Asynchronous YOSO *a la* Paillier

Ivan Bjerre Damgård[1], Simon Holmgaard Kamp[2], Julian Loss[2], and Jesper Buus Nielsen[1]

[1] Aarhus University
[2] CISPA Helmholtz Center for Information Security

Abstract. We present the first complete adaptively secure asynchronous MPC protocol for the YOSO (You Speak Only Once) setting. In contrast to many previous MPC constructions in the YOSO model, we provide a full stack implementation that does MPC, role assignment and total order broadcast. Therefore, our construction is also the first to provide adaptively secure asynchronous total order broadcast and MPC that is sub-quadratic in the number of parties and does not require threshold fully homomorphic encryption. Instead, our protocols rely on threshold additively homomorphic Paillier encryption. Our total-order broadcast protocol has complexity optimal in the message length. This optimality also implies that the amortized complexity of handling a secure multiplication is linear in the number of parties.

Table of Contents

1 Introduction

In multiparty computation (MPC), a set of parties want to compute an agreed function on a set of privately held inputs such that the intended output is computed, but this is the only new information released. A very special case is Total Order Broadcast (TOB), where the goal is to provide a public "ledger" that all parties can write to, and agreement is guaranteed on what was written and the order in which data arrived. Security guarantees hold assuming at most some maximal number of parties are corrupted by an adversary.

In this paper, we focus on MPC and TOB secure against adaptive corruption, where the adversary decides during the protocol who to corrupt, and we assume an asynchronous network where any message sent is eventually delivered, but the delay is arbitrary and decided by the adversary. In this model, it is possible to do information theoretically secure MPC based on secret sharing, but in any such construction, parties must maintain a large secret state corresponding to the size of the entire computation. In this paper, we will instead focus on constructions using public-key cryptography, where the secret state can be reduced. In particular, the so-called CDN paradigm [CDN01] is based on threshold additively homomorphic encryption. Here, the only secret state parties need to maintain is a set of shares of the private key, the state of the computation is contained in public ciphertexts. In [CDN01] it was shown that *synchronous* MPC with adaptive security can be obtained from the CDN paradigm using the Paillier cryptosystem.

The YOSO (You Speak Only Once) paradigm [GHK+21] is a more recent promising paradigm for MPC for a large set of M parties, where the work is done by a sequence of small committees of size $n \ll M$. Each committee only speaks once, and is anonymous until it speaks to hide from an adaptive adversary. Because the committee size is much smaller than M, one can hope to have better complexity than conventional solutions for M parties. It was shown in [GHK+21] how to adapt the CDN paradigm for the YOSO setting: each committee hold shares of the secret key, that are handed down from the previous committee. This remains the most efficient approach to YOSO MPC. For this to work, a construction for Role Assignment (RA) is required. In a nutshell, RA provides a way to choose anonymous committees and a way for one committee to send secret messages to the next. A popular approach is to base this on private information retrieval (PIR), see [GHM+21] for instance.

Some caveats remain, however, in the previous YOSO literature: MPC protocols for this setting require RA and TOB but often assume they are given for free. More seriously, they are not proven secure against a fully adaptive adversary, even though the motivation for YOSO is adaptive attacks. Instead they use a model where the adversary corrupts a set of parties initially and then committees are chosen randomly from the set of parties. Moreover, to the best of our knowledge, no YOSO MPC has been shown secure in the asynchronous setting.

A different line of research has instead focused on adaptively secure and asynchronous consensus, in particular [BKLZL20] presents the first Byzantine agreement protocol in this model that is subquadratic in the number of parties. As a building block they implement a protocol for Agreement on a Core Set (ACS) used to reach agreement on a set of inputs and often used to construct a TOB by sequentially running instances of ACS. However, the

protocol is based on quite heavy machinery: it makes non-blackbox use of threshold fully homomorphic encryption. It in particular runs key generation for FHE inside FHE.

1.1 Our Contribution

In this paper, we bring together the two lines of research mentioned above and present the first full-stack implementation of adaptively secure YOSO MPC for an asynchronous network. We present protocols in this setting for TOB, RA and MPC that are secure assuming at most $(1/3-c)M$ of the M parties are corrupted, where c can be any constant. Our protocols are based on threshold Paillier/Damgård-Jurik encryption and integer commitments based on factoring. We make black-box use of these tools in the sense that we exploit only the algebraic properties they natively possess and our communication complexity does not depend on the circuit complexity of the primitives we use.

Our TOB is the first asynchronous and adaptively secure sub-quadratic TOB that does not need threshold FHE. It is also communication optimal, in that for long enough messages to broadcast, the cost is linear in the number of parties. This improves on recent work [BLZLN22] by giving a simplified and more efficient extension protocol for the asynchronous setting that avoids the use of cryptographic accumulators.

Our RA protocol is based on a new PIR protocol that makes black-box use of Paillier/Damgård-Jurik encryption and hence is well suited for implementing the client side of the PIR in MPC. In comparison, the RA protocol from [GHM+21] (that was also based on PIR) makes non-blackbox use of threshold FHE. The amortized communication complexity for creating an anonymous committee is $O(M \operatorname{poly}(\lambda, \log M))$ where λ is the security parameter.

Our MPC protocol is based on the CDN paradigm, so we have a sequence of committees where each committee reshares the secret Paillier key for the next committee. We show a new technique for this that keeps the size of the shares constant. In contrast, in all previous YOSO protocols based on CDN, the share sizes grew for every resharing. Finally, because the TOB is optimal, the amortized communication complexity of our secure multiplication is linear in the number of parties, if sufficiently many multiplications are done in parallel.

1.2 Technical Overview

Asynchronous YOSO from Paillier Encryption The YOSO MPC uses a version of the CDN [CDN01] paradigm adapted to the asynchronous model. We work in the random oracle model and assume we are given set-up data, such as the public key for the Paillier scheme.

Once we have an asynchronous MPC we follow the idea from [GHM+21] of doing YOSO role assignment by simulating a PIR client in MPC. As a main technical contribution we show how to do this without heavy machinery such as threshold FHE: we first propose a novel 2-message PIR protocol based on Damgård-Jurik/Paillier encryption. It has overhead logarithmic in the size of the database, and has the major advantage that its client part is very well suited for implementation in a CDN MPC based on Paillier, since the message from the server is effectively a Paillier encryption of the database item the client wants. With

this machinery, we sample a random party to be member of a committee as follows: intially, each party P_i publishes Paillier encryptions of a random tag tag_i and a one-time pad otp_i. We consider sets of pairs of encryptions as the database, retrieve a random one of these via PIR, randomize them inside MPC, and finally output the randomized pair of ciphertexts. We decrypt the tag, allowing the selected party P_i to recognize his tag. Using the encrypted one-time pad, we can send results computed inside MPC anonymously to P_i. We simply add result and one-time pad inside the encryption and then we decrypt.

Using a variant of this idea, we get a major simplification compared to previous YOSO protocols. Any such protocol needs to enable secret data to be sent from one committee to the next. Earlier protocols do this by assigning a (randomized) public key anonymously to each committee member, a so-called role key and the previous committee can then encrypt data for the next one.

In our approach, we do not need role keys at all. Instead, we do as follows: say P_i needs to send a secret message to P_j. We compute and decrypt the sum of one-time pads chosen by the two players, $\mathsf{otp}_i + \mathsf{otp}_j$. Once this value is public, P_i can compute otp_j and use it to one-time-pad encrypt the message she is to send to P_j, without knowing the identity of P_j.

We need to use this idea for sending shares of the secret Paillier key from one committee to the next. As can be seen, this approach creates a circular situation where (shares of) the secret Pailler key are encrypted using one-time pads that are in turn protected under Paillier encryption[3]. So, we need to assume that our cryptosystem is secure despite this. We believe this non-standard assumption is a fair price to pay for achieving our goal of making black-box use of the primitives and avoiding generic techniques. It is also one that is not easily avoided: It would be extremely inefficient and non black-box to generate new key set-up for Paillier inside MPC. So, the modulus N we are given as set-up must be used for the entire protocol.[4] Moreover, the shares of the secret key must be protected under some form of encryption as they need to be communicated to the parties who need them. The encryption scheme we have available is Paillier, so some form of circularity must result.

In our protocol, we need to generate many random encrypted bits. We cannot use the trick from [DFK+06], at it relies on computing square roots modulo a prime which cannot be done efficiently modulo N. So we propose the novel idea of deriving the bit from the Jacobi symbol of a random encrypted number modulo N. This has the added advantage of giving us a straight-line simulatable coin-flip in the random oracle model.

We achieve adaptive security using techniques similar to those used in [DN03] where a synchronous, adaptive secure, but non-YOSO CDN-style protocol was presented. A challenge, however, is that the UC simulator needs to be given a trapdoor allowing it to fake decryptions of ciphertexts that contain incorrect values, and the solution from [DN03] is incompatible with YOSO. We suggest a new solution exploiting that we have a random oracle. In adapting [DN03] we also adapt the *single inconsistent party* technique to work in the asynchronous YOSO setting for the first time.

[3] The issue does not come from the use of one-time pads, the circularity would also be there if we used role keys.

[4] Using class group based encryption is not an option, this would make generation of new set-up easy, but would require us to use a generic PIR protocol for role assignment.

Efficient Subquadratic Total Order Broadcast for the YOSO Model As part of our overall YOSO MPC protocol, we present a highly efficient total order broadcast (TOB) protocol with subquadratic communication complexity for adaptive corruptions in a hybrid model assuming a perfect coin flip functionality. We use our YOSO MPC to implement the coin-flip with subquadratic communication complexity. Our protocol relies on the [BKLZL20] paradigm of having an initial setup that allows flipping λ coins to run a TOB, and then using the TOB to recompute a new setup for future iterations. While [BKLZL20] is the first work to break the n^2 communication barrier for binary Byzantine agreement (BBA) in the asynchronous setting with adaptive corruptions, it is concretely quite heavy on communication and does not explain how to obtain an efficient TOB from BBA. We improve on the state of the art by taking the protocol for Agreement on a Core Set from [KN23] and adapting it to the YOSO framework. This allows us to get subquadratic communication with adaptive corruptions, by moving to the setting with suboptimal resilience against $T < (1-\epsilon)M/3$ corruptions studied in [BKLZL20,BLZLN22] where we can sample a fresh committee with honest supermajority for every round of the protocol. In this setting, we instantiate the RB protocol by [DXR21] for committees by using Reed-Solomon codes with reconstruction thresholds linear in n over a field of size at least M. This results in a subquadratic extension protocol to attain RB with optimal complexity in the message length, similar to the work of Banghale et al. [BLZLN22]. However, our protocol appears to be substantially simpler and more efficient, as it requires neither accumulators nor any application of advanced concentration bounds such as McDiarmid's inequality. Finally, we add some machinery to reach agreement on how much setup was consumed by the protocol. This allows the TOB to only consume expected constant number of RA committees per block, as opposed to the worst case λ.

2 Preliminaries

Paillier and Damgård-Jurik encryption Most of the material in this subsection is taken from [DJN10] where more details and proofs of the facts stated here can also be found.

We use several variants of the Paillier cryptosystem. It works with an RSA modulus $N = pq$ where the prime factors p, q are of form $p = 2p' + 1, q = 2q' + 1$ and p', q' are also primes. We use the Damgård-Jurik generalization of Paillier, where the plaintext space is N^s for $s \geq 1$, and the basic form of an encryption of a message $m \in \mathbb{Z}_{N^s}$ with randomness $r \in \mathbb{Z}_N^*$ is $(N+1)^m r^{N^s} \bmod N^{s+1}$. This encryption scheme is well-known to be additively homomorphic modulo N^s. Moreover, it is CPA-secure under the well-known Decisional Composite Residuosity Assumption (DCRA). It can be shown that this assumption for $s = 1$ implies the same assumption for any polynomial size s. To simplify matters in the following, we will assume that the randomness for the encryption is always chosen to have Jacobi symbol +1. It can be shown that this variant is CPA secure under the same assumption.

A lossy version. We will make use of a variant where the element $(N + 1)$ is replaced by an encryption of 1, denoted by w_s, and determined as follows. We consider values of s up to some maximum value S. We set $w_S = (N+1)u_S^{N^S} \bmod N^{S+1}$, for random u_S. We then

set $w_s = w_S \bmod N^{s+1}$ for $s < S$. It is easy to see that all w_s are encryptions of 1. We will consider w_S as part of the public key. Using this notation, we define

$$E_{N,w_s}(m;r) = w_s^m r^{N^s} \bmod N^{s+1} \ .$$

It is easy to see that $E_{N,w_s}(m;r)$ is actually an encryption of m according to the basic scheme above, it therefore also inherits the homomorphic property of the basic scheme. Concretely, we have for plaintexts m_1, m_2, a and randomness r_1, r_2 that

$$E_{N,w_s}(m_1;r_1)^a \cdot E_{N,w_s}(m_2;r_2) \bmod N^{s+1} = E_{N,w_s}(am_1 + m_2 \bmod N^s; r_1^a r_2 \bmod N)$$

The point of the variant using w_s is that it is a possibly lossy encryption scheme. Namely, if we replace w_S by a random encryption $\bar{w}_S$ of 0, then an "encryption" of any m, $E_{N,\bar{w}_s}(m;r) = \bar{w}_s^m r^{N^s} \bmod N^{s+1}$ will actually be an encryption of 0. However, given only the public key, the two forms cannot be distinguished, as an encryption of 0 is indistinguishable from an encryption of 1.

How to decrypt For decryption, we define a decryption exponent d_S constructed by the Chinese remainder theorem such that $d_S \equiv 0 \bmod \phi(N)$ and $d_S \equiv 1 \bmod N^S$. We can use d_S to decrypt any $E_{N,w_s}(m;r)$ where $s \leq S$, as we have:

$$E_{N,w_s}(m;r)^{d_S} = (N+1)^{md_S}((ru_s)^{N^s})^{d_S} \bmod N^{s+1} = (N+1)^m \bmod N^{s+1}.$$

We can then get the message by exploiting the fact that discrete logs modulo $(N+1)$ are easy to compute. Later, we will define a threshold version of the cryptosystem where d_S is shared among a set of parties, who can then collaborate to raise a ciphertext to power d_S, thus effectively decrypting it. In the following, we will sometimes suppress the randomness from the notation for readability, and write $E_{N,w_s}(m)$ instead of $E_{N,w_s}(m;r)$.

Commitment Scheme We will need a commitment scheme allowing commitment to integers. We will use the scheme suggested by Fujisaki and Okamoto [FO99], for a formal treatment, see [DF02]. Here, we describe the properties we need informally. The scheme is based on a modulus N' of the same form as our Paillier modulus, but chosen independently. The public key for the scheme is now $\mathsf{ck} = (N', \alpha, \beta)$ where α, β are random squares mod N', and we assume ck is given as set-up. A commitment to integer x using randomness r is denoted $\mathsf{Com}_{\mathsf{ck}}(x;r) = \alpha^x \beta^r \bmod N'$. Here, x is an integer in an interval $[0..2^b]$, and r is chosen uniformly from an interval $[0..2^K]$. We return to the choice of K below. The commitment scheme is clearly homomorphic over both inputs to the commitment function, that is, we have

$$\mathsf{Com}_{\mathsf{ck}}(x;r) \cdot \mathsf{Com}_{\mathsf{ck}}(x';r') = \mathsf{Com}_{\mathsf{ck}}(x+x';r+r') \ .$$

It is well known that this is statistically hiding. Computationally binding and soundness of the standard Σ-protocol for proving that you know how to open a commitment follow from the strong RSA assumption.

Moreover, the scheme is equivocable: the public key can be generated together with a trapdoor td, namely the discrete log of α base β, and there is an PPT algorithm $\mathsf{Eq}(\mathsf{td})$ that

will first output a commitment c, and then on input $x \in [0..2^b]$ will produce r such that $c = \mathsf{Com}_{\mathsf{ck}}(x; r)$ where the distribution of c, r is statistically close to the one generated by an honest committer that commits to x and then opens the resulting commitment.

Non-Interactive Zero-Knowledge Proofs In several cases, we will require non-interactive zero-knowledge proofs of knowledge. They allow a prover to prove, for a predicate P and a given (public) value x that they know a witness w such that $P(x, w) = 1$. We will denote such a proof by $\mathsf{NIZK}(w : P(x, w) = 1)$. These proofs need to be unconditionally simulation sound, statistical zero-knowledge and on-line extractable, i.e., there is an efficient extractor that can extract a witness from a successful prover without rewinding. In all cases, we can achieve this in the random oracle model: we first construct a Σ-protocol for the relation in question using standard techniques, and then we apply the Fishlin transformation to get non-interactive proofs. More details can be found in Appendix F. With one exception, all Σ-protocols introduce only a constant factor overhead in communication which means that after the Fischlin transformation, we get an overhead factor of $\lambda / \log \lambda$, where λ is the security parameter. The exception is the protocol for so-called 2-level ciphertexts from Appendix D, which is a double discrete log type protocol that inherently seems to require binary challenges and therefore gives us an overhead factor λ.

Linear Integer Secret Sharing The secret decryption exponent will be secret-shared among the members of a committee using *Linear Integer Secret Sharing* (LISS) [DT06]. For details on LISS and the verifiable secret sharing scheme we build on top, see Appendix E. We list here some notation and properties we will need in the main text.

A *sharing vector* $\boldsymbol{v}_d$ has c entries and contains the secret d as its first entry. The other entries are randomness used for the sharing. $\mathsf{sh}(d, \boldsymbol{v}_d)$ is the vector of ℓ shares resulting from applying the sharing algorithm to $\boldsymbol{v}_d$ which concretely means multiplying $\boldsymbol{v}_d$ by a public matrix M_{share} with integer entries, to get the vector of shares. As is well known, a secret sharing scheme comes with an access structure, a family of subsets of the players that are qualified to reconstruct the secret. A *valid sharing vector* $\boldsymbol{v}_d$ is one where the random entries are sufficiently large compared to d, more precisely, the bit-length of random numbers in the sharing vector required to share a δ-bit number is $\delta' = \delta + k$ where k is a statistical security parameter. In the appendix it is shown that as long as the sharing vector is valid, the scheme is statistically secure. That is, for any unqualified set A and any δ-bit value d that is shared, the shares given to A have distribution statistically indistinguishable from a shares of a default value, say 0.

Computational Assumption We now specify the computational assumption our protocol is based on. As explained in the introduction it is a circular security type assumption.

On a high level, we define a game between the adversary and an interactive agent we call a privacy preserving box denoted PPBox. The box generates a Paillier key pair, gives the public parameters to the adversary and then sets its initial state d to be the secret Pailler key. The public data contains a ciphertext w_S and the goal of the adversary is to decide whether

Game $\mathsf{CSO}_A^{b_{enc},b_{sk}}$

Init: b_{enc}, b_{sk} are assumed to be bits. Initially generate $(N, w_S, d_S) \leftarrow \mathsf{Gen}$ and give (N, w_S) to A. Let $w_S = E_{N,w_S}(b_{env})$. Send w_S to A. Delete the randomness used for the encryption. If $b_{sk} = 1$ set $\mathsf{st} = d_S$, else set $\mathsf{st} = 0$. Set a counter $i = 0$. The box is parametrized by a sequence of randomized polynomial time computable functions $f_1, f_2, \ldots$ and a predicate ω.

Update State. The adversary A issues this query with an auxiliary input x, the box sets $\mathsf{st} = f_i(\mathsf{st}, x)$ and sets $i = i + 1$.

Encryption queries The adversary issues this query with the specification of polynomial time computable functions ϕ, ψ. The box returns $\overline{E_{N,w_S}(\phi(\mathsf{st}))}$ (and deletes the randomness used for the encryption).

Leakage Queries The adversary issues this query with the specification of a polynomial time computable function g. The box returns $g(\mathsf{st})$.

Guess: At any time, the adversary may end the game by outputting a guess $g \in \{0, 1\}$.

Fig. 1. The Circular Selective Opening Game

w_S contains 0 or 1. The box maintains state which equals d_S initially. The adversary can issue three types of queries to the box:

- A state update query comes with an input x. The box will update its state by setting $\mathsf{st} = f_i(x, \mathsf{st})$ where f_i is a randomized function used for the i'th update. The sequence of functions $f_1, f_2, \ldots$ is fixed and part of the definition of the box.
- An encryption query comes with the specification of efficiently computable function ϕ. The box returns $\overline{E_{N,w_S}(\phi(\mathsf{st}))}$ to the adversary. The encryption is a variant of Paillier encryption that comes with the definition of the box, and is essentially the same as Paillier, in a sense defined below. The randomness used for the encryption is not part the state and is deleted.
- A leakage query that comes with the specification of a function (denoted g_i for the i'th query). The box returns $g_i(\mathsf{st})$ to the adversary. However, the functions g_i cannot be chosen freely. The sequences of functions $g_1, g_2, \ldots, \psi_1, \psi_2, \ldots$ must be *admissible*, which by definition means that for all j it must be the case that $\omega(g_1, \ldots, g_j) = 1$ where ω is a fixed predicate that is part of the definition of the box. Loosely speaking, ω should be designed such that the extra information released does not help the adversary, we define this more precisely below.

Definition 1. *We say that* PPBox *is* encryption secure *if its encryption scheme* $\overline{E_{N,w_S}(\cdot)}$ *uses the same public key* N, w_S *as Paillier encryption, is CPA secure under the DCRA assumption (like Paillier) and is lossy if* w_S *is an encryption of* 0.

As an example of an encryption scheme that would satisfy this definition, one can think of $\overline{E_{N,w_S}(m)} = (E_{N,w_S}(R), R + m)$ where R is a random one-time pad. So, when the bit b_{enc} in the CSO game is 0, the encryptions returned from the box are lossy, they are independent of the input state.

Some notation: Let D_{A,d_S} be the distribution of the responses to the leakage queries when running $\mathsf{CSO}_A^{0,1}$, and let the $D_{A,0}$ be the distribution of the responses when running $\mathsf{CSO}_A^{0,0}$.

The idea here is to define a fixed distribution that is independent of d_S, and then we want the leakage to look like this distribution.

Definition 2. *An adversary A is* admissible *with respect to* PPBox *if it always issues an admissible sequence of queries when talking to* PPBox. PPBox *is said to be* privacy preserving *if it is the case that when A is admissible, the distribution D_{A,d_S} is statistically independent of the secret input d_S. More precisely, D_{A,d_S} and $D_{A,0}$ are statistically indistinguishable.*

The following assumption essentially says that even if the PPBox uses the secret key d_S internally, the encryptions are CPA secure, specifically, one cannot tell if w_S contains 0 or 1.

Definition 3 (Circular Selective Opening Security). *We say that Paillier/Damgård-Jurik encryption is CSO secure if for any encryption secure and privacy preserving* PPBox *and any PPT adversary A admissible with respect to* PPBox*, it holds that $\mathsf{CSO}_A^{1,1}$ is computationally indistinguishable from $\mathsf{CSO}_A^{0,1}$.*

We will assume in the following that Paillier/Damgård-Jurik encryption is CSO secure. As evidence in favour of this assumption we can note the following lemma:

Lemma 1. *For any encryption secure and privacy preserving* PPBox*, and any admissible adversary A, $\mathsf{CSO}_A^{0,1}$ is statistically indistinguishable from $\mathsf{CSO}_{A,0}^{0,0}$, and $\mathsf{CSO}_A^{0,0}$ is computationally indistinguishable from $\mathsf{CSO}_A^{1,0}$ assuming standard CPA security of Paillier encryption.*

Proof. The first conclusion follows because the ciphertexts are lossy in both cases and PPBox is privacy preserving. The second one follows by an easy reduction because $\mathsf{CSO}_A^{0,0}$ or $\mathsf{CSO}_A^{1,0}$ can be simulated perfectly given w_S and the public Paillier key, as the secret Paillier key is not used in any of these games. □

Given Lemma 1, our assumption is equivalent to claiming that $\mathsf{CSO}_A^{1,1}$ is computationally indistinguishable from $\mathsf{CSO}_A^{1,0}$. The only difference between these two games is that the encryptions sent from the box contain information related to the secret key d_S in one case and not in the other. If one is willing to believe that this difference does not matter to the adversary (note the randomness for encryptions is never revealed) then the adversary is left with the leakage which should not help, as the box is privacy preserving. Put differently, to break the assumption, one needs to exploit in some highly non-trivial way that the encryptions one is given relate to the secret key. To the best of our knowledge, no such approach is known.

It is worth noting that we do not actually need the assumption in full generality. We only need one specific version of PPBox that updates its state by generating a sequence of fresh secret sharings of d_S, the encryption queries return encryptions of the shares generated, and the leakage queries are only allowed to reveal an unqualified subset of any set of shares. However, as we believe the more general version is true, we chose it for better readability.

We emphasize that even though our assumption is interactive and non-standard, this does not trivializes the proof: the actual protocol contains many ingredients that are not

present in the game, such as commitments, zero-knowledge proofs, one-time pad encryption etc. So we are very far from claiming that the protocol is secure under the assumption that it is secure.

Protocol PIR, to be executed by a client C and a server S.
S has a database consisting of numbers in $\mathbb{Z}_N$, $y_0, \ldots, y_{L-1}$, and we assume for simplicity that L is a two-power, $L = 2^\lambda$. C wants to retrieve y_t for some $1 \leq t \leq L$.

1. C writes t in binary as $t_0, t_1, ..., t_{\lambda-1}$ where t_0 is the *most* significant bit. C sends the following to S:
$$c_0 = E_{N,w_1}(t_0), \ldots, c_{\lambda-1} = E_{N,w_\lambda}(t_{\lambda-1}) \ .$$
2. S sets $(y_0^0, ..., y_{L_1}^0) = (y_0, ..., y_{L-1})$, and now executes the following loop for $i = 0$ to $\lambda - 1$:
   (a) Set $\bar{c}_i = E_{N,w_i}(1) \cdot c_i^{-1} \bmod N^{i+1}$, where $E_{N,w_i}(1)$ is an encryption of 1 with default randomness 1. Note that $\bar{c}_i$ is an encryption of the bit $1 - t_i$.
   (b) Define $L_i = L/2^i$. This step takes as input the list $(y_0^i, ..., y_{L_i-1}^i)$, and outputs a list $(y_0^{i+1}, ..., y_{L_i/2-1}^{i+1})$. Namely, for $j = 0$ to $L_i/2 - 1$, set
   $$y_j^{i+1} = c_i^{y_j^i} \cdot \bar{c}_i^{y_{j+L_i/2}^i} \bmod N^{i+1}.$$
   Note that y_j^{i+1} is an encryption of $t_i \cdot y_j^i + (1 - t_i) \cdot y_{j+L/2^i}^i$, and hence the loop produces encryptions of the first or the second half of the incoming list, depending on t_i.
3. The list output by the last loop step above has lenght $L_\lambda = L/2^\lambda = 1$. S sends the single ciphertext on the list y_0^λ to C.
4. Note that we have
$$y_0^\lambda = E_{N,w_{\lambda-1}}(E_{N,w_{\lambda-2}}(\ldots E_{N,w_1}(y_t))),$$
so by doing $\lambda - 1$ decryption steps, C can retrieve y_t, as desired.

Fig. 2. The PIR protocol.

3 A PIR protocol based on Damgård-Jurik encryption

In this section, we present a PIR protocol (Fig. 2), in the standard form with a single client and server, where we assume that the client has the secret key for an instance of the Damgård-Jurik scheme as defined above. In a later section we show how to transplant it to a multiparty setting, where several parties play the role of both server and client and we will see how this can be used to do role assignment.

A first important observation that will come in handy is the following: the plaintext space of the encryption function $E_{N,w_s}(\cdot;\cdot)$ is $\mathbb{Z}_N^s$, but this is also the ciphertext space of $E_{N,w_{s-1}}(\cdot;\cdot)$. Therefore, given ciphertexts $E_{N,w_{s-1}}(m)$, $E_{N,w_s}(m')$ and the public key, one can compute $E_{N,w_s}(m')^{E_{N,w_{s-1}}(m)} \bmod N^{s+1} = E_{N,w_s}(m' \cdot E_{N,w_{s-1}}(m) \bmod N^s)$.

The PIR protocol is clearly secure based on CPA security: S cannot distinguish C's message from a set of random encryptions, and if S is semi-honest, C will always get correct output. The protocol requires only 2 messages which is clearly optimal.

Note that we can allow the input y_i's to be longer than the modulus, say numbers in $\mathbb{Z}_{N^v}$, we just need to define C's message to be

$$c_0 = E_{N,w_v}(t_0), \ldots, c_{\lambda-1} = E_{N,w_{v+\lambda}}(t_{\lambda-1}),$$

and adjust S's part accordingly. If we denote the length of a data item by k and the length of N by κ, one finds that the communication complexity is $O(\lambda k + \lambda^2\kappa)$. Thus, for large k, the overhead over just sending a data item in the clear is only a factor λ, logarithmic in the size of the database.

4 UC Modelling

We use the UC framework in [CCL15] as it is asynchronous, allows to model interactive functionality and is simple and sufficient for our study, as we work with a fixed set of parties $\mathbb{P} = \{P_1, \ldots, P_M\}$. We model functionalities for total-order broadcast ($\mathcal{F}_{\text{TOB}}$) and YOSO MPC and coin-flip ($\mathcal{F}_{\text{MPC+CF}}$). We also have a helper $\mathcal{F}_{\text{SETUP}}$ for setting up values. As the underlying model of communication we assume authenticated point-to-point communication with atomic send, where a party in one instruction can send messages to several users, and it is guaranteed that all messages be delivered even if the party gets corrupted right after the sending. See [BKLZL20] for further discussion. For completeness an ideal functionality is given in Fig. 28. When formulating ideal functionalities we talk about the adversary having to eventually deliver certain values to guarantee liveness. We discuss in Appendix G.1 how this is formalized, but the exact formulation is not consequential for the security of our protocols.

Functionality $\mathcal{F}_{\text{TOB}}$

Init: Let $L = \epsilon$ be the empty ledger. For each party P let $\ell_P = 0$ be the number of blocks delivered at P. Let $b = 0$ be the number of blocks produced so far. Let b_P be the number of blocks requested by P.

Broadcast: On input $(\text{BROADCAST}, \mathsf{mid}, m)$ from party $P(\mathsf{mid})$ add (mid, m) to $\mathsf{Accepted}$ and leak $(\text{BROADCAST}, \mathsf{mid}, m)$ to the adversary.

Set Wait Predicate: On input (WAIT, W) from honest party P let $b_P \leftarrow b_P + 1$ and let $W^{b_P} = W$. We require that all honest parties agree on W^{b_P}.

Next Batch: On input $B = ((\mathsf{mid}_1, m_1), \ldots, (\mathsf{mid}_\ell, m_\ell))$ from the adversary where W^{b+1} is defined and where $(\mathsf{mid}_j, m_j) \in \mathsf{Accepted}$ for $j = 1, \ldots, \ell$ and $W^{b+1}(L, B) = \top$, update $L \leftarrow L \| B$ and remove each (mid_j, m_j) from $\mathsf{Accepted}$. Update $b \leftarrow b + 1$.

Deliver: On input $(\text{DELIVER}, P)$ from the adversary where $\ell_P < |L|$ update $\ell_P = \ell_P + 1$ and output $L[\ell_P]$ to P.

Eventual Liveness: If $b_P > b$ for all honest parties and $W^{b+1}(L, \mathsf{Accepted}) = \top$ then eventually the adversary must produce block number $b + 1$. Furthermore, if $(\mathsf{mid}, m) \in \mathsf{Accepted}$ and $P(\mathsf{mid})$ is honest, then eventually (mid, m) must be added to a block. Finally, if $\ell_P < |L|$ then eventually the adversary will deliver a new message to P.

