How to Withstand Mobile Virus Attacks, Revisited

Joshua Baron[1] Karim El Defrawy[1] Joshua Lampkins[2*]
Rafail Ostrovsky[2,3]
{jwbaron,kmeldefrawy}@hrl.com, jlampkins@math.ucla.edu, rafail@cs.ucla.edu
[1] HRL Laboratories, LLC, Malibu, CA
[2] Department of Mathematics, UCLA, Los Angeles, CA
[3] Departments of Computer Science, UCLA, Los Angeles, CA

Abstract

Secure Multiparty Computation (MPC) protocols allow a set of distrusting participants to securely compute a joint function of their private inputs without revealing anything but the output of the function to each other. In 1991 Ostrovsky and Yung introduced the *proactive security model*, where faults spread throughout the network, analogous to the spread of a virus or a worm. More specifically, in the proactive security model, the adversary is not limited in the number of parties it can corrupt but rather in the *rate* of corruption with respect to a "rebooting" rate. In the same paper, Ostrovsky and Yung showed that constructing a general purpose MPC protocol in the proactive security model is indeed feasible when the rate of corruption is a constant fraction of the parties. Their result, however, was shown only for stand-alone security and incurred a large polynomial communication overhead for each gate of the computation. In contrast, protocols for "classical" MPC models (where the adversary is limited to corrupt in total up to a fixed fraction of the parties) have seen dramatic progress in reducing communication complexity in recent years.

The question that we consider in this paper is whether continuous improvements of communication overhead in protocols for the "classical" stationary corruptions model in the MPC literature can lead to communication complexity reductions in the proactive security model as well. It turns out that improving communication complexity of proactive MPC protocols using modern techniques encounters two fundamental roadblocks due to the nature of the mobile faults model: First, in the proactive security model there is the inherent impossibility of "bulk pre-computation" to generate cryptographic material that can be slowly consumed during protocol computation in order to amortize communication cost (the adversary can easily discover pre-computed values if they are not refreshed, and refreshing is expensive); second, there is an apparent need for double-sharing (which requires high communication overhead) of data in order to achieve proactive security guarantees. Thus, techniques that were used to speed up classical MPC do not work, and new ideas are needed. That is exactly what we do in this paper: we show a novel MPC protocol in the proactive security model that can tolerate a $\frac{1}{3} - \epsilon$ (resp. $\frac{1}{2} - \epsilon$) fraction of moving faults, is perfectly (resp. statistically) UC-secure, and achieves near-linear communication complexity for each step of the computation. Our results match the asymptotic communication complexity of the best known results in the "classical" model of stationary faults [DIK10]. One of the important building blocks that we introduce is a new near-linear "packed" proactive secret sharing (PPSS) scheme, where the amortized communication and computational cost of maintaining each individual secret share is just a constant. We believe that our PPSS scheme might be of independent interest.

Keywords: Proactive security, secure multiparty computation, secret sharing.

1 Introduction

Secure multiparty computation (MPC) is a notion central to cryptography. MPC protocols allow a set of distrusting parties $P_1, ..., P_n$, with private inputs $x_1, ..., x_n$, to jointly compute a function f while

*The work of this author was performed while at HRL Laboratories, LLC.

guaranteeing correctness of its evaluation and privacy of inputs for honest parties. The study of secure computation was initiated by [Yao82] for two parties and [GMW87] for many parties. The information-theoretic setting was introduced by [BOGW88] and [CCD88], where assuming private channels MPC protocols were shown to tolerate less than 1/3 of malicious parties. Assuming a broadcast channel, [RB89] shows how protocols can tolerate less than 1/2 of malicious parties. Fixed bounds on adversary's corruption limit can be viewed as unrealistic for protocols that have very long execution times, especially when considering so-called "reactive" functionalities that never stop executing, e.g., continuously running control loops. Constructing MPC protocols that guarantee security against stronger adversary models and at the same time satisfy low communication and computational complexity bounds has been an important program in cryptography, and has seen significant progress, e.g., [IKOS08, DIK$^+$08, DIK10, BFO12].

An approach to deal with an adversary's ability to eventually corrupt *all* parties is the so-called proactive security model [OY91], which introduces the notion of a *mobile adversary* motivated by the persistent corruption of participants. A mobile adversary is one that can corrupt all parties in a distributed protocol during the execution but with the following limitations: (1) only a constant fraction of parties can be corrupted during any round; (2) parties periodically get rebooted to a clean initial state, guaranteeing small fraction of corrupted parties, assuming that the corruption rate is not more than the reboot rate[1].

We remark that we model rebooting to a clean initial state including global computation information (e.g., the circuit to be computed, the identities of the other parties in the computation, access to perfectly secure point-to-point channels and to a broadcast channel). The [OY91] model also assumes that an adversary does not have the ability to predict or reconstruct the randomness used by parties in any uncorrupted period of time, as demarcated by rebootings.

This paper's goal is to construct an MPC protocol under the *proactive security model* with low communication complexity. We specifically consider two questions (1) can we construct proactive *UC-secure* MPC protocols and (2) how much can we improve the communication complexity of previous proactive protocols, both for secret sharing and MPC?

