteq \mathcal{P}_{\text{HONEST}}\} \\
\mathcal{S}_{\text{MAL}} &= \{j \in \mathcal{S} \mid \mathcal{C}_j \subseteq \mathcal{P}_{\text{MAL}}\} \\
\mathcal{S}_{\text{CRASHABLE}} &= \mathcal{S} \setminus (\mathcal{S}_{\text{HONEST}} \cup \mathcal{S}_{\text{MAL}})
\end{aligned}$$

If $j \in \mathcal{S}_{\text{HONEST}}$, then all parties in committee j are honest. Therefore, $\mathcal{F}_j$ is secure and also has guaranteed output delivery. This will correspond to S_j being secure in the outer protocol. If $j \in \mathcal{S}_{\text{MAL}}$, then all parties in committee j are malicious. Therefore, $\mathcal{F}_j$ provides no security. This will correspond to S_j being malicious in the outer protocol. If $j \in \mathcal{S}_{\text{CRASHABLE}}$, then at least one party in committee j is honest and at least one party is malicious. Therefore $\mathcal{F}_j$ provides privacy and correctness, but some or all honest parties might not learn the output. If at some point a party in committee j does not get an output, then $\mathcal{F}_j$ will abort. This corresponds to S_j crashing in the outer protocol. We will let the honest party in $\mathcal{C}_j$ which received output $\perp$ inform all other parties that $\mathcal{F}_j$ has aborted. Overall, this will correspond to a crash-stop corruption of S_j in the outer protocol.

By the way we have set the parameters of the threshold expander graph it follows that if $|\mathcal{P}_{\text{MAL}}| \leq n \cdot \mathsf{t}_{\text{MAL,COMB}}$ then $|\mathcal{S}_{\text{MAL}}| \leq m \cdot \mathsf{t}_{\text{MAL,OUT}}$. We have therefore almost perfectly emulated the entities $\mathsf{U}_1, \ldots, \mathsf{U}_n, \mathsf{S}_1, \ldots, \mathsf{S}_m$ of the outer protocol with the needed adversary structure as long as $|\mathcal{P}_{\text{MAL}}| \leq n \cdot \mathsf{t}_{\text{MAL,COMB}}$. The only significant difference between $\mathsf{U}_1, \ldots, \mathsf{U}_n, \mathsf{S}_1, \ldots, \mathsf{S}_m$ and $\mathsf{P}_1, \ldots, \mathsf{P}_n$, $\mathcal{F}_1, \ldots, \mathcal{F}_m$ is the fact that in the outer protocol, the entities $\mathsf{U}_1, \ldots, \mathsf{U}_n, \mathsf{S}_1, \ldots, \mathsf{S}_m$ can send private messages to each other, whereas most of the entities $\mathsf{P}_1, \ldots, \mathsf{P}_n$, $\mathcal{F}_1, \ldots, \mathcal{F}_m$ cannot send private messages to each other. This is going to give us the so-called *secret communication problems* when we emulate the protocol in Fig. 3.

Distribution of input shares To give inputs, each U_i sends a share to S_j. In the emulated outer protocol this corresponds to P_i inputting a message to $\mathcal{F}_j$. If $\mathsf{P}_i \notin \mathcal{C}_j$, this is not allowed.

Server Communication As part of the evaluation, each S_j sends a message to S_k. In the emulated outer protocol, this corresponds to $\mathcal{F}_j$ sending a message to $\mathcal{F}_k$. This is not allowed since ideal functionalities cannot communicate.

Distribution of output shares To give outputs, S_j sends a share to U_i. In the emulated outer protocol this corresponds to $\mathcal{F}_j$ outputting a message to P_i. If $\mathsf{P}_i \notin \mathcal{C}_j$, this is not allowed.

Another problem is that in the outer protocol, if a server S_i crashes, it will by definition notify the other servers. However, now the code of S_i is "trapped" inside $\mathcal{F}_i$ so S_i must notify S_j via the parties $\mathcal{C}_i$ and $\mathcal{C}_j$ and there might be corrupted parties among $\mathcal{C}_i \cup \mathcal{C}_j$. We call this the *abort propagation problem*. Handling of the the abort propagation problem is described in Fig. 6.

The parties run a copy of the outer protocol. The code and state for S_i will be inside $\mathcal{F}_i$. If S_i crashes inside $\mathcal{F}_i$, then $\mathcal{F}_i$ will also crash, i.e., it will enter a state with $\mathsf{State} = \text{CRASH}$ and will output CRASH to all parties. As we will describe later, there will be other events which can trigger $\mathcal{F}_i$ to crash. In all those cases, we want that all other $\mathcal{F}_j$ learn that $\mathcal{F}_i$ has crashed. This is handled as follows:

Define Crashing We say that $\mathcal{F}_i$ is crashed if it enters a state where $\mathsf{State} = \text{CRASH}$. If this happens, it outputs CRASH to all $\mathsf{P} \in \mathcal{C}_i$

Crash Alerting If at any point during the execution a party $\mathsf{P} \in \mathcal{C}_i$ sees $\mathcal{F}_i$ output CRASH, then for $j = 1 \dots m$, P broadcasts (CRASH, i) to all parties in $\mathcal{C}_i \cup \mathcal{C}_j$.

Crash Recording A crash alert is received as follows.