Fig. 3. Total-Order Broadcast

Total Order Broadcast Our ideal functionalities will have a notion of batches, which is just one "round" of broadcast or function evaluation. To not confuse with the synchronous notion of round we call them batches. In a fully asynchronous network not all parties can be guaranteed to give inputs to each batch. We introduce a notion of a wait predicate W which tells the ideal functionality when it can produce the next batch. For TOB this is formulated via a notion of blocks of messages and such a block being a valid extension of the current ledger. A block B is a non-empty sequence of bit-strings B and a ledger L is a sequence of blocks. A wait predicate $W(L, B) \in \{\top, \bot\}$ judges if B may extend L. We require that if $W(L, B) = \top$ then $W(L, B \| B') = \top$ for all blocks B' and $W(L, \pi(B)) = \top$ for all permutations of the messages in B. Messages are named by a message identifier mid. We assume that mid contains the name of the party allowed to send it, and we denote this party by $P(\mathsf{mid})$. We assume that $P(\mathsf{mid})$ uses mid at most once. The ideal functionality is given in Fig. 3. The formalisation is straightforward and uses known design patterns. For completeness there is a motivation in Appendix G.3

YOSO MPC The functionality in Fig. 4 for MPC also proceeds in batches, each evaluating one function f. Since the network is asynchronous we cannot ensure that all parties can give input. We therefore again give a wait predicate W. This is a monotone predicate—if $W(Q) = \top$ then $W(Q \cup Q') = \top$—judging whether we may evaluate f after having seen inputs from parties in Q. All parties are assumed to agree on W in a given round and the function f to compute. In each batch a secret uniformly random permutation $\pi : \mathbb{Q} \to \mathbb{Q}$ is chosen, assigning party P to play role $R = \pi(P)$. Each P gives two inputs x_P and z_P. We think of x_P as its input as the party P and z_P as its input as role $R = \pi(P)$. Note that f is not given π.[5] Instead we give f the inputs z_P sorted on roles.

Each batch has a public output y as well as a secret output y_R for role R. Only parties who gave input can get a secret output. The reason we need to hear from a party to give it secret output is that we want adaptive security, so the secret key needs to grow linear in the amount of data which can be sent under a fixed public key [Nie02]. So, parties would occasionally need to refresh their public keys under which we send them secret outputs. We opted for a simple solution establishing a non-committing encryption channel between the MPC and P. When generating a new batch P will pick a fresh one-time pad $\mathsf{otp}_P^{\mathsf{b}}$ and send $E_{(N,w_S)}(\mathsf{otp}_P^{\mathsf{b}})$ on the TOB. It then deletes all randomness used to compute $E_{(N,w_S)}(\mathsf{otp}_P^{\mathsf{b}})$ and keeps only $\mathsf{otp}_P^{\mathsf{b}}$. The YOSO MPC takes $E_{(N,w_S)}(\mathsf{otp}_P^{\mathsf{b}})$ as input and post $c_R^{\mathsf{b}} = y_R^{\mathsf{b}} + \mathsf{otp}_P^{\mathsf{b}} \bmod N$ on the TOB. In the simulation we can equivocate by lying about $\mathsf{otp}_P^{\mathsf{b}}$.

Note that when a party P is corrupted leakage of the secret output y_R^{b} only happens between batch b being ordered by the first honest party, where y_R^{b} may be generated by the adversary in **Generate Batch**, and batch b being delivered at P. It models that in the implementation, from when c_R^{b} is posted on the TOB and until it was delivered at P, a corruption of P will leak $\mathsf{otp}_P^{\mathsf{b}}$ and hence y_R^{b}. Once c_R^{b} was delivered to P it will delete $\mathsf{otp}_P^{\mathsf{b}}$ and y_R^{b} in the implementation.

[5] In the implementation we do not pass information on π to the MPC evaluating f. If f leaked information on π in y it would destroy our simulation strategy.

Functionality $\mathcal{F}_{\text{MPC+CF}}^{\mathsf{Pal},\gamma}$ for MPC and coin-flip. Parametrised by a number of coin-flips γ.

Init: When activated the first time do the following.

1. Generate $((N, w_S), d_S) \leftarrow \mathsf{Pal.Gen}$ and output (N, w_S) to all parties.
2. Leak d_S to the adversary.
3. Let $\mathsf{b} = 0$ be the number of batches produced.
4. For $P \in \mathbb{P}$ let $\mathsf{bo}_P = 0$ and $\mathsf{bd}_P = 0$ be the number of batches ordered by respectively delivered at P.

Order Next Batch: On input $(\textsc{NextBatch}, f, W, x_P, z_P \in \mathbb{Z}_N)$ from honest party P where $\mathsf{bo}_P = \mathsf{bd}_P$ do the following:

1. Leak $(\textsc{NextBatch}, f, W, P)$ to the adversary.
2. Update $\mathsf{bo}_P \leftarrow \mathsf{bo}_P + 1$.
3. Let $f^{\mathsf{bo}_P} = f$, $W^{\mathsf{bo}_P} = W$, $x_P^{\mathsf{bo}_P} = x_P$, and $z_P^{\mathsf{bo}_P} = z_P$. We assume that honest parties agree on f^b and W^b for a given b.

Generate Batch: We call $\mathbb{Q}$ qualified for some batch b if $\mathsf{bo}_P \geq b$ for all honest $P \in \mathbb{Q}$ and $W^b(\mathbb{Q}) = \top$. On input $(\textsc{NextBatch}, \mathbb{Q} \subseteq \mathbb{P}, \{\mathsf{vk}_P, (x_P^{\mathsf{b}+1}, z_P^{\mathsf{b}+1})\}_{P \in \mathbb{C}})$ from the adversary, where $\mathbb{Q}$ is qualified for batch $\mathsf{b}+1$ and $\mathbb{C}$ is the set of corrupted parties in $\mathbb{Q}$, do:

1. Update $\mathsf{b} \leftarrow \mathsf{b} + 1$.
2. Let $\mathbb{P}^{\mathsf{b}} = \mathbb{Q}$.
3. Let $\mathbb{R}^{\mathsf{b}} = \mathbb{P}^{\mathsf{b}}$ and pick a uniformly random permutation $\pi^{\mathsf{b}} : \mathbb{P}^{\mathsf{b}} \to \mathbb{R}^{\mathsf{b}}$. We say that party $P \in \mathbb{P}^{\mathsf{b}}$ has role $R_P^{\mathsf{b}} = \pi^{\mathsf{b}}(P) \in \mathbb{R}^{\mathsf{b}}$.
4. Sample $(y^{\mathsf{b}}, \{(R, y_R^{\mathsf{b}})\}_{R \in \mathbb{R}^{\mathsf{b}}}) \leftarrow f^{\mathsf{b}}\left(\{(P, x_P^{\mathsf{b}})\}_{P \in \mathbb{P}^{\mathsf{b}}}, \{(R_P^{\mathsf{b}}, z_P^{\mathsf{b}})\}_{P \in \mathbb{P}^{\mathsf{b}}}\right)$, where $\{(P, x_P^{\mathsf{b}})\}_{P \in \mathbb{P}^{\mathsf{b}}}$ is given as a vector sorted on P and $\{(R_P^{\mathsf{b}}, z_P^{\mathsf{b}})\}_{P \in \mathbb{P}^{\mathsf{b}}}$ is sorted on R_P^{b}.
5. Give $(y^{\mathsf{b}}, \{(P, R_P^{\mathsf{b}}, y_{R_P^{\mathsf{b}}}^{\mathsf{b}}\}_{P \in \mathbb{C}^{\mathsf{b}}})$ to the adversary.
6. For $j = 1, \ldots, \gamma$ sample uniformly random $\mathsf{coin}_j^{\mathsf{b}} \in \{0,1\}^\lambda$.

Deliver Batch: On input $(\textsc{DeliverBatch}, P)$ from the adversary, where $\mathsf{bd}_P < \mathsf{b}$, update $\mathsf{bd}_P = \mathsf{bd}_P + 1$ and output y^{bd_P} to P. If $P \in \mathbb{P}^{\mathsf{bd}_P}$ then also output $(R = \pi^{\mathsf{bd}_P}(P), y_R^{\mathsf{bd}_P})$ to P.

Flip Coin: On input $(\textsc{Flip}, b, j)$ from P where $b \leq \mathsf{bd}_P$ and $j \in [\gamma]$ record $(\textsc{Flip}, P, b, j)$ and output coin_j^b to the adversary.

Deliver Coin: On input $(\textsc{DeliverCoin}, P, b, j)$ from the adversary, where $b \leq \mathsf{b}$ and $j \in [\gamma]$ output $(\textsc{Flip}, b, j, \mathsf{coin}_j^b)$ to P.

Eventual Liveness: If $\mathsf{bo}_P \geq \mathsf{b}$ for all honest $P \in \mathbb{P}$, then the adversary must eventually input a valid $(\textsc{NextBatch}, \cdot, \cdots)$. If $\mathsf{bd}_P < \mathsf{b}$ for some honest P then the adversary must eventually input $(\textsc{DeliverBatch}, P)$. If for some (b, j) the value $(\textsc{Flip}, Q, b, j)$ is recorded for all honest $Q \in \mathbb{P}$ and P is honest, then the adversary must eventually input $(\textsc{DeliverCoin}, P, b, j)$.

Corruption: On corruption of P output $y_{R=\pi^{\mathsf{bd}_P}(P)}^b$ to the adversary for all b where $\mathsf{bd}_P < b \leq \mathsf{b}$.

Fig. 4. The Ideal Functionality for MPC and Coin-Flip

Note that we leak d_S to the adversary. This means that encryption under (N, w_S) is not secure in a context where $\mathcal{F}_{\text{MPC+CF}}$ is *used* as hybrid functionality. This is fine, as we only need that encryption under (N, w_S) is secure when we *implement* $\mathcal{F}_{\text{MPC+CF}}$. The reason why we leak d_S is that when we prove security we need the simulator to know d_S to be able to do straight-line simulation.

Functionality $\mathcal{F}_{\text{SETUP}}$ for setup parametrised by a number of parties M, commitment key generator Gen, Paillier encryption $\mathsf{Pal} = (\mathsf{Gen}, E, D)$, committee size n, initial number of committees eno.

1. Generate $((N, w_S), d_S) \leftarrow \mathsf{Pal.Gen}$, $\mathsf{ck} \leftarrow \mathsf{Gen}$, $c^* \leftarrow E_{N,w_1}(0)$, where $S = 2\log(M)$, $w_S = (N+1)u^{N^S} \bmod N^{S+1}$. Output $((N, w_S), \mathsf{ck}, c^*)$ to all parties.
2. Run internally the generation of one batch of $\mathsf{RoleBatches}^{\mathsf{eno}}$ protocol, letting all (virtual) parties play honestly and delivering all messages instantaneously, generating public and private data for the first batch of committees. Use the wait predicate $|\mathbb{Q}| = M$ such that all parties get output. Output the public data to all parties and hand the private data to the relevant committee members.
3. Let $P_1^c, \ldots, P_{2n}^c$ be the parties selected for the first committee pair in the previous step. Sample $(\mathsf{pv}^c, \mathsf{sv}_1^c, \ldots, \mathsf{sv}_{2n}^c) \leftarrow \mathsf{VSS}(d_S)$.[a] Output pv^c to all parties and sv_i^c to P_i^c.

[a] The VSS produces a public part pv and a secret part sv for each party, see Section 5.1 for details.

Fig. 5. Setup for implementing $\mathcal{F}_{\text{MPC+CF}}$

Setup In our implementation we use RA and MPC to implement threshold decryption, and threshold decryption to implement RA and MPC. To get off the ground we need that the Pailier public and secret keys have been generated, that there are some initial RA committees and that an initial committee is given a secret sharing of d_S. We assume this is all done in a setup phase modelled by $\mathcal{F}_{\text{SETUP}}$. To simplify the specification, we let $\mathcal{F}_{\text{SETUP}}$ do the set-up for the committees by running one batch of $\mathsf{RoleBatches}^{\mathsf{eno}}$ producing one batch of RA in its head and output the resulting data to the relevant parties. This is just a compact way of saying that we want set-up data in the same format that $\mathsf{RoleBatches}^{\mathsf{eno}}$ generates them, as this is also the format of the setup data that it consumes. This protocol is found in Fig. 12 and is used in the global protocol for setting up data for the next batch of committees.

Once the setup is done, all committee members share one-time pad key material with all parties they need to communicate with secretly, and these one-time pads are committed to the sending and receiving party. This allows committees to verifiably reshare d_S for the next committees.

5 Implementing $\mathcal{F}_{\text{MPC+CF}}$

In this section we give an implementation of $\mathcal{F}_{\text{MPC+CF}}$. In the following we will define several protocols of similar form: all members of a committee send a message of some form to $\mathcal{F}_{\text{TOB}}$. This message must include a zero-knowledge proof that the message was computed correctly, and a proof that the given party was indeed assigned the role that involves sending the

message. Looking ahead, this last proof will be an opening of a ciphertext that the party published earlier, and the proof is considered valid if the ciphertext contains a tag that was assigned to the role in question.

In all such cases, the wait predicate will say that $\mathcal{F}_{\text{TOB}}$ should wait for at least $n - t$ messages from the committee, of the correct form, where the proofs verify. Here, $t < n/2$ is the bound we assume on the number of corrupted players in the committee. In the protocol descriptions "the message is sent to $\mathcal{F}_{\text{TOB}}$" tacitly implies the above.

We use some basic subprotocols for secure computation on ciphertexts of form $E_{N,w_s}(m; r)$. They include protocols for secure multiplication on encrypted values and creating random encrypted numbers and bits. As these are mostly standard, we list them in Appendix H. We will use the following notation.

- $\mathsf{Multiply}(E_{N,w_s}(x_1), \ldots, E_{N,w_s}(x_l)) = E_{N,w_s}(x)$ is shorthand for invoking the multiplication protocol a number of times on the ciphertexts $E_{N,w_s}(x_i)$ to obtain $E_{N,w_s}(x)$ where $x = \prod_i x_i \bmod N^s$.

5.1 Protocols for Threshold Decryption and Resharing

The secret decryption exponent will be verifiable secret-shared among the members of a committee using *Linear Integer Secret Sharing* (LISS). Using LISS instead of the integer version of Shamir sharing will allow us to keep the size of shares from growing. For details on LISS and the VSS we use, see Appendix E.

A secret d will always be shared among the members of a pair of committees, namely it is additively shared among the members of the *additive committee*, and each additive share is shared with threshold t among the members of the *threshold committee*. The shares are computed from a sharing matrix M_{SH} and a *sharing vector* $\boldsymbol{v}_d$. This vector contains the secret as its first entry and the other entries are the randomness used for the sharing. The vector of shares will then be $\mathsf{sh}(d, \boldsymbol{v}_d) = M_{\text{SH}} \cdot \boldsymbol{v}_d$.

We number the shares in $\mathsf{sh}(d, \boldsymbol{v}_d)$ such that the first n entries are the additive shares. Further, for $1 \leq i \leq n$ and a qualified set A of the threshold committee, we let $\boldsymbol{r}_A^i$ denote the reconstruction vector that players in A can use to reconstruct the additive share $s_i = \mathsf{sh}(d, \boldsymbol{v}_d)[i]$, while $\boldsymbol{r}_A$ is the reconstruction vector used to reconstruct the secret itself. I_i contains the set of indices of elements in $\mathsf{sh}(d, \boldsymbol{v}_d)$ that are threshold shares of the additive share s_i. Finally, each player in the threshold committee gets several shares, we therefore use $u(i)$ to denote the index of the player holding the i'th share.

We say that a committee pair *holds* d_S if it is the case that all honest players in the committees have valid shares of d_S, according to some sharing vector $\boldsymbol{v}_{d_S}$, and if every share s_i has been committed to by $P_{u(i)}$ using a commitment α_i (this must hold even if $P_{u(i)}$ is corrupt).

Discussion of the Decryption Protocol For the implementation of threshold decryption, we assume a committee pair assigned to handle each batch of ciphertexts to decrypt. We maintain the invariant that when a committee pair is about to decrypt a batch, it holds d_S. This

Protocol RandomizeCiphertext Recall that c^* is the ciphertext from the global setup and H is the random oracle, here assumed to output a random number from $\mathbb{Z}_{N^{s+1}}$. Also, we assume L is a unique label assigned to this batch of ciphertexts[a]. When a batch to decrypt appears on the ledger, do the following for each ciphertext $\bar{c}$ to decrypt (in parallel):

1. Call the DecryptNoRandomize protocol on input $H(L)$ and let R be the resulting plaintext.
2. Output $c = \bar{c} \cdot \mathsf{Multiply}(c^*, H(\bar{c}, R)) \bmod N^{s+1}$.

[a] No randomness is required here, L could just be a counter, for instance.

Fig. 6. The RandomizeCiphertext protocol.

Protocol Decrypt
When a batch to decrypt appears on the ledger, the committee pair assigned to do the batch will execute the operations below. The committee pair is assumed to hold d_S.

1. For each ciphertext $\bar{c}$ in the batch, set $c = \mathsf{Randomize}(\bar{c})$.
2. For each member of the additive committee P_i, each $c = E_{N,w_s}(m)$ in the batch and share $s_i = \mathsf{sh}(d_S, \boldsymbol{v}_{d_S})[i]$, P_i sets:

$$d_{c,i} = c^{s_i} \bmod N^{s+1}$$
$$\pi_{c,i} = \mathsf{NIZK}(s_i, v_i : d_{c,i} = c^{s_i} \bmod N^{s+1}, \alpha_i = \mathsf{Com}_{\mathsf{ck}}(s_i; v_i)) \ ,$$

   and send the *decryption message* $(d_{c,i}, \pi_{c,i})$ to all parties.
3. All parties: Once $n - t$ decryption well-formed decryption messages have been received, run Π_{GATHER} (cf. Fig. 15) with the received set as input. The input is justified by consisting of at least $n - t$ well-formed decryption shares.
4. For each member P_u of the threshold committee: Let D be the set of all decryption messages finally received via Π_{GATHER}. Send D to all parties. In addition, for each additive share s_i for which a decryption message was not received, for each share s_j where $j \in I_i, u(j) = u$, and for each ciphertext c in the batch, set:

$$d_{c,i,j} = c^{s_j} \bmod N^{s+1} \ ,$$
$$\pi_{c,i,j} = \mathsf{NIZK}(s_j, v_j : d_{c,i,j} = c^{s_j} \bmod N^{s+1}, \ \alpha_j = \mathsf{Com}_{\mathsf{ck}}(s_j; v_j))$$

   and send the *back-up message* $(d_{c,i,j}, \pi_{c,i,j})$ to all parties.
5. All parties: Once messages from a subset A (of size at least $n-t$ parties) of the threshold committee have been received, then do the following for each ciphertext $c = E_{N,w_s}(m)$ in the batch: For each additive share s_i where a decryption message $d_{c,i}$ was not received, use the reconstruction vector $\boldsymbol{r}_A^i$ to compute

$$d_{c,i} = \prod_{j, P_{u(j)} \in A} d_{c,i,j}^{\boldsymbol{r}_A^i[j]} \bmod N^{s+1},$$

   then compute

$$\prod_{i=1}^{n} d_{c,i} = (N+1)^m \bmod N^{s+1}$$

   and compute m from this result.

Fig. 7. The Decrypt protocol.

is ensured for the first pair by $\mathcal{F}_{\text{SETUP}}$, and later by using the resharing protocol we discuss below.

We prove in Appendix I that the invariant is maintained and that hence the decryption protocol outputs correct plaintexts. The idea for decryption is that each member of the additive committee issues a decryption message by raising the ciphertext to its share. These messages are distributed to all parties using a protocol called Gather.[6] It ensures than when a set of n parties (of which at most t are corrupt) all try to send a message to all parties, all receivers will get messages from at least $n-t$ senders, and there will be a set of at least $n-t$ senders that all receivers have heard from, the so-called *core*. The parties might not agree on the core, as some parties might have heard from more parties than those in the core. For details, see Section B.6. Once the Gather protocol is done, we let the threshold committee supply back-up messages that allow reconstruction of all missing decryption messages. We use this set-up, rather than a single threshold committee, for technical reasons, in order to be able to show adaptive security using the so-called single inconsistent party technique.

Also for the sake of adaptive security a subprotocol RandomizeCiphertext (Fig. 6) is included. When decrypting values that serve as output from the global protocol, we call RandomizeCiphertext on the ciphertext to be decrypted before the actual decryption. This is a trick that allows us to "correct" the ciphertext that appears in the simulation. The simulation was run on dummy inputs, so the ciphertext likely contains the wrong plaintext. When simulating RandomizeCiphertext we let c^* be an encryption of 1 and we program the random oracle $H(\bar{c}, R)$ to output an encryption of the correction we need to add. If course, in the protocol c^* be an encryption of 0 and nothing is added.

For all other decryption operations, the call to RandomizeCiphertext is omitted, we denote this by DecryptNoRandomize. For input $\bar{c}$, Randomize($\bar{c}$) will denote the ciphertext output from this protocol[7].

The Reshare protocol. This protocol is described in detail in Appendix E. Here we sketch the main ideas. We want a committee pair C_1 that holds d_S to send secret messages to a receiving committee pair C_2 such that C_2 ends up holding d_S. For each threshold share s_i held by $P_{u(i)}$ in C_1, we ask $P_{u(i)}$ to verifiably secret share s_i among the members of C_2. This is possible because $P_{u(i)}$ is committed to s_i, so $P_{u(i)}$ will also commit to entries in a sharing vector and send resulting shares to each member in C_2 using one-time pad encryption (which we show later how to set up). Because the one-time pad is also committed, it can be shown in zero-knowledge that the shares sent are correct. Since the receiver is also committed to the one-time pad she can commit to the received share and prove that the commitment is correct.

We can assume that VSS messages are delivered on the ledger from at least $n-t$ parties and this is enough that receiving parties can do reconstruction on the received shares of the

[6] For efficiency, the decryption protocol does not use $\mathcal{F}_{\text{TOB}}$ to communicate its output, instead we borrow a more efficient subprotocol from the implementation of $\mathcal{F}_{\text{TOB}}$.

[7] This subprotocol uses the multiplication protocol, which in turn uses decryption. In order for this not to become circular, the decryptions in the multiplication protocol are done without the randomization step.

s_i's to get shares of d_S. By the homomorphic property of the commitment scheme we can get commitments to the new shares of d_S and so C_2 now holds d_S.

Note that for privacy, the shares of s_i need to be larger than s_i which in turn needs to be larger than d_S. So if we naively continue resharing this way, the size of the shares will grow indefinitely. But, as we show in the appendix, this can be solved by involving an auxiliary committee C_{AUX} that will VSS a random sufficiently large value R to both C_1 and C_2. Then members of C_1 will publicly reconstruct $d_S - R$ using linearity which allows the members of C_2 to adjust their shares of R into shares of d_S. Since the size of shares sent to C_2 only need to depend on the size of d_S, this keeps the share size constant.

5.2 Role Assignment

In this section we first show some basic building blocks and at the end of the section we show how to put them together to get role assignment.

The MPC version of the PIR Protocol. For our implementation of assigning roles to random parties, we will use a version of the PIR protocol where the client side is executed in YOSO MPC. For this, we make a useful observation: Say we are given 2 ciphertexts of form $E_{N,w_s}(E_{N,w_{s-1}}(m_1; r_1); u_1)$ and $E_{N,w_s}(E_{N,w_{s-1}}(m_2;_2 r); u_2)$. If we apply the multiplication protocol to these two ciphertexts, we get:

$$E_{N,w_s}(E_{N,w_{s-1}}(m_1; r_1) \cdot E_{N,w_{s-1}}(m_2; r_2) \bmod N^s; u_3) = E_{N,w_s}(E_{N,w_{s-1}}(m_1 + m_2; r_1 r_2); u_3),$$

thus implicitly adding the two underlying plaintexts.

Protocol MPCPIR

1. Take the next available set of encrypted bits
$$c_0 = E_{N,w_{s+1}}(t_0), \ldots, c_{\lambda-1} = E_{N,w_{s+\lambda}}(t_{\lambda-1})$$
from the ledger. Let t be the number represented by the bits $t_0, \ldots, t_{\lambda-1}$, where t_0 is the most significant bit.
Let $h_1, \ldots, h_{M'}$ be the input list of ciphertexts, all with N^s as plaintext space, i.e., they are all contained in $\mathbb{Z}^*_{N^{s+1}}$.
2. All parties executes the server side of the PIR protocol using $h_1, \ldots, , h_{M'}$ as the database. The output is a ciphertext $c = E_{N,w_{\lambda+s}}(E_{N,w_{\lambda+s-1}}(\ldots E_{N,w_{s+1}}(h_t)))$.
3. Set $\tilde{c}_{\lambda+s} = c$ and execute the following loop, for $i = \lambda + s, \lambda + s - 1, \ldots, s$:
   (a) Take the next $d_{i-1} = E_{N,w_i}(E_{N,w_{i-1}}(0; r); r')$ available on the ledger, and set $u_i = \mathsf{Multiply}(\tilde{c}_i, d_{i-1})$.
   (b) Send u_i to the Decrypt protocol and let $\tilde{c}_{i-1}$ be the result. We have $\tilde{c}_{i-1} = E_{N,w_{i-1}}(0; r) \cdot p_i \bmod N^i$, where p_i is the plaintext contained in $\tilde{c}_i$. However, p_i is itself a ciphertext, so $\tilde{c}_{i-1}$ is a random ciphertext with the same content as p_i.
4. The final ciphertext produced by the loop is a random ciphertext $\tilde{c}_s$ containing the same plaintext as h_t. Output $\tilde{c}_s$.

Fig. 8. The MPCPIR protocol.

We now design a protocol outputting a random and randomized ciphertext, taken from a global set of ciphertexts $h_1, ..., h_{M'}$, where each ciphertext has been broadcast by some party, and where each $h_i \in \mathbb{Z}_{N^{s+1}}$ for some s.

We will ensure that the communication complexity is sublinear in M'. We do this by modifying the PIR protocol from Section 3, so it can be executed by a set of committees, where we think of the h_j's as the database, and a (set of) committees play the role of the client.

More concretely, the high-level idea is that we first use the protocol for encrypted random bits to get what corresponds to the client's first message in the PIR protocol,

$$c_0 = E_{N,w_{s+1}}(t_0), \ldots, c_{\lambda-1} = E_{N,w_{\lambda+s}}(t_{\lambda-1}) \ ,$$

where the t_i's are random bits and in this application $\lambda = \log M'$. Note that we need the encryptions of the t_i's to have large enough plaintext space to accomodate the size of the h_j's. See Appendix H for the protocol for generating encrypted random bits.

Given that these ciphertexts are on the ledger, any party can consider the $h_1, \ldots, h_{M'}$ to be the database and locally execute the server's part of the PIR. This will result in a ciphertext $E_{N,w_{\lambda+s}}(E_{N,w_{\lambda+s-1}}(\ldots E_{N,w_{s+1}}(h_t)))$, containing a randomly chosen h_t. Since the server side of the PIR is deterministic, any honest party will arrive at the same ciphertext. However, it would be insecure to decrypt this, even partially. For instance, if we directly remove all but one layer of encryption, to get $E_{N,w_{s+1}}(h_t)$, this would reveal information on which element is chosen, as this encryption was computed in the first stage of the loop. Concretely, it is of form $E_{N,w_{s+1}}(h_t) = c_0^{h_j} \bar{c}_0^{h_{j+M'/2}}$, for some j, where h_t is either h_j or $h_{j+M'/2}$. An adversary can just try all possibilities for j.

We therefore need to randomize the encryption at each level h_t before decrypting anything. For this purpose, we assume that we have available on the ledger a sufficient number of random "two level" encryptions of 0, concretely random ciphertexts of form $E_{N,w_{i+1}}(E_{N,w_i}(0))$ for $i = s+1, \ldots, \lambda+s$. Such a set of ciphertexts can be used to randomize the involved ciphertexts before decryption.

Looking ahead, there will be several lists of ciphertexts coming from the same set of parties and we will need to select from each list, using the same random index for all of them. This is easy, by simply using the same set of encrypted index bits each time.

The CreateInputData protocol Our role assignment protocol below needs to start from the assumption that, for each player P in a set $\{P_1, \ldots, P_{M'}\}$, we are given a set of ciphertexts and commitments that have been published on the ledger by P. For this, we use the $\mathsf{CreateInputData}$ protocol, Fig. 9.

The predicate $\mathsf{pred}^{\text{SND}}(\mathsf{otp}_i^{\text{SND}}, v_i, r_i, c_i^{\text{SND}}, h_i^{\text{SND}})$ is satisfied if

$$h_i^{\text{SND}} = E_{N,w_S}(\mathsf{otp}_i^{\text{SND}}; v_i), \ c_i^{\text{SND}} = \mathsf{Com}_{\mathsf{ck}}(\mathsf{otp}_i^{\text{SND}}; r_i)$$

Likewise, $\mathsf{pred}^{\text{REC}}(\mathsf{otp}_i^{\text{REC}}, v_i, r_i, c_i^{\text{REC}}, h_i^{\text{REC}})$ is satisfied if

$$h_i^{\text{REC}} = E_{N,w_S}(\mathsf{otp}_i^{\text{REC}}; v_i'), \ c_i^{\text{REC}} = \mathsf{Com}_{\mathsf{ck}}(\mathsf{otp}_i^{\text{REC}}; r_i') \ .$$

Protocol CreateInputData
Each party P:

1. Choose tag at random in $\mathbb{Z}_{N^S}$. For $i = 1, \ldots, \mathsf{max}$ choose random $\mathsf{otp}_i^{\text{REC}}, \mathsf{otp}_i^{\text{SND}} \in \mathbb{Z}_{N^S}$.
2. Form ciphertexts $h = E_{N,w_S}(\mathsf{tag})$,
$$\{h_i^{\text{SND}} = E_{N,w_S}(\mathsf{otp}_i^{\text{SND}}; v_i), h_i^{\text{REC}} = E_{N,w_S}(\mathsf{otp}_i^{\text{REC}}; v_i') |\ i = 1, \ldots, \mathsf{max}\}$$
3. Form commitments
$$\{c_i^{\text{SND}} = \mathsf{Com}_{\mathsf{ck}}(\mathsf{otp}_i^{\text{SND}}; r_i), c_i^{\text{REC}} = \mathsf{Com}_{\mathsf{ck}}(\mathsf{otp}_i^{\text{REC}}; r_i')|\ i = 1, \ldots, \mathsf{max}\}.$$
4. Compute proofs $\pi_0, \{\pi_i^{\text{SND}}, \pi_i^{\text{REC}} \mid i = 1, \ldots, \mathsf{max}\}$ w.r.t. predicates $\mathsf{pred}^{\text{SND}}, \mathsf{pred}^{\text{REC}}$:
$$\begin{aligned}\pi_0 &= \mathsf{NIZK}(\mathsf{tag}, r_P : h = E_{N,w_S}(\mathsf{tag}_P; r_P))\\ \pi_i^{\text{SND}} &= \mathsf{NIZK}(\mathsf{otp}_i^{\text{SND}}, v_i, r_i : \mathsf{pred}^{\text{SND}}(\mathsf{otp}_i^{\text{SND}}, v_i, r_i, c_i^{\text{SND}}, h_i^{\text{SND}}))\\ \pi_i^{\text{REC}} &= \mathsf{NIZK}(\mathsf{otp}_i^{\text{REC}}, v_i', r_i' : \mathsf{pred}^{\text{REC}}(\mathsf{otp}_i^{\text{REC}}, v_i, r_i, c_i^{\text{REC}}, h_i^{\text{REC}}))\end{aligned}$$
5. Let $I_P = \mathsf{InputData}(\mathsf{tag}_P, \{\mathsf{otp}_i^{\text{SND}}, \mathsf{otp}_i^{\text{REC}} |\ i = 1, \ldots, \mathsf{max}\})$ denote the ordered set of ciphertexts, commitments and zero-knowledge proofs created above[a]. Erase the v_i, v_i' (but keep the otp's, tag_P, r_P and the r_i, r_i').
6. Send I_P to $\mathcal{F}_{\text{TOB}}$. Let the wait predicate be that $M - T$ such I_P appeared from distinct parties with valid proofs.

[a] for better readability, we suppress in the $\mathsf{InputData}$ function the randomness used in creating the data set.

Fig. 9. The CreateInputData protocol.

To explain the data created by P: If P is chosen for a role, P's tag will be decrypted, allowing P to detect that it has the role. The $\mathsf{otp}_i^{\text{SND}}, \mathsf{otp}_i^{\text{REC}}$'s are randomly chosen one-time pads that will allow P to send and receive data anonymously. The parameter max is chosen such that there are enough one-time pad material to accommodate the data to be sent to and from P. We fix max later, but we note already here that asymptotically it only depends on the size of the committees, and not on the total number of players.

The party P will later need to prove that it uses the pads correctly, and we will use the commitments for this, which is why P keeps the randomness for the commitments. We can maintain adaptive security since they can be equivocated. But the ciphertexts cannot, so the randomness for these is erased (except for the encryption of tag that never has to be equivocated).