1.1 Related Work

Ostrovsky and Yung introduced the proactive security model in [OY91]. The same paper also contains the first proactive secret sharing (PSS) scheme and proactive MPC (PMPC) protocol. Following [OY91], there has been significant follow up work on PSS schemes, both in the synchronous and asynchronous models (see Table 1 for a comparison). The most efficient scheme is [HJKY95], which has $O(n^2)$ communication complexity per secret share. By contrast, our work has $O(1)$ (amortized) communication complexity per secret share. Further, we are not aware of any UC-secure PSS scheme until this work. We note that our work is in the synchronous model; extending our work to the asynchronous model is an interesting open question.

In addition to proactive secret sharing, there has also been substantial research on proactively secure threshold encryption and signature schemes (e.g., [FGMY97a, FGMY97b, Rab98, CGJ$^+$99, FMY01, Bol03, JS05, JO08, ADN06]).

The only known PMPC protocol is due to [OY91]. The protocol is proven secure in the stand-alone corruption model and requires at least $O(Cn^3)$ communication complexity. By contrast, the PMPC protocol in this paper is UC-secure and has near-linear communication complexity.

[1]Deciding which parties to reboot is outside the scope of this paper; this paper only requires that the corruption rate is not exceeded. One option is that that parties are rebooted when anti-virus and/or intrusion detection systems detect a compromise. Another option is to periodically select a set of parties at random and reboot them. However, if parties are randomly rebooted, perfect security is unattainable because with (very) low probability the same set of parties may always be rebooted.

Paper	Network	Security	Threshold	Communication Complexity
[WWW02]	synch.	cryptographic	$t/n < 1/2$	$\exp(n)$
[ZSvR05]	asynch.	cryptographic	$t/n < 1/3$	$\exp(n)$
[CKLS02]	asynch.	cryptographic	$t/n < 1/3$	$O(n^4)$
[Sch07]	asynch.	cryptographic	$t/n < 1/3$	$O(n^4)$
[HJKY95]	synch.	cryptographic	$t/n < 1/2$	$O(n^2)$
This Paper	synch.	perfect	$t/n < 1/3-\epsilon$	$O(1)$
This Paper	synch.	statistical	$t/n < 1/2-\epsilon$	$O(1)$

Table 1: Comparison of Proactive Secret Sharing (PSS) Schemes. Threshold is for each reboot phase. Our communication complexity is amortized per bit.

1.2 Roadblocks to Proactive MPC (PMPC) with Low Communication

Naively, one might think that constructing a PMPC protocol would be a simple matter of starting with an existing MPC protocol that relies on secret sharing and replacing the secret sharing scheme with a PSS scheme, i.e., a scheme that has "proactivized" the original secret sharing scheme. While this simple strategy may work, i.e., MPC combined with PSS yields PMPC, it is not the case that simply combining PSS and MPC protocols leads to an *efficient* PMPC protocol due to the following roadblocks:

PSS Communication Complexity: A typical construction for (honest majority) MPC protocols computes arithmetic gates on secret-shared inputs. Since there exist PSS schemes (see Table 1), a naive attempt to construct a PMPC protocol would be to use a PSS scheme instead of a standard secret sharing protocol for an honest majority MPC protocol. However, such a construction results in high communication complexity. A far more efficient PSS scheme is needed that incurs *constant* (amortized) communication complexity per secret shared to yield a more efficient PMPC protocol.

Efficient Proactive Share Redistribution: Related to the issue of high communication complexity, since a mobile adversary can eventually corrupt *all* parties, if a normal secret sharing scheme is used, then the adversary would eventually recover all shares of all secrets, compromising the security of the computation. To mitigate this, PSS schemes such as [HJKY95] and [Sch07] have a renewal procedure by which sharings of secrets are re-randomized so that old shares "expire." Additionally, once parties are rebooted, since they no longer have the state required to participate in the computation, such state must be jointly reconstructed for them by the rest of the parties. To remedy this, the PSS schemes in [HJKY95] and [Sch07] incorporate a recovery procedure by which rebooted parties can recover the required state shares to continue the computation. Since data recovery occurs over the course of the computation, the repeated communication cost of redistribution contributes significantly to the overall communication complexity of the PMPC protocol. All of the PSS schemes listed in Table 1 have a communication complexity for redistribution of at least $O(n^2)$ per secret, where n is the number of parties. The redistribution protocol described in this paper is much more efficient, with an amortized communication complexity of $O(1)$ per secret.

MPC Pre-computation: A common technique utilized in recent MPC literature [DIK10, DIK$^+$08, BFO12] to reduce communication complexity is the use of a pre-computation phase to distribute input-independent information for use throughout the rest of the protocol. A challenge facing constructing a PMPC protocol is that any pre-processed data generated at the outset of the protocol must be periodically redistributed throughout the computation until it is used. Therefore, "proactivizing" existing pre-computation techniques naively would greatly increase the communication complexity of the PMPC protocol. Therefore, a different approach to distribute input-independent shares is required.