- If at any point during the execution, a party $\mathsf{P}_k \in \mathcal{C}_j$ receives a broadcast (CRASH, i) from a party in $\mathcal{C}_i$ to $\mathcal{C}_i \cup \mathcal{C}_j$ then P_k inputs (CRASH, i) to $\mathcal{F}_j$.
- If $\mathcal{F}_j$ receives input (CRASH, i) from all parties in $\mathcal{C}_j$ then it inputs (CRASH, i) to S_j as if coming from S_i in a run of the outer protocol. If $\mathcal{F}_j$ receives input (CRASH, i) from all honest parties but some corrupted party did not give input (CRASH, i) then $\mathcal{F}_j$ does a crash. [a] If $\mathcal{F}_j$ receives input (CRASH, i) from some honest parties but some other honest party did not input (CRASH, i), then $\mathcal{F}_j$ does a complete break down.

[a] Recall that in the UC model ideal functionalities know which parties are corrupt.

Fig. 6. Crash Handling Part of π_{COMB}

We handle all three secret communication problems by letting the entities that need to communicate share a secret key which is used to encrypt the given message using an Authenticated One-Time Encryption scheme. Then the authenticated ciphertext c can be sent in cleartext between the two involved entities. This solves the problem as an ideal functionality $\mathcal{F}_j$ for instance can output c to all members of $\mathcal{C}_j$ which can then all send c to all the members of $\mathcal{C}_k$ who will input it to $\mathcal{F}_k$. Our way to solve the secret communication problems is significantly more complicated than the approach in [IPS08] and other player emulation protocols. The reason is that previous techniques incur an overhead of at least n. For instance, in [IPS08] each message is secret shared among all parties, which means that messages will become a factor n longer. We need constant overhead. This is not an issue for [IPS08] as they consider n to be a constant. Also, the technique in [IPS08] do not guarantee termination if there is just one corrupted party.

To have servers and players share keys, we use a subprotocol to do so. This introduces another problem. It is possible that all committees contain at least one corrupt player. When a player is generating a key with a committee, a problem may arise. This can occur because either the player is corrupt or the committee contains at least one dishonest member. It is imperative that a server with at least one honest member must get the key from each honest user or abort. Otherwise, the corrupt parties can prevent honest parties from giving inputs. We employ player elimination techniques [HMP00] to solve this problem.

Distribution of input shares When U_i needs to send a message m to S_j and they are both honest there will exist a random secret key K_j^i which is held by P_i and which is inside $\mathcal{F}_j$. Then P_i computes $c = \mathsf{Enc}_{K_j^i}(m)$ and sends it to each $\mathsf{P}_k \in \mathcal{C}_j$. Then each P_k inputs c to $\mathcal{F}_j$. Let c_k denote the value input by P_k. Then the virtual server $\mathcal{F}_j$ computes $m_k = \mathsf{Dec}_{K_j^i}(c)$. If $|\{m_k\}_{k\in\mathcal{C}_j} \setminus \{\perp\}| = 1$, then let m be the unique value in $\{m_k\}_{k\in\mathcal{C}_j} \setminus \{\perp\}$. Otherwise, let $m = \perp$. Notice that if P_i is honest and there is at least one honest $\mathsf{P}_k \in \mathcal{C}_j$, then the correct ciphertext will be input to the virtual server and therefore $m \in \{m_k\}_{k\in\mathcal{C}_j}$. Furthermore, no corrupted committee member can input another valid ciphertext. In particular, when the correct message is not received, either P_i is corrupted or $j \in \mathcal{S}_{\text{MAL}}$.

Server Communication When S_i needs to send a message m to S_j and they are both honest there will exist a random secret key $K_{i,j}$ which is inside $\mathcal{F}_i$ and $\mathcal{F}_j$. Then $\mathcal{F}_i$ computes $c = \mathsf{Enc}_{K_{i,j}}(m)$ and outputs it to all $\mathsf{P}_k \in \mathcal{F}_i$. Then all $\mathsf{P}_k \in \mathcal{F}_i$ sends c to all $\mathsf{P}_l \in \mathcal{C}_j$ and they all input all the ciphertexts they received. The virtual server decrypts all ciphertexts and sets m to be the unique message different from $\perp$ if it exists and $\perp$ otherwise. If $\mathcal{C}_i$ crashes the message is also set to $\perp$. Assume that $\mathcal{C}_i$ is not crashed and $\mathcal{C}_j$ is not corrupted. Then all the honest parties $\mathsf{P}_k \in \mathcal{C}_i$ sent the correct ciphertext and no party knows the secret key, so the only correct ciphertext input to the virtual server is the correct one. Hence m arrives correctly. Assume then that that $\mathcal{C}_i$ is crashed and $\mathcal{C}_j$ is not corrupt. Then the message is set to $\perp$ as it should be.

Distribution of output shares When S_j needs to send a message m to U_i and they are both honest there will exist a random secret key K_j^i which is held by P_i and which is inside $\mathcal{F}_j$. Then $\mathcal{F}_j$ computes $c = \mathsf{Enc}_{K_j^i}(m)$ and outputs it to all parties $\mathsf{P}_k \in \mathcal{C}_j$, who all forward it to P_i. The party decrypts all ciphertexts and sets m as above. Assume that $\mathcal{C}_j$ is not crashed or corrupted and that P_i is honest. Then all the honest parties $\mathsf{P}_k \in \mathcal{C}_j$ sent the correct ciphertext and no party knows the secret key, so the only correct ciphertext sent to P_i is the correct one. Hence m arrives correctly. Assume then that $\mathcal{C}_j$ is crashed and that P_i is honest. Then the message is set to $\perp$ as it should be. Assume that $\mathcal{C}_j$ is corrupted and that P_i is honest. Then any message might arrive, but this is allowed as it correspond to S_j being corrupted. Similarly if P_i is corrupted.

It should be clear that the above emulation of the outer protocol should work as long as the security of the encryption scheme is not broken. We are, however, still left with the problem of getting the keys in place. We describe the key distribution protocols below.