The NewRole protocol. The NewRole protocol assumes that CreateInputData has been executed, and a set of M' parties have had their data delivered on the ledger, so we have as input data sets $I_1, \ldots, I_{M'}$.

It then uses the MPCPIR protocol to select a party P_t at random and output randomized versions of the ciphertexts published by P_t. This is done by running MPCPIR $\mathsf{max} + 1$ times in parallel with the same encrypted index bits, where the j'th call will use as database the list formed by taking the j'th ciphertext from each party. We get randomized ciphertexts $\tilde{h}, \{\tilde{h}_i^{\text{SND}}, \tilde{h}_i^{\text{REC}} \mid i = 1 \ldots \mathsf{max}\}$, where the first one contains P_t's tag. Finally, this tag is

Protocol $\mathsf{SecretChannels}(C_1, C_2)$

The protocol assumes that we have outputs from $\mathsf{NewRole}$ for all members of committee pairs C_1, C_2. For each member P_j of the threshold committee in C_1 and each member P_u in C_2, let $(\mathsf{tag}_j, \{\tilde{h}^{\text{SND}}_{i,j}, \tilde{h}^{\text{REC}}_{i,j} \mid i = 1, \ldots, \mathsf{max}\})$ be the output data for P_j and $(\mathsf{tag}_u, \{\tilde{h}^{\text{SND}}_{i,u}, \tilde{h}^{\text{REC}}_{i,u} \mid i = 1, \ldots, \mathsf{max}\})$ the output data for P_u. Then execute as follows:

Set up one time pads The following steps are repeated $\mathsf{max}_{pad} + 1$ times in parallel:

1. Take the next unused ciphertexts $\tilde{h}^{\text{SND}}_{i,j}, \tilde{h}^{\text{REC}}_{i,u}$ from the two sets of data and decrypt $\tilde{h}^{\text{SND}}_{i,j} \cdot \tilde{h}^{\text{REC}}_{i,u} \bmod N^{S+1}$ to get a number $\mathsf{sum}_{j,u}$.
2. P_j locally looks up the ciphertext h in its original data set from $\mathsf{CreateInputData}$ that corresponds to $\tilde{h}^{\text{SND}}_{i,j}$ (based on that ciphertext's position in the ordered list) and sets $\mathsf{otp}^{\text{SND}}_{i,j}$ to be the otp in h.
3. P_u locally looks up h in its original data set from $\mathsf{CreateInputData}$ that corresponds to $\tilde{h}^{\text{REC}}_{i,u}$ and sets $\mathsf{otp}^{\text{REC}}_{i,j}$ to be the otp in h.
4. All parties store $\mathsf{sum}_{j,u}$ as associated to the channel from P_j to P_u. P_u computes locally $\mathsf{otp}^{\text{SND}}_{i,j} = \mathsf{sum}_{j,u} - \mathsf{otp}^{\text{REC}}_{i,u} \bmod N^S$, where $\mathsf{otp}^{\text{REC}}_{i,u}$ is the one-time pad contained in $\tilde{h}^{\text{REC}}_{i,u}$. Note that $\mathsf{otp}^{\text{SND}}_{i,j}$ is also held by P_j, and so can be used for secret communication.
5. P_j assigns the commitment $c^{\text{SND}}_{i,j}$ to $\mathsf{otp}^{\text{SND}}_{i,j}$ that it created earlier to the channel from P_j to P_u. P_u creates a commitment $\mathsf{Com}_{\mathsf{ck}}(\mathsf{otp}^{\text{SND}}_{i,j}; r_{i,j})$ and assigns it to the channel from P_j to P_u, as well as a ZK proof $\pi_{i,j}$ that the commitment was correctly formed. This proof may be used later when P_u executes its role.

Select one-time pad Let otp_1 be the first shared pad set up previously. Both P_j and P_u call the random oracle to compute $H(\mathsf{otp}_1)$ and use this to select a random subset of the remaining max_{pad} one-time pads. The size of the subset equals the number of shares P_u holds in our secret-sharing scheme. Erase all one-time pads except the ones selected by $H(\mathsf{otp}_1)$.

Fig. 10. The SecretChannels protocol.

decrypted. As shorthand for a call to this protocol we write:

$$\mathsf{NewRole}(I_1, \ldots, I_{M'}) = (\mathsf{tag}, \{\tilde{h}^{\text{SND}}_i, \tilde{h}^{\text{REC}}_i \mid i = 1, \ldots, \mathsf{max}\}) \ .$$

The SecretChannels protocol We now specify the $\mathsf{SecretChannels}$ protocol, see Figure 10, used for setting up key material so that data can be sent secretly from one committee to another. The protocol assumes that two committee pairs C_1, C_2 have been formed by calling $\mathsf{NewRole}$ a sufficient number of times, such that for each committee member we have a set of outputs from that protocol: a tag and a set of encrypted one-time pads. The goal is to set up key material such that a message can be sent from each member of the threshold committee in C_1 to all members of C_2. More precisely, for each P_j in the threshold committee in C_1 and and each P_u in C_2, the two parties share some secret one-time pads and both parties are committed to the pads.

Note that, at the time the protocol is executed, the commitments will only be known to the involved parties, but when a party executes a role, it will become known which original data set $\mathsf{InputData}(\mathsf{tag}, \{\mathsf{otp}^{\text{SND}}_i, \mathsf{otp}^{\text{REC}}_i \mid i = 1..\mathsf{max}\})$ it created and now all parties can compute the relevant commitment.

The parameter max_{pad} is $O(n)$, the precise value is discussed in Appendix E. The protocol sets up many shared one-time pads and then deletes all but a subset of them. This seems strange but is done for technical reasons, to be able to prove adaptive security. As discussed

Protocol ExecuteRole
Each party P:

1. When the GenerateInputData is to be executed P generates a data set I_P as specified there and sends it to $\mathcal{F}_{\text{TOB}}$. In particular, I_P contains a ciphertext $c_P = E_{N,w_S}(\mathsf{tag}_P; r_P)$.
2. If I_P is delivered on $\mathcal{F}_{\text{TOB}}$, then for each execution of NewRole generating public output $\mathsf{out}_{\mathsf{tag}} = (\mathsf{tag}, \{\tilde{h}_i^{\text{SND}}, \tilde{h}_i^{\text{REC}} \mid i = 1, \ldots, \mathsf{max}\})$, check if $\mathsf{tag}_P = \mathsf{tag}$. If so, store a pointer to $\mathsf{out}_{\mathsf{tag}}$, as P now has the corresponding role.
3. When the SecretChannels protocol is executed, if it enters a state where $\mathsf{tag}_j = \mathsf{tag}$ or $\mathsf{tag}_u = \mathsf{tag}$, then execute the local computation specified in SecretChannels for P_j or P_u. As a result, P has a number of one-time pads for each connection to another party that it will need (for receiving shares of the secret Paillier key, or for sending it to the next committee), as well as a commitment to the pad.
4. When the data on the ledger indicates that P should execute its role, P first computes a proof r_P that it has the role associated to tag. The proof can be verified by anyone by checking that the ciphertext c_P as found in I_P on the ledger satisfies $c_P = E_{N,w_S}(\mathsf{tag}; r_P)$.
5. P executes its part of the Reshare protocol (from appendix E) in the role of a receiving party, that is, it uses its known one-time pads to decrypt ciphertexts sent by a previous committee and computes its shares of the secret key and commitments to them, as well as ZK proofs that the commitments contain correct shares.
6. P computes Dec, its contribution according to the Decrypt protocol for the set of ciphtertexts to be decrypted by the current commitee. If needed, P computes Contr, its contribution to MPC subprotocols such as Multiply.
7. Finally P computes a VSSmessage, as its contribution to the Reshare protocol in the role of the sending party. Here, the commitments from the SecretChannels protocol will be used.
8. P send $(\mathsf{tag}, r_P, \mathsf{Dec}, \mathsf{Contr}, \mathsf{VSSmessage})$ to $\mathcal{F}_{\text{TOB}}$.

Fig. 11. The ExecuteRole protocol.

above, we use a single inconsistent party technique based on rewinding, implying that during the proof the channel between two players may be used several times. To avoid reusing one-time pads, the simulator reprograms the random oracle after rewinding to select different one-time pads.

The Execute Role Protocol. The final ingredient we need is the specification of how parties execute roles, i.e., how a party should prepare for playing a role, and how the single message to be sent should be put together. This is described in the ExecuteRole protocol, Fig. 11. For better readability, we do not specify all parts in full detail. These details can be found in the other protocols referred to in ExecuteRole.

Protocol $\mathsf{RoleBatches}^{\mathsf{eno}}$ parametrised by a number M of parties, $\mathsf{Pal} = (\mathsf{Gen}, E, D)$, and a number eno which is an upper bound on the number of committees needed to run one batch of $\mathsf{RoleBatches}^{\mathsf{eno}}$.

Init:
1. Let $c > 0$ be a constant, called the honesty gap.
2. Let $M = (3 + c)T$ be the number of parties, where T is the maximal tolerable corruption.
3. Let $n = \lambda$ and let $\phi = \ell 2n$ be the largest multiple of $2n$ such that $\phi \leq (c/2)T$. Assume that $\ell = \mathsf{eno}$. We later discuss how to handle when this is not the case.
4. Learn (N, w_S) from $\mathcal{F}_{\text{SETUP}}$.
5. As below, define eno committees each with a secret sharing of d_S from the output of $\mathcal{F}_{\text{SETUP}}$.
6. Each $P \in \mathbb{P}$ lets $\mathsf{bo}_P = 0$ and $\mathsf{bd}_P = 0$.

Next Batch: On input (NEXTBATCH) to P, where $\mathsf{bo}_P = \mathsf{bd}_P$, update $\mathsf{bo}_P \leftarrow \mathsf{bo}_P + 1$ and:
1. Run CreateInputData to have each P send I_P on $\mathcal{F}_{\text{TOB}}$.
2. Wait for valid I_P to appear from $P \in \mathbb{Q}$ with $|\mathbb{Q}| = M - T$.
3. Let $\mathbb{P}^{\mathsf{bo}_P} \leftarrow \mathbb{Q}$. Let $\mathbb{P}^{\mathsf{bo}_P} = \{P_1, \ldots, P_{M'}\}$.
4. For $j = 1, \ldots, \ell = \lambda|\mathbb{Q}|$, run $(\mathsf{tag}_j, \{\tilde{h}_i^{\text{SND}}, \tilde{h}_i^{\text{REC}} \mid i = 1, \ldots, \mathsf{max}\}) \leftarrow \mathsf{NewRole}(I_1, \ldots, I_{M'})$.
5. If there are not $|\mathbb{P}^{\mathsf{b}}|$ unique values tag_j, then terminate. Otherwise, sort the outputs of NewRole lexicographically on tag and for each unique tag map the first occurrence $(\mathsf{tag}, \cdot)$ unto a unique role $R \in \mathbb{P}^{\mathsf{b}}$.
6. The above gives $M - T$ roles R with unique $(\mathsf{tag}_R, \{\tilde{h}_{R,i}^{\text{SND}}, \tilde{h}_{R,i}^{\text{REC}} \mid i = 1, \ldots, \mathsf{max}\})$. Take the first ϕ of these roles and use them to form $\ell = \mathsf{eno}$ committees $C_1^{\mathsf{bd}_P}, \ldots, C_{\mathsf{eno}}^{\mathsf{bd}_P}$ of size $2n$. Break each into a threshold committee and an additive committee of sizes n.
7. Run $\mathsf{SecretChannels}(C_{\mathsf{eno}}^{\mathsf{bd}_P - 1}, C_1^{\mathsf{bd}_P})$ and for $c = 1, \ldots, \mathsf{eno} - 1$, run $\mathsf{SecretChannels}(C_c^{\mathsf{bd}_P}, C_{c+1}^{\mathsf{bd}_P})$.
8. Let $\mathsf{bd}_P \leftarrow \mathsf{bd}_P + 1$.

Fig. 12. The protocol for one batch of Role Assignment.

Iterative Role Assignment We finally describe the protocol $\mathsf{RoleBatches}^{\mathsf{eno}}$ for assigning roles for our MPC protocol. For each batch, we first run CreateInputData, so each party P creates a data set I_P containing ciphertexts, commitments and zero-knowledge proofs and broadcasts it. Later, when P executes its role by sending a message, it will include the randomness for the ciphertext h in I_P so it can be verified that tag_i contained in h is indeed the tag assigned to the role in question. Intuitively, this prevents corrupt parties from taking over someone else's role. To do this, you must make a ciphertext containing someone else's tag, and for any given batch, nothing is decrypted until all relevant ciphertexts are on the ledger.

While running CreateInputData, we wait only for $M' = M - T$ contributions to not deadlock. Let $\mathbb{Q}$ be the set of parties that got their contributions delivered. We then do $\lambda(M-T)$ calls to NewRole. Each call selects implicitly a random anonymous party P in $\mathbb{Q}$, and the call also outputs a tag tag which cannot be linked to P but was chosen randomly by P in the previous phase. Since honest parties will choose colliding tags with negligible probability, we can use this to throw away duplicates, such that each P gets one role. If there are less than $M-T$ unique tags, then we abort. The probability that any given $P \in \mathbb{Q}$ is not hit after $\lambda(M-T)$ tries is the negligible $(1-(M-T)^{-1})^{\lambda(M-T)}) \to e^{-\lambda}$, so we can ignore it.

Note that we let $n = \lambda$ and let $\phi = \ell 2n$ be the largest multiple of $2n$ such that $\phi \leq (c/2)T$. If $\ell <$ eno this will not give use eno committees. In this case we simply run the five first steps m times in parallel for $m\ell \geq$ eno. When we run a protocol like NewRole some m times in parallel we use the same committees in each run, so it consumes a number of committees constant in m. Ergo, $\mathsf{RoleBatches}^{\mathsf{eno}}$ consumes a number of committees constant in eno. We can therefore set eno large enough to generate enough committees for running the next batch without circularity in how we compute eno. In fact, we set eno sufficiently larger than for this to have sufficiently many committees left over in order to do the batches of MPC function evaluation in $\mathcal{F}_{\text{MPC+CF}}$.

The reason why we only use $(c/2)M$ out of $M-T$ roles to form committees is that once we used and executed $(c/2)M-1$ roles and revealed which parties were behind them, there are still $(M-T)-(c/2)T \geq (2+(c/2))T$ parties left unrevealed. So, even if the adversary concentrates its corruptions on this set it can corrupt the party behind the last unrevealed role of the last committee with probability at most $1/(2+(c/2))$. Therefore, by a Chernoff bound, the probability that $\geq (1/2)\lambda$ parties out of the $n = \lambda$ parties on a committee are corrupted before they executed their role is $\text{negl}(\lambda)$. Hence each committee has honest majority of executed roles except with negligible probability, as required for all our sub-protocols.

5.3 Putting the Pieces Together

Let $\mathcal{F}_{\text{MPC}}$ be $\mathcal{F}_{\text{MPC+CF}}$ with the Flip Coin and Deliver Coin commands removed. To implement $\mathcal{F}_{\text{MPC}}$ parties run a number of RoleBatches. Some of the generated roles are used for running the next RoleBatches, while some are used for running the MPC part of $\mathcal{F}_{\text{MPC}}$. To supply input to $\mathcal{F}_{\text{MPC}}$ we extend CreateInputData such that parties contribute their party input and role input by sending $E_{N,w_1}(x_i)$ and $E_{N,w_1}(z_i)$ along with the usual encryptions of tag and otp's. All encryptions are augmented by proofs of plaintext knowledge. The role inputs $E_{N,w_1}(z_i)$ are permuted along with the encryptions of otp's by running NewRole with the same random index on the set of encryptions $E_{N,w_1}(z_i)$. The encrypted inputs x_i and permuted encryptions of tag, otp, and z_i now enter the MPC. The order of the encryptions of tag, otp and z_i define how they are mapped onto roles and therefore the permutation π in $\mathcal{F}_{\text{MPC}}$. We then use the sub-protocols for random bits, adding, and multiplying to compute a circuit evaluating f^{b} on the contributed inputs, producing encrypted outputs. The public output y^{b} is decrypted. For private output going to a role, we add the output to the relevant otp inside the encryption and decrypt. This defines a protocol π_{MPC} implementing $\mathcal{F}_{\text{MPC}}$, the proof of this fact is found in Appendix I.

We the construct a protocol π_{CF} implementing $\mathcal{F}_{\text{MPC+CF}}$ in the $\mathcal{F}_{\text{MPC}}$-hybrid model. This is done by securely computing a function f which divides the parties into committees and robustly secret shares a random value onto each committee. To flip the coin the committee members all send their shares and verification values to all other parties. Details and proof are found in Appendix J. By UC composition we then have an implementation of $\mathcal{F}_{\text{MPC+CF}}$.

In this implementation, we assume we have access to an instance of $\mathcal{F}_{\text{TOB}}$. In Section 6 we show how to implement $\mathcal{F}_{\text{TOB}}$ from an ideal functionality $\mathcal{F}_{\text{CF}}$ for coin-flip, which in turn will just be the coin-flip part of $\mathcal{F}_{\text{MPC+CF}}$. This seems cyclical, but importantly, since we open coin-flips by reconstructing robust secret sharings, the implementation of *openings* of coin-flips on $\mathcal{F}_{\text{MPC+CF}}$ does not use $\mathcal{F}_{\text{TOB}}$. We can therefore let $\mathcal{F}_{\text{MPC+CF}}$ implement a surplus of coin-flips in batch b. During the implementation of batch $\mathsf{b}+1$ these can be used to implemented the instance of $\mathcal{F}_{\text{TOB}}$ used in $\mathsf{b}+1$. The implementation of $\mathcal{F}_{\text{MPC+CF}}$ works equally well if each batch uses a separate $\mathcal{F}_{\text{TOB}}$. This finally gives a protocol which we call $\mathsf{MPCCF}^{\mathsf{Pal}}$, and from the theorems proved in Appendix I and J, we get:

Theorem 1. *When for a constant c at most $T < M/(3+c)$ parties are adaptive corrupted and we set $n = \lambda$ then for a large enough constant eno we have that if Pal is CSO-secure (Definition 3), then $\mathsf{MPCCF}^{\mathsf{Pal}}$ securely implements $\mathcal{F}_{MPC+CF}^{\mathsf{Pal},1/3,\gamma}$ in the $(\mathcal{F}_{SETUP}, \mathcal{F}_{TOB}, \mathcal{F}_{ATOMIC\text{-}SEND})$-model with a random oracle. Here γ can be any polynomial. If $M \geq 2\mathsf{eno}$ and we use the $\mathcal{F}_{TOB}$ implementation from Section 6 then the amortized communication in bits to generate one committee of size $n = \lambda$ is $M \operatorname{poly}(\lambda, \log M)$. If $\Omega(M \operatorname{poly}(\lambda, \log M))$ multiplication gates are handled in parallel, the amortized complexity of a secure multiplication is $M \operatorname{poly}(\lambda)$.*

As for the communication note that in each basic run we generate $(c/2)T = \Theta(M)$ roles. To do this we use a number of committees which is $\operatorname{poly}(\lambda, \log M)$ as we have a constant number of runs of parallel protocols each with $\operatorname{poly}(\lambda, \log M)$ rounds. Each committee consists of λ parties, so we consume $\operatorname{poly}(\lambda, \log M)$ roles. To generate $\Theta(M)$ roles a total of $O(M)\operatorname{poly}(\lambda, \log M)$ group elements are posted on $\mathcal{F}_{\text{TOB}}$. Therefore the amortized number of group elements posted on $\mathcal{F}_{\text{TOB}}$ per generated committee is $O(\lambda)\operatorname{poly}(\lambda, \log M) = \operatorname{poly}(\lambda, \log M)$, and the same holds for generating $\mathsf{eno} \in \operatorname{poly}(\lambda, \log M)$ committees. Since our implementation of $\mathcal{F}_{\text{TOB}}$ uses communication $O(ML) + \operatorname{poly}(\lambda, \log M)$ to broadcast L bits to all parties, the amortized number of bits sent and received by each party to generate eno committees is $\operatorname{poly}(\lambda, \log M)$.

Handling one multiplication gate uses a constant number of committees and requires broadcasting $\operatorname{poly}(\lambda)$ bits. If sufficiently many multiplications are done in parallel in the theorem, the broadcasts require communication $O(M \operatorname{poly}(\lambda))$ bits and the cost of generating the committees will "amortize away".

The reason why we generate $\ell = \lambda|\mathbb{Q}|$ roles in Item 4 in **Next Batch** is that we need each party to be sampled at least once for roles which are outputs of $\mathcal{F}_{\text{MPC}}$. For roles only used for committee formation to run $\mathsf{RoleBatches}$, this is an overkill, as we throw away $M - T - \phi = \Theta(M)$ roles. It is enough to ensure that ϕ roles are hit at least once. When $M = \Theta(\lambda)$ this allows to reduce the overhead from λ to $O(1)$ using a simple concentration bound. We discuss this in Appendix A.

6 Consensus

We present a protocol Π_{TOB} securely implementing $\mathcal{F}_{\text{TOB}}$. For implementing $\mathcal{F}_{\text{TOB}}$ we need an ideal functionality for asynchronous coin-flip $\mathcal{F}_{\text{CF}}$. Like for the coin related sub-interface of $\mathcal{F}_{\text{MPC+CF}}$, when the first honest party asks for the next coin it is flipped and shown to the adversary, and after the last honest party asks for the next coin, the adversary must eventually deliver the coin to all honest parties. For later convenience we assume $\mathcal{F}_{\text{CF}}$ has a command (coin-index) telling a party P_i how many coins it received from $\mathcal{F}_{\text{CF}}$ so far.

6.1 Implementing $\mathcal{F}_{\mathsf{TOB}}$ from $\mathcal{F}_{\mathsf{CF}}$

We give a high-level description of the protocol Π_{TOB} instantiating $\mathcal{F}_{\text{TOB}}$. It sequentially runs instances of an Agreement on Core Set (ACS) protocol Π_{ACS} where the inputs of each party satisfies the wait predicate. The ACS protocol is heavily inspired by the one in [KN23], but with several changes to make it YOSO. The protocol Π_{TOB} uses small committees, but we do not use role assignment to sample these committees. The role assignment establishes private channels to future roles which is are not needed by Π_{TOB}. We therefore use simple self-nomination from VRFs as in [GHM+17]. This is concretely much more efficient. We elaborate in Appendix B.1. The decentralized nature of sampling the roles for Π_{TOB} via self-nomination means that the committees will be of variable size, and that there is no straightforward way to talk about the index of a role on the committee. Since the ACS protocol in [KN23] uses the fact that each party corresponds to an integer in $[M]$ to describe its causal past via M-bit vectors and to elect a leader using a $\log M$-bit coin, it cannot simply be instantiated with the self-nominated committees without some structural changes.

6.2 Agreement on Core Set

In Appendix B we present a modified version of the protocol for ACS from [KN23]. Beyond some syntactical changes, we make the following three changes:

1. For each instance of an activation rule the set of parties who "speak" is a committee specifically elected for that role. This is implemented using sortition on a unique identifier associated with that instance of the activation rule as described in Appendix B.1. We do not explicitly define the identifiers, but a natural choice is to concatenate the protocol name, session identifier, and a unique activation rule name.
2. The underlying Causal Cast (CC) primitive is instantiated using a novel RB protocol in Appendix B.3. This allows the CC protocol to give output to all parties, as we by design do not know who is in the next committee.
3. Leader election is separated from CC and implemented directly in the context of graded block selection (Appendix B.7) using an extra round of communication and the coin-flip functionality $\mathcal{F}_{\text{CF}}$ in Fig. 29. The resulting protocol has the property that if a party selects a block with grade 2, then *all possible justified outputs* (cf. Appendix B.2) in the following round have grade 2.

4. By using the justified grade of the block selected in the preceding round, as justifier that we did not yet terminate, we make sure that all parties terminate in adjacent rounds, and that the round number becomes a justified output of the protocol.

The first three changes make the ACS protocol YOSO and compatible with self-nominated committees. The final change allows a substantial optimization in the amount of setup that needs to be recomputed when instantiating $\mathcal{F}_{\text{CF}}$ with a YOSO protocol that requires setup for each coin flip.

Many natural instantiations of the coin flip requires a setup to be computed for each round of the protocol. The number of rounds can be bounded by $O(\lambda)$ but only an expected constant number of setups are actually used. So there is a multiplicative factor $O(\lambda)$ overhead on the communication and computation required to compute setups. We cannot *a priori* repurpose the unused setups for later rounds, because a party cannot determine from its local view, whether another honest party requested a coin and thus leaked it to the adversary. Exposing the justified output round will allow us to reach agreement on an upper bound on the number of setups that could have been used in each iteration by supplying them as input to the next round. This will in turn allow reducing the number of coin flipping setups being consumed by each decision of Π_{TOB} to expected constant.

6.3 Total-Order Broadcast

We present a protocol Π_{TOB} implementing $\mathcal{F}_{\text{TOB}}$ with ledgers, blocks and wait predicates as defined in Section 4. A straightforward implementation would be to have parties who want to broadcast a message on the TOB send the message to all parties, have an elected committee collect these messages and then, when their local blocks satisfy the wait predicate, propose them in Π_{ACS}. But as each message could be included by multiple proposers this would result in a worst-case multiplicative communication overhead of $O(n)$. Instead we will have each party who wants to broadcast a message on the TOB send the message through reliable broadcast with a $O(\log M)$ bit message identifier, mid, and then have the proposers include the message in their block by referring to the message using mid. Each block proposer will also add the round number it got as part of the output of Π_{ACS} in the previous round to its block. After agreement on a subset of the blocks is reached, we can take the minimum output round, r, included in the set of blocks and via Adjacent Output Round property conclude that no honest party participated in round $r + 2$ or later. Thus the corresponding coins remain unpredictable to the adversary and their setups can be repurposed.

We require that all blocks have the same size up to a constant factor. Otherwise, the communication in each round could be dominated by a large block which does not make it into the core. To get around this issue, we require that all blocks include references to between $(\max(\alpha, W_{\#}))$ and $2(\max(\alpha, W_{\#}))$ messages, which means that the size of blocks that get included in the core in each epoch is not asymptotically dominated by the remaining blocks. For the concrete complexity analysis in Theorem 3 we assume that α is at least λ.

Theorem 2. *When for a constant c at most $T < M/(3 + c)$ parties are adaptive corrupted and we sample committees as in Lemma 2, then we have that Π_{TOB} implements $\mathcal{F}_{TOB}$ in the $\mathcal{F}_{CF}$-model.*

Protocol Π_{TOB} described from the view of party P. We use the definitions of ledger, blocks and wait predicates from Section 4.

Init: Each party P initialises the empty ledger $L_{\mathsf{P}} = \epsilon$, a broadcast index keeping track of how many messages were broadcast by P, $c_{\mathsf{P}} = 0$, a batch index keeping track of how many wait predicates are set for P, $b_{\mathsf{P}} = 0$, and a set of dispersed messages pending inclusion in L, $\mathsf{Pending}_{\mathsf{P}} = \emptyset$. It also initialises two instances of the coin functionality $\mathcal{F}_{\mathrm{CF}}^{0}$ and $\mathcal{F}_{\mathrm{CF}}^{1}$ and corresponding coin counters $\mathsf{coin}_0 = 0$ and $\mathsf{coin}_1 = 0$.

Broadcast message: On input $(\textsc{Broadcast}, \mathsf{mid}, m)$ party $P(\mathsf{mid})$ starts an instance of Π_{RB} on the message m with session identifier $(\textsc{Broadcast}, \mathsf{mid}, c_{\mathsf{P}})$ with the input justifier that Π_{RB} has given output for all sessions $(\textsc{Broadcast}, \mathsf{mid}, c)$ with $c \in [1; c_{\mathsf{P}})$. Finally, it lets $c_{\mathsf{P}} = c_{\mathsf{P}} + 1$.

Schedule message: On output m from an instance of Π_{RB} with session identifier $(\textsc{Broadcast}, \mathsf{mid}, c)$ add $(P(\mathsf{mid}), c, m)$ to $\mathsf{Pending}_{\mathsf{P}}$.

Set Wait Predicate: On input $(\textsc{wait}, W)$ let $b_P \leftarrow b_P + 1$ and let $W^{b_P} = W$.

Deliver: When P has output (C, r) from Π_{ACS} with session identifier $|L| + 1$ and for each pair $(\mathsf{P}', c) \in C$ there is a an entry in $\mathsf{Pending}_{\mathsf{P}}$ of the form (P', c, m) it does the following: Adds each of the messages m (ignoring duplicates) in order to a block which is added to L_{P} and removes the corresponding entries from $\mathsf{Pending}_{\mathsf{P}}$. Lets $\mathsf{LocalOutputRound}_{|L|+1} = r$, and lets r' be the minimal justified output round included in C. It then adds $r' + 1$ to $\mathsf{coin}_{|L|+1 \mod 2}$ and inputs $(\textsc{next-coin})$ to $\mathcal{F}_{\mathrm{CF}}^{|L|+1 \mod 2}$ until querying it for input $(\textsc{coin-index})$ returns $\mathsf{coin}_{|L|+1 \mod 2}$.[a]

Propose Block: When $|\mathsf{Pending}_{\mathsf{P}}| \geq \max(W_{\#}^{|L_{\mathsf{P}}|+1}, \alpha)$ and $W^{|L_{\mathsf{P}}|+1}(L_{\mathsf{P}}, \mathsf{Pending}_{\mathsf{P}}) = \top$, P starts running Π_{ACS} with session id $|L_{\mathsf{P}}|+1$. The input block is defined as follows: We say an element $(\mathsf{P}', c, m) \in \mathsf{Pending}_{\mathsf{P}}$ is referenced by (P', c). Party P computes a block B consisting of references to at most $W_{\#}^{|L_{\mathsf{P}}|+1}$ messages in $\mathsf{Pending}_{\mathsf{P}}$ where $W^{|L_{\mathsf{P}}|+1}(L_{\mathsf{P}}, B)$[b] and remove those messages from $\mathsf{Pending}_{\mathsf{P}}$. Then add references to the $\max(W_{\#}^{|L_{\mathsf{P}}|+1}, \alpha, \mathsf{Pending}_{\mathsf{P}})$ oldest messages from $\mathsf{Pending}_{\mathsf{P}}$ to a block B' and remove those messages from $\mathsf{Pending}_{\mathsf{P}}$. Finally let B'' be a block including only the locally observed output round from the previous batch, $\mathsf{LocalOutputRound}_{|L_{\mathsf{P}}|}$ and let $B''' = B \| B' \| B''$ be the input to Π_{ACS}. The block is justified by consisting of a justified output round number from the previous iteration of Π_{ACS} and references to messages that are sent through RB in the step above, these messages satisfying the wait predicate and the number of references being in the interval $[\max(W_{\#}^{|L_{\mathsf{P}}|+1}, \alpha); \max(W_{\#}^{|L_{\mathsf{P}}|+1}, \alpha)]$.

[a] In plain English, it uses the agreement on how many coins were consumed in past iterations to skip past any coins that it has not used but which potentially could have been used by other parties, so that all parties are synchronised when they start flipping coins in the next instance of Π_{ACS}.

[b] This can be computed efficiently as described in Section 4.

Fig. 13. Total-Order Broadcast

Proof. Follows from Theorem 4, Theorem 3, and Lemma 2.

References

Bai16. Leemon Baird. The swirlds hashgraph consensus algorithm: Fair, fast, byzantine fault tolerance. SWIRLDS TECH REPORT SWIRLDS-TR-2016-01, 2016. `https://www.swirlds.com/downloads/SWIRLDS-TR-2016-01.pdf`.

BKLZL20. Erica Blum, Jonathan Katz, Chen-Da Liu-Zhang, and Julian Loss. Asynchronous byzantine agreement with subquadratic communication. pages 353–380, 2020.

BLZLN22. Amey Bhangale, Chen-Da Liu-Zhang, Julian Loss, and Kartik Nayak. Efficient adaptively-secure byzantine agreement for long messages. pages 504–525, 2022.

Bra87. Gabriel Bracha. Asynchronous byzantine agreement protocols. *Inf. Comput.*, 75(2):130–143, nov 1987.

CCL15. Ran Canetti, Asaf Cohen, and Yehuda Lindell. A simpler variant of universally composable security for standard multiparty computation. pages 3–22, 2015.

CDN01. Ronald Cramer, Ivan Damgård, and Jesper Buus Nielsen. Multiparty computation from threshold homomorphic encryption. pages 280–299, 2001.