Per-Round Corruption Rate: Round complexity in a PMPC protocol must be carefully managed in order to obtain a maximal per-round corruption rate. More specifically, one must assume some limitation on how quickly a mobile adversary can corrupt new parties. To make this assumption concrete, one assumes that there is some threshold fraction of parties the adversary can corrupt per communication round. In order to asymptotically maximize the per-round corruption rate, one must construct a PMPC

protocol with only a constant number of rounds between share redistributions.

Handling Corruption: In the non-proactive case, the total communication complexity in an MPC protocol due to verifying party corruption is low because the number of corrupted parties over the course of the protocol is bounded and therefore the communication complexity is independent of the circuit size. However, in the proactive security model, the number of corrupted parties over the course of the entire protocol is $O(Dn)$ (where D is the depth of the circuit) and therefore care must be taken in ensuring that the communication complexity to handle corrupt parties is low.

1.3 Main Ideas Behind Our Protocols

In this paper, we construct the first perfectly UC-secure proactive MPC protocol with near-linear communication complexity using the following new ideas.

Packed Proactive Secret Sharing (PPSS). This paper presents the first "packed" proactive secret sharing (PPSS) scheme. In particular, we construct the perfectly UC-secure PPSS scheme by extending techniques presented in [DIK10, FY92] to the proactive security model. The new scheme allows one to share *many* secrets with constant (amortized) communication complexity per secret. In order to renew, i.e., re-randomize, sharings of secrets, the parties generate random masking polynomials. To renew a polynomial f that stores a block of secrets via the evaluation points $(f(\beta^1), \ldots, f(\beta^\ell))$, parties generate a random polynomial R which evaluates to zero at $\beta^1, \ldots, \beta^\ell$ and set the renewed polynomial to be $f + R$. In contrast to previous schemes (such as [HJKY95] and [Sch07]), our protocol amortizes random polynomial generation by using hyper-invertible matrices to more efficiently construct batches of random polynomials in a new way.

The process of reconstructing new shares for rebooted parties is a major communication bottleneck in previous work [Sch07, HJKY95]. Our redistribution protocol combines double sharing with packed sharing (the first protocol to do so) to achieve an $O(1)$ per-secret amortized communication complexity, improving upon previous protocols by a factor of n. In order to use these double sharings, it is necessary to verify that the players shared their shares correctly. This is accomplished using hyper-invertible matrices in a novel way. Previous protocols [BTH08, DIK$^+$08] have used hyper-invertible matrices for error correction by multiplying a vector of data structures by a hyper-invertible matrix and having each party check one entry/data structure in the resultant vector for errors. In [DIK$^+$08], the data structures are packed sharings; in [BTH08], the data structures are pairs of Shamir-sharings that share the same value. In our protocol, each data structure is a collection of sharings and corresponding double sharings.[2] After multiplication by a hyper-invertible matrix, each player checks one entry in the resultant vector to make sure that the double sharings contain the correct shares. By using hyper-invertible matrices with these larger data structures, which is a technique unique to this paper, we are able to check the correctness of the double sharings without asymptotically increasing the communication complexity of the protocol.

Proactive MPC (PMPC) Techniques. The main approach for our PMPC protocol is to compute the circuit layer by layer, while proactively redistributing the parties' secret shares after each layer. Therefore, if each layer is computed using subprotocols that are secure in the *non-proactive* setting up to a threshold t, and our PPSS share redistribution protocol is also secure up to threshold t, then the overall PMPC protocol is *proactively* secure up to a threshold t between proactive refreshes. The approach to construct our PMPC protocol is realized as follows:

During initialization of the protocol, the circuit is transformed so that each layer contains either only addition gates or only multiplication gates; each gate has two inputs, multiplication gates have one output, and addition gates have either one or two outputs. This transformation is necessary in order to do arithmetic with block-shared secrets, and it does not asymptotically increase the size of the circuit (measured as the number of gates plus the number of wires).

[2] [BTH08] uses the phrase "double sharing" differently than in this paper.

Each party then shares their inputs using the PPSS scheme. The circuit is then evaluated layer by layer. Before each layer, the secrets are permuted so that they are in the correct order for performing the arithmetic operations for that layer. For an addition layer, the gates are computed by locally adding shares. For a multiplication layer, the gates are computed using a standard technique.

After each layer, the parties execute the new redistribution protocol. This re-randomizes the sharings of the secrets so that old shares are erased and leak no information about current sharings to an adversary that obtained them. Redistribution also allows parties that have been rebooted to recover the required shares to continue the computation. This is where our new PPSS scheme is critically utilized.

Once all the layers of the circuit have been evaluated, the parties reconstruct the shares to obtain their outputs.

The initial PMPC protocol that is constructed has a corruption rate threshold less than $(1/3-\epsilon)n$ and subsequently uses Bracha committees [Bra87] to obtain a protocol with the desired threshold. However, one must overcome an additional difficulty because using Bracha committees might violate the need to have a constant number of rounds between