Key Generation We use a key generation protocol $\mathsf{KeyGeneration}_{i,j}^{U\leftrightarrow S}$ run between P_i and the parties in $\mathcal{C}_j$. It is invoked by all parties in the same round by giving the input (KEY, kid, m), where kid is a fresh key id and m is the length of the message that will later be encrypted. It proceeds as follows:

1. Let $C = \mathcal{C}_j$;
2. If $C = \emptyset$, then P_i terminates the protocol. Otherwise it proceeds as follows.
3. P_i samples $K_j^i \leftarrow \mathcal{K}_{m,\lambda}$;
4. P_i samples an authenticated secret sharing $\{K_{j,k}^i\}_{k\in C} \leftarrow \mathsf{share}(K_j^i)$ among the parties C.
5. For $k \in C$, party P_i sends $K_{j,k}^i$ to P_k and P_k inputs $K_{j,k}^i$ to $\mathcal{F}_j$.
6. Let $\hat{K}_{j,k}^i$ be the value of $K_{j,k}^i$ received by $\mathcal{F}_j$;
7. $\mathcal{F}_j$ computes $\hat{K}_j^i \leftarrow \mathsf{rec}(\{\hat{K}_{j,k}^i\}_{k\in C})$;
8. If $\hat{K}_j^i \neq \perp$, then $\mathcal{F}_j$ outputs SUCCESS to all parties in $\mathcal{C}_j$ and stores $(kid, m, \hat{K}_j^i)$. All parties terminate the protocol.
9. If $\hat{K}_j^i = \perp$, then $\mathcal{F}_j$ outputs $\{\hat{K}_{j,k}^i\}_{k\in C}$ to all parties in $\mathcal{C}_j$;
10. Each party $\mathsf{P}_k \in \mathcal{C}_j$ broadcasts $\{\hat{K}_{j,k}^i\}_{k\in C}$ to $\mathcal{C}_j \cup \{\mathsf{P}_i\}$.
11. If $\mathsf{P}_k \in \mathcal{C}_j$ sees that not all parties from $\mathcal{C}_j$ broadcast the same values, then P_k inputs CRASH to $\mathcal{F}_j$ and waits for two rounds to let the crash propagate.
12. If $\mathcal{F}_j$ crashed during the above, then each $\mathsf{P}_k \in \mathcal{C}_j$ broadcasts CRASH to $\mathcal{C}_j \cup \{\mathsf{P}_i\}$;
13. If $\mathcal{F}_j$ did not crash during the above but still some $\mathsf{P}_k \in \mathcal{C}_j$ broadcast CRASH, then all honest $\mathsf{P}_k \in \mathcal{C}_j$ inputs CRASH to $\mathcal{F}_j$;
14. If P_i did not see any $\mathsf{P}_k \in \mathcal{C}_j$ broadcast CRASH, then P_i received $\{\hat{K}_{j,k}^i\}_{k\in C}$ from all parties. It then finds k such that $\hat{K}_{j,k}^i \neq K_{j,k}^i$ and broadcasts k to $\mathcal{C}_j$;
15. If P_i does not broadcast $k \in C$ then P_i is corrupt and the protocol terminates with the output being some dummy key.
16. If P_i does broadcast $k \in C$, then each $\mathsf{P} \in \mathcal{C}_j$ sets $C \leftarrow C \setminus \{k\}$ and inputs k to $\mathcal{F}_j$;
17. Unless all $\mathsf{P} \in \mathcal{C}_j$ input the same k, $\mathcal{F}_j$ will crash. Otherwise it sets $C \leftarrow C \setminus \{k\}$;
18. P_i and all parties in $\mathcal{C}_j$ go to Step 2.

Fig. 7. User-Server Key-generation Communication Part of π_{COMB}

Generating Input Keys. The basic idea behind generating the key K_i^j is to let P_i generate it and distribute an authentic secret sharing $\{K_{i,k}^j\}_{k\in\mathcal{C}_j} \leftarrow \mathsf{share}(K_{i,k}^j)$ among the parties $\mathsf{P}_k \in \mathcal{C}_j$ who will input the shares to $\mathcal{F}_j$ which will in turn compute $K_i^j \leftarrow \mathsf{rec}(\{K_{i,k}^j\}_{k\in\mathcal{C}_j})$ and store it for later use. The main problem arises when the reconstruction fails. This will prevent P_i from giving (secure)

User-Server communication We use a communication protocol $\mathsf{Send}_{i,j}^{U\to S}$ run between P_i and the parties in $\mathcal{C}_j$. It is invoked by all parties in the same round by giving the input (SEND, kid), where some (kid, K, m) is stored. In addition P_i inputs $x \in \{0,1\}^m$. It proceeds as follows.

1. Delete (kid, K, m).
2. If $x \in \{0,1\}^m$ then P_i computes $c = \mathsf{Enc}_{\lambda,K}(x)$ and sends c to all parties $\mathsf{P}_k \in \mathcal{C}_j$.
3. Each P_k inputs the received c to $\mathcal{F}_j$. Let c_k be the value received from P_k.
4. For $k \in \mathcal{C}_j$ the ideal functionality computes $x_k = \mathsf{Dec}_{\lambda,m,K}(c_k)$.
5. If there exists $x \in \{0,1\}^m$ such that $\{x\} = \{x_k\}_{k\in\mathcal{C}_j} \setminus \{\perp\}$, then store $(\text{MESSAGE}, kid, x)$. Otherwise, store $(\text{MESSAGE}, kid, \perp)$.

Server-User communication We use a communication protocol $\mathsf{Send}_{j,i}^{S\to U}$ run between P_i and the parties in $\mathcal{C}_j$. It is invoked by all parties in the same round by giving the input (SEND, kid), where some (kid, K, m) is stored. It proceeds as follows.