DF02. Ivan Damgård and Eiichiro Fujisaki. A statistically-hiding integer commitment scheme based on groups with hidden order. In *Advances in Cryptology—ASIACRYPT 2002: 8th International Conference on the Theory and Application of Cryptology and Information Security Queenstown, New Zealand, December 1–5, 2002 Proceedings 8*, pages 125–142. Springer, 2002.

DFK+06. Ivan Damgård, Matthias Fitzi, Eike Kiltz, Jesper Buus Nielsen, and Tomas Toft. Unconditionally secure constant-rounds multi-party computation for equality, comparison, bits and exponentiation. In *Theory of Cryptography Conference*, pages 285–304. Springer, 2006.

DJN10. Ivan Damgård, Mads Jurik, and Jesper Buus Nielsen. A generalization of paillier's public-key system with applications to electronic voting. *International Journal of Information Security*, 9:371–385, 2010.

DN03. Ivan Damgård and Jesper Buus Nielsen. Universally composable efficient multiparty computation from threshold homomorphic encryption. pages 247–264, 2003.

DT06. Ivan Damgård and Rune Thorbek. Linear integer secret sharing and distributed exponentiation. In *International Workshop on Public Key Cryptography*, pages 75–90. Springer, 2006.

DXR21. Sourav Das, Zhuolun Xiang, and Ling Ren. Asynchronous data dissemination and its applications. pages 2705–2721, 2021.

Fis05. Marc Fischlin. Communication-efficient non-interactive proofs of knowledge with online extractors. In *Annual International Cryptology Conference*, pages 152–168. Springer, 2005.

FO99. Eiichiro Fujisaki and Tatsuaki Okamoto. Statistical zero-knowledge protocols to prove modular polynomial relations. *IEICE transactions on fundamentals of electronics, communications and computer sciences*, 82(1):81–92, 1999.

GHK+21. Craig Gentry, Shai Halevi, Hugo Krawczyk, Bernardo Magri, Jesper Buus Nielsen, Tal Rabin, and Sophia Yakoubov. YOSO: You only speak once - secure MPC with stateless ephemeral roles. pages 64–93, 2021.

GHM+17. Yossi Gilad, Rotem Hemo, Silvio Micali, Georgios Vlachos, and Nickolai Zeldovich. Algorand: Scaling byzantine agreements for cryptocurrencies. Cryptology ePrint Archive, Report 2017/454, 2017. `https://eprint.iacr.org/2017/454`.

GHM+21. Craig Gentry, Shai Halevi, Bernardo Magri, Jesper Buus Nielsen, and Sophia Yakoubov. Random-index PIR and applications. pages 32–61, 2021.

KKNS21. Idit Keidar, Eleftherios Kokoris-Kogias, Oded Naor, and Alexander Spiegelman. All you need is DAG. In Avery Miller, Keren Censor-Hillel, and Janne H. Korhonen, editors, *PODC '21: ACM Symposium on Principles of Distributed Computing, Virtual Event, Italy, July 26-30, 2021*, pages 165–175. ACM, 2021.

KN23. Simon Holmgaard Kamp and Jesper Buus Nielsen. Byzantine agreement decomposed: Honest majority asynchronous total-order broadcast from reliable broadcast. *IACR Cryptol. ePrint Arch.*, page 1738, 2023.

LN23. Julian Loss and Jesper Buus Nielsen. Early stopping for any number of corruptions. *IACR Cryptol. ePrint Arch.*, page 1813, 2023.

Nie02. Jesper Buus Nielsen. Separating random oracle proofs from complexity theoretic proofs: The non-committing encryption case. pages 111–126, 2002.

A Concrete Optimisations

The reason why we generate $\ell = \lambda|\mathbb{Q}|$ keys in Item 4 in **Next Batch** in Fig. 12 is that we insist that each $P \in \mathbb{Q}$ needs to be sampled at least once in the PIR so that it gets a role. This gives a clearer analysis but is very wasteful. We run $\mathsf{NewRole}$ $\ell = \lambda|\mathbb{Q}|$ times to output only $|\mathbb{Q}|$ roles, an overhead of λ. Note, however, that if one generates $|\mathbb{Q}|$ keys instead of $\lambda|\mathbb{Q}|$ keys, then a given P gets a role with constant positive probability. An honest and corrupted parties in $\mathbb{Q}$ are hit with equal probability. Furthermore, we can use the tags to compute how many parties got a role key. Repeating this in sequence until there are $|\mathbb{Q}|$ unique parties which got a role key will take expected $O(\log|\mathbb{Q}|)$ rounds. And we have that $O(\log|\mathbb{Q}|) = O(\log M) = O(\log\lambda)$, as M is polynomial in λ. This replaces an overhead λ by $\log\lambda$. Furthermore, for the roles used to form committees in **Renew Setup** we only use $\phi \leq (c/2)T$ roles out of the $M - T = (2 + (c/2))T$ roles generated. There is no reason to generate role keys for the parties in $\mathbb{Q}$ which are not used to form committees. If we generate $|\mathbb{Q}| = M - T$ random role keys, then by a Chernoff bound, we will have $(c/2))T$ unique roles except with negligible probability, saving an additional $\log M$ factor for internal committees.

B Justified ACS with Adjacent Output Rounds and Player Replaceable Committees

We adapt the protocol ACS protocol from [KN23] to the YOSO setting. The high-level changes are described in Section 6.2.

B.1 Sampling Committees using Cryptographic Sortition

Our TOB protocol does not rely on private channels and the committees can simply self-nominate using the cryptographic sortition implementation from [GHM+17], which is described in terms of parties who each have a weight w and where the sum of corrupted weight must be less than a third of the total weight W by some constant fraction. In short a party P_i who has w_i units of the total weight W computes $(h, \pi, j) = \mathsf{Sortition}(\mathsf{sk}_i, \mathsf{seed}, n, \mathsf{role}, w, W)$ to privately check how many parties they are emulating on the committee for role, and other parties can verify this using $\mathsf{VerifySort}(\mathsf{pk}_i, h, \pi, \mathsf{seed}, n, \mathsf{role}, w, W)$. Assuming the seed was chosen before the secret key of each user, the probability that each unit of weight gets to emulate a role is $\frac{n}{W}$. We present the protocol in the standard setting of M parties of equal weight, by simply fixing $w_i \coloneqq 1$ for all P_i and $W \coloneqq M$. Since the assignment of each party (or equivalently unit of weight) to a role is drawn independently from a Bernoulli distribution we can apply lemma 24 from [BKLZL20] which we restate (slightly simplified).

Lemma 2 (Lemma 24 from [BKLZL20]). *If $n \leq M$, $0 < \epsilon < 1/3$, and $T \leq (1-2\epsilon)M/3$ bounds the number of corruptions, then a committee C_{role} sampled as above satisfies the following except with probability negligible in n:*

1. *C_{role} contains fewer than $(1+\epsilon)\cdot n$ parties.*
2. *C_{role} contains more than $((1+\epsilon/2)\cdot 2\cdot n)/3$ honest parties.*

3. C_{role} *contains fewer than* $(1-\epsilon)\cdot n/3$ *corrupted parties.*

In the following protocols we will be using $\mathsf{P} \in \mathsf{C}_{\mathsf{role}}$ as shorthand for a party P who was elected for the committee of role and who implicitly sends proofs of this along with messages. This does not affect communication complexity as the parties are already sending a $O(\lambda)$ bit hash. Additionally we use n as the expected size of $\mathsf{C}_{\mathsf{role}}$, and assume that $t \coloneqq (1-\epsilon)\cdot n/3$ is an upper bound on the number of corrupted parties on $\mathsf{C}_{\mathsf{role}}$. We can think of n as a statistical security parameter. Additionally, except for the committees used to implement RB, the committees only need to have honest majority, so we can have concretely smaller committees in those instances. For simplicity we let n be security parameter λ.

B.2 Eventual Justifiers

We will use the definition of Eventual Justifiers from [KN23]. An eventual justifier is a predicate evaluated on a message and a party's local view. They are required to be monotone and propagating. Monotone in the sense that a party seeing a message as justified in their view, does not at some later point consider it to not be justified. Propagating in the sense that a message being justified at one party means that it will eventually be justified at all honest parties. A recent example of justifiers being used in a synchronous model include [LN23].

Definition 4 (Justifier [KN23]). *For a message identifier* mid *we say that* J^{mid} *is a justifier if the following properties hold.*

Monotone: *If for an honest* P *and some time* τ *it holds that* $J^{\mathsf{mid}}(m, \mathsf{P}, \tau) = \top$ *then at all* $\tau' \geq \tau$ *it holds that* $J^{\mathsf{mid}}(m, \mathsf{P}, \tau') = \top$.

Propagating: *If for honest* P *and some point in time* τ *it holds that* $J^{\mathsf{mid}}(m, \mathsf{P}, \tau) = \top$, *then eventually the execution will reach a time* τ' *such that* $J^{\mathsf{mid}}(m, \mathsf{P}', \tau') = \top$ *for all honest parties* P'.

The justifiers are often used as an explanation for why a party sent a particular message. It is natural to implement this by considering the predicate satisfied when a subset of the messages you received would prompt you to send the same message if you were performing the same role as the sender. Most of the protocol definitions in this section will have justifiers on inputs and outputs, meaning that all inputs and outputs can be guaranteed to have certain properties. This holds even for adversarial inputs and outputs, in the sense that if they are accepted as justified in the view of an honest party, then they satisfy some predicate. We say in that case that all *possible justified outputs* of a protocol satisfy the predicate. Similarly, most protocol messages will come with justifiers which are used to reason about all *possible justified messages* of some type satisfying some property. In many cases this means that an adversary might lie about what message they should have sent in the protocol, but it will still be a message that can be explained as something an honest party would have sent based on a valid sequence of events, and therefore a message that is as good as what an honest party would have sent. We refer to [KN23] for a formal definition of possible justified messages and outputs. Combining justifiers with RB means that Byzantine parties cannot equivocate

and have to give a explanation for why they send each message, which in many cases can combine to make an adversarial message have all the relevant properties we require of an honest one.

B.3 Reliable Broadcast for long messages

We present a protocol for reliable broadcast (RB) it has the usual properties of Bracha's RB[Bra87].

Definition 5 (Reliable Broadcast). *A protocol for M parties $\mathsf{P}_1, \ldots, \mathsf{P}_M$, where all parties have input mid. The message identifier mid contains the identity of a sender P_s.*

Validity: *If honest P_s has input (mid, m) and an honest P_i has output (mid, m') then $m' = m$.*

Agreement: *For all honest outputs (mid, m) and (mid, m') it holds that $m = m'$.*

Eventual Output 1: *If P_s is honest and has input $(\mathsf{mid}, \cdot)$, and all honest P_j start running the protocol, then eventually all honest P_i have output $(\mathsf{mid}, \cdot)$.*

Eventual Output 2: *If an honest P_j has output $(\mathsf{mid}, \cdot)$, and all honest parties start running the protocol, then eventually all honest P_i have output $(\mathsf{mid}, \cdot)$.*

Our protocol for subquadratic and message length optimal RB follows the blueprint of algorithm 4 by Das et al. [DXR21], but relies on a distinct self-nominated committee to perform each activation rule. At a high level: after receiving the message from the designated sender the remaining parties essentially run Bracha's RB protocol on a hash of the message while distributing shares of a Reed-Solomon encoding of the message. Each party P_i is supposed to receive the i^{th} share of the encoding from each party sending an echo message. So if all honest parties receive the same message from the sender, then they forward a matching hash and shares to each other party. It follows that if any message is supported by the echo messages of a supermajority (making it unique), then any honest party P_i who sends a ready message includes the i^{th} share of that message. We restate it in Fig. 14 to illustrate how it can have subquadratic communication when $T < (1-\epsilon)M/3$ by instantiating it with committees.

We only need to observe that it is YOSO (i.e., each committee member sends only one message) and that when $T < (1-\epsilon)M/3$ we can by Lemma 2 sample committees of size $O(\lambda)$ where at most n parties are corrupted and more than twice as many are honest. Then lowering the degrees of the polynomials used in the Reed-Solomon code to t means that from any set of distinct shares greater than $3t$ of which at most t shares are incorrect one can reconstruct the message. To be able to send a distinct share to all M parties we need to pick the polynomials of the Reed-Solomon code to be over a field that is larger than M. Concretely the messages just need to be of size $\log(M) \cdot \lambda$ to dominate the total combined sized of the points sent by the committee, resulting in message length optimality, but since all parties will be including a cryptographic hash, a signature, and the output of a VRF from sortition in their messages, the dominating cost will be λ parties sending λ bits to M parties. This lowers the communication complexity of broadcasting an $|m|$ bit message from $O(M(|m| + M\lambda))$ in [DXR21] to $O(M(|m| + \lambda^2))$.

Protocol Π_{RB}

Send: P_s sends m to all parties.

Echo: On receiving the first valid message m_i from P_s each party $\mathsf{P}_i \in \mathsf{C}_{\mathsf{echo}}$ computes the Reed-Solomon encoding and hash of their value $\mathsf{D}_i = (s_1, \ldots, s_n) = \mathsf{Encode}(m_i)$, $h_i = \mathsf{H}(m_i)$, and sends $(\mathsf{echo}, s_j, h_i)$ to each party $\mathsf{P}_j \in \mathcal{P}$.

Ready 1: On receiving messages of the form $(\mathsf{echo}, s_i^1, h)$ with the same values of s_i^1 and h from $n-t$ distinct parties in $\mathsf{C}_{\mathsf{echo}}$ each party $\mathsf{P}_i \in \mathsf{C}_{\mathsf{ready}}$ who has not yet sent a ready message sends $(\mathsf{ready}, s_i^1, h)$ to all parties.

Ready 2: On receiving messages of the form $(\mathsf{ready}, \cdot, h)$ from $t+1$ distinct parties in $\mathsf{C}_{\mathsf{ready}}$ and messages of the form $(\mathsf{echo}, s_i^2, h)$ from $t+1$ distinct parties $\mathsf{C}_{\mathsf{echo}}$ with the same hash h and share s_i^2, each party $\mathsf{P}_i \in \mathsf{C}_{\mathsf{ready}}$ who has not yet sent a ready message sends $(\mathsf{ready}, s_i^2, h)$ to all parties.

Output: On receiving messages of the form $(\mathsf{ready}, s'_j, h)$ from at least $n-t$ distinct parties each party P tries to reconstruct from the shares received. To reconstruct, P first removes any shares from parties sending more than one share and then, if reconstruction using the remaining shares is successful, outputs the result and terminates. This step is repeated each time a new ready message is received, until successful reconstruction.

Fig. 14. Reliable Broadcast

The main insight is that if two committees with honest supermajority perform the echo and ready roles, and the reconstruction threshold of the Reed-Solomon code is less than a third of the committee size, the proof still goes through: If an honest party, P, terminates because they saw $n-t$ ready messages, then (as in Bracha's original protocol) $t+1$ honest parties sent ready, and at least one of them, P', did so because they saw $n-t$ echo messages. At least $t+1$ of matching echo messages seen by P' came from honest parties, and the messages of these $t+1$ honest parties will eventually arrive at the remaining honest parties (with a different set of matching shares). So now every honest party will eventually see $t+1$ echo messages (from the parties who sent echo to P') with the same hash and same share, $t+1$ ready messages (from the honest parties who sent ready to P) with the same hash, allowing them to reconstruct the message from the shares in the consistent ready messages, and send their own share in a ready message. All honest parties sending a ready message in turn allows every honest party to terminate.

B.4 Causal Cast

The concept of Causal Cast (CC)[KN23] is an abstraction over DAG based protocols that utilize the structure of a DAG to infer what a party would have said in a protocol that they are in an abstract sense running without directly sending the messages. It can be thought of as a tool to describe protocols in this paradigm (notable examples include [Bai16,KKNS21]) without needing to explicitly consider the structure of the DAG. Using the terminology of CC a message of a party in the protocol is a *computed message* when it can be inferred by pointing to previous messages instead of explicitly sending the message. This concept was pioneered and dubbed "virtual voting" in [Bai16]. If a message, such as a message in a block, needs to be introduced to the DAG, then it is instead a *free-choice message*. For motivation of the remaining concepts we refer to [KN23]. What is important for our purposes is that

CC is used black box in [KN23] and that we can implement it in the YOSO model by giving YOSO implementations of Reliable Broadcast and $\mathcal{F}_{\text{CF}}$, which are in turn used black box to implement CC in [KN23]. We provide a YOSO RB in Appendix B.3 and assume access to an ideal coin functionality $\mathcal{F}_{\text{CF}}$. However, there is the caveat that the ideal coin functionality does not immediately provide an implementation of Leader Election, because the committees are sampled using sortition. We show how to get around this hurdle in Appendix B.7.

Definition 6 (Causal Cast[KN23]). *A protocol for M parties $\mathsf{P}_1, \ldots, \mathsf{P}_M$ is called a Causal Cast (CC) if it has the following properties.*

Free-Choice Send: *A party P_i can have input $(\text{CC-SEND}, \mathsf{mid}, m)$ where mid is a free choice identifier $\mathsf{P}_i = \mathsf{P}^{\mathsf{mid}}$ and $J_{IN}^{\mathsf{mid}}(m) = \top$ at $\mathsf{P}^{\mathsf{mid}}$ at the time of input.*

Computed-Message Send: *A party P_i can have input $(\text{CC-SEND}, \mathsf{mid}, m, \mathsf{mid}_1, \ldots, \mathsf{mid}_\ell)$, where mid is a computed-message identifier, $\mathsf{P}_i = \mathsf{P}^{\mathsf{mid}}$, P_i earlier gave outputs $(\text{CC-DEL}, \mathsf{mid}_j, m_j)$ for $j = 1, \ldots, \ell$, and*

$$\bot \neq m = \mathsf{NextMessage}^{\mathsf{mid}}((\mathsf{mid}_1, m_1), \ldots, (\mathsf{mid}_\ell, m_\ell)) \ .$$

Constant Send: *A party P_i can have input $(\text{CC-SEND}, \mathsf{mid}, m)$ where mid is a constant identifier. In that case it is guaranteed that all parties eventually have the same input $(\text{CC-SEND}, \mathsf{mid}, m)$.*

Free-Choice Validity: *A party P_i can have output $(\text{CC-DEL}, \mathsf{mid}, m)$, where mid is a free-choice identifier. It then holds that $J_{IN}^{\mathsf{mid}}(m) = \top$ at P_i at the time of output. Furthermore, if $\mathsf{P}_j = \mathsf{P}^{\mathsf{mid}}$ is honest, then P_j had input $(\text{CC-SEND}, \mathsf{mid}, m)$.*

Coin Flip Validity: *A party P_i may output $(\text{CC-DEL}, \mathsf{mid}, m)$ where mid is a coin-flip identifier mapping to an instance of $\mathcal{F}_{CF}$ as defined in Fig. 29 and the index of a coin: ℓ. In that case P_i has previously received output $\mathcal{F}_{CF}.L[\ell]$ from $\mathcal{F}_{CF}$.*

Computed-Message Validity: *A party P_i can have output $(\text{CC-DEL}, \mathsf{mid}, m, \mathsf{mid}_1, \ldots, \mathsf{mid}_\ell)$, where mid is a computed-message identifier. In that case P_i earlier gave outputs $(\text{CC-DEL}, \mathsf{mid}_j, m_j, \ldots)$ for $j = 1, \ldots, \ell$, and $\bot \neq m = \mathsf{NextMessage}^{\mathsf{mid}}((\mathsf{mid}_1, m_1), \ldots, (\mathsf{mid}_\ell, m_\ell))$.*

Constant Validity: *A party P_i can have output $(\text{CC-DEL}, \mathsf{mid}, m)$. In that case it immediately before had input $(\text{CC-SEND}, \mathsf{mid}, m)$.*

Liveness: *If an honest party P_i had input $(\text{CC-SEND}, \mathsf{mid}, \ldots)$ or some honest party had output $(\text{CC-DEL}, \mathsf{mid}, \ldots)$ and all honest parties are running the system, then eventually all honest parties have output $(\text{CC-DEL}, \mathsf{mid}, \ldots)$.*

Agreement: *For all possible justified outputs $(\text{CC-DEL}, \mathsf{mid}, m, \ldots)$ and $(\text{CC-DEL}, \mathsf{mid}, m', \ldots)$ it holds that $m' = m$.*

We will also be using the notion of Justified Causal Cast protocols ([KN23]) in which outputs are associated with a message identifier mid and can be sent as a computed message $(\text{CC-SEND}, \mathsf{mid}, m, \mathsf{mid}_1, \ldots, \mathsf{mid}_\ell)$, in which case the message identifiers $\mathsf{mid}_1, \ldots, \mathsf{mid}_\ell$ justify the output. For a Justified Causal Cast protocol Π we will use $\Pi.J_{\text{OUT}}$ to denote its output justifier. This will be useful for reporting justified outputs of subprotocols.

Remark 1 (Honest Majority Committees). For the remaining protocols in this section we only need "honest majority" committees, by which we mean that the following holds except with probability negligible in n:

1. $\mathsf{C}_{\mathsf{role}}$ contains fewer than $(1+\epsilon)\cdot n$ parties.
2. $\mathsf{C}_{\mathsf{role}}$ contains more than $((1+\epsilon)\cdot n)/2$ honest parties.
3. $\mathsf{C}_{\mathsf{role}}$ contains fewer than $(1-\epsilon)\cdot n/2$ corrupted parties.

This is implied by the bounds in Lemma 2, but in practice allows sampling committees that are concretely smaller by picking an appropriately smaller n and letting $t \coloneqq (1-\epsilon)\cdot n/2$.

B.5 Justified Gather

We restate the definition Justified Gather (Definition 7) and the protocol Π_{GATHER} (Fig. 15) which implements it. Nothing changes from [KN23] apart from notation and each activation rule being performed by parties who self-nominate using sortition as described in Appendix B.1.

Definition 7 (Justified Gather). *A protocol for M parties $\mathsf{P}_1, \ldots, \mathsf{P}_M$. There is an input justifier J_{IN} and an output justifier J_{OUT} specified by the protocol. All honest P_i have an input B_i for which $J_{IN}(B_i) = \top$ at P_i at the time the input is given.*

Liveness: *If all honest parties start running the protocol with a J_{IN}-justified input then eventually all honest parties have a J_{OUT}-justified output.*

Justified Blocks: *For all possible justified outputs U and all (potentially corrupt) P_i and all $(\mathsf{P}_i, B_i) \in U$ it holds that $J_{IN}(B_i) = \top$.*

Validity: *For all possible justified outputs U and all honest P_i and all $(\mathsf{P}_i, B_i) \in U$ it holds that P_i had input B_i.*

Agreement: *For all possible justified outputs U and U' and all $(\mathsf{P}_i, B_i) \in U$ and $(\mathsf{P}_i, B_i') \in U'$ it holds that $B_i = B_i'$.*

Large Core: *For all possible justified outputs $(U^1, \ldots, U^m)$ it holds that $|\bigcap_{k=1}^{m} U^k| \geq n - t$.*

The proof that Π_{GATHER} is a Justified Gather protocol is presented in [KN23]. The only change is that the dimensions of the table T and the combinatorial argument changes from using a fixed committee size n_{C} and corruption bound t_{C} with $t_{\mathsf{C}} < n_{\mathsf{C}}/2$ to reasoning about committees of random but bounded size elected using sortition. Concretely, let $n_{\mathsf{Gather;2}}$ be the number of parties in $\mathsf{C}_{\mathsf{Gather;2}}$ and $n_{\mathsf{Gather;3}}$ be the number of parties in $\mathsf{C}_{\mathsf{Gather;3}}$, then the table will have $n_{\mathsf{Gather;2}}$ rows and $n_{\mathsf{Gather;3}}$ columns. But this does not change the conclusion, as in either case the $n-t$ sets included in the unions will (except with negligible probability) be more than half than the maximal actual committee size for all other committees in the protocol. This holds even with the bounds in Remark 1.

B.6 Justified Graded Gather

We restate the definition of a Justified Graded Gather protocol from [KN23] in Definition 8 and the protocol, $\Pi_{\text{GRADEDGATHER}}$ implementing it in Fig. 16.

1. The input of P_i is B_i with $J_{\text{IN}}(B_i) = \top$. Party P_i lets $U_i^0 = \{(\mathsf{P}_i, B_i)\}$. The singleton set is justified by B_i satisfying J_{IN}.
2. For $r \in [1;3]$ each party $\mathsf{P}_i \in \mathsf{C}_{\mathsf{Gather;r\text{-}1}}$ causal casts U_i^{r-1} as a computed message. Then each party P_i collects incoming U_j^{r-1} from parties $\mathsf{P}_j \in \mathsf{C}_{\mathsf{Gather;r\text{-}1}}$, lets P_i^r be the set of P_j it heard from, waits until $|P_i^r| \geq n - t$ and lets
$$U_i^r = \bigcup_{\mathsf{P}_j \in P_i^r} U_j^{r-1}.$$
The message is justified by being computed from the set P_i^r where $|P_i^r| \geq n - t$.
3. Finally, P_i outputs U_i^3.

Fig. 15. Protocol Π_{GATHER}.

Definition 8 (Justified Graded Gather). *A protocol for M parties $\mathsf{P}_1, \ldots, \mathsf{P}_M$. There is an input justifier J_{IN} and an output justifier J_{OUT} specified by the protocol. All honest P_i have an input B_i for which $J_{\text{IN}}(B_i) = \top$ at P_i at the time the input is given.*

Liveness: *If all honest parties start running the protocol with a J_{IN}-justified input then eventually all honest parties have a J_{OUT}-justified output.*

Justified Blocks: *For all possible justified outputs (U, T) and all (potentially corrupt) P_i and all $(\mathsf{P}_i, B_i) \in U$ it holds that $J_{\text{IN}}(B_i) = \top$.*

Sub Core: *For all possible justified outputs $((U^1, T^1), \ldots, (U^m, T^m))$ it holds that $T^i \subseteq \bigcap_{k=1}^m U^k$ for all $i \in [m]$.*

Validity: *For all possible justified outputs (U, T) and all honest P_i and all $(\mathsf{P}_i, B_i) \in U$ it holds that P_i had input B_i.*

Agreement: *For all possible justified outputs (U, T) and (U', T') and all $(\mathsf{P}_i, B_i) \in S$ and $(\mathsf{P}_i, B_i') \in U'$ it holds that $B_i = B_i'$.*

Large Sub Core: *For all possible justified outputs $((U^1, T^1), \ldots, (U^m, T^m))$ it holds that $|\bigcap_{k=1}^m T^k| \geq n - t$.*

As in Appendix B.5 besides the committee being self-nominating nothing substantial changes, and the proof follows from the one in [KN23] because the bounds in Remark 1 imply intersection between any two subsets of the committee of size $n - t$.

1. The input of P_i is B_i with $J_{\text{IN}}(B_i) = \top$. All parties run Π_{GATHER} with P_i inputting B_i justified by J_{IN}. Let the output of P_i be U_i'. If $\mathsf{P}_i \in \mathsf{C}_{\mathsf{GradedGather}}$ it then causal casts U_i' as a computed-message justified by $\Pi_{\text{GATHER}}.J_{\text{OUT}}$.
2. Party P_i collects U_j' from parties $\mathsf{P}_j \in \mathsf{C}_{\mathsf{GradedGather}}$, lets P_i be the set of P_j it heard from and waits until $|P_i| \geq n - t$.
3. Party P_i outputs
$$(U_i, T_i) = \Big(\bigcup_{\mathsf{P}_j \in P_i} U_j' \, , \quad \bigcap_{\mathsf{P}_j \in P_i} U_j' \Big).$$
The outputs are justified by being computed as above from justified sets.

Fig. 16. $\Pi_{\text{GRADEDGATHER}}$

B.7 Justified Graded Block Selection

Justified Graded Block Selection as defined in [KN23] allows a set of parties to input a justified block and get one as output alongside a grade with the following properties:

Definition 9 (Justified Graded Block Selection[KN23]). *A protocol for M parties $\mathsf{P}_1, \ldots, \mathsf{P}_M$. There is an input justifier J_{IN} and an output justifier J_{OUT} specified by the protocol. All honest P_i have an input B_i for which $J_{\text{IN}}(B_i) = \top$ at the time the input B_i is given. The output of the protocol is a block C_i justified by J_{OUT}.*

Liveness: *If all honest parties start running the protocol with a J_{IN}-justified input then eventually all honest parties have a J_{OUT}-justified output.*
Justified Output: *$J_{\text{IN}}(C_i) = \top$ holds for all possible J_{OUT}-justified outputs C_i.*
Graded Agreement: *For all possible justified outputs (C_i, g_i) and (C_j, g_j) it holds that $|g_i - g_j| \leq 1$. Furthermore, if both $g_i, g_j > 0$ then $C_i = C_j$.*
Positive Agreement: *There exists $\alpha > 0$ such that with probability at least $\alpha - \text{negl}$ all possible justified outputs of at least $n - t$ parties will have grade $g_i = 2$.*
Stability: *If there are possible justified outputs (C_i, g_i) and (C_j, g_j) with $C_i \neq C_j$ then there exist two justified inputs B_i and B_j with $B_i \neq B_j$.*

We will start out by presenting a protocol, $\Pi_{\mathsf{WeakGradedSelectBlock}}$, with a weakened version of the Positive agreement property:

Weak Positive Agreement There exists $\alpha > 0$ such that with probability at least $\alpha - \text{negl}$ some honest P_i will have output (C_i, g_i) with $g_i = 2$.

This will in turn be used to implement a full fledged Justified Graded Block Selection protocol with a strengthened Stability property in Appendix B.8.

$\Pi_{\mathsf{WeakGradedSelectBlock}}$ is presented in Fig. 17. It is using the core principles from the corresponding primitive in [KN23] but needs a few modifications to function in our setting where the committees are self-nominated. The main challenge is that parties do not have a description of the committee. They do not even know its exact size, so we cannot in a straightforward manner map a random string to a member of the committee. We will instead for each committee member we have seen in our accumulated set, feed the coin output together with their party identifier through the random oracle, H, and obtain a string which was unpredictable before the core of the accumulated sets were fixed. We will locally regard the party who has the lexicographically least string as a leader candidate and then gossip candidates to get graded agreement on a leader and their block. If it happens that the committee member with the least string is in a sub core, T_i, then P_i gets a grade 2 output. Due to the Large Sub Core property of $\Pi_{\text{GradedGather}}$ this happens with good constant probability as at least one honest T_i is fixed when the first honest party initiates gives input to $\mathcal{F}_{\text{CF}}$.

When instantiating this $\Pi_{\mathsf{WeakGradedSelectBlock}}$ as shown in Fig. 17 with $\mathcal{F}_{\text{CF}}$ from Fig. 29, then it satisfies Definition 9 with Weak Positive Agreement.

Lemma 3. *$\Pi_{\mathsf{WeakGradedSelectBlock}}$ satisfies Justified Graded Block Selection Definition 9 with Weak Positive Agreement.*

1. The input of P_i is B_i with $J_{\text{IN}}(B_i) = \top$.
2. The parties run $\Pi_{\text{GRADEDGATHER}}$ with input B_i and input justifier J_{IN}. Let the output of P_i be (U_i, T_i).
3. After getting output from $\Pi_{\text{GRADEDGATHER}}$ the parties input NEXT-COIN to $\mathcal{F}_{\text{CF}}$ and then COIN-INDEX to get the corresponding coin identifier ℓ.
4. On output $(\ell, coin)$ from $\mathcal{F}_{\text{CF}}$ if $\mathsf{P}_i \in \mathsf{C}_{\mathsf{FirstRoundCandidate}}$: for each $\mathsf{P}_j \in U_i$ let $ticket_j = \mathsf{H}(\mathsf{P}_j \| coin)$, let P_k be the one with the lexicographically least $ticket_k$ value, and send $(\mathsf{FirstRoundCandidate}, \mathsf{P}_k)$ as a Causal Cast message computed from the set U_i and $coin$.
5. On receiving $n - t$ $(\mathsf{FirstRoundCandidate}, \mathsf{P}_k)$ messages from parties in $\mathsf{C}_{\mathsf{FirstRoundCandidate}}$ each $\mathsf{P}_i \in \mathsf{C}_{\mathsf{SecondRoundCandidate}}$: If all the relayed P_k are identical lets $b_i = \top$, and otherwise $b_i = \bot$ and finally sends $(\mathsf{SecondRoundCandidate}, \mathsf{P}_k)$ as a Causal Cast computed message based on the set of received $\mathsf{FirstRoundCandidate}$ messages.
6. On receiving $n-t$ $(\mathsf{SecondRoundCandidate}, b_j)$ messages from parties in $\mathsf{C}_{\mathsf{SecondRoundCandidate}}$ each P_i: lets n_i be the number of messages where $b_j = \top$, if $n_i > 0$ lets P_k be a party included in $n-t$ $\mathsf{FirstRoundCandidate}$ messages, and outputs

$$(C_i, g_i) = \begin{cases} (B_k, 2) & \text{if } n_i \geq n - t \wedge \exists(\mathsf{P}_k, B_k) \in T_i \\ (B_k, 1) & \text{if } n_i > 0 \wedge \exists(\mathsf{P}_k, B_k) \in U_i \setminus T_i \\ (B_i, 0) & \text{if } \nexists(\mathsf{P}_k, \cdot) \in U_i \vee n_i = 0\ . \end{cases}$$

   and if it is on the committee $\mathsf{C}_{\mathsf{GradedSelectBlock}}$ Causal Casts (C_i, g_i). The output is justified by being computed as above from justified values.

Fig. 17. $\Pi_{\mathsf{WeakGradedSelectBlock}}$

Proof. Liveness and Justified Output are trivial. Stability holds because the output justification transitively refers to a justified input through computed messages. We argue Graded Agreement: Assume a party P_i has output $(B_k, 2)$. Then $n_i \geq n-t$ and $B_k \in T_i$. Now for any other party P_j the Sub Core property of $\Pi_{\text{GRADEDGATHER}}$ ensures that $B_k \in U_j$, and because all second round messages were sent through CC (i.e. without equivocations) $n_i \geq n-t$ implies $n_j \geq n-2t > 0$. So, we have taken care of $|g_i - g_j| \leq 1$ as P_j must now have grade at least 1. (If no party has grade 2 there is nothing to show.) Now consider any party P_i with output grade at least 1. This party had $n_i > 0$, and thus received at least one $(\mathsf{SecondRoundCandidate}, \top)$ message justified by $n-t$ $(\mathsf{FirstRoundCandidate}, \mathsf{P}_k)$ messages on the same party P_k which by intersection and the messages being sent through CC implies that no other party P_l can be included in $n-t$ $\mathsf{FirstRoundCandidate}$ messages, which in turn means that only B_k can get a grade of 1 or 2. We finally argue Weak Positive Agreement: When modelling H as a random oracle: except with negligible probability there are no collisions among the outputs of H, and the probability that the party, P_i, which has the lexicographically least $ticket_i$ value globally in a set S, is in any subset of size $c|S|$ where the subset is independent from $coin$ is $c - \text{negl}$. Let P_j be the first honest party to give input NEXT-COIN to $\mathcal{F}_{\text{CF}}$. In particular P_j already had output (T_j, U_j) from $\Pi_{\text{GRADEDGATHER}}$ while the value of $coin$ was unpredictable, so an adversary cannot have chosen T_j to correlate with $coin$. By the Large Sub Core property of $\Pi_{\text{GRADEDGATHER}}$ the intersection of all possible justified T values has size at least $n - t$. In particular, the probability that P_i has the lexicographically least $ticket_i$ value is in T_j is

at least $\frac{n-t}{t}$ − negl.[8] Assume $\mathsf{P}_i \in T_j$. Then by Sub Core $\mathsf{P}_i \in U_k$ for all possible justified U_k and thus $(\mathsf{FirstRoundCandidate}, \mathsf{P}_i)$ is the only justifiable $\mathsf{FirstRoundCandidate}$ message, which means that $(\mathsf{SecondRoundCandidate}, \top)$ is the only justifiable $\mathsf{SecondRoundCandidate}$ message and the output of P_j must be $(B_i, 2)$.

B.8 Justified Strongly Stable Graded Block Selection

For our construction it will be useful to make sure that when instances of a Justified Graded Block Selection are run sequentially with the outputs being fed back as justified inputs to the next iteration, then whenever a party gets an output with grade 2 all other parties receive grade 2 in the next iteration. This is ensured by adding the following Strong Stability property.

Definition 10 (Justified Strongly Stable Graded Block Selection). *A Justified Graded Block Selection protocol that additionally satisfies the following Strong Stability property.*

Strong Stability: *If there is a possible justified output (C_i, g_i) with $g_i < 2$ then there exist two justified inputs B_i and B_j with $B_i \neq B_j$.*

Given a protocol $\Pi_{\mathsf{WeakGradedSelectBlock}}$ satisfying Definition 9 with Weak Positive Agreement we construct $\Pi_{\mathsf{StronglyStableGradedSelectBlock}}$ by adding two rounds of Causal Cast in Fig. 18 and show that it satisfies Definition 10.

Protocol $\Pi_{\mathsf{StronglyStableGradedSelectBlock}}$

1. The input of P_i is B_i with $J_{\text{IN}}(B_i) = \top$.
2. Party P_i runs $\Pi_{\mathsf{WeakGradedSelectBlock}}$ with B_i as input using $J_{\text{IN}}(B_i)$ as justifier, and gets output $(B_i^1, \cdot)$.
   If $\mathsf{P}_i \in \mathsf{C}_{\mathsf{Upgrade1}}$ it causal casts $(\mathsf{Upgrade1}, B_i^1)$ justified by $\Pi_{\mathsf{WeakGradedSelectBlock}}.J_{\text{OUT}}$.
3. Party P_i collects justified messages $(\mathsf{Upgrade1}, B_j^1)$ from at least $n-t$ parties $\mathsf{P}_j \in P_i^1 \subseteq \mathsf{C}_{\mathsf{Upgrade1}}$ and lets

$$(B_i^1, h_i) = \begin{cases} (B, 1) & \text{if } \exists B : |\{\mathsf{P} \in P_i^1 | \mathsf{P} \text{ sent } (\mathsf{Upgrade1}, B)\}| \geq n - t \\ (\perp, 0) & \text{otherwise.} \end{cases}$$

   If $\mathsf{P}_i \in \mathsf{C}_{\mathsf{Upgrade2}}$ it causal casts $(\mathsf{Upgrade2}, B_i^2, h_i)$ justified by P_i^1.
4. Party P_i collects justified messages $(\mathsf{Upgrade2}, B_j^2, h_j)$ from at least $n-t$ parties $\mathsf{P}_j \in P_i^2 \subseteq \mathsf{C}_{\mathsf{Upgrade2}}$.

$$(C_i, g_i) = \begin{cases} (B, 2) & \text{if } \forall \mathsf{P} \in P_i^2 : \mathsf{P} \text{ sent } (\mathsf{Upgrade2}, B, 1) \\ (B, 1) & \text{if } \exists \mathsf{P} \in P_i^2 : \mathsf{P} \text{ sent } (\mathsf{Upgrade2}, B, 1) \\ (B_i, 0) & \text{if } \forall \mathsf{P} \in P_i^2 : \mathsf{P} \text{ sent } (\mathsf{Upgrade2}, \perp, 0) \ . \end{cases}$$

   Output (C_i, g_i) with output justifier P_i^2.

Fig. 18. $\Pi_{\mathsf{StronglyStableGradedSelectBlock}}$

[8] Note that while we informally refer to the committees as having honest majority, Remark 1 in fact specifies that the honest parties outnumber the adversary by a number which is a constant fraction of the committee size, so the probability of terminating is constant in every round.

Lemma 4 (Strong Stability). *$\Pi_{\mathsf{StronglyStableGradedSelectBlock}}$ is a Justified Strongly Stable Graded Block Selection protocol.*

Proof. Liveness and Justified Output is trivial. To get soft grade $h = 1$ one has to see the same block from a majority. Since all block are sent through RB, at most one block can have votes from a majority. It follows that blocks with grade $g > 0$ are identical as they justified by at least one block with soft grade $h = 1$. It is impossible for one party to get grade $g = 0$ and another to get grade $g = 2$ as each requires a majority of votes on soft grades $h = 0$ and $h = 1$ respectively. Finally the strong stability follows from the input values being justified, so if only one block is justified by J_{IN}, then all parties get $h = 1$ and $g = 2$. Note that the Justified output property means that the inner protocol $\Pi_{\mathsf{WeakGradedSelectBlock}}$ preserves the input justifier. For the same reason positive agreement holds, in fact a stronger statement holds: a single party getting grade $g = 2$ in the $\Pi_{\mathsf{WeakGradedSelectBlock}}$ results in everyone getting grade $g = 2$.

B.9 Justified Block Selection with Adjacent Output Round Agreement

We now present a modified version of the Justified Block Selection primitive from [KN23]. It satisfies all of the properties of the original primitive, but adds a round number to the output and the guarantee that all parties give output in adjacent rounds. As in the original protocol the parties repeatedly execute a Justified Graded Block Selection protocol until a block is selected with grade 2 which guarantees that all other parties selected the same block with at at least grade 1. Because we are using a version with Strong Stability, all honest parties will output in adjacent rounds, and moreover it is impossible to cook up a justification for outputting in a round where an honest party could not have given output.

Definition 11 (Justified Block Selection with Adjacent Output Round Agreement). *A protocol for M parties $\mathsf{P}_1, \ldots, \mathsf{P}_M$. There is an input justifier J_{IN} and an output justifier J_{OUT}. All honest P_i have an input B_i for which $J_{\text{IN}}(B_i) = \top$ at the time the input was given. The output of the protocol is a block (C_i, r_i) justified by J_{OUT}.*

Liveness: *If all honest parties start running the protocol with a J_{IN}-justified input then eventually all honest parties have a J_{OUT}-justified output.*
Justified Output: *$J_{\text{IN}}(C_i) = \top$ holds for all possible J_{OUT}-justified outputs C_i.*
Agreement: *For all possible justified outputs (C_i, r_i) and (C_j, r_j) it holds that $C_i = C_j$.*
Adjacent Output Round Agreement: *For all possible justified outputs (C_i, r_i) and (C_j, r_j) it holds that $|r_i - r_j| \leq 1$, and if $(\cdot, r)$ is a justified output then no honest party sent a message in any round $r' > r + 1$.*

The protocol $\Pi_{\mathsf{SelectBlock}}$ is identical to the protocol in [KN23] except it is adapted to use committees, it has a round number added to its output, and it uses Strongly Stable Graded Block Selection as subprotocol instead of Graded Block Selection[9]. So it still implements Justified Block Selection by the proof in [KN23]. We only need to argue that it additionally satisfies the Adjacent Output Round Agreement property.

[9] Note that the former is just a special case of the latter

Protocol $\Pi_{\mathsf{SelectBlock}}$

1. Each party P_i initialises $\mathsf{GaveOutput}_i = \bot$.
2. Each party $\mathsf{P}_i \in \mathsf{C}_{\mathsf{SelectBlockInput}}$ with input B_i where $J_{\text{IN}}(B_i) = \top$, lets $B_i^0 = B_i$ and $g_i^0 = 0$ and Causal Casts (B_i^0, g_i^0), which is justified by $J_{\text{IN}}(B_i^0) = \top$ and $g_i^0 = 0$.
3. For rounds $r = 1, \ldots$ each part P_i with $\mathsf{GaveOutput}_i = \bot$ runs $\Pi_{\mathsf{StronglyStableGradedSelectBlock}}$ where:
   (a) P_i has input B_i^{r-1}.
   (b) The input of P_i is justified by a justified (B_i^{r-1}, g_i^{r-1}) with $g_i^{r-1} < 2$.
   (c) P_i eventually gets justified output (B_i^r, g_i^r).
4. In addition to the above loop each P_i runs the following *echo rules*:
   – In the first round r where $\mathsf{GaveOutput}_i = \bot$ and $g_i^r = 2$, set $\mathsf{GaveOutput}_i = \top$ and output $(C_i, r_i) = (B_i^r, r)$. The output justifier is the justifier for (B_i^r, g_i^r).[a] If $\mathsf{P}_i \in \mathsf{C}_{\mathsf{EchoOutput}}$ it causal casts (B_i^r, g_i^r) with justifiers as a computed message.
   – In the first round r where $\mathsf{GaveOutput}_i = \bot$ and where some justified (B_j^ρ, g_j^ρ) propagated from $\mathsf{P}_j \neq \mathsf{P}_i$ with $g_j^\rho = 2$, set $\mathsf{GaveOutput}_i = \top$, and output $(C_i = B_j^\rho, \rho)$. The output justifier is the justifier for (B_j^ρ, g_j^ρ).

[a] Note that as the inputs had grade less than 2 this justifies the protocol not terminating earlier, forcing even corrupt parties to output a round number in which an honest party could have terminated.

Fig. 19. $\Pi_{\mathsf{SelectBlock}}$

Lemma 5 (Adjacent Output Round Agreement). *For all possible justified outputs of* $\Pi_{\mathsf{SelectBlock}}$ (C_i, r_i) *and* (C_j, r_j) *it holds that* $|r_i - r_j| \leq 1$.

Proof. Consider any output (C_i, r_i) justified by the justifier for $(B_i^{r_i} = C_i, g_i^{r_i} = 2)$, and any output (C_j, r_j) justified by the justifier $(B_j^{r_j} = C_j, g_j^{r_j} = 2)$. If $r_i = r_j$ we are done, so assume they are different and without loss of generality that $r_i \leq r_j$. Since the output in round r_i was justified by a grade 2, then by Graded Agreement all possible justified outputs from $\Pi_{\mathsf{WeakGradedSelectBlock}}$ in round r_i contain the same block. Which in turn by Strong Stability implies that all justified outputs in round $r_i + 1$ have grade 2, and thus that there are no justified input to $\Pi_{\mathsf{WeakGradedSelectBlock}}$ in round r_i+2. It follows that if $(\cdot, r)$ is a justified output of $\Pi_{\mathsf{SelectBlock}}$, then no honest party initiated $\Pi_{\mathsf{WeakGradedSelectBlock}}$ for any round $r' > r + 1$.

B.10 Justified Agreement on Core Set with Adjacent Output Round Agreement

We present a YOSO protocol for ACS Π_{ACS}, which again is almost identical to the protocol in [KN23], except that it is adapted to use YOSO committees and and adds a round number to its output. Since this is just the output from $\Pi_{\mathsf{SelectBlock}}$, Π_{ACS} inherits the Adjacent Output Round Agreement property. This just adds some auxiliary information to the output and has no effect on the validity of the proofs showing that it satisfies the remaining properties of ACS. In conclusion Π_{ACS} satisfies Definition 12 which is identical to the ACS definition from [KN23], except that it includes the Adjacent Output Round Agreement property.:

Definition 12 (Justified ACS with Adjacent Output Round Agreement). *A protocol for* M *parties* $\mathsf{P}_1, \ldots, \mathsf{P}_M$ *with input and output justifiers* J_{IN} *and* J_{OUT}. *All honest* P_i *have an input* B_i *for which* $J_{\text{IN}}(B_i) = \top$ *at the time of input.*

Liveness: *If all honest parties start running the protocol with a J_{IN}-justified input then eventually all honest parties have a J_{OUT}-justified output.*

Validity: *For all possible J_{OUT}-justified outputs $(U, \cdot)$ and all honest P_i and all $(\mathsf{P}_i, B_i) \in U$ it holds that P_i had input B_i.*

Justified Blocks: *For all possible justified outputs $(U, \cdot)$ and all (potentially corrupt) P_i and all $(\mathsf{P}_i, B_i) \in U$ it holds that $J_{IN}(B_i) = \top$.*

Agreement: *For all possible justified outputs $(U_i, \cdot)$ and $(U_j, \cdot)$ it holds that $U_i = U_j$.*

Large Core: *For all possible justified outputs $(U, \cdot)$ it holds that $|S| \geq n - t$.*

Adjacent Output Round Agreement: *For all possible justified outputs (U_i, r_i) and (U_j, r_j) it holds that $|r_i - r_j| \leq 1$ and if $(\cdot, r)$ is a justified output then no honest party sent a message in any round $r' > r + 1$.*

Protocol Π_{ACS}

1. The input of P_i is B_i with $J_{\text{IN}}(B_i) = \top$.
2. If $\mathsf{P}_i \in \mathsf{C}_{\mathsf{ACSPropose}}$ is causal casts B_i. This message is justified by $J_{\text{IN}}(B_i) = \top$ and B_i having been reliably broadcast by P_i.
3. Party P_i collects at least $n-t$ justified B_j from parties $\mathsf{P}_j \in \mathsf{Collected}_i$ and lets $U_i = \{(\mathsf{P}_j, B_j)\}_{\mathsf{P}_j \in \mathsf{Collected}_i}$. This value is justified by each B_j being justified and $|U_i| \geq n - t$.
4. Run $\Pi_{\mathsf{SelectBlock}}$ where P_i inputs U_i. The input justifier of $\Pi_{\mathsf{SelectBlock}}$ is to check that U_i is justifiable as defined in the above step.
5. Party P_i gets output (C_i, r_i) from $\Pi_{\mathsf{SelectBlock}}$ and outputs (C_i, r_i). The output justifier is that (C_i, r_i) is a justified output from the above $\Pi_{\mathsf{SelectBlock}}$.

Fig. 20. Π_{ACS}

C Proofs for Total-Order Broadcast

In this section we will switch from simulation based security to game-based security and prove that the protocol Π_{TOB} defined in Fig. 13 satisfies the following definition.

Definition 13 (Game-based TOB). *We say that a protocol for M parties Π_{TOB} is a game-based secure TOB if for all PPT environments corrupting at most $T < (1 - \epsilon)M/3$ parties the following properties hold except with negligible probability for at random run $\text{Exec}_{\Pi_{\mathsf{TOB}}, \mathcal{E}}$ in the $(\mathcal{F}_{CF}, \mathcal{F}_{\text{ATOMIC-SEND}})$-hybrid model. Each party P holds a ledger L_{P}.*

Agreement *For any honest P and P', either $L_{\mathsf{P}} \sqsubseteq L_{\mathsf{P}'}$ or $L_{\mathsf{P}'} \sqsubset L_{\mathsf{P}}$.*

Validity *For each honest party P with ledger L_{P}. If (mid, m) is in a block in L_{P} and $P(\mathsf{mid}) = P_i$ and P_i is honest, then P_i sent (mid, m). Additionally, when looking at L_{P} as a sequence of blocks $B_1, \ldots, B_b$, the wait predicate W is satisfied for each prefix, i.e. $W^i(B_1 || \ldots || B_{i-1}, B_i)$ for each batch i.*

Liveness *Assuming new wait predicates W^i and messages satisfying them are continuously input to the protocol and that all honest parties get the same wait predicates in the same batches, then all messages input to the protocol are eventually added to L_P for all honest parties P.*

The UC and game based definitions are equivalent as long as there are no secret inputs to simulate, only correctness properties. Appendix C.1 elaborates.

Theorem 3. *The protocol Π_{TOB} described in Fig. 13 satisfies the game-based definition of Total-Order Broadcast in Definition 13. Assuming block size, $\alpha = \Omega(\lambda)$ it uses expected $O(M(\beta + \iota\lambda^2))$ bits of communication to order ι messages of combined size β in the coin flip hybrid model, and additionally needs setup for expected amortized $O(1)$ coin flips per batch to be computed to instantiate the coin flip.*

Proof. Agreement follows immediately from agreement of ACS and RB. Validity with respect to the individual messages in the block follows from the validity of RB, while the validity with respect to the wait predicate follows from all blocks input to ACS being justified by individually satisfying the wait predicate and the fact that a list of messages that satisfy the wait predicate will still satisfy it after permuting it or adding more message to it (cf. Section 4). Finally liveness follows from liveness of the subprotocols. Note that unless Π_{TOB} is continuously updated with new wait predicates that all parties agree on, the liveness property is an empty statement. $\Pi_{\mathsf{SelectBlock}}$ terminates in expected constant rounds, and since the protocol repurposes unused setups at most an expected constant number of new setups needs to be computed per batch. Each batch of the TOB consists of reliable broadcasts of the messages referenced in the block and one instance of Π_{ACS} to agree on which messages makes it into each batch and in which order. To produce new batch Π_{TOB} runs one instance of Π_{ACS} to agree on a set of blocks which references messages that were previously RBed. We first account for the cost RBing messages across all batches which for ι messages of combined size β is $O(M(\beta + \iota\lambda^2))$ when using Π_{RB}. In each batch a committee of expected λ parties RB a block which references within a constant of $\max(\alpha, W_\#)$ messages, which has communication complexity $O(\lambda M(\max(\alpha, W_\#) \log M + \lambda^2))$. Even if they all happen to reference the same same set of messages, this gives a per message cost of $O(\frac{M\lambda^3}{\max(\alpha, W_\#)})$, which since we assumed $\alpha = \Omega(\lambda)$ is $O(M\lambda^2)$ and thus dominated by the cost of RBing the message we accounted for above. Finally to run Π_{ACS} on the proposed blocks, a sequence of an expected constant number of committees of expected size λ need to send descriptions of subsets of the preceding committee through RB. This can again be done by each committee member sending a λ^2 bit list of public keys through the RB protocol which for each list has communication complexity $O(M\lambda^2)$. So, each invocation of Π_{ACS} contributes $O(M\lambda^3)$ bits of communication. Again since we assumed at least $\alpha = \Omega(\lambda)$ messages per epoch this $O(M\lambda^2)$ per message and dominated by the RB of the messages. In conclusion we get a communication complexity of $O(M(\beta + \iota\lambda^2))$ to add ι messages of combined size β to the ledger, which is optimal if the average message size if $\Omega(\lambda^2)$.

C.1 Equivalence Between Game Based and UC TOB

The following theorem shows that game-based security implies simulation based security for TOB.

Theorem 4. *If Π_{TOB} is a game-based TOB then Π_{TOB} UC security implements $\mathcal{F}_{TOB}$ in the $(\mathcal{F}_{CF}, \mathcal{F}_{ATOMIC\text{-}SEND})$-hybrid model against $T < (1-\epsilon)M/3$ adaptive corruptions.*

Proof. We have to prove that there exists a PPT simulator $\mathcal{S}$ such that $\text{Exec}_{\Pi_{\mathsf{TOB}},\mathcal{E}} \approx \text{Exec}_{\mathcal{F}_{\text{TOB}},\mathcal{S},\mathcal{E}}$ for all PPT environments $\mathcal{E}$ doing at most $T < (1-\epsilon)M/3$ adaptive corruptions. The simulator $\mathcal{S}$ proceeds as follows. Note that whenever an input is given to $\mathcal{F}_{\text{TOB}}$ information is leaked which allows $\mathcal{S}$ to compute this input. The simulator will run Π_{TOB} on these inputs including copies of $\mathcal{F}_{\text{CF}}$ and $\mathcal{F}_{\text{ATOMIC-SEND}}$. It lets the environment $\mathcal{E}$ interact with $\mathcal{F}_{\text{CF}}$ and $\mathcal{F}_{\text{ATOMIC-SEND}}$ as in $\text{Exec}_{\Pi_{\mathsf{TOB}},\mathcal{E}}$. Whenever the copy of Π_{TOB} run by $\mathcal{S}$ produces an output then $\mathcal{S}$ uses its influence over $\mathcal{F}_{\text{TOB}}$ to make the copy of $\mathcal{F}_{\text{TOB}}$ in $\text{Exec}_{\mathcal{F}_{\text{TOB}},\mathcal{S},\mathcal{E}}$ produce the same outputs to the same parties as Π_{TOB}. To be able to do this it is clearly enough that Π_{TOB} is a game-based TOB, as this ensures its outputs are always possible outputs of $\mathcal{F}_{\text{TOB}}$. □

When we work with game-based security definitions for sub-protocols, in all cases the properties are tacitly meant to hold hold except with negligible probability for all PPT environments corrupting at most $T < (1-\epsilon)M/3$ parties and for a random run $\text{Exec}_{\Pi_{\mathsf{TOB}},\mathcal{E}}$ in the $(\mathcal{F}_{\text{CF}}, \mathcal{F}_{\text{ATOMIC-SEND}})$-hybrid model.

D Two-level ciphertexts

In this section, we show how parties in a committee can generate random ciphertexts of form $E_{N,w_{i+1}}(E_{N,w_i}(0))$ for $i = 1, \ldots, \lambda - 1$ and prove they are correctly formed. These can then be combined to get a random two-level encryption of 0, and such a ciphertext can in turn be used to randomize encryptions in the multiparty version of the PIR protocol.

For this purpose, we will need that a party P, can publish a "two-level" encryption of 0: $c = E_{N,w_{i+1}}(E_{N,w_i}(0))$ and give a non-interactive zero-knowledge proof that the ciphertext was correctly formed. We will denote such a proof for the above relation by $\mathsf{NIZK}((r,s) : c = E_{N,w_{i+1}}(E_{N,w_i}(0;r);s))$. Below, we present and analyse a Σ-protocol for this relation which can then be made non-interactive in the random oracle model, as usual. The protocol is novel, and is an application of techniques known from so-called double discrete log proofs.

Random Two-level Encryptions The protocol in Fig. 21 produces a random two-level encryption of 0. Several instances can be run in parallel to create any desired number of outputs.

For correctness of $\mathsf{RandTwoLevel}$, note that the plaintext inside the output ciphertext d is the product of the plaintexts inside the d_u's which are themselves ciphertexts. Hence, d is an encryption of

$$\prod_{u=0}^{t} E_{N,w_i}(0; s_u) \bmod N^{i+1} = E_{N,w_i}(0; \prod_{u=0}^{t} s_u \bmod N) \ .$$

Protocol RandTwoLevel

1. On input i, where $0 \leq i < S$, each member P_u of the current committee computes $d_u = E_{N,w_{i+1}}(E_{N,w_i}(0; r_u); s_u)$ and sends d_u, π_u to $\mathcal{F}_{\text{TOB}}$, where
$$\pi_u = \mathsf{NIZK}((r_u, s_u) : d_u = E_{N,w_{i+1}}(E_{N,w_i}(0; r_u); s_u)) \ .$$
2. Once $h \geq n - t$ valid contributions $(d_1, \pi_1), \ldots, (d_h, \pi_h)$ are delivered from $\mathcal{F}_{\text{TOB}}$, output $d = \mathsf{Multiply}(d_1, \ldots, d_h)$

Fig. 21. The protocol for generating random two-level encryptions of 0

So, d is itself a two-level encryption of 0, where at least one contribution to the randomness comes from an honest player since there are $t + 1$ contributions. We can therefore think of the protocol as implementing a functionality that produces random two-level encryptions of 0.

The protocol in Fig. 22 allows a party, here denoted by P, to publish a two-level encryption of 0: $E_{N,w_{i+1}}(E_{N,w_i}(0))$ and convince a verifier V in (honest verifier) zero-knowledge that is correctly formed. This protocol can then be turned into a non-interactive proof using standard tools (see Appendix F), to form the proof needed in the RandTwoLevel protocol.

Protocol TwoLevel.

1. P sends the ciphertext $c = E_{N,w_{i+1}}(E_{N,w_i}(0; r); s)$ to V, and then the proof below is executed k times in parallel.
2. (a) P computes $E_{N,w_i}(0; r')$ for random r' and sends to V
$$a = c^{E_{N,w_i}(0;r')} s'^{N^{i+1}} \bmod N^{i+2} = E_{N,w_{i+1}}(E_{N,w_i}(0; rr' \bmod N); ss' \bmod N)$$
   (b) V sends a random bit e to P.
   (c) If $e = 0$, P sends $z_1 = r'$ and $z_2 = s'$ to V. If $e = 1$, P sends $z_1 = rr' \bmod N$ and $z_2 = ss' \bmod N$.
   (d) If $e = 0$, V checks that $a = c^{E_{N,w_i}(0;z_1)} z_2^{N^{i+1}} \bmod N^{i+2}$.
   If $e = 1$, V checks that $a = E_{N,w_{i+1}}(E_{N,w_i}(0; z_1); z_2)$.

Fig. 22. The Σ-protocol for two-level ciphertexts

Theorem 5. *For any $i = 0, \ldots, S-1$, the* TwoLevel *protocol is a Σ-protocol for the relation*
$$\{((N, w_i, w_{i+1}, c), (r, s)) \mid c = E_{N,w_{i+1}}(E_{N,w_i}(0; r); s)\},$$
i.e., P knows r, s such that $c = E_{N,w_{i+1}}(E_{N,w_i}(0; r); s)$ and the protocol is complete and perfect honest verifier zero-knowledge.

Proof. Completeness is clear from inspection of the protocol. For special soundness (which is well-known to imply standard knowledge soundness), we assume we are given two accepting

conversations (a, e, z_1, z_2) and (a, e', z_1', z_2') with $e \neq e'$, and must show that we can efficiently find valid values for r, s. Assume without loss of generality that $e = 0$. Then we have

$$c^{E_{N,w_i}(0;z_1)} z_2^{N^{i+1}} \bmod N^{i+2} = a = E_{N,w_{i+1}}(E_{N,w_i}(0; z_1'); z_2')$$

which implies

$$c^{E_{N,w_i}(0;z_1)} = E_{N,w_{i+1}}(E_{N,w_i}(0; z_1'); z_2' z_2^{-1} \bmod N)$$

One can now compute $E_{N,w_i}(0; z_1)^{-1} \bmod N^{i+1}$, and observe that for some integer w we have $E_{N,w_i}(0; z_1)^{-1} \cdot E_{N,w_i}(0; z_1) = 1 + wN^{i+1}$. Now, by raising both side of the above equation to $E_{N,w_i}(0; z_1)^{-1}$, we get

$$c = E_{N,w_{i+1}}(E_{N,w_i}(0; z_1' z_1^{-1} \bmod N); z_2' z_2^{-1} c^{-w} \bmod N) \ .$$

Thus we see that we can compute valid values for r, s given the two conversations.

Finally, for honest verifier zero-knowledge, we show a simulator which first chooses random bit e and random $z_1, z_2 \in \mathbb{Z}_N^*$. Then, if $e = 0$ it sets $a = c^{E_{N,w_i}(0;z_1)} z_2^{N^{i+1}} \bmod N^{i+2}$. If $e = 1$ it sets $a = E_{N,w_{i+1}}(E_{N,w_i}(0; z_1); z_2)$, and finally, it outputs (a, e, z_1, z_2). It is clear that e, z_1 and z_2 have the right distribution, and a is set to be the only correct value, given e, z_1 and z_2. Hence the simulation is perfect. □

E Linear Integer Secret Sharing

The material on linear integer secret sharing in this section is taken from [DT06], while the later section on non-interactive VSS is a contribution of this paper.

We will need to secret share the secret decryption exponent which of course "lives in the exponent". So, the natural choice would be a sharing scheme that uses arithmetic modulo the order of the group. However, all parties need to be able to run the scheme and the group order is not public knowledge, so we resort to secret-sharing over the integers instead. Earlier work[GHK+21] has used integer variants of Shamir's scheme, but this leads to technical difficulties due to the fact that the Lagrange coefficients needed for interpolation are not integers. In earlier work, this implies that the size of the shares grow each time the secret key is re-shared. We will instead use a linear integer sharing scheme (LISS), which allows us to avoid this problem and simplify the protocols, at the cost of a larger share size in the beginning.

A LISS is defined by a *sharing matrix* M with integer entries, c columns and ℓ rows (one can think of M as replacing the Van der Monde matrix from Shamir's scheme). One can share a secret number $s \in [0, 2^b]$ for a publicly known upper bound b among n players. To do this, choose a column vector $\boldsymbol{v}_s$ with s as the first entry and sufficiently large random numbers in the other entries (we make this precise in a moment). Shares are computed as the product $M \cdot \boldsymbol{v}_s$. This corresponds to evaluating the polynomial in a number of points in Shamir's scheme.

Each row of M is labelled with an index in $[1, n]$, we say that each row is owned by a player. Our notation for this is that row i is owned by player number $u(i)$. Each entry in

$M \cdot \boldsymbol{v}_s$ corresponds to a row, and the entry is handed to the player who owns that row. We will refer to an entry in $M \cdot \boldsymbol{v}_s$ as a *share*, but note that each player may receive several shares.

For each player set A we let M_A denote the matrix we obtain by selecting from M only the rows owned by players in A. If A is qualified to reconstruct the secret, there exists a *reconstruction vector* $\boldsymbol{r}_A$ with the property that

$$\boldsymbol{r}_A \cdot (M_A \cdot \boldsymbol{v}_s) = s,$$

this corresponds to the interpolation in Shamir's scheme.