1. F_j deletes (kid, K, m).
2. F_j computes $c = \mathsf{Enc}_{\lambda,K}(x)$ and sends c to all parties $\mathsf{P}_k \in \mathcal{C}_j$.
3. Each $\mathsf{P}_k \in \mathcal{C}_j$, awaits c from $\mathcal{F}_j$.
4. Each $\mathsf{P}_k \in \mathcal{C}_j$, sends c to P_i. Let c_k be the value received from P_k.
5. For $k \in \mathcal{C}_j$, P_i computes $x_k = \mathsf{Dec}_{\lambda,m,K}(c_k)$.
6. If there exists $x \in \{0,1\}^m$ such that $\{x\} = \{x_k\}_{k\in\mathcal{C}_j} \setminus \{\perp\}$, then store $(\text{MESSAGE}, kid, x)$. Otherwise, store $(\text{MESSAGE}, kid, \perp)$.

Fig. 8. User-Server Communication

inputs to $\mathcal{F}_j$. Unfortunately the error can arise either due to P_i being corrupted or some $\mathsf{P}_k \in \mathcal{C}_j$ being corrupted. In the later case $\mathcal{C}_j$ is crashable but might not be crashed, in which case P_i must be able to give secure inputs, as an honest U_i can give secure inputs to an honest S_j in the outer protocol.

We describe how to handle the case when reconstruction fails. First $\mathcal{F}_j$ will output to all parties in $\mathcal{C}_j$ all the shares $\{\hat{K}_{i,k}^j\}_{k\in\mathcal{C}_j}$ that was input. If this does not crash $\mathcal{C}_j$ then all parties $\mathsf{P}_k \in \mathcal{C}_j$ will broadcast $\{\hat{K}_{i,k}^j\}_{k\in\mathcal{C}_j}$ to $\mathcal{C}_j \cup \{\mathsf{P}_i\}$. If all parties $\mathsf{P}_k \in \mathcal{C}_j$ do not broadcast the same values, then all honest parties $\mathsf{P}_k \in \mathcal{C}_j$ will crash $\mathcal{C}_j$ by sending CRASH to all parties and $\mathcal{F}_j$. Otherwise, P_i will identify the indices k such that $\hat{K}_{i,k}^j \neq K_{i,k}^j$ and will broadcast the indices to $\mathcal{C}_j \cup \{\mathsf{P}_i\}$. If P_i does not do so, then P_i is corrupted and the parties in $\mathcal{C}_j$ will ignore all future messages from P_i. If parties were excluded, then the above procedure is repeated, but now P_i secret shares only among the committee members that were not excluded. Notice that only corrupted parties are excluded. Therefore, if $\mathcal{C}_j$ is not corrupted, the procedure will terminate before all committee members were excluded, at which point K_i^j was added to $\mathcal{F}_j$. If eventually all committee members were excluded, P_i will consider $\mathcal{C}_j$ corrupted.

Generating Committee Keys. The basic idea behind generating $K_{i,j}$ is to let $\mathcal{F}_i$ generate $K_{i,j}$ and sample an authentic secret sharing $\{K_{i,j,k}\}_{k\in\mathcal{C}_i} \leftarrow \mathsf{share}(K_{i,j})$ and output $K_{i,j,k}$ to P_k. Then P_k inputs $K_{i,j,k}$ to $\mathcal{C}_j$ using the method

Key Generation We use a key generation protocol $\mathsf{KeyGeneration}_{i,j}^{S\leftrightarrow S}$ run between the parties in $\mathcal{C}_i \cup \mathcal{C}_j$. It is invoked by all parties in the same round by giving the input (KEY, kid, m), where kid is a fresh key id and m is the length of the message that will later be encrypted. It proceeds as follows:

1. All parties in $\mathcal{C}_i \cup \mathcal{C}_j$ set $C = \mathcal{C}_i$;
2. If $C = \emptyset$, then terminate and use a dummy key. Otherwise proceed as follows.
3. $\mathcal{F}_i$ samples $K_{i,j} \leftarrow \mathcal{K}_{m,\lambda}$;
4. $\mathcal{F}_i$ samples an authenticated secret sharing $\{K_{i,j,k}\}_{k\in C} \leftarrow \mathsf{share}(K_j^i)$ among the parties C.
5. For $k \in C$, the functionality $\mathcal{F}_i$ outputs $K_{i,j,k}$ to P_k.
6. For $k \in C$, party P_k uses the code in Fig. 8 to send $K_{i,j,k}$ to $\mathcal{F}_j$.
7. Let $\hat{K}_{i,j,k}$ be the value of $K_{i,j,k}$ received by $\mathcal{F}_j$ (if the transmission fails, then $\hat{K}_{i,j,k} = \perp$ which will trigger a reconstruction error which is handled below);
8. $\mathcal{F}_j$ computes $\hat{K}_{i,j} \leftarrow \mathsf{rec}(\{\hat{K}_{i,j,k}\}_{k\in C})$;
9. If $\hat{K}_{i,j} \neq \perp$, then $\mathcal{F}_j$ outputs SUCCESS to all parties in $\mathcal{C}_j$ and stores $(\text{KEY}, kid, i, j, m, \hat{K}_{i,j})$. All parties terminate the protocol.
10. If $\hat{K}_{i,j} = \perp$, then $\mathcal{F}_j$ outputs $\{\hat{K}_{i,j,k}\}_{k\in C}$ to all parties in $\mathcal{C}_j$.
11. Each party $\mathsf{P}_k \in \mathcal{C}_j$ broadcasts $\{\hat{K}_{j,k}^i\}_{k\in C}$ to $\mathcal{C}_i \cup \mathcal{C}_j$.
12. If $\mathsf{P}_k \in \mathcal{C}_j$ sees that not all parties from $\mathcal{C}_j$ broadcasts the same values, then P_k inputs CRASH to $\mathcal{F}_j$ and waits for two rounds to make the crash propagate. In this case no key is needed.
13. If $\mathcal{F}_i$ does not consider $\mathcal{F}_j$ crashed during the above, then all $\mathsf{P}_k \in \mathcal{C}_i$ inputs $\{\hat{K}_{j,k}^i\}_{k\in C}$ to $\mathcal{F}_i$. If they do not all input the same value, $\mathcal{F}_i$ will crash.
14. If $\mathcal{F}_i$ did not crash it will find k such that $K_{i,j,k} \neq \hat{K}_{i,j,k}$ and outputs k to all parties in $\mathcal{C}_i$. Following the usual patterns, they will all broadcast k to $\mathcal{C}_i \cup \mathcal{C}_j$, crash $\mathcal{F}_i$ if there is not agreement and otherwise let all parties input k to $\mathcal{F}_j$ which will crash if there is not agreement.
15. If neither $\mathcal{F}_i$ nor $\mathcal{F}_j$ is crashed, then set $C \leftarrow C \setminus \{k\}$ and go to Step 2.