If A is not qualified to reconstruct, there exists a *sweeping vector* $\boldsymbol{w}_A$, which has 1 as it's first entry, and further has the property that $M_A \cdot \boldsymbol{w}_A$ is the all-0 vector. It can be shown that the existence of $\boldsymbol{w}_A$ implies statistical privacy of the scheme. Namely, we define w_{max} to be the maximal numeric value of any entry occurring in any $\boldsymbol{w}_A$. Then, a *valid sharing vector* for $s \in [0, 2^b]$ is a vector $\boldsymbol{v}_s$ with s in the first entry, and with the other entries chosen uniformly from $[0, 2^{b+\log_2(w_{max})+k}]$, where k is the security parameter.

Lemma 6. *For any two secrets $s, s' \in [0, 2^b]$, the distributions of shares seen by A from sharing s or s', with valid sharing vectors, are statistically indistinguishable.*

Proof. (Sketch) if s was shared using $\boldsymbol{v}_s$, one could instead have shared s' using $\boldsymbol{v}_s + (s' - s)\boldsymbol{w}_A$, and the players in A would receive exactly the same shares. But the numbers in $\boldsymbol{v}_s$ are a factor 2^k larger than those in $(s' - s)\boldsymbol{w}_A$, so $\boldsymbol{v}_s + (s' - s)\boldsymbol{w}_A$ is statistically indistinguishable from a valid sharing vector for s', so the lemma follows.

It has been shown [DT06] that LISS schemes with polynomial share size exist for the threshold case, where any majority of the players are qualified to reconstruct, this follows from the fact that polynomial size monotone formulae exist for computing the majority function. We let M_{th} denote such a scheme for n players. It is also straightforward to see that simple additive secret sharing can be realized in this formalism, using a matrix M_{add} with $n+1$ columns and n rows. namely, M_{add} is the identity matrix, except that the first row has -1 in the last n entries. Notably, for these schemes, all entries in all sweeping vectors are 1 or -1, so the numbers in a valid sharing vector just need to be k bits longer than the secret.

In this paper, we will need to share a secret among the members of two committees, that we will call the *additive committee* and the *threshold committee*. This is needed for technical reasons, to obtain adaptive security. The idea is to share the secret additively among the members of the additive committee, and then each additive share is shared among the members of the threshold committee. It is not hard to see that this can be phrased as one LISS scheme using a matrix M that we can build from M_{add} and M_{th}. We will not do the straightforward (but very tedious) details of this.

Note that, in the scheme defined by M the sets qualified to reconstruct the secret will be, either all members of the additive committee, or a majority of the threshold committee.

Verifiable Secret Sharing We require a verifiable LISS scheme that can be used to do distributed exponentiation in the groups $\mathbb{Z}^*_{N^{s+1}}$, for $s = 1, \ldots, S$. We will use a Pedersen-style construction for this, where the idea is that the secret and the sharing vector are committed to using the integer commitment scheme $\mathsf{Com}_{\mathsf{ck}}(\cdot;\cdot)$ we described in the preliminaries. We then define the algorithm $\mathsf{VSSshare}$, in figure 23.

Algorithm $\mathsf{VSSshare}$.

1. To share a secret d, choose a valid sharing vector $\boldsymbol{v}_d$ and compute $\mathsf{sh}(d, \boldsymbol{v}_d) = M \cdot \boldsymbol{v}_d$.
2. Choose a vector $\boldsymbol{r}$ containing randomness values for the commitment scheme and compute commitments to each entry $\boldsymbol{v}_d[j]$ as $\beta_j = \mathsf{Com}_{\mathsf{ck}}(\boldsymbol{v}_d[j]; \boldsymbol{r}[j])$ as well as $\mathsf{ra}(\boldsymbol{r}) \coloneqq M \cdot \boldsymbol{r}$.
3. Output
$$\mathsf{VSS}(d, \boldsymbol{v}_d, \boldsymbol{r}) = (\mathsf{sh}(d, \boldsymbol{v}_d), \mathsf{ra}(\boldsymbol{r}), \beta_1, \ldots \beta_c).$$

Fig. 23. The $\mathsf{VSSshare}$ algorithm.

Note that $\beta_1 = \mathsf{Com}_{\mathsf{ck}}(d; \boldsymbol{r}[1])$ serves as a commitment to the secret.

From the output produce by the VSS, anyone can compute commitments α_i to the i's share $s_i = \mathsf{sh}(d, \boldsymbol{v}_d)[i]$, and if the dealer has executed the VSS correctly, she can also open the α_i's. To see this, let $\boldsymbol{m}_i$ be the ith row of M, then $\mathsf{sh}(d, \boldsymbol{v}_d)[i] = \boldsymbol{m}_i \cdot \boldsymbol{v}_d$, and $\mathsf{ra}(\boldsymbol{r})[i] = \boldsymbol{m}_i \cdot \boldsymbol{r}$. Then $\alpha_i := \mathsf{Com}_{\mathsf{ck}}(\mathsf{sh}(d, \boldsymbol{v}_d)[i]; \mathsf{ra}(\boldsymbol{r})[i])$ is indeed a commitment to s_i that an honest dealer can open, and we claim that

$$\alpha_i = \prod_{j=1}^{c} \beta_j^{\boldsymbol{m}_i[j]} \tag{1}$$

This holds by the homomorphic property of the commitments. Indeed, we have:

$$\begin{aligned}
\alpha_i &= \prod_{j=1}^{c} \beta_j^{\boldsymbol{m}_i[j]} && (2)\\
&= \prod_{j=1}^{c} (\mathsf{Com}_{\mathsf{ck}}(\boldsymbol{v}_d[j]; \boldsymbol{r}[j]))^{\boldsymbol{m}_i[j]} && (3)\\
&= \mathsf{Com}_{\mathsf{ck}}(\boldsymbol{m}_i \cdot \boldsymbol{v}_d; \boldsymbol{m}_i \cdot \boldsymbol{r}) && (4)\\
&= \mathsf{Com}_{\mathsf{ck}}(\mathsf{sh}(d, \boldsymbol{v}_d)[i]; \mathsf{ra}(\boldsymbol{r}[i])). && (5)
\end{aligned}$$

Non-interactive VSS In the usual form of a VSS, players would receive shares and opening information for the α_i's and complain if they fail verification. However, we will need a non-interactive VSS, i.e., a possibly corrupt dealer can broadcast a single message to a set of n players and as a result they obtain valid shares of the secret the dealer had in mind, or they all conclude that the dealer is corrupt. To ensure that the receivers are committed to their shares, we do the following: the dealer will compute $\mathsf{VSS}(d, \boldsymbol{v}_d)$, encrypt the individual shares for each player, and attach non-interactive zero-knowledge proofs that the correct shares are encrypted. The global protocol calling the VSS will make sure that the receiver

will be committed to the decryption key and can therefore commit to the received share and prove in zero-knowledge that the commitment is correct.

We will be using a variant of one-time pad encryption in our VSS, the keys for this are assumed to have been set up by the $\mathsf{SecretChannels}$ protocol, Fig. 10. That protocol only supports one-time pads of limited length due to the fact that we can only do Pailler ciphertexts of a certain size. Using a single pad you can encrypt κ bits where $\kappa = \log_2(N^S) - k$ where k is a statistical security parameter. Shares in our VSS will be too large to encrypt using a single pad, so we will be using a tuple consisting of several pads as keys. A second issue is that for our proof of adaptive security to go through, the encryption needs to make a fresh random choice of the one-time pad actually added to the message at the time of *encryption*. This has to do with a step in the proof where we need rewinding. To cater for this, we will assume several pads have been set up and committed for each message to encrypt, and the sender will choose one of them for the actual encryption.

The notation for this is that a single one-time is denoted otp while a tuple of pads is denoted $\vec{\mathsf{otp}}$. In order to not clutter up the notation, we do not include the number of pads in a tuple explicitly but simply assume that there enough to cater for the share to be encrypted. In practice we will need $O(n)$ pads.

Concretely, for a message (an integer) m and one-time pad tuple $\vec{\mathsf{otp}}$, the encryption $E_{\vec{\mathsf{otp}}}(m)$ is done by first breaking m in pieces of κ bits each and encrypting each piece individually. We will use $\mathsf{Com}_{\mathsf{ck}}(\vec{\mathsf{otp}}; \boldsymbol{v})$ as shorthand for a tuple of individual commitments to the pads in $\vec{\mathsf{otp}}$. Then, to encrypt one κ-bit message a under a pad otp, you output the ciphertext $E_{\vec{\mathsf{otp}}}(a) = (a + \mathsf{otp})$.

Finally, we write $m = \sum_{\nu=0}^{w} a_\nu 2^{\nu\kappa}$, assume we have a long enough tuple $\vec{\mathsf{otp}}$ and define the encryption by $c = E_{\vec{\mathsf{otp}}}(m) = (E_{\mathsf{otp}_1}(a_1), \ldots, E_{\mathsf{otp}_w}(a_w))$. Decryption is denoted $D_{\vec{\mathsf{otp}}}(c)$ and is trivial by subtracting the pads.

Given $\alpha = \mathsf{Com}_{\mathsf{ck}}(m; v), \delta = \mathsf{Com}_{\mathsf{ck}}(\vec{\mathsf{otp}}; \boldsymbol{r})$ and a ciphertext $c = E_{\vec{\mathsf{otp}}}(m)$, we require a non-interactive zero-knowledge proof of knowledge of the data $m, \boldsymbol{v}, \mathsf{otp}, r$ used for forming α, δ, that the ciphertext contains m, and also that m is in a bounded interval $[0, 2^a]$. We denote such a proof by

$$\mathsf{NIZK}(m, v, \vec{\mathsf{otp}}, \boldsymbol{r} : \alpha = \mathsf{Com}_{\mathsf{ck}}(m; v), \delta = \mathsf{Com}_{\mathsf{ck}}(\vec{\mathsf{otp}}; \boldsymbol{r}),\ c = E_{\vec{\mathsf{otp}}}(m),\ m \in [0..2^a])\ .$$

In the following we will use this proof in a case where m is a share of some secret. The secret will be verified to be at most 2^b, so it follows that the entries in the sharing vector should be chosen from a bounded size interval as explained above, and given the sharing matrix M this implies an upper bound on the size of any share, we denote this bound by $2^{sh(b)}$, and will use it as the bound 2^a in the proof.

We also require a non-interactive zero-knowledge proof of knowledge that the prover knows how to open a set of commitments $\beta_1, \ldots, \beta_c$, and that the integer committed to in β_1 is in the interval $[0, 2^b]$, recall that the secret is committed to via β_1. We denote such a proof by

$$\mathsf{NIZK}(\{a_j, b_j\}_{j=1}^{c} : \{\beta_j = \mathsf{Com}_{\mathsf{ck}}(a_j; b_j)\}_{j=1}^{c},\ a_1 \in [0, 2^b])\ .$$

We will assume that the proof systems are statistical zero-knowledge, on-line extractable and unconditionally simulation sound, as explained in Section 2. A dealer in our VSS will proceed using the NonIntVSSshare algorithm, in Fig. 24.

Algorithm NonIntVSSshare

This protocol assumes that the dealer has committed to one-time pad tuples $\vec{\mathsf{otp}}_i, i = 1, \ldots, \ell$, as $\delta_i = \mathsf{Com}_{\mathsf{ck}}(\vec{\mathsf{otp}}_i; \boldsymbol{r}_i)$, and that the δ_i's are already published to all parties.

1. To share a value $d \in [0, 2^b]$, first compute $\mathsf{VSS}(d, \boldsymbol{v}_d, \boldsymbol{r}) = (\mathsf{sh}(d, \boldsymbol{v}_d), \mathsf{ra}(\boldsymbol{r}), \beta_1, \ldots \beta_c)$ as defined above.
2. Compute
$$\pi_0 = \mathsf{NIZK}(\{\boldsymbol{v}_d[j], \boldsymbol{r}[j]\}_{j=1}^c : \{\beta_j = \mathsf{Com}_{\mathsf{ck}}(\boldsymbol{v}_d[j]; \boldsymbol{r}[j])\}_{j=1}^c,\ \boldsymbol{v}_d[1] \in [0, 2^b]).$$
3. For $i = 1, \ldots, \ell$, let $s_i = \mathsf{sh}(d, \boldsymbol{v}_d)[i]$ and $v_i = \mathsf{ra}(\boldsymbol{r})[i]$, and let $\alpha_i = \mathsf{Com}_{\mathsf{ck}}(s_i; v_i)$. Compute a ciphertext $c_i = E_{\vec{\mathsf{otp}}_i}(s_i)$.
4. For $i = 1, \ldots, \ell$ compute
$$\pi_i = \mathsf{NIZK}(s_i, v_i, \mathsf{otp}_i, \boldsymbol{r}_i : \alpha_i = \mathsf{Com}_{\mathsf{ck}}(s_i; v_i), \delta_i = \mathsf{Com}_{\mathsf{ck}}(\vec{\mathsf{otp}}_i; \boldsymbol{r}_i),$$
$$c_i = E_{\vec{\mathsf{otp}}_i}(s_i),\ s_i \in [0..2^{sh(b)}])$$
5. Output $\mathsf{NonIntVSSshare}(2^b, d, \boldsymbol{v}_d, \boldsymbol{r}, \{\vec{\mathsf{otp}}_i, \boldsymbol{r}_i |\ i = 1, \ldots, \ell\}) := (\beta_1, \ldots, \beta_c, \{c_i, \pi_i |\ i = 1 \ldots \ell\})$.

Fig. 24. The NonIntVSSshare algorithm.

Anyone can then use the algorithm VSSverify, in Fig. 25, to check what the dealer sent.

Algorithm $\mathsf{VSSverify}(2^b)$

1. Given $\mathsf{NonIntVSSshare}(2^b, d, \boldsymbol{v}_d, \boldsymbol{r}, \{\vec{\mathsf{otp}}_i, \boldsymbol{r}_i \,|\, i = 1, \ldots, \ell\}) := (\beta_1, \ldots, \beta_c, \{c_i, \pi_i |\ i = 1 \ldots \ell\})$, verify π_0 against public values $\beta_1, \ldots, \beta_c$. For $i = 1, \ldots, \ell$ compute α_i using equation (1), verify the proof π_i against public values α_i, c_i and δ_i. Reject if any proof fails, otherwise accept.

Fig. 25. The VSSverify algorithm.

If the dealer's message verifies, the intended receivers will be able to obtain valid shares of the committed secret, by decrypting the relevant ciphertexts, and constructing commitments to the resulting shares. This assumes that receivers are committed to the same one-time pads that were used to form the VSS message, and our global protocol does this via the SecretChannels protocol. The details are given in protocol VSSreceive, Fig. 26.

After executing VSSreceive, we say that the player set *holds* $\mathsf{VSS}(d, \boldsymbol{v}_d)$ *with share bound* v if there exists $d, \boldsymbol{v}_d$ such that for all players, a share s_i has been committed to via commitment $\tilde{\alpha}_i$, and we have $s_i = \mathsf{sh}(d, \boldsymbol{v}_d)[i]$, and $s_i \leq 2^v$.

Note that, by the linearity property of the sharing scheme and the commitments, if a set of parties holds both $\mathsf{VSS}(d, \boldsymbol{v}_d) = (\mathsf{sh}(d, \boldsymbol{v}_d), \beta_1, \ldots \beta_c, \tilde{\alpha}_1, \ldots, \tilde{\alpha}_\ell)$ and $\mathsf{VSS}(d, \boldsymbol{v}_d) =$

Protocol VSSreceive
This protocol is to be executed by a set of parties to compute committed shares of a secret. Input is a VSS message $\mathsf{NonIntVSSshare}(2^b, d, \boldsymbol{v}_d, \boldsymbol{r}, \{\mathsf{otp}_i, \boldsymbol{r}_i \,|\, i = 1, \ldots, \ell\}) := (\beta_1, \ldots, \beta_c, \{c_i, \pi_i \,|\, i = 1 \ldots \ell\})$ that passes the check in $\mathsf{VSSverify}(2^b)$. Also we assume that for every share index i, party $P_{u(i)}$ has committed to the one-time pad tuple $\vec{\mathsf{otp}}_i$ via the commitment $\rho_i = \mathsf{Com}_{\mathsf{ck}}(\vec{\mathsf{otp}}_i; \boldsymbol{u}_i)$.

1. Given $\mathsf{NonIntVSSshare}(2^b, d, \boldsymbol{v}_d, \boldsymbol{r}, \{\vec{\mathsf{otp}}_i, \boldsymbol{r}_i \,|\, i = 1, \ldots, \ell\}) := (\beta_1, \ldots, \beta_c, \{c_i, \pi_i \,|\, i = 1 \ldots \ell\})$, set $s_i = D_{\underset{\mathsf{otp}_i}{\rightarrow}}(c_i)$. Set $\tilde{\alpha}_i = \mathsf{Com}_{\mathsf{ck}}(s_i; r_{s_i})$ to be a commitment to s_i and compute a zero-knowledge proof $\tilde{\pi}_i$ containing s_i based on the public VSS message and ρ_i.

Fig. 26. The VSSreceive protocol.

$(\mathsf{sh}(d', \boldsymbol{v}_{d'}), \beta'_1, \ldots \beta'_c, \tilde{\alpha}'_1, \ldots, \tilde{\alpha}'_\ell)$, we can define linear operations on these, for a public integer γ, as

$$\mathsf{VSS}(d, \boldsymbol{v}_d) + \gamma \cdot \mathsf{VSS}(d', \boldsymbol{v}_{d'}) =$$

$$(\mathsf{sh}(d, \boldsymbol{v}_d) + \gamma \cdot \mathsf{sh}(d', \boldsymbol{v}_{d'}), \beta_1(\beta'_1)^\gamma, \ldots \beta_c(\beta'_c)^\gamma, \tilde{\alpha}_1(\tilde{\alpha}'_1)^\gamma, \ldots, \tilde{\alpha}_\ell(\tilde{\alpha}'_\ell)^\gamma) \ .$$

Clearly this new object is also a VSS, we have

$$\mathsf{VSS}(d, \boldsymbol{v}_d) + \gamma \cdot \mathsf{VSS}(d', \boldsymbol{v}_{d'}) = \mathsf{VSS}(d + \gamma d', \boldsymbol{v}_d + \gamma \boldsymbol{v}_{d'}) \ .$$

It follows that if a committee pair holds several VSS objects, it also holds (by local operations on shares) a VSS containing a given linear function applied to the underlying secrets, with a share bound that is easy to compute using the coefficients in the linear function.

We now show that the VSSreceive protocol works as expected:

Lemma 7. *Assume we are given* otp_i, r_i, u_i *such that for* $i = 1, \ldots, \ell$, $\rho_i = \mathsf{Com}_{\mathsf{ck}}(\mathsf{otp}_i; u_i)$, $\delta_i = \mathsf{Com}_{\mathsf{ck}}(\mathsf{otp}_i; r_i)$, *and an adversary creating a dealer's VSS message*

$$\mathsf{NonIntVSSshare}(2^b, d, \boldsymbol{v}_d, \boldsymbol{r}, \{\vec{\mathsf{otp}}_i, \boldsymbol{r}_i \,|\, i = 1, \ldots, \ell\}).$$

If the event that the VSS messages verifies but the receiving players do not hold $\mathsf{VSS}(d, \boldsymbol{v}_d)$ *with share bound* $sh(b)$, *happens with non-negligible probability, then we can construct an algorithm that breaks the binding property of the commitment scheme.*

Proof. Suppose we are given an adversary that can make the event specified in the lemma happen with non-negligible probability. We take a public commitment key ck as input and run the adversary with this key. Note that the zero-knowledge proofs given are straight-line extractable. This concretely means that if the dealer's message verifies, then (except with negligible probability) from the dealer's oracle calls and the proof π_0, one can extract opening information for all the β_j's, to get a sharing vector $\boldsymbol{v}_d$. In particular we can extract d from β_1. Then, from Eq. (1), one can compute opening information for all the α_i's and by construction of Eq. (1) this will show how to open the α_i's to reveal a set of valid shares $\{s_i\}$ of d, where $s_i = \mathsf{sh}(d, \boldsymbol{v}_d)[i]$. On the other hand, from the proof π_i one can extract the

content s_i' of c_i as well as a way to open α_i, to get s_i'. The assumption that we are given otp_i, r_i, u_i clearly implies that the commitment $\tilde{\alpha}_i$ constructed by the receiver contains s_i'. Thus, if the adversary's attack is successful, it must be that for some i $s_i \neq s_i'$. We can then output the two ways to open α_i and break binding. □

Note that the reduction from the above proof can be used in context of our global protocol because the commitment key is given as setup, and is chosen independently from everything else—provided that the global protocol ensures that even corrupt parties know the otp-values and randomness used for the relevant commitments. This is done by the SecretChannels protocol, where the commitments to the otp-values are done and parties must prove that they know the committed values.

We note that we will not require an explicit reconstruction protocol for the VSS, instead the shares held by the players will be used for decryption as detailed in the main protocol. Basically, all (honest) players in the additive committee will contribute to the decryption of a given ciphertext. Since contributions from the entire additive committee would be required to decrypt, we have the threshold committee fill in for the those additive players that do not contribute.

As for privacy, the intuition is clear: an unqualified set of shares reveals essentially nothing about the secret, and the commitments and zero-knowledge proofs leak (statistically) no additional information. However, we will need to show adaptive security of the global protocol, so we need to be careful. To this end, we will use in our security proof given elsewhere the so-called single inconsistent player (SIP) technique. Here, the idea is that the simulator is set up such that it knows a complete set of shares for players in the additive and threshold committees, however, these shares determine some dummy value. The simulator selects at random a player from the additive committee to be the SIP. When simulating decryption, it can fake the contribution from the SIP to make the output plaintext be correct. Therefore, as long as the SIP is not corrupted and its contribution is delivered on time, the simulation will be statistically indistinguishable. This happens with probability at least $1/n$. The reason why we need the SIP it to be delivered on time is that parties that do not have their contribution delivered on time will have their contribution computed by the threshold committee, which for the SIP would reveal the inconsistency.

op0opi

The Reshare Protocol We are now ready to specify the protocol for resharing. Basically all threshold members in committee pair C_1 holding $\mathsf{VSS}(d_S, \boldsymbol{v}_{d_S})$ will VSS their shares among all members of committee pair C_2. They can then use a reconstruction vector to get new shares of d_S. In the protocol, the receiving committee waits for non-interactive VSS messages from C_1 to be delivered from $\mathcal{F}_{\text{TOB}}$. We can assume that these messages pass the check in the VSSverify algorithm, as otherwise they would be rejected by $\mathcal{F}_{\text{TOB}}$.

Protocol Reshare

In this protocol, committee pair C_1 holding d_S will reshare to committee pair C_2 such that C_2 eventually holds d_S. It is assumed that for each share s_j held by C_1, $P_{u(j)}$ shares a one-time pad tuple $\overrightarrow{\mathsf{otp}}_i^j$ with $P_{u(i)}$ in C_2. Furthermore $P_{u(j)}$ has commitment $\delta_i^j = \mathsf{Com}_{\mathsf{ck}}(\overrightarrow{\mathsf{otp}}_i^j; r_i^j)$ to $\overrightarrow{\mathsf{otp}}_i^j$, while $P_{u(i)}$ has commitment $\rho_i^j = \mathsf{Com}_{\mathsf{ck}}(\overrightarrow{\mathsf{otp}}_i^j; u_i^j)$

1. For each member P_u of the threshold committee in C_1, and each share s_j for which $u(j) = u$, P_u sets $\beta_1^j = \alpha_j$, where α_j is the commitment to s_j. Then compute and send to $\mathcal{F}_{\text{TOB}}$:

$$\mathsf{NonIntVSSshare}(2^{sh(b)}, s_j, \boldsymbol{v}_{s_j}, \boldsymbol{r}_j, \{\overrightarrow{\mathsf{otp}}_i^j, \boldsymbol{r}_i^j \mid i = 1, \ldots, \ell\})$$

2. Once VSS messages from a subset A of size at least $n - t$ are delivered from $\mathcal{F}_{\text{TOB}}$, parties in C_2 run $\mathsf{VSSreceive}$ for each VSS message. C_2 now holds $\mathsf{VSS}(s_j, \boldsymbol{v}_{s_j})$ for j such that $P_{u(j)} \in A$. The committee uses the reconstruction vector $\boldsymbol{r}_A$ to compute

$$\mathsf{VSS}(d_S, \sum_j \boldsymbol{r}_A[j] \cdot \boldsymbol{v}_{s_j}) = \sum_j \boldsymbol{r}_A[j] \cdot \mathsf{VSS}(s_j, \boldsymbol{v}_{s_j}) \ .$$

Fig. 27. The Reshare protocol.

F Σ-protocols and Non-interactive Zero-Knowledge Proofs

In this section we sketch the (well-known) techniques one can use to get Σ-protocols for the relevant relations. Once we have these, we can get NIZK's in the random oracle model, using the Fischlin transform [Fis05].

For the case of two-level Paillier ciphertexts, the required protocol was already described and analysed in appendix D.

Otherwise, we use two main types of proofs in our protocols, namely those that are used in the MPC protocols, and those that are used in the Role Assignment, Decryption and Reshare protocols.

The first type of proofs work only on Paillier ciphertexts, and the protocols we need for this can be found in [DJN10], they can be used here with no essential change.

The second type of proofs work with integer commitments, Paillier ciphertexts, and in some cases one-time pad encrypted values, where message and one-time pad have been committed to.

Recall that a commitment to integer x is of form $\alpha^x \beta^r \bmod N'$, where r is randomly chosen in a large enough interval (we do not need the low-level details of the scheme for this discussion). We will later use $\mathsf{Com}(z)$ as shorthand for a commitments to z. Note that commitments are linearly homomorphic, we have $\mathsf{Com}(z) \cdot \mathsf{Com}(z')^f = \mathsf{Com}(z + fz' \bmod N')$.

Consider now the type of proof we require in the Reshare protocol where we are given commitments $\mathsf{Com}_{\mathsf{ck}}(\mathsf{otp}; r_1), \mathsf{Com}_{\mathsf{ck}}(z; r_2)$ and one-time pad ciphertext $c = \mathsf{otp} + z$ for some otp, z and we require a non-interactive zero-knowledge proof of knowledge of $\mathsf{otp}, z, r_1, r_2$ such that commitment and ciphertext are of the claimed form. This is straightforward by giving a standard proof of knowledge for the commitments to otp and z, multiplying the

commitments to get a commitment to the sum $z + \mathsf{otp}$ and proving that this commitment contains c.

In most cases the message to encrypt in the Reshare cannot be encrypted securely using just one one-time pad as these are only generated as numbers with a fixed bit length, so only κ-bit number are encrypted in one go. So the protocol splits the message (share) m to encrypt in w chunks of κ-bits such that $m = \sum_{\nu}^{w} 2^{\nu w} z_\nu$. To handle this, the prover commits to m and all individual m_ν's, proves using a standard range proof that $0 \leq m_\nu \leq 2^\kappa$ for each ν and any one can use the homomorphic property of commitments to verify that m and the m_ν's satisfy $m = \sum_{\nu}^{w} 2^{\nu w} z_\nu$. Finally, given commitments to w one-time pads, the method above can be used to encrypt each m_ν and prove that the ciphertexts are correctly formed. All the sigma protocols can be done in parallel.

We also need, for the Create Input Data protocol, a proof that given commitment $\mathsf{Com}(x)$ and Paillier ciphertext $E_{N,w_S}(x)$ contain the same value. For this, the prover uses a standard range proof to show that x is in the plaintext space of the encryption scheme, so $0 \leq x < N^S$ and then, since x sits in the exponent both for the commitment and the ciphertext, a standard equality of discrete log protocol can be used to show that the two contain the same value.

For the Decrypt protocol, we need a proof that for a committed value s_i, Paillier ciphertext c, and $c^{s_i} \bmod N^s$, that the committed value s_i has indeed been used to compute the last value. Again, since s_i sits in the exponent, this is case of equality of discrete logs.

For the Secret Channels protocol we need a proof for the following scenario: the prover has committed to $\mathsf{Com}(\mathsf{otp}_1)$, we are given $sum = \mathsf{otp}_1 + \mathsf{otp}_2 \bmod N^S$ and the prover commits to $\mathsf{otp}_2 = \mathsf{sum} - \mathsf{otp}_1 \bmod N^S$ and we want a proof that this commitment is correct. This is a standard case of proving modular relations inside commitments and the techniques from [DF02] can be used here.

There is, however, one technicality we need to address: we want, of course, that the protocol has negligible soundness error, and we want to avoid having to use binary challenges and repeat protocols many times. For this to work out, it needs to be the case that all group elements involved in the statement to prove are in a group with only exponentially large prime factors in the order. This is potentially an issue as these elements may be adversarially generated in some cases. For elements coming from the set-up we are fine, as they are generated honestly by the set-up. But this is not the case for Paillier ciphertexts to decrypt, for instance. We would be fine if we knew that the values in question were in the subgroup of squares in $\mathbb{Z}_N^*$. But we cannot verify membership in this subgroup efficiently. To solve this, we simply square the elements of the ciphertext and do the proof on the result. This has the effect of multiplying the plaintext by 2, but since 2 is relatively prime to N, this factor can be divided out after decryption.

G Supplementary UC Formalization

G.1 Eventual Liveness

When modelling asynchronous security we will as usual have to talk about an event *eventually* happening, for instance saying that if a message is sent then it is eventually delivered. We

now discuss how we model this. When specifying an ideal functionality we will as usual let the adversary specify when certain events E happen. We sometimes say that under certain preconditions C the adversary must *eventually* make E happen. This means that whenever C becomes true it holds at some future point in time that either C stopped being true or E became true. As an example we might say that if m was sent from an honest party which is still honest, then eventually m will be delivered, where C is "m was sent by an honest party which is still honest and" and E is "m was delivered by the adversary". We call this an eventual event. We say that a protocol π using ideal functionalities $\mathcal{F}_1, \ldots, \mathcal{F}_\ell$ and implementing $\mathcal{F}$ is live if it holds that when all eventual events on $\mathcal{F}_1, \ldots, \mathcal{F}_\ell$ happened, then π made all eventual events of $\mathcal{F}$ happen. This way we do not need to define what it means for an event to eventually happen, rather we just require that protocols are "eventuality-preserving". As an example, if $\mathcal{F}_1$ is a functionality for sending messages on point-to-point channels and $\mathcal{F}$ the ideal functionality for broadcast then eventuality-preserving liveness could be of the form "if all messages sent on the point-to-point channels by honest parties in π have been delivered then all messages broadcast by honest parties via π have been delivered". This is a crude model but good enough for our study.

G.2 Ideal Functionality for Atomic Send

The ideal functionality for atomic send is given in Fig. 28.

Functionality $\mathcal{F}_{\text{ATOMIC-SEND}}$

Init: Let $\mathsf{Accepted} = \emptyset$.
Broadcast: On input $(\text{ATOMIC-SEND}, (\mathsf{mid}_1, m_1, P_1), \ldots, (\mathsf{mid}_\ell, m_\ell, P_\ell))$, where $P(\mathsf{mid}_i) = P$ for $i = 1, \ldots, \ell$, add each $(\mathsf{mid}_i, m_i, P_i)$ to $\mathsf{Accepted}$ and leak $(\text{ATOMIC-SEND}, (\mathsf{mid}_1, m_1), \ldots, (\mathsf{mid}_\ell, m_\ell))$ to the adversary. We assume that honest parties use each mid at most once.
Deliver: On input $(\text{DELIVER}, (\mathsf{mid}, m, R))$, where $(\mathsf{mid}, m, R) \in \mathsf{Accepted}$ or $P(\mathsf{mid})$ is currently corrupted, remove (mid, m, R) from $\mathsf{Accepted}$ and output (mid, m) to R.
Eventual Liveness: If $(\mathsf{mid}, m, R) \in \mathsf{Accepted}$ the adversary must eventually input $(\text{DELIVER}, (\mathsf{mid}, m, R))$.

Fig. 28. Atomic Send

G.3 Discussion of $\mathcal{F}_{\text{TOB}}$

For each batch the parties input a wait predicate indicating which messages should be collected. We assume that all honest parties agree on the wait predicate W for each block. That means that we only prove an implementation secure under this condition and that, on the other hand, when we use $\mathcal{F}_{\text{TOB}}$ in a protocol π then π must guarantee that all honest parties input the same W to $\mathcal{F}_{\text{TOB}}$. We cannot guarantee that all messages attempted to be sent are delivered in a given round, as the network is asynchronous. We therefore let the adversary choose which messages are delivered. This happens in **Next Batch**, where it picks the next block. However, the adversary is only allowed to pick a block valid by W.