Fig. 9. Server-Server Communication Part of π_{COMB} (Key Generation)

for when U_k gives input to $\mathcal{F}_j$. Recall that when U_k gives input to $\mathcal{F}_j$ it will succeed unless $\mathcal{F}_j$ crashes or $\mathcal{C}_j$ detects U_k as being corrupted. If $\mathcal{F}_j$ crashes there is no need to generate a key. If $\mathcal{C}_j$ detects P_k as corrupted, they all broadcast this to $\mathcal{C}_i \cup \mathcal{C}_j$ and P_k. Notice that if this happens, then either P_k is corrupted, and it is secure to excluded it, or $\mathcal{C}_j$ is corrupt, which corresponds to S_j being corrupt, and hence there is no reason to keep the key secret, so again it is secure to exclude P_k. Let $\mathcal{C}_i' \subseteq \mathcal{C}_i$ be the parties that were not excluded. If any parties were excluded, then $\mathcal{F}_i$ generates a new key $K_{i,j}$ and samples an authentic secret sharing $\{K_{i,j,k}\}_{k\in\mathcal{C}_i'} \leftarrow \mathsf{share}(K_{i,j})$ and outputs $K_{i,j,k}$ to P_k. The procedure is repeated until $\mathcal{C}_i' = \emptyset$ or $\mathcal{C}_i$ crashed or $\mathcal{C}_j$ crashed or in some attempt all keys $\{K_{i,j,k}\}_{k\in\mathcal{C}_i'}$ were successfully input to $\mathcal{F}_j$. In the three first cases, either $\mathcal{C}_i$ or $\mathcal{C}_j$ is corrupted and there is no need for a key. In the last case, $\mathcal{F}_j$ computes $K_{i,j} \leftarrow \mathsf{rec}(\{K_{i,j,k}\}_{k\in\mathcal{C}_i'})$. If $K_{i,j} \neq \perp$, then the key is the same as generated by $\mathcal{F}_i$ unless the security of the secret sharing scheme was broken. Assume then

Communication Defines a procedure $\mathsf{Send}_{i,j}^{S \to S}$. It is invoked by all parties in the same round by giving the input (SEND, kid), where some $(\text{KEY}, i, j, kid, K, m)$ is stored inside $\mathcal{F}_i$ and some $(\text{MESSAGE}, i, j, kid, x \in \{0,1\}^m)$ is stored inside $\mathcal{F}_i$. It proceeds as follows:

1. $\mathcal{F}_i$ deletes $(\text{KEY}, i, j, kid, K, m)$;
2. $\mathcal{F}_i$ computes $c = \mathsf{Enc}_{\lambda,K}(x)$ and outputs c to all parties $\mathsf{P}_k \in \mathcal{C}_i$.
3. Each $\mathsf{P}_k \in \mathcal{C}_i$ sends c to all parties $\mathsf{P}_l \in \mathcal{C}_j$. The parties P_l might receive conflicting values, in which case they keep them all.
4. For $l \in \mathcal{C}_j$ each P_l inputs all the values c received from parties $\mathsf{P}_k \in \mathcal{C}_i$ to $\mathcal{F}_j$. The functionality accepts at most $|\mathcal{C}_i|$ values from each party.
5. Let $\mathcal{A}$ denote the set of ciphertexts c received by $\mathcal{F}_j$. There might be up to $|\mathcal{C}_i| \cdot |\mathcal{C}_j|$ such values.
6. If there exist $x \in \{0,1\}^m$ such that $\{x\} = \{\mathsf{Dec}_{\lambda,K,m}(c)\}_{c \in \mathcal{A}} \setminus \{\perp\}$, then store $(\text{MESSAGE}, i, j, kid, x)$. Otherwise, store $(\text{MESSAGE}, i, j, kid, \perp)$.

Fig. 10. Server-Server Communication Part of π_{COMB} (Communication)

$K_{i,j} = \perp$. Since we are in a situation which might correspond to both S_i and S_j being honest (if for instance $\mathcal{C}_i$ and $\mathcal{C}_j$ are crashable but not crashed) we have to handle $K_{i,j} = \perp$ by trying again. When $K_{i,j} = \perp$ the virtual server $\mathcal{F}_j$ will output this to $\mathcal{C}_j$ along with $\{K_{i,j,k}\}_{k \in \mathcal{C}_i'}$ which will all broadcast the shares to $\mathcal{C}_i \cup \mathcal{C}_j$. If they do not all broadcast the same value, then the honest parties in $\mathcal{C}_j$ will crash $\mathcal{C}_j$ which is safe as there must be a corrupted party in $\mathcal{C}_j$. If they all broadcast the same value, denote this value by $\{K_{i,j,k}\}_{k \in \mathcal{C}_i'}$. Then all parties in $\mathcal{C}_i$ will give these values to $\mathcal{F}_i$. Again, if they do not all give the same values, the $\mathcal{F}_i$ will crash. Otherwise $\mathcal{F}_i$ will find the indices k for which the wrong shares arrived at $\mathcal{F}_j$. This only happens if P_k is corrupted, so it is not safe to remove P_k from the set of parties among which the secret sharing is done and try again. The code is given in Fig. 9 and Fig. 10.