Note that this means that if the messages input by the honest parties do not satisfy W then $\mathcal{F}_{\text{TOB}}$ might deadlock. This is a feature. It is the obligation of the protocol using $\mathcal{F}_{\text{TOB}}$ to pick W and send messages m such that W gets satisfied in each round. On the other hand this allows an implementation of $\mathcal{F}_{\text{TOB}}$ to wait until it saw messages satisfying W. Note that $\mathcal{F}_{\text{TOB}}$ is guaranteed to produce the next block once all honest parties requested it and their messages satisfy W. It might, however, produce the block *before* all honest parties ordered the block. We could not possibly wait for all honest parties to have ordered the block in an asynchronous network. Notice, however, that at least one honest party must have ordered the next block before it can be produced as W^{b+1} needs to be defined. The wait predicate W^{b+1} can therefore be used to control that the block is not produced too early—it might for instance say that the next block is valid only if there are messages from enough parties that at least one must be honest. Notice, finally that even though we cannot ensure that a message sent in round b will make it into block $b+1$ we *do* require that any message sent will eventually make it into *some* block. Finally, we require that all blocks are eventually delivered to all honest parties. This all in all means that all messages broadcast by honest parties are eventually delivered to all honest parties.

Looking forward, the implementation for $\mathcal{F}_{\text{TOB}}$ will use small committees and YOSO role assignment. It might seem puzzling that this is not reflected in $\mathcal{F}_{\text{TOB}}$. However, the committees are an implementation detail, not a part of the specification of total-order broadcast.

G.4 Threshold Coin-Flip

The ideal functionality for coin-flip is given in Fig. 29. It can trivially be implemented given $\mathcal{F}_{\text{MPC+CF}}$. We introduce it as a separate ideal functionality to not use $\mathcal{F}_{\text{MPC+CF}}$ in its full glory when implementing $\mathcal{F}_{\text{TOB}}$ from coin-flip.

Functionality $\mathcal{F}_{\text{CF}}$ for coin flip.

Init: For each party P let $\ell_P = 0$ be the number of coins delivered at P and let $b_P = 0$ be the number of coins ordered by P. Let $b = 0$ be the number of coins flipped so far. Let L be the empty ledger.

Order next coin: On input (NEXT-COIN) from honest P leak (NEXT-COIN, P) to the adversary, let $b_P = b_P + 1$, flip uniformly random $c_{b_P} \in \{0,1\}^\kappa$ if c_{b_P} was not already flipped, and give c_{b_P} to the adversary.

Coin Index: On input (COIN-INDEX) from honest P output b_P to P.

Next coin: On input (NEXT-COIN) from the adversary where $b_P > b$ for some honest P, let $b \leftarrow b + 1$, and let $L = L \| c_b$.

Deliver: On input (DELIVER, P) from the adversary where $\ell_P < |L|$ update $\ell_P = \ell_P + 1$ and output $(\ell_P, L[\ell_P])$ to P.

Eventual Liveness: If $b_P > b$ for all honest P the adversary eventually calls **Next coin** again. Furthermore, if $\ell_P < |L|$ then eventually the adversary inputs (DELIVER, P) again.

Fig. 29. Threshold Coinflip

H Basic Protocol for Secure Computation

This section contains some basic protocols for secure computation that we will need for the implementation of $\mathcal{F}_{\text{MPC}}$. Many of these are standard, but we incorporate some new ideas, as accounted for in the text below.

Before we dive into the protocols, a remark on the committees used for the execution. In the RoleBatches protocol we discussed how to set the parameter eno large enough that we have sufficiently many committee pairs for running one batch of RoleBatches. But we do not just want to run RoleBatches, we also want to do MPC to implement $\mathcal{F}_{\text{MPC+CF}}$. For any polynomial γ we can set eno sufficiently larger to form γ additional committees. Some of these will be used in Decrypt and will be committee pairs as the ones used for role assignment. However, some of them do not need a secret sharing of the secret key as they do not perform decryption, we will call these *MPC committees*. The MPC committees will consist of just n parties, so when we say that all parties on an MPC committee acts, only n parties do something. They will perform one round in some sub-protocol and post their messages on $\mathcal{F}_{\text{TOB}}$. If these messages need decryption some double committee will handle it via Decrypt.

Secure Multiplication. For secure multiplication, we use the standard idea of producing multiplication triples and using one triple later for each multiplication. The protocol for making triples is in Fig. 30. It lets all parties compute the output triple locally from data on the ledger. Once a triple is produced it is placed on an ordered list, and when we say in the following that we use the next available triple, we mean that all parties take the next unused triple on the list.

Protocol Triple.
This protocol is parametrized by s, and will produce ciphertexts in $\mathbb{Z}_{N^{s+1}}$.

1. Each member P_u of the MPC committee assigned to do this protocol instance chooses random a_u, r_u, computes $c_u = E_{N,w_s}(m_u; r_u)$ and sends to $\mathcal{F}_{\text{TOB}}$ the pair
$$(c_u, \mathsf{NIZK}(a_u, r_u : c_u = E_{N,w_s}(a_u; r_u)) \ .$$
2. Once $n-t$ valid pairs appear on the ledger, all parties compute $c_a = \prod_u c_u \bmod N^{s+1}$ where the product is over the u's that appeared. We have $c_a = E_{N,w_s}(a)$ where $a = \sum_u a_u \bmod N^s$.
Each member P_v of the next MPC committee chooses random b_v, r_v, s_v. They compute $c_v = E_{N,w_s}(b_v; r_v)$ and $c'_v = c_a^{b_v} \cdot E_{N,w_s}(0; s_v) \bmod N^{s+1}$. They send to $\mathcal{F}_{\text{TOB}}$ the tuple:
$$(c_v, c'_v, \mathsf{NIZK}(b_v, r_v, s_v : c_v = E_{N,w_s}(b_v; r_v),\ c'_v = c_a^{b_v} \cdot E_{N,w_s}(0; s_v) \bmod N^{s+1}).$$
3. Once $n-t$ valid tuples appear on the ledger, all parties compute $c_b = \prod_v c_v \bmod N^{s+1}$ and $c_{ab} = \prod_v c'_v \bmod N^{s+1}$, where the product are over the v's that appeared. We have $c_b = E_{N,w_s}(b)$ where $b = \sum_v a_v \bmod N^s$.
And, because each c'_v contains $b_v a \bmod N^s$, we have $c_{ab} = E_{N,w_s}(ab \bmod N^s)$ where $d = ab \bmod N^s$.
All parties output (c_a, c_b, c_{ab}).

Fig. 30. The Triple protocol.

Protocol Multiply.
This protocol is parametrized by s, and will do secure multiplication on ciphertexts in $\mathbb{Z}_{N^{s+1}}$. The input consists of two ciphertexts $c_x = E_{N,w_s}(x), c_y = E_{N,w_s}(y)$.

1. Let (c_a, c_b, c_{ab}) be the next available triple. All parties compute $c_\epsilon = c_x(c_a)^{-1} \bmod N^{s+1}$ and $c_\delta = c_y(c_b)^{-1} \bmod N^{s+1}$. Send c_ϵ, c_δ to the DecryptNoRandomize protocol for decryption.
2. Once the decryption results ϵ, δ are returned, all parties compute and output

$$c_{xy} = c_{ab} \cdot c_b^\epsilon \cdot c_a^\delta \cdot (N+1)^{\epsilon\delta} \bmod N^{s+1}.$$

It is straightforward to see that $c_{xy} = E_{N,w_s}(xy \bmod N^s)$.

Fig. 31. The Multiply protocol.

The Triple and Multiply protocols can be run in parallel as many times as we want. In the following, we use $c = \mathsf{Multiply}(c_1, \ldots, c_a)$ as shorthand for invoking Multiply an appropriate number of times on ciphertexts $c_1 = E_{N,w_s}(x_1), \ldots, c_a = E_{N,w_s}(x_a)$ to obtain $c = E_{N,w_s}(x_1 \cdot \cdots \cdot x_a \bmod N^s)$. This can be done in a standard tree structure and will then consume $\log a$ consecutive commitees, but one can also use the well-known Bar-Ilan and Beaver constant round technique to use only a constant number of MPC committees. We will not go into details with this.

The Multiply protocol uses a decryption step when it consumes a triple. For this case, the decryption protocol (Fig. 7) is run without calling the RandomizeCiphertext subprotocol. We refer to this as DecryptNoRandomize. This is done because RandomizeCiphertext itself calls Multiply and we need to avoid circularity.

The first part of the Triple protocol where the random ciphertext c_a is created can be used stand-alone, and based on this we can do inversion using a well-known trick, also from Bar-Ilan and Beaver. Namely, given a ciphertext $c = E_{N,w_s}(x)$, let an MPC committee create $c_a = E_{N,w_s}(a)$ for a random a, and then decrypt $\mathsf{Multiply}(c, c_a)$, to get $e = xa \bmod N^s$. Finally all players compute $c_a^{e^{-1}} \bmod N^{s+1} = E_{N,w_s}(x^{-1} \bmod N^s)$. We will refer to this ciphertext as $\mathsf{Inverse}(c)$.

Creating Random Encrypted Bits To make a random encrypted bit, we call the random oracle on input a label for this instance of the protocol, which produces an encryption of a random value x. The idea is now to compute securely a new encryption containing the Jacobi symbol $\left(\frac{x}{N}\right)$ of x modulo N which can easily be converted to a random bit. All parties compute the output ciphertext locally from data on the ledger. Once an encrypted bit is produced it is placed on an ordered list, and when we say in the following that we use the next available encrypted bit, we mean that all parties take the next unused ciphertext on the list. The protocol is found in Fig. 32.

The required zero-knowledge proof in the RandBit protocol is easy to construct by observing that the two ciphertexts in question are either of form $c_u = E_{N,w_s}(1; r_u), c'_u = E_{N,w_s}(1; r'_u)$ (if $b_u = 0$) or $c_u = E_{N,w_s}(\alpha; r_u), c'_u = E_{N,w_s}(-1; r'_u)$ (if $b_u = 1$). Either case can be proved by

Protocol RandBit.

This protocol is parametrized by s, and will create a random bit inside a ciphertext in $\mathbb{Z}_{N^{s+1}}$. The random oracle H is here assumed to output a random value modulo N^{s+1}. Also, we assume a fixed number $\alpha \in \mathbb{Z}_{N^s}$ has been chosen such that it's Jacobi symbol mod N is -1.

1. Let ℓ be a unique label for this instance of the protocol, and let $c_x = H(\ell)$, then for some x we have $c_x = E_{N,w_s}(x)$.
2. Each party P_u on the first MPC committee assigned to this protocol instance will choose a random bit b_u, random r_u, r'_u and compute $c_u = E_{N,w_s}(\alpha^{b_u}; r_u), c'_u = E_{N,w_s}((-1)^{b_i}; r'_u)$. Note that if this is correctly done, c'_u contains the Jacobi symbol of the number inside c_u, and this Jacobi symbol is random. and send to $\mathcal{F}_{\text{TOB}}$ the triple:
$$(c_u, c'_u, \mathsf{NIZK}(b_u, r_u, r'_u : c_u = E_{N,w_s}(\alpha^{b_u}; r_u), c'_u = E_{N,w_s}((-1)^{b_i}; r'_u)).$$
3. Once $h = n - t$ valid triples $(c_{u_j}, c'_{u_j} \pi_{u_j})$ appear on the ledger, set $c_a = \mathsf{Multiply}(c_{u_1}, \ldots, c_{u_h})$ and $c'_a = \mathsf{Multiply}(c'_{u_1}, \ldots, c'_{u_h})$. Since the Jacobi symbol is multiplicative, we will have that $c_a = E_{N,w_s}(a), c'_a = E_{N,w_s}((\frac{a}{N}))$ for an a which is not random, but its Jacobi symbol is.
4. Decrypt $\mathsf{Multiply}(c_x, c_a)$ to get $xa \bmod N^s$, and decrypt $\mathsf{Multiply}(c_x, c_a)$ to get $\sigma = (\frac{x}{N})(\frac{a}{N}) = (\frac{xa}{N})$. Therefore $c'_x := (c'_a)^\sigma \bmod N^{s+1} = E_{N,w_s}((\frac{x}{N}))$.
5. All parties compute and output
$$c_b = (c'_x \cdot (N+1))^{2^{-1} \bmod N^s} \bmod N^{s+1}.$$
This operation ensure that c_b will contain $((\frac{x}{N}) + 1)/2$ which is indeed a 0/1 value.

Fig. 32. The RandBit protocol.

an efficient standard Σ protocol for proof of plaintext knowledge, so a standard or-protocol construction based on this will give a sound protocol that does not reveal the choice of b_u.

I Proofs of Security for the YOSO MPC Protocol

In section 5.3 we specified a protocol π_{mpc} for implementing $\mathcal{F}_{\text{MPC}}$. The goal in this section is to show the following:

Theorem 6. *When for a constant c at most $T < M/(3+c)$ parties are adaptive corrupted and we set $n = \lambda$ then for a large enough constant* eno *we have that if Pailler encryption is CSO-secure, then π_{mpc} securely implements $\mathcal{F}_{MPC}^{\mathsf{Pal},1/3,\gamma,m}$ in the $(\mathcal{F}_{SETUP}, \mathcal{F}_{TOB}, \mathcal{F}_{ATOMIC\text{-}SEND})$-model with a random oracle. Here γ and m can be any polynomials.*

We prove the theorem for $m = 1$ for notational convenience. The proof trivially adapts to $m > 1$.

We first prove a technical lemma for a common proof pattern, where we argue that a property holds in all hybrids if it holds in one of the hybrids. We argue this from the hybrids being indistinguishable. But we at the same time argue that the hybrids are indistinguishable by the property holding in all hybrids. This seeming circularity can be broken when the violation of the property can be detected in polynomial time by the adversary.

Lemma 8 (Everywhere from Anywhere). *Consider two processes D^0 and D^1 which each run through a number of steps. Let A be a PPT adversary being shown a view of $D = D^b$*

and having to guess b. Let Ab *and* Ba *be events defined on both processes. Think of* $\mathsf{Ba}(D^b)$ *as something bad happening. We let* $\mathsf{Ab}(D^b)$ *(about to happen) denote the event that* $\mathsf{Ba}(D^b)$ *did not happen yet, but it will happen in the next step. Let* $D^b_{\to\mathsf{Ab}}$ *denote the process where we run* D^b *up until* $\mathsf{Ab}(D^b)$ *happens and then stops. So,* $A(D^b_{\to\mathsf{Ab}})$ *would have to make its guess using its view at this step just before* Ba *happens. Assume that* $A(D^b)$ *can detect* $\mathsf{Ab}(D^b)$ *in poly-time from its view in the game with* D^b*. Assume furthermore that we can prove the following for all PPT A:*

1. $|\Pr[A(D^0_{\to\mathsf{Ab}}) = 0] - \Pr[A(D^1_{\to\mathsf{Ab}}])| = \text{negl}$.
2. $\mathsf{Ba}(D^0)$ *happens with negligible probability in* $A(D^0)$.

Then for all PPT A

1. $|\Pr[A(D^0) = 0] - \Pr[A(D^1) = 0]| = \text{negl}$.
2. $\mathsf{Ba}(D^1)$ *happens with negligible probability in* $A(D^1)$.

Proof. We prove that last conclusion first. To see this observe that if $\Pr[\mathsf{Ba}(D^1)] \neq \text{negl}$ then because $\mathsf{Ba}(D^1)$ implies $\mathsf{Ab}(D^1)$ and A can detect $\mathsf{Ab}(D^1)$ in poly-time, we can consider the A outputting 1 when $\mathsf{Ab}(D^b)$ happens and outputs 0 if the execution ends without $\mathsf{Ab}(D^b)$ happening. From $\Pr[\mathsf{Ba}(D^0)] = \text{negl}$ we have that $\Pr[\mathsf{Ab}(D^0)] = \text{negl}$, so $\Pr[\mathsf{Ab}(D^0)] = \text{negl}$. Assume that for the sake of contradiction that it is not the case that $\mathsf{Ba}(D^1)$ happens with negligible probability in $A(D^1)$. From $\Pr[\mathsf{Ba}(D^1)] \neq \text{negl}$ we get that $\Pr[\mathsf{Ab}(D^1)] \neq \text{negl}$. But then $\Pr[\mathsf{Ab}(D^0)] = \text{negl}$ and $\Pr[\mathsf{Ab}(D^1)] \neq \text{negl}$. But by construction $\Pr[A(D^0)] = \Pr[A(D^0)_{\to\mathsf{Ab}}]$ and $\Pr[A(D^1)] = \Pr[A(D^1)_{\to\mathsf{Ab}}]$. This contradicts $|\Pr[A(D^0) = 0] - \Pr[A(D^1) = 0]| = \text{negl}$. Now since for all PPT A we have proven that $\mathsf{Ba}(D^1)$ happens with negligible probability in both $A(D^0)$ and $A(D^1)$ and we have assumed that that $|\Pr[A(D^0_{\to\mathsf{Ab}}) = 0] - \Pr[A(D^1_{\to\mathsf{Ab}}])| = \text{negl}$, we have that $|\Pr[A(D^0) = 0] - \Pr[A(D^1) = 0]| = \text{negl}$, as desired. □

We start by showing that the Decrypt and Reshare protocols in Figs. 7 and 27 produce correct outputs. Recall that there is a committee pair assigned to handle each batch of ciphertexts to decrypt. We maintain the invariant that when a pair is about to decrypt a batch, it holds $\mathsf{VSS}(d_S, \boldsymbol{v}_{d_S})$ with share bound 2^b, where d_S is the decryption exponent. This is ensured for the first pairs by $\mathcal{F}_{\text{SETUP}}$, and later by resharing d_S for the next committee pairs. At all times, we will use α_i to denote the commitment by which the player holding share s_i is committed to s_i.

Note that we have no issue of ciphertexts to decrypt being ill-formed: any number is an encryption of some message.

Regarding the share bound after resharing, assume we use the specific sharing scheme mentioned in Appendix E, where the share matrix is of size polynomial in n, say we have at most n^a rows and columns, and where all entries in the sharing matrix, sweeping and recombination vectors are 1 or -1. Then, by inspection of the operations done, one obtains that if the share bound is 2^b before resharing, it will be $2^{2a\log n+b+2k}$ after the resharing where k is the statistical security parameter. Namely, resharing a number with at most b bits first results in additive shares of size at most $b + k$ bits. These are now in turn shared using the threshold scheme. For this we need sharing vectors with $(b + k) + k = b + 2k$

bits entries, resulting in threshold shares with $a \log n + b + 2k$ bits[10]. These are combined using a recombination vector, which adds another $a \log n$ bits. This is already better than the approach using the integer version of Shamir's scheme, as here one gets a factor $n!$ multiplied on shares for each resharing, so that we add $\Omega(n \log n) + k$ bits to the share size for each resharing. We also sketched a protocol that even allows us to reduce the share size to a fixed amount.

For later use in the argument for security of our role assignment and MPC protocol, we want to argue that the decryption and reshare protocols work correctly. These proofs and many of the ones to follow make the following assumption:

Assumption HCNF (Honest committees, no forgeries): all committees used have honest majority and no corrupt player gets to execute a role she was not assigned, i.e., no corrupt player successfully forges the message an honest player was supposed to send to execute a role.

If this assumption does not hold the protocol may, for instance, abort early or a corrupt player can send a message for a role she does not hold. In a hybrid we construct later we will be able to show that the assumption holds, namely at that point, all ciphertexts are lossy and so the adversary cannot know the information she would need to break the HCNF. One now wants to say that if the assumption failed in the protocol, we could distinguish the protocol from the hybrid. But on the other hand, we have used HCNF to argue indistinguishability of protocol and hybrid, so it seems the argument is circular. But this is not really the case: if HCNF does fail in the protocol, there is a first time where this happens and up to that point, there is no difference. So we can use this first offending event to distinguish. This line of reasoning was formalized in Lemma 8.

A remark on the use of commitments. In the following, we will use the binding property of the commitment scheme and soundness of the zero-knowledge proofs to argue correctness of the protocols for decryption and resharing. The arguments for this are always of the same type: a party proves in zero-knowledge that the message she sends is correctly computed from public information and private data she is committed to. By knowledge soundness we can extract the committed information. However, we know by the invariant that the party is committed to correct shares of the secret key and other data, so if the extracted information is different, we can break the binding property by a straightforward reduction. This type of argument was also used in Lemma 7, and we will not repeat it below, but simply state the soundness of the zero-knowledge proofs implies what we want.

In later hybrids we construct, we will generate the public commitment key with associated trapdoor information so we can equivocate the commitments of honest players. However, we can still assume that corrupt players cannot break the binding property. This is because we show that protocol and hybrid are either statistically indistinguishable or computationally indistinguishable under an assumption unrelated to the commitments. Therefore a break of the binding property could be used to distinguish and must occur with negligible probability.

[10] Note that all players must prove in zero-knowledge that the shares they encrypt are in range, so corrupt players cannot force shares to be too large.

Correctness of Decryption. We have:

Lemma 9. *Under HCNF, the invariant is maintained by protocol* Reshare *except with negligible probability. That is, the j'th committee doing decryption holds* $\mathsf{VSS}(d_S, \boldsymbol{v}_{d_S})$ *with share bound* $2^{(S+1)\log N + k + (j-1)(a\log n + 2k)}$.

Proof. The fact that the committee pair holds a VSS of d_S follows for the first pair from the fact that it receives shares and commitments from the ideal functionality. For subsequent committee pairs, note that the reshare protocol only considers VSS's that pass the check in VSSverify, and the proof of Lemma 7 implies that, except with negligible probability, for each received VSS all players in the receiving committee are committed to shares of a well-defined value with a well-defined share vector. So such a VSS for player P_i is indeed of form $\mathsf{VSS}(z, \boldsymbol{v}_z)$ for some $z, \boldsymbol{v}_z$. Moreover, this value is equal to the original share s_i held by the sending player, as α_i defining that share is used as β_1^i in the VSS. Since there is honest majority in the sending committee by assumption, at least $n - t > t$ players will send a valid VSS message, and no corrupt messages can successfully claim to play the same roles, so the receiving committee can wait for $n - t$ VSS messages, and so reconstruction is always possible. The share bound follows immediately from the discussion above, and the fact that d_S it self is a number with at most $(S+1)\log N$ bits. □

Lemma 10. *Under HCNF, the decryption protocol outputs correct plaintexts except with negligible probability.*

Proof. We first consider a ciphertext c that is output by the initial call to RandomizeCipertext. Since the committee holds a VSS of the correct d_S, by Lemma 9, each α_i is a commitment containing the correct i'th share s_i of d_S. Hence, by soundness of the zero-knowledge proof $\pi_{c,i}$, we can assume that, except with negligible probability, each decryption message $d_{c,i}$ that is delivered from the Gather protocol (and relayed by the threshold committee) is of form $d_{c,i} = c^{s_i} \bmod N^{s+1}$, where s_i is the corresponding additive share of d_S.

Consider now some fixed but arbitrary receiving party, and assume that there is some decryption message $d_{c,i}$ that this party did not receive in any of the messages coming from a set A of parties in the threshold committee. This means that all parties in A (claim they) did not get $d_{c,i}$ and so, for their last step message to be valid, they must have included a valid back-up message for $d_{c,i}$. We therefore have back-up messages $\{d_{c,i,j} \mid P_{u(j)} \in A\}$. Again by soundness of the zero-knowledge proofs, we can assume that $d_{c,i,j} = c^{s_j} \bmod N^{s+1}$. So, since A is a qualified set (by the assumption on honest majority in the committee), by the properties of the reconstruction vector $\boldsymbol{r}_A^i$, it follows immediately that

$$\prod_{j, P_{u(j)} \in A} d_{c,i,j}^{\boldsymbol{r}_A^i[j]} = d_{c,i} = c^{s_i} \bmod N^{s+1} ,$$

and so we can conclude that the final product computed satisfies

$$\prod_{i=1}^{n} d_{c,i} = c^{\sum_{i=1}^{n} s_i} = c^{d_S} \bmod N^{s+1}$$

resulting in correct decryption of c.

Considering the randomization step, recall that it outputs

$$c = \bar{c} \cdot \mathsf{Multiply}(c^*, H(\bar{c}, R)) \bmod N^{s+1} \ ,$$

where $\bar{c}$ is the original input ciphertext. The Multiply protocol relies for correctness on the decryption protocol without the randomization, but this is what we just proved is correct. By the homomorphic property, we can therefore assume that the plaintext contained in c is of form $m + \alpha \cdot \beta \bmod N^s$, where m, α, β are the plaintexts contained in $\bar{c}, H(str)$ and c^*, respectively. In the set-up for the protocol, c^* is generated such that $\beta = 0$, and so we conclude that c also contains m and the decryption is correct. □

Main idea for proof of security We now outline the idea for the proof of security of π_{mpc}: We first show a UC simulator UCsim, it will do a straight-line simulation (as required for UC) which is possible because it knows the factorization of N and hence can do most of its job by simply following the protocol. The only caveat occurs when the protocol decrypts an output corresponding to an output from $\mathcal{F}_{\text{MPC}}$, such as a public key for a role. In such a case, the ciphertext produced in the simulation cannot be assumed to contain the right value. UCsim fixes this by changing the ciphertext c^* from the set-up such that it contains 1 instead of 0. This allows it to engineer the randomization step in the decryption such that the "randomized" ciphertext that is actually decrypted contains the correct value. Therefore, at the end of the day, the only difference between simulation and real protocol is that some of the Paillier ciphertexts that are never decrypted contain different values in the two cases.

However, since UCsim knows the factorization of N (it gets it from $\mathcal{F}_{\text{MPC}}$), we cannot directly appeal to CPA security of Paillier to say that this difference cannot be detected. Instead, we exploit the fact that when proving indistinguishability of simulation and protocol, we are no longer doing UC simulation, so rewinding is allowed. We show that, even without the factorization of N, we can emulate both the real protocol and the simulation using rewinding and a variant of the single inconsistent player (SIP) technique. This way, we define two processes called ProtRewind and SimRewind producing views for the environment that are information theoretically indistinguishable from the protocol and from the simulation, respectively. Note that in doing this, we are rewinding the environment. This is allowed as we do it as a proof technique. The UC simulator itself is straight line. Finally, skipping many details, we use the fact that the SIP technique allows us to simulate decryption without knowing the secret Paillier key and the computational assumptions we make to argue that ProtRewind and SimRewind are computationally indistinguishable.

The UC Simulator The high-level approach of UCsim, shown in Figure 33, is standard: emulate the honest parties by following the protocol, and when output is generated or a new party is corrupted, adjust the internal state so it matches what the functionality requires. We will use the variant of the UC framework where there is no explicit adversary, and the environment Z acts also as adversary.

Towards understanding the simulator, we note a few points: decryption of Paillier ciphertext occurs in two cases: one case is when a ciphertext is decrypted as a part of the

multiplication subprotocol, where we consume a multiplication triple, or when the decryption corresponds to a private output for an honest player. The other case is called an *output decryption* where decryption occurs, corresponding to an output that $\mathcal{F}_{\text{MPC}}$ leaks to the simulator; either a public output or a private output for a corrupt player. Here, the result of the decryption is dictated by $\mathcal{F}_{\text{MPC}}$, and the simulator takes measures to ensure that the correct output is generated.

When an honest player is corrupted, the player may possess several different types of data as listed below. Note that, in general, a player always deletes data that is no longer needed, so the only data found in memory are the latest one-time pads and shares to be used for a role that is not executed yet.

- *Private output:* the player may hold private output from the MPC.
- *One time pads:* the player may hold some one-time pads to be used for receiving shares from a previous committee or sending shares to the subsequent one. The player may also hold a one-time pad, if it is waiting for private output from the MPC.
- *Shares:* the player may hold shares of the Paillier decryption key.

It can be seen in the simulation that the simulator handles corruptions in a very simple way: it has a simulated state st_i for the corrupted player P_i containing simulated output that the player has, and has not yet been deleted. Now, the simulator receives the correct output out_i from $\mathcal{F}_{\text{MPC}}$, and it simply replaces the simulated output in st_i by out_i and hands the resulting state to Z. Let us explain intuitively why this does not create any inconsistencies that Z could use to tell it is in the simulation: In the real protocol, P_i gets output by first broadcasting an encrypted one-time pad otp_i, and then later the value $\mathsf{out}_i + \mathsf{otp}_i \bmod N$ is decrypted in public, allowing P_i to compute out_i. In the simulation, some random value rnd_i was decrypted, so when the simulation claims that the output was out_i, it implicitly claims that the original one-time pad was $\mathsf{rnd}_i - \mathsf{out}_i \bmod N$. This is most likely not the value used for the simulated encryption of the pad, but Z cannot detect that the claim is false, as it only knows the encryption of the pad, and the randomness for the encryption has been deleted by P_i by construction.

Finally, the simulator can fail, if it is not able to program the oracle as needed. Intuitively, this should not happen, as the programming is done before R is decrypted, so the environment would have to guess R from $E_{N,w_s}(R)$ to make an offending call. As the secret Paillier key is used in the simulation we cannot immediately argue that failure happens with negligible probability, but we will do so later, in a hybrid that is shown indistinguishable without assuming that offending calls are unlikely.

As a consequence of these observation and Lemma 10, we get:

Lemma 11. *Under HCNF, the Π_{MPC} protocol and the protocol instance run by* UCsim *produce correct outputs, except with negligible probabilty.*

Proof. In the protocol, one sees by simple inspection of the subprotocols, that as long as the decryption produces correct results, each subprotocol works correctly. This is because they all consist of homomorphic evaluation on ciphertexts, inputs supported by zero-knowledge proofs of plaintext knowledge and decryption. Further, the supply of random bits comes

Simulator UCsim for the Role Assignment protocol.

Intialize. Receive N and the factors of N from $\mathcal{F}_{\text{MPC}}$. Use this to emulate $\mathcal{F}_{\text{SETUP}}$, with one adjustment: the ciphertext c^* is generated as an encryption of 1 instead of 0. Initialize a copy of all honest players and give them the simulated set-up data from $\mathcal{F}_{\text{SETUP}}$. Send all private set-up data meant for corrupted players to Z. Initialize a copy of $\mathcal{F}_{\text{TOB}}$.

Main Process Execute the code of the honest players according to the protocol (while Z plays for the corrupted players). The interface of the internal emulation of $\mathcal{F}_{\text{TOB}}$ is connected to Z and the honest players as in the real protocol.

- The random oracle H is emulated using standard lazy sampling of random values, however, certain inputs are handled in a special way as detailed below.
- Whenever a corrupt player supplies a ciphertext, it is always accompanied by a zero-knowledge proof, from which the simulator straight-line extracts the corresponding secret data used to generate the message. As a result, the simulator knows plaintext and randomness for all ciphertexts in the global state, as well as all shares of the secret Paillier key held by the committees.
- When an output decryption occurs, execute the decryption subroutine below.
- When a corruption occurs, execute the corruption handling subroutine below.

Output Decryption Let $\bar{c} \in \mathbb{Z}^*_{N^{s+1}}$ be the ciphertext to be decrypted, and let z be the output generated by $\mathcal{F}_{\text{MPC}}$. Let m be the plaintext contained in $\bar{c}$, which most likely is different from z. To fix this, the simulator does the following:

1. Let L be the label of the relevant batch of ciphertexts to decrypt. The simulator chooses R at random and programs $H(L)$ to be an encryption of R. Once the batch appears on the ledger, the simulator programs $H(\bar{c}, R)$ to be a random encryption of $z - m$. If the random oracle has been called before with an input containing R, the simulator fails and aborts.
2. Since c^* contains 1, the output $c = \bar{c} \cdot \mathsf{Multiply}(c^*, H(\bar{c}, str)) \bmod N^{s+1}$ from RandomizeCiphertext contains z. The simulator can therefore let the rest of the decryption proceed honestly.

Corruption Handling When a player P_i is corrupted, the simulator is given whatever output out_i that $\mathcal{F}_{\text{MPC}}$ has given to P_i and has not yet been deleted. The simulator also has the internal state st_i of P_i as generated in the simulation so far. Note that st_i will contain output values generated for P_i in the simulation. The simulator replaces these values by out_i and hands the resulting state to Z.