Putting the Pieces Together. We now describe how to put the pieces together. The combined protocol is given in Fig. 11. Since the tools we use are information-theoretically secure, the information theoretic security of π_{COMB} is fairly straight forward to argue, using the arguments we gave above for the security of the individual sub-protocols.

Termination. To analyze termination, we use that each party is in at most $d = O(1)$ committees and that $n = O(m)$. Let $\delta = m/(8nd)$. If less than δn parties are corrupted, there will be at most $d\delta n = m/8$ committees which even contain a corrupted member. Therefore the total number of corrupted committees plus crashable committees will be at most $m/8$. Since the outer protocol is secure (including termination guarantee) against $m/8$ malicious corruptions, it follows that the combined protocol guarantees termination against δn malicious corruptions.

Formation For $j = 1, \ldots, m$ initialize $\mathcal{F}_j$ with committee $\mathcal{C}_j$ as defined above. The code of $\mathcal{F}_j$ is describe in the figures above . Below, we describe further behaviour of $\mathcal{F}_j$.

Crash Handling Start running the sub protocols in Fig. 6. If $\mathcal{F}_j$ learns that $\mathcal{F}_i$ is crashed as part of the crash handling, then this is added to the current state $y^r_{j,j}$ of S_j below as in a run of the outer protocol.

Key Generation For all U_i and S_j and messages x of length m and with id kid to be sent from U_i to S_j or S_j to U_i, run $\mathsf{KeyGeneration}^{U \leftrightarrow S}_{i,j}(kid, m)$. For all S_i and S_j and messages x of length m and with id kid to be sent from S_i to S_j in the outer protocol, run $\mathsf{KeyGeneration}^{S \leftrightarrow S}_{i,j}(kid, m)$. All parties wait a number of rounds which upper bounds the worst case running time of all the sub protocols to stay synchronized.

Computation Now emulate the outer protocol π_{OUT} as follows.

- **Inputs** For $i = 1, \ldots, n$ party P_i has input x_i.
- **Server initialization** For $j, k = 1, \ldots, m$ functionality $\mathcal{F}_j$ initializes S_j by letting $y^0_{k,j}$ be the empty string.
- **Generation of input shares** For $i = 1, \ldots, n$ user P_i samples $(x_{i,1}, \ldots, x_{i,m}) \leftarrow \mathsf{In}_{\text{OUT}}(x_i; t_i)$ for a random tape t_i.
- **Distribution of input shares** For $i = 1, \ldots, n$ and $j = 1, \ldots, m$ user P_i sends $x_{i,j}$ to server $\mathcal{F}_j$ using $\mathsf{Send}^{U \to S}_{i,j}$.
- **Embedding of input shares** For $j = 1, \ldots, m$ functionality $\mathcal{F}_j$ sets $y^0_{j,j} \leftarrow (x_{1,j}, \ldots, x_{n,j}, r_j)$ for a random tape r_j.
- **Evaluation rounds** For $r = 1, \ldots, \mathsf{R}_{\text{OUT}}$ round r is emulated as follows:
  - **Transition**
    For $j = 1, \ldots, m$ functionality $\mathcal{F}_j$ computes $(y^r_{j,1}, \ldots, y^r_{j,m}) \leftarrow \mathsf{Tr}_{\text{OUT}}(r, y^{r-1}_{1,j}, \ldots, y^{r-1}_{m,j})$.
  - **Communication**
    For $j, k = 1, \ldots, m$ functionality $\mathcal{F}_j$ sends $y^r_{j,k}$ to functionality $\mathcal{F}_k$ using $\mathsf{Send}^{S \to S}_{j,k}$
- **Generation of output shares** For $j = 1, \ldots, m$ functionality S_j computes $(z_{j,1}, \ldots, z_{j,n}) \leftarrow \mathsf{Tr}_{\text{OUT}}(\mathsf{R}_{\text{OUT}} + 1, y^{\mathsf{R}_{\text{OUT}}}_{1,j}, \ldots, y^{\mathsf{R}_{\text{OUT}}}_{m,j})$
- **Distribution of output shares** For $j = 1, \ldots, m$ and $i = 1, \ldots, n$ server S_j sends $z_{j,i}$ to party P_i using $\mathsf{Send}^{S \to U}_{j,i}$.
- **Output reconstruction** For $i = 1, \ldots, n$ party P_i computes $z_i \leftarrow \mathsf{Out}_{\text{OUT}}(z_{1,i}, \ldots, z_{n,i})$.

Fig. 11. The combined protocol π_{COMB}

Complexity. We now address the complexity of the combined protocol when run in inner-hybrid model. We count one computational step by some $\mathcal{F}_j$ as 1 towards the complexity. We count one computational step by some P_i as 1 towards the complexity. We count the sending of a message x by some P_i as $|x|$ towards the complexity. We count the broadcast of one bit to $\log(n)$ parties as $\text{polylog}(n)$. Notice that throughout the protocol, we ever only broadcast to sets of parties of constant size, as all committees have constant size. Let c denote the complexity of running the outer protocol as a plain protocol. We want to compute the complexity of running the combined protocol and show that it is of the form $O(c \cdot \text{polylog}(n)) + \text{poly}(n, \lambda)$. This would show that the overhead

of the outer protocol is $\mathrm{OH} = \mathrm{polylog}(n)$. The emulation of the computation of the outer protocol clearly introduces no other overhead than the abort handling, key generation and the encryption of messages. It is clear that crash handling sends at most $O(n^2)$ messages of constant size. This can be swallowed by the $\mathrm{poly}(n, \lambda)$ term. It is clear that one attempt of a key generation of a key of length k will have complexity $O(k)$, as secret sharing is done among a constant number of parties and secret sharing and reconstruction is linear and we broadcast a constant number of messages in one attempt. Since C initially has constant size and each attempt of generating a key sees the size of C go down by at least 1 and the procedure stops when $C = \emptyset$, it follows that key generation has complexity $O(k)$. By assumption the overall complexity of key generation and sending a message is therefore $O(k) + \mathrm{poly}(\lambda)$. There are in the order of $2n + m^2\mathsf{R}_{\mathrm{OUT}}$ messages, so the total complexity of sending the encrypted messages of total length M will be $O(M) + (2n + m^2) \cdot \mathrm{poly}(\lambda)$. The total length M of the messages is already counted as part of the complexity c and can therefore be swallowed by the $O(c \cdot \mathrm{polylog}(n))$ terms. The remaining $(2n + m^2) \cdot \mathrm{poly}(\lambda)$ can be swallowed by $\mathrm{poly}(n, \lambda)$ as $m = O(n)$.

Theorem 5. *For all $c \in [0, 1)$ there exists a protocol π for the inner-hybrid model for all $f \in \mathsf{NC}$ secure against malicious, adaptive corruption of up to cn parties and with termination guarantee against a non-zero constant fraction of corruptions with $\mathrm{OH} = \mathrm{polylog}(n) \cdot \log(\mathrm{size}(f))$.*

We can use the UC theorem to replace each $\mathcal{F}_j$ by a suitable inner protocol. Using the fact that all $\mathcal{C}_j$ have constant size along with Theorem 2, this gives a combined protocol for the model with OT and a broadcast between sets of parties of constant size with an overhead of $\mathrm{OH} = \mathrm{polylog}(n) \cdot \log(\mathrm{size}(f))$.

When we want to tolerate that more than half the parties are corrupted, there is no way to implement the broadcast from scratch. We can, however, weaken the assumption on broadcast to only having access to $\mathrm{poly}(n)$ broadcasts which are all performed prior to the protocol being run. They might even be performed prior to knowing f. The broadcasts can either be used to set up a public-key infrastructure and then rely on signatures. They can also be used to run the setup phase of the protocol from [PW92] which can then be used to implement an unbounded of number of broadcasts in the online phase.

The protocol from [PW92] has information theoretic security. To broadcast between c parties the protocol from [PW92] has complexity $\mathrm{poly}(c) \cdot \lambda$ to broadcast one bit. Since we broadcast $\mathrm{poly}(n, \lambda)$ bits among log-size sets this will all in all contribute with a complexity of $\mathrm{poly}(n, \lambda)$, which does not affect the overhead.

Corollary 1. *For all $c \in [0, 1)$ there exists an information-theoretically secure protocol π secure against adaptive, malicious corruption of up to cn parties and with termination guarantee against a non-zero constant fraction of corruptions for the hybrid model with initial broadcast and oblivious transfer for all $f \in \mathsf{NC}$ with $\mathrm{OH} = \mathrm{polylog}(n) \cdot \log(\mathrm{size}(f))$.*

We can similarly use Theorem 3 to get a protocol for the CRS model and initial broadcast between log-size sets of parties. In this case we will only get

computational security, and we might therefore as well go for the weaker model where we assume a PKI instead of initial broadcasts. Given a PKI we can implement the broadcasts using for instance the protocol in [DS83].

Corollary 2. *For all $c \in [0, 1)$ there exists a protocol π secure against adaptive, malicious corruption of up to cn parties and with termination guarantee against a non-zero constant fraction of corruptions for the (PKI, CRS)-hybrid model for all $f \in \mathsf{NC}$ with $\mathrm{OH}_{\mathrm{ARITH}} = \lambda \cdot \mathrm{polylog}(n) \cdot \log(\mathrm{size}(f))$.*

Acknowledgments

This work is supported by European Research Council Starting Grant 279447. Samuel Ranellucci is supported by NSF grants #1564088 and #1563722. This work is partially supported by the H2020-LEIT-ICT project SODA, project number 731583. The authors would also like to thank the anonymous reviewers for their valuable comments and suggestions. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation.

References

[BCP15] Elette Boyle, Kai-Min Chung, and Rafael Pass. Large-scale secure computation: Multi-party computation for (parallel) RAM programs. In *Advances in Cryptology–CRYPTO 2015*, pages 742–762. Springer, 2015.

[Bra87] Gabriel Bracha. An o (log n) expected rounds randomized byzantine generals protocol. *Journal of the ACM (JACM)*, 34(4):910–920, 1987.

[BSFO12] Eli Ben-Sasson, Serge Fehr, and Rafail Ostrovsky. Near-linear unconditionally-secure multiparty computation with a dishonest minority. In *Advances in Cryptology–CRYPTO 2012*, pages 663–680. Springer, 2012.

[CDD+15] Ronald Cramer, Ivan Bjerre Damgård, Nico Döttling, Serge Fehr, and Gabriele Spini. Linear secret sharing schemes from error correcting codes and universal hash functions. In Elisabeth Oswald and Marc Fischlin, editors, *Advances in Cryptology - EUROCRYPT 2015 - 34th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Sofia, Bulgaria, April 26-30, 2015, Proceedings, Part II*, volume 9057 of *Lecture Notes in Computer Science*, pages 313–336. Springer, 2015.