Fig. 33. The UCsim simulator.

from the random oracle, so is correctly distributed. For the simulation, UCsim engineers the output ciphertexts so they contain the correct values, and Lemma 10 guarantees that these values are actually decrypted. □

Processes. In the following, a *process* is an algorithm that runs the environment Z is its head, as well as some number of other parties. The output of a process is whatever Z outputs. The Protocol process runs the protocol composed with Z and the resource functionalities the protocol requires, random oracle H, $\mathcal{F}_{\text{SETUP}}$ and $\mathcal{F}_{\text{TOB}}$. The Simulation process is the standard ideal process in the UC framework, it runs UCsim composed with Z and $\mathcal{F}_{\text{MPC}}$.

We first define $\mathsf{Simulation}^{\text{ZKSIM}}$ and $\mathsf{Protocol}^{\text{ZKSIM}}$, which are exactly the same as Simulation and Protocol, respectively, except that all ZK proofs done by honest parties are simulated.

We proceed to define two new processes $\mathsf{ProtRewind}(d_S)$ and $\mathsf{SimRewind}(d_S)$. Both take the secret Paillier key d_S as input.

To give a precise description, we recall that both simulation and protocol proceed in *batches*: in a batch, the first step is that all parties create input data for the role assignment and some parties may give input to whatever secure computation is specified to start in this batch. We then think of the protocol execution in the batch after this point as being split into *epochs*. There is an epoch for every committee pair that is active during the batch. An epoch starts at the time the *sending* committee reshares the secret Paillier key for the next *receiving* committee, or more precisely, at the time where the first honest party in the sending committee starts executing the Reshare protocol. Note that the first epoch will not start until all inputs for the batch have been specified. The receiving committee pair C, that receives shares of the secret key, will be the sending committee in the next epoch. The epoch ends when the first member of C starts executing the Reshare protocol. This will not happen until it can be seen on the ledger that the previous committee has done its work.

As a final prerequisite, note that there is an initial committee pair C_1 that receives from $\mathcal{F}_{\text{SETUP}}$ a VSS of the secret Paillier key d_S. $\mathcal{F}_{\text{SETUP}}$ is executed in both protocol and simulation (with a change in the simulation that is irrelevant here). The $\mathsf{ProtRewind}(sec)$ process can be found in Figure 34.

We then define the $\mathsf{SimRewind}(sec)$ process. We get it by modifying the simulation in exactly the same way as we modified the protocol to get $\mathsf{ProtRewind}(sec)$. More precisely, UCSim works by directly executing the protocol. In $\mathsf{SimRewind}(sec)$, we will modify the execution of each epoch to use rewinding exactly as in ProtRewind.

By various lemmas proved below, we will first be able to conclude the following

$$\mathsf{Protocol} \approx_s \mathsf{Protocol}^{\text{ZKSIM}} \approx_p \mathsf{ProtRewind}(d_S)$$

$$\mathsf{SimRewind}(d_S) \approx_p \mathsf{Simulation}^{\text{ZKSIM}} \approx_s \mathsf{Simulation}\ ,$$

where $\approx_p$ denotes perfect indistinguishability and $\approx_s$ denotes statistical indistinguishability. And then, using the computational assumptions we make we will, via a number of intermediate hybrids, conclude that $\mathsf{ProtRewind}(d_S) \approx_c \mathsf{SimRewind}(d_S)$, which implies the result we want. Here $\approx_c$ denote computational indistinguishability.

Process ProtRewind(sec).

Intialize. It is assumed that sec is a number chosen from the same domain as the secret Pailier key d_S. Emulate $\mathcal{F}_{\text{SETUP}}$, with one modification: Choose a valid sharing vector $\boldsymbol{v}_{sec}$ and give $\mathsf{VSS}(sec, \boldsymbol{v}_{sec})$ to C_0. Now, execute the protocol composed with Z and random oracle H, as specified, except that each epoch is done differently, as specified below. For every zero-knowledge proof from a corrupt player, extract on-line the witness used. As a result of this, and because the actions of honest players can be observed, the process knows the content of every Pailler ciphertext and every VSS share computed in the protocol.

Epochs execution. We specify how each epoch is done:

1. At the start of the epoch, let the sending committee execute the Reshare protocol as specified. For the receiving committee pair (C_{add}, C_{th}) receiving a VSS in this epoch, where C_{add} is the additive committee and C_{th} is the threshold one, do as follows:
   (a) Pick a random member of C_{add} to be the single inconsistent player (SIP).
   (b) When the committee pair does decryption of a batch of ciphertexts, execute the Decrypt protocol as is, except that the decryption messages of the SIP are computed in a special way. Let c be a ciphertext to decrypt after the randomization step (see Fig. 7), and note that the decryption result m is known. Let P_u be the SIP. Let the (correctly computed) decryption messages of the other parties in C_{add} be $\{d_{c,i} |\ i \neq u\}$ and set

$$d_{c,u} = (N+1)^m \prod_{i \neq u} d_{c,i}^{-1} \bmod N^{s+1}.$$

   Let P_u send $d_{c,i}, \pi_{c,u}$ using the Gather protocol, where $\pi_{c,u}$ is a simulated proof.
2. If at any point, the SIP in the additive committee is corrupted, or is not in the core set after the Gather protocol is done, rewind the entire process to the state it had at the start of the epoch, and go to step 1. If the epoch ends without rewinding happening, consider the set of honest parties that are in the core set after Gather (since we assume honest majority in committees, and the core has size at least $n - t$, there must be at least 1 such party). Select a random party P in this set. If P is not the SIP, rewind.
   If no rewind took place, continue to the next epoch.

Fig. 34. The ProtRewind process.

Lemma 12. *We have*

$$\mathsf{Protocol} \approx_s \mathsf{Protocol}^{ZKSIM},$$

$$\mathsf{Simulation}^{ZKSIM} \approx_s \mathsf{Simulation}.$$

Proof. This is immediate by statistical zero-knowledge of the proofs we use—even under adaptive corruption—as an honest party always deletes randomness and witness immediately after sending its single message. □

Lemma 13. *Under HCNF, we have*

$$\mathsf{Protocol}^{ZKSIM} \approx_p \mathsf{ProtRewind}(d_S),$$

$$\mathsf{SimRewind}(d_S) \approx_p \mathsf{Simulation}^{ZKSIM},$$

and each of the two processes run in expected polynomial time.

Proof. We argue that $\mathsf{Protocol}^{\text{ZKSIM}} \approx_p \mathsf{ProtRewind}(d_S)$: Note that the only difference between the two processes is that the decryption step is executed differently in the two cases, while the output plaintext is always the correct one that is actually contained in the ciphertext to decrypt.

Moreover, note that when $\mathsf{ProtRewind}(d_S)$ creates an execution of an epoch, it has exactly the same distribution as in $\mathsf{Protocol}^{\text{ZKSIM}}$. The decryption message from the SIP is computed in a different way, but its distribution remains the same, namely we maintain, even in the rewinding process that the decryption messages for ciphertext c and plaintext m are computed correctly from the corresponding shares for all parties except the SIP, and then the SIP's message is fixed by the equation $\prod_i d_{c,i} = (N+1)^m \bmod N^{s+1}$. Note that we do not get the decryption messages from the parties when they send them, which would give a problem with rushing. We compute them from the corresponding shares, which are known as we know the contents of all encryptions sent by corrupted parties. We are merely computing the correct decryption messages for the SIP in an indirect way from the result and the share of all other parties. This will perfectly give the same decryption message. In particular, this means that Z has no information on which party we choose as the SIP, all parties in C_{add} behave the same probability in the view of Z. This in turn implies that all decisions that lead to the choice of the player P at the end of an epoch are taken independently of the choice of SIP. Therefore, the probability that we do not rewind and keep the view for Z that we generated is exactly $1/n$, and the decision to rewind or not is independent of the actual view. This, and the fact that each view we make going forward has the right distribution, implies the first conclusion.

The second conclusion follows by essentially the same argument, we leave the details to the reader.

Finally, the claim on the expected run time follows from the fact that everything in the protoocol is polynomial time so the only question is regarding the expected number of times we rewind. Since there is one sending committees in an epoch, then since each decryption avoids rewinding with an independent probability $1/n$, the probability we get a complete simulation of an epoch is $1/n$ and hence the expected number of rewinds is n. □

$\mathsf{PPBox}(b)$.
After the initialize step where the Paillier key is generated, this PPBox will compute a series of sharings and resharings of d_S or a default value, according to the value of b. On request, it will output Paillier encryptions related to the shares. On getting a leakage query it will output a subset of the shares it has generated. The sequence of corruption commands is required to be admissible: only an unqualified set of each committee pair can be corrupted. Moreover, it is not allowed to call the Rewind command more than $2n$ times for any value of i_{VSS} (see definition below).

Intialize. When asked to initialize, generates the public and secret Paillier key N, d_S, as well as $w_S = E_{N,w_S}(1)$. Return N, w_S, and if $b = 1$, set $d = d_S$, else set $d = 0$.
Define initial committee pair C_1 and mark all members as honest. Compute a set of shares $\mathsf{sh}(d, \boldsymbol{v}_d)$ and assign it to the members of C_1.
i_{VSS} will be the index of the committee pair who will reshare it state next time. Set $i_{VSS} = 1$.

New Committee This command makes the box define a new committee. Committees are numbered sequentially.

One-time pad encryption This command points to an existing committee pair C_i. For each honest member of C_i, choose random one-time pads as in the $\mathsf{CreateInputData}$ protocol. Paillier encrypt each pad and return the ciphertexts. Note that in the protocol each pad will be used for communicating with a specific party from a different committee. If that party is corrupt, return also the pad in clear.

Reveal sums This command points to two consecutive committees C_i, C_{i+1} For each pair of honest members P_j, P_u of C_i, C_{i+1}, consider the set of one-time pads that would be used when doing the $\mathsf{SecretChannels}$ protocol for P_j, P_u. It consists of set of max_{pad} pairs $\mathsf{otp}_{\text{SND}}, \mathsf{otp}_{\text{REC}}$ held by P_j, P_u respectively. For each pair, return $\mathsf{otp}_{\text{SND}} + \mathsf{otp}_{\text{REC}}$. Choose a random subset $S_{j,u}$ of the pairs as in $\mathsf{SecretChannels}$. If one of P_j, P_u are corrupt accept the set $S_{j,u}$ as input.
Keep storing all the pairs, but now list only the pads in $S_{j,u}$ as assigned to P_j and P_u.

Fig. 35. The PPBox we use, Part I. See also Fig. 36

.

$\mathsf{PPBox}(b)$ (cont.).

Reshare On receiving this command, for every share s_i held by an honest member P_j of the threshold committee in $C_{i_{VSS}}$, compute a set of sub-shares $\mathsf{sh}(s_i, \boldsymbol{v}_{s_i})$, and let P_u be the receiver of the sub-share. For the encryption of sub-shares to be sent between P_j and P_u, use the pads in the set $S_{j,u}$. Specifically, for each individual sub-share $\mathsf{sh}(s_i, \boldsymbol{v}_{s_i})[v]$, let $\vec{\mathsf{otp}}_{i,v}$ be the one-time pad tuple assigned for encryption of the share. Return $E^{\rightarrow}_{\mathsf{otp}_{i,v}}(\mathsf{sh}(s_i, \boldsymbol{v}_{s_i})[v])$.

Choose a new unused subset and set $S_{j,u}$ to be this subset. List the otp's in the new subset as assigned to P_j, P_u.

Recombine Assumes that Reshare has been called. Input is a qualified set A of parties in the threshold committee of $C_{i_{VSS}}$, and for each s_i assigned to a corrupt member in A, a set of shares of form $\mathsf{sh}(s_i, \boldsymbol{v}_{s_i})$ must be supplied. Let $\boldsymbol{r}_A$ be the recombination vector for A and compute a new set of shares of d as

$$\sum_{i, P_{u(i)} \in A} \boldsymbol{r}_A[i] \cdot \mathsf{sh}(s_i, \boldsymbol{v}_{s_i}).$$

Assign these shares to $C_{i_{VSS}+1}$.

Rewind Assumes that Recombine has been called. Delete the shares defined for $C_{i_{VSS}+1}$, and delete the sub-shares generated for $C_{i_{VSS}}$. Mark all members of $C_{i_{VSS}+1}$ as honest.

Advance VSS index Let $i_{VSS} = i_{VSS} + 1$.

Corrupt This command points to a member of an existing committee pair. Mark the member P as corrupt. Return all shares and otp's currently assigned to P. If P was in the additive committee holding share s_i, return also all threshold shares of s_i held by members in the threshold committee.

Fig. 36. The PPBox we use, Part II. See also Fig. 35.

In the proof of the following lemma, we will use a PPBox specified in Fig. 35. It is designed to allow emulation of the resharing and decryption steps in the protocol. The defined commands are syntactically a bit different from those defined in the generic version in Section 2. However, the commands defined can be understood as special types of encryption, state update or leakage queries. Namely, the New Committee, Recombine, Rewind and Advance VSS index are state update queries, the One-time pad encryption and Reshare commands are encryption queries, while the Corrupt Command is a leakage query. Moreover the box assigns a set of one-time pads that can be used for encryption of each sub-share (when doing the Reshare command). However, it uses a new set of pads from the set for every Rewind command and the set is assumed to be large enough to allow for $2n$ rewind commands without reusing any pad. It will refuse to do more Rewinds. In addition the box also releases an encryption of a one-time pad for the receiver and the sum of the pads for sending and receiving. Therefore, the encryption of a share s used by the box effectively is of form

$$E_{N,w_S}(\mathsf{otp}_{\mathrm{SND}}), E_{N,w_S}(\mathsf{otp}_{\mathrm{REC}}), \mathsf{otp}_{\mathrm{REC}} + \mathsf{otp}_{\mathrm{SND}}, \mathsf{otp}_{\mathrm{SND}} + s \ .$$

Since the pads are chosen uniformly, this is clearly as secure as Paillier encryption itself, and so the box is encryption secure, as defined in Section 2. Finally, since the corruption queries cannot corrupt more than an unqualified set, the box is privacy preserving. Note that rewinding makes all members of the affected committee honest again so new corruptions can be done, but all shares are deleted, so this not a problem. Note also that corruption of

someone in an additive committee returns the threshold shares of the party's additive share, even if some that are held by members that are still honest. This is also not a problem since the threshold shares tell you nothing about the secret that was not already present in the additive share. So, as long as not all members of the additive committee are corrupted, the information returned is statistically independent of the secret.

Therefore if one plays the CSO game with this PPBox, CSO-security will imply that one cannot efficiently distinguish $\mathsf{PPBox}(0)$ from $\mathsf{PPBox}(1)$.

In the proof of the following lemma, we will also use an assumption saying that during protocol or simulation, Z does not call the random oracle on a certain input:

The No Random Oracle Call assumption (NROC): Assume that the SecretChannels protocol has been used to set up otp's for sending from P_j to P_u, and let $H(\mathsf{otp}_1)$ be the output they use for selecting the subset of otp's to retain. If both parties are honest immediately after they are done with SecretChannels, then Z never calls H on input otp_1.

Intuitively, this is reasonable, because the assumed honesty of parties implies that in the view of Z, otp_1 is only known in encrypted form and if any of the parties is corrupted later, otp_1 has been erased. We will assume NROC for now and later use CSO security and Lemma 8 to prove it.

Lemma 14. *Under HCNF and NROC, and assuming Paillier encryption is CSO-secure we have*

$$\mathsf{ProtRewind}(d_S) \approx_c \mathsf{ProtRewind}(0),$$

$$\mathsf{SimRewind}(d_S) \approx_c \mathsf{SimRewind}(0).$$

Proof. We will show that $\mathsf{ProtRewind}(d_S) \approx_c \mathsf{ProtRewind}(0)$, the proof of the other conclusion is essentially the same. We will use the CSO game with the PPBox specified in Figure 35.

We define a process Hybrid that interacts with Z and also plays the CSO game with $\mathsf{PPBox}(b)$. It runs in the same way as Protrewind, with the following modifications

1. Instead of emulating the set-up itself, it calls Intitialize on PPBox and gives the resulting N, w_S to all players. The rest of the set-up is done by following the $\mathsf{RoleBatches}^{\mathsf{eno}}$ protocol, so it is enough to describe how Hybrid emulates that protocol, which is done in the next steps.
2. To emulate an instance of $\mathsf{RoleBatches}^{\mathsf{eno}}$, recall that the NewRole protocol uses a set of random encrypted bits to select a player for a role. These bits are determined by the random oracle, it outputs a random ciphertext where the bit is determined by the Jacobi symbol of the plaintext. For all calls of this type to the random oracle, the process programs the oracle to output a ciphertext containíng a random plaintext chosen by the process. The process now knows the entire vector $\boldsymbol{b}$ of bits that determine which players are selected for which roles, but $\boldsymbol{b}$ is still correctly distributed.
3. Call New Committee on PPBox enough times that the box has defined eno committee pairs.

4. For each C_i in the new set of committee pairs, call One-time pad encryption. For each pair C_i, C_{i+1} call Reveal sums. For each pair the process now has encryptions of one-time pads for C_i when acting as sender E_i^{SND}, encrypted pads chosen for C_{i+1} when acting as receiver E_{i+1}^{REC}, and finally the sums of corresponding pads S_i.
5. For each committee pair C_i we now have E_i^{REC}, E_i^{SND}, and for each member of this committee pair note that the set of encryptions contain exactly the encryptions of one-time pads that the party having the corresponding role would publish in the CreateInputData protocol. Since the process knows $\boldsymbol{b}$, it knows which party, say P, will be assigned to the role. If P is corrupt, call Corrupt on PPBox and let Z decide the actions of P. If P is honest construct its input data I_P as follows:
   (a) Choose a random tag and add $E_{N,w_S}(\mathsf{tag})$ to I_P.
   (b) Add the relevant encrypted pads from E_i^{REC}, E_i^{SND} to I_P.
   (c) Add commitments in a form so they can be equivocated and add simulated zero-knowledge proofs as required to I_P

   The process publishes the constructed I_P on behalf of P.
6. The process lets the NewRole executions run according to the protocol.
7. In the executions of SecretChannels, the sum of a set of pairs $\mathsf{otp}_{\text{SND}}, \mathsf{otp}_{\text{REC}}$ is decrypted. Note that the process knows the sum. If both sender and receiver were honest when the input data was created, the sum is one of the entries in one of the S_i's released from the box. If one of the parties were corrupt, we get from PPBox the summand, say otp, it chose for the corrupted player, and we can extract from the zero-knowledge proofs the value otp' chosen by the adversary. Then the sum from PPBox can be adjusted by adding $\mathsf{otp}' - \mathsf{otp}$.

   Therefore, the process knows what is decrypted during SecretChannels, and this holds for all other ciphertexts decrypted, as argued when we specified ProtRewind. With this knowledge, the process can emulate the Decrypt protocols, happening in each epoch, as described next.
8. An epoch is done as follows:
   (a) At the start of an epoch the process calls Reshare on PPBox. This returns, for all honest members of the sending committee $C_{i_{VSS}}$ a set of encrypted (sub)shares. For each such member, the process extends this to a properly formatted VSS message by adding equivocable commitments and simulated zero-knowledge proofs as needed. Then include this in the message sent by the honest party.
   (b) Let A be the set of parties whose VSS messages are delivered on the ledger. Extract from the zero-knowledge proofs in the VSSs from corrupt players the (sub)shares they have chosen. Call Recombine on PPBox with input A and the extracted sets of shares. PPBox now has a valid sharing of the master secret assigned to $C_{i_{VSS}+1}$.
   (c) Select a SIP in $C_{i_{VSS}+1}$ as in ProtRewind and corrupt on PPBox all additive members in $C_{i_{VSS}+1}$ except for the SIP. Using the shares returned, the process can now emulate the execution of decryption exactly as in ProtRewind.
   (d) If the SIP is corrupted, a rewind is required. If $2n$ rewinds were done already, give up and stop. Else, call Rewind on PPBox and go to Step 8a.

(e) If the epoch is completed, call Advance VSS index on PPBox and proceed to the next epoch, or next batch if all epochs are done.

9. If a party is corrupted at a point where it has supplied input data in the beginning of a batch, but has not yet executed its role, corrupt this party on PPBox to get its one-time pads and (perhaps) shares, equivocate the commitments created for the party earlier to hold the (now known) values and return the resulting data to Z
10. Once the process has ended, output the bit that Z outputs.

It is now easy to verify that if the game with PPBox is played when the secret bit to guess is 1, we get a process that is statistically indistinguishable from $\mathsf{ProtRewind}(d_S)$. While if the bit is 0 we are statistically indistinguishable from $\mathsf{ProtRewind}(0)$.

Namely, PPBox prepares otp's and shares exactly as it would be done in the real protocol and Hybrid adds commitments and zero-knowledge proofs to match the format in the protocol. There are only two differences. First, when a rewind occurs, PPBox switches to a new subset of one time pads for share encryption, while ProtRewind does not. However, under NROC this difference cannot be detected: The issue only occurs if the sender and receiver we consider are both honest when the epoch starts, so they were also honest when SecretChannels was executed, so NROC applies. And, PPBox's selection of subset always has the right distribution. While the choice of subset may not match the value that would come from the random oracle, if Z has not called the oracle on the relevant input, it has no information on the output.

The second difference is that Hybrid never does more than $2n$ rewinds in an epoch, but since ProtRewind stops at every iteration with probability $1/n$ it clearly does more than $2n$ rewinds with negligible probability.

Hence Z's distinguishing advantage equals (except for a negligible amount) the advantage we have in winning the game, which is negligible by assumption. So the lemma follows. □

For the next two lemmas, we note that CPA security of Paillier encryption clearly follows from CSO-security, which we assume throughout.

We define a modification of $\mathsf{SimRewind}(0)$: $\mathsf{SimRewind}^{\mathrm{PROT}}(0)$, where we make the process look more like the protocol: we set the ciphertext c^* from the set-up used in the RandomizeCiphertext protocol so it contains 0 as in the protocol and we drop the programming of the random oracle that UCSim uses.

Lemma 15. *Under HCNF, NROC and CPA security of Paillier, we have*

$$\mathsf{SimRewind}(0) \approx_c \mathsf{SimRewind}^{PROT}(0).$$

Moreover an offending call to the random oracle that would make UCSim *fail, occurs with negligible probability in* $\mathsf{SimRewind}(0)$.

Proof. Follows immediately by CPA security of Paillier, as the only effect of the changes occur inside Paillier ciphertexts that we can control by programming the setup or the random oracle. Moreover, to make an offending call, Z would need to guess the value inside a ciphertext before it is opened. □

From the above lemma and the previous ones, we can now conclude that the offending call also must occur with negligible probability in Simulation (and the intermediate processes). By Lemma 8 this means that it holds in all hybrids that there is a negligible probability that there are offending calls to the random oracle.

We define processes $\mathsf{ProtRewind}_{\text{LOSSY}}(0)$ and $\mathsf{SimRewind}^{\text{PROT}}_{\text{LOSSY}}(0)$ where we replace the ciphertext w_S from the set-up by a ciphertext containing 0. This means that all ciphertexts generated by parties or in set-up are lossy, i.e., they actually contain 0, instead of the plaintext the party had in mind. Note, however, that it is still possible to extract witnesses from zero-knowledge proofs, showing how a ciphertext was formed, and hence which plaintext the party had in mind.

Lemma 16. *Under HCNF, NROC and CPA security of Paillier, we have*

$$\mathsf{SimRewind}^{PROT}(0) \approx_c \mathsf{SimRewind}^{PROT}_{LOSSY}(0),$$

$$\mathsf{ProtRewind}(0) \approx_c \mathsf{ProtRewind}_{LOSSY}(0).$$

Proof. Follows immediately from CPA security of Paillier, as the only effect of the changes occur inside a Paillier ciphertext from the setup. □

In both $\mathsf{SimRewind}^{\text{PROT}}_{\text{LOSSY}}(0)$ and $\mathsf{ProtRewind}_{\text{LOSSY}}(0)$, all information about random bits used for selecting parties for roles has been removed. Also, all one-time pads and tags chosen by honest parties are information theoretically hidden from Z until they are (possibly) decrypted. Therefore the corruptions done by Z are decided independently of the distribution of roles to parties, and the tags chosen by corrupt parties in CreateInputData are chosen independently of the tags of honest players, and will therefore collide with those tags with negligible probability. Finally, Z has no information on which value two honest players use in selecting the subset of one-time pads to use in the Secretchannels protocol. We therefore conclude that the HCNF and NROC assumptions hold in these processes, and by Lemma 8, we conclude that they also holds in all other processes, except with negligible probability.

It therefore follows by Lemma 11 that the original Protocol process generates correct outputs and this is preserved over the processes we have derived from the protocol. The same can be concluded for the processes derived from Simulation. We conclude that the output generated by $\mathsf{SimRewind}^{\text{PROT}}_{\text{LOSSY}}(0)$ and $\mathsf{ProtRewind}_{\text{LOSSY}}(0)$ have statistically indistinguishable distributions, i.e., indistinguishable from correct outputs from the MPC. Further, even though Simulation and processes derived from it use dummy inputs for honest players to the MPC, this cannot be detected once ciphertexts are lossy. Also observe that, except for the choice of inputs, the protocol is executed in exactly the same way in $\mathsf{SimRewind}^{\text{PROT}}_{\text{LOSSY}}(0)$ and $\mathsf{ProtRewind}_{\text{LOSSY},}(0)$. We conclude that

Lemma 17. $\mathsf{SimRewind}^{PROT}_{LOSSY}(0) \approx_s \mathsf{ProtRewind}^{LOSSY}(0)$

This concludes the proof of Theorem 6.

J Secure Coinflips Based on MPC

In this section we show how to implement $\mathcal{F}_{\text{MPC+CF}}$ based on $\mathcal{F}_{\text{MPC}}$. Thus, we need to describe a protocol that runs assuming $\mathcal{F}_{\text{MPC}}$ is available. The only new thing we need to implement is the coin flip, and this is done by asking $\mathcal{F}_{\text{MPC}}$ to evaluate an appropriate function and give the output to a number of committees. Concretely, the function $f_{\text{RSS}}(P_1, \ldots, P_n)$ creates a standard robust secret sharing of a random value modulo N, playing the role of the coin. The members of the committee $P_1, \ldots, P_n$ each learn a share and can then later reveal the value of the coin by sending shares to all players. Here, $P_1, \ldots, P_n$ should be understood as roles, that map to n random actual parties in the set $\mathbb{P}^{\mathsf{b}}$ receiving output from $\mathcal{F}_{\text{MPC}}$ in the given batch. The function f^{b} just outputs to n consecutive roles in $\mathbb{R}^{\mathsf{b}}$, as these roles are already randomly permuted using π^{b}.

We first describe the function to compute:

1. $f_{\text{RSS}}(R_1, \ldots, R_n)$ chooses random values $\mathsf{coin}, a_1, \ldots, a_t \in \mathbb{Z}_N^*$, defines a polynomial
$$p(X) = \mathsf{coin} + a_1 X + \cdots + a_t X^t \bmod N$$
and sets $\mathsf{sh}_j = p(j),\ j = 1, \ldots, n$.
2. For $i = 1, \ldots, n$, chooses $\alpha_i \in \mathbb{Z}_N^*$ at random. For $i, j = 1, \ldots, n$, choose $\beta_{i,j} \in \mathbb{Z}_N^*$ at random. Set $\mathsf{mac}_{i,j} = \alpha_i \mathsf{sh}_j + \beta_{i,j} \bmod N$.
3. Define the output out_j, for $i = j, \ldots, n$ as follows:
$$\mathsf{out}_j = \mathsf{sh}_j, \alpha_j, \beta_{j,1}, \ldots, \beta_{j,n}, \mathsf{mac}_{1,j}, \ldots, \mathsf{mac}_{n,j},$$
and output out_j to P_j.

The idea is that sh_j is the actual share of coin, the α_j and $\beta_{j,i}$ are keys that can verify authentication codes for other shares, and the $\mathsf{mac}_{i,j}$-values are authentication codes for sh_j that can be verified using key material from other players. It is well known that these macs are information theoretically secure: given $\mathsf{sh}_j, \mathsf{mac}_{i,j}$, producing a different pair $\mathsf{sh}'_j, \mathsf{mac}'_{i,j}$ satisfying $\alpha_i \mathsf{sh}'_j + \beta_{i,j} = \mathsf{mac}'_{i,j}$ requires that you guess α_i, which happens with negligible probability $1/N$.

In the protocol below, players will receive a subset of the out_j's, some of which may be incorrect. Given a received value sh_j, we will say that sh_j *has support from* R_i if it is the case that $\mathsf{mac}_{i,j} = \alpha_i \mathsf{sh}_j + \beta_{i,j}$, where $\alpha_i, \beta_{i,j}$ are the values received from R_i.

In addition to the above, when party P gives input to $\mathcal{F}_{\text{MPC}}$ they also give a role input $z_P = \mathsf{vk}_P$, where $(\mathsf{vk}_P, \mathsf{sk}_P) \leftarrow \mathsf{Sig.Gen}$ is a key-party for an EUF-CMA secure signature scheme. Party P keeps the signing key sk_P. The function f_{RSS} computes the identity function on the vk_P, i.e., it outputs $\{(R_P = \pi(P), \mathsf{vk}_P)\}_{P \in \mathbb{Q}}$ sorted on R_P. When P sends its message as role R_P it signs with sk_P and the receivers use the role name R_P to look up (R_P, vk_P) and verifies with vk_P. This ensure that only the genuine P can act as R_P. We skip these details below and just assume that one P can send in the name of R_P.

The protocol to implement $\mathcal{F}_{\text{MPC+CF}}$ in Figure 37 is very simple and is only described for a single coin, the extension to several coins is trivial.

We first show that the protocol has liveness and outputs the correct value.

Protocol Coinflip.

1. The protocol connects the interface of $\mathcal{F}_{\text{MPC+CF}}$ directly to the corresponding interface of $\mathcal{F}_{\text{MPC}}$, except for the part relating to coinflip.
2. If a coinflip ordered in the current batch, ask $\mathcal{F}_{\text{MPC}}$ to compute $f_{\text{RSS}}(R_1, \ldots, R_n)$, where $R_1, \ldots, R_n$ stands for the next n available roles.
3. The Flip Coin command is implemented by having P_j who has role R_j send out_j to all players and signing using sk_{P_j}. The receivers look up (R_j, vk_{P_j}) in vk and uses vk_{P_j} to verify the signature.
4. Each player waits until it has received $n-t$ shares that have support from at least $n-t$ roles. Interpolate a value coin from the first $t+1$ such shares and outputs coin.

Fig. 37. Protocol for Coin-Flip

Lemma 18. *Except with negligible probability, each player in the* Coinflip *protocol eventually outputs the correct value of coin.*

Proof. It follows from the above security property of the macs that (with overwhelming probability) no incorrect share can have support from more than t players. Thus, a share having support from $n - t > t$ players can be assumed to be correct. On the other hand, a correct share will eventually have support from all $n-t$ honest players, since all $n-t$ honest messages are eventually delivered. So, a player can safely wait until it gets $n-t$ shares with enough support, and since these shares can be assumed correct, the output is correct. □

Theorem 7. *The* Coinflip *protocol implements $\mathcal{F}_{MPC+CF}$ in the $\mathcal{F}_{MPC}$-hybrid model.*

Proof. We describe a simulator for the protocol. When a coin is ordered, the simulator evaluates f_{RSS} but sets $\mathsf{coin} = 0$. It hands the resulting out_j-values to P_j for corrupt P_j and stores the honest values. Until the coin is to be revealed, if a new player P_j is corrupted, hand out_j to P_j. When coin is leaked from $\mathcal{F}_{\text{MPC+CF}}$, interpolate a polynomial $g(X)$, such that $g(0) = \mathsf{coin}$ and $g(j) = 0$ for all corrupt P_j. For each honest R_i, update out_i as follows: set $\mathsf{sh}_i = \mathsf{sh}_i + g(i)$, and update all $\mathsf{mac}_{j,i}$ values such that the new value of sh_i and the new macs verify against all the key material. This is trivial by solving n linear equations. Send the resulting out_i values on behalf of the honest players. When enough messages are delivered to honest player P, send a Deliver command to $\mathcal{F}_{\text{MPC+CF}}$.

It is straightforward to verify that this simulation is statistically indistinguishable from the protocol. The data from honest players and resource functionality seen by the environment have exactly the same distribution as in the real protocol, so by (the proof of) Lemma 18, the only source of error is when a corrupt player successfully forges a share, which happens with negligible probability. □