[CDI+13] Gil Cohen, Ivan Bjerre Damgård, Yuval Ishai, Jonas Kölker, Peter Bro Miltersen, Ran Raz, and Ron D Rothblum. Efficient multiparty protocols via log-depth threshold formulae. In *Advances in Cryptology–CRYPTO 2013*, pages 185–202. Springer, 2013.

[CPS14] Ashish Choudhury, Arpita Patra, and Nigel P Smart. Reducing the overhead of MPC over a large population. In *Security and Cryptography for Networks*, pages 197–217. Springer, 2014.

[DIK+08] Ivan Damgård, Yuval Ishai, Mikkel Krøigaard, Jesper Buus Nielsen, and Adam Smith. Scalable multiparty computation with nearly optimal work and resilience. In *Advances in Cryptology–CRYPTO 2008*, pages 241–261. Springer, 2008.

[DIK10] Ivan Damgård, Yuval Ishai, and Mikkel Krøigaard. Perfectly secure multiparty computation and the computational overhead of cryptography. In Henri Gilbert, editor, *Advances in Cryptology - EUROCRYPT 2010, 29th Annual International Conference on the Theory and Applications of Cryptographic Techniques, French Riviera, May 30 - June 3, 2010. Proceedings*, volume 6110 of *Lecture Notes in Computer Science*, pages 445–465. Springer, 2010.

[DKL+13] Ivan Damgård, Marcel Keller, Enrique Larraia, Valerio Pastro, Peter Scholl, and Nigel P Smart. Practical covertly secure MPC for dishonest majority–or: Breaking the SPDZ limits. In *Computer Security–ESORICS 2013*, pages 1–18. Springer, 2013.

[DPSZ12] Ivan Damgård, Valerio Pastro, Nigel Smart, and Sarah Zakarias. Multiparty computation from somewhat homomorphic encryption. In *Advances in Cryptology–CRYPTO 2012*, pages 643–662. Springer, 2012.

[DS83] Danny Dolev and H. Raymond Strong. Authenticated algorithms for byzantine agreement. *SIAM J. Comput.*, 12(4):656–666, 1983.

[DZ13] Ivan Damgård and Sarah Zakarias. Constant-overhead secure computation of boolean circuits using preprocessing. In *Theory of Cryptography*, pages 621–641. Springer, 2013.

[GWZ09] Juan A. Garay, Daniel Wichs, and Hong-Sheng Zhou. Somewhat non-committing encryption and efficient adaptively secure oblivious transfer. In Shai Halevi, editor, *Advances in Cryptology - CRYPTO 2009, 29th Annual International Cryptology Conference, Santa Barbara, CA, USA, August 16-20, 2009. Proceedings*, volume 5677 of *Lecture Notes in Computer Science*, pages 505–523. Springer, 2009.

[HM00] Martin Hirt and Ueli Maurer. Player simulation and general adversary structures in perfect multiparty computation. *Journal of cryptology*, 13(1):31–60, 2000.

[HMP00] Martin Hirt, Ueli M. Maurer, and Bartosz Przydatek. Efficient secure multiparty computation. In Tatsuaki Okamoto, editor, *Advances in Cryptology - ASIACRYPT 2000, 6th International Conference on the Theory and Application of Cryptology and Information Security, Kyoto, Japan, December 3-7, 2000, Proceedings*, volume 1976 of *Lecture Notes in Computer Science*, pages 143–161. Springer, 2000.

[IKOS08] Yuval Ishai, Eyal Kushilevitz, Rafail Ostrovsky, and Amit Sahai. Cryptography with constant computational overhead. In *Proceedings of the fortieth annual ACM symposium on Theory of computing*, pages 433–442. ACM, 2008.

[IKP+16] Yuval Ishai, Eyal Kushilevitz, Manoj Prabhakaran, Amit Sahai, and Ching-Hua Yu. Secure protocol transformations. In Matthew Robshaw and Jonathan Katz, editors, *Advances in Cryptology - CRYPTO 2016 - 36th Annual International Cryptology Conference, Santa Barbara, CA, USA, August 14-18, 2016, Proceedings, Part II*, volume 9815 of *Lecture Notes in Computer Science*, pages 430–458. Springer, 2016.

[IPS08] Yuval Ishai, Manoj Prabhakaran, and Amit Sahai. Founding cryptography on oblivious transfer - efficiently. In David Wagner, editor, *Advances in Cryptology - CRYPTO 2008, 28th Annual International Cryptology Conference, Santa Barbara, CA, USA, August 17-21, 2008. Proceedings*, volume 5157 of *Lecture Notes in Computer Science*, pages 572–591. Springer, 2008.

[KMTZ13] Jonathan Katz, Ueli Maurer, Björn Tackmann, and Vassilis Zikas. Universally composable synchronous computation. In *TCC*, pages 477–498, 2013.

[PW92] Birgit Pfitzmann and Michael Waidner. Unconditional byzantine agreement for any number of faulty processors. In Alain Finkel and Matthias Jantzen, editors, *STACS 92, 9th Annual Symposium on Theoretical Aspects of Computer Science, Cachan, France, February 13-15, 1992, Proceedings*, volume 577 of *Lecture Notes in Computer Science*, pages 339–350. Springer, 1992.

[SS96] Michael Sipser and Daniel A. Spielman. Expander codes. *IEEE Trans. Information Theory*, 42(6):1710–1722, 1996.

[WC81] Mark N. Wegman and Larry Carter. New hash functions and their use in authentication and set equality. *J. Comput. Syst. Sci.*, 22(3):265–279, 1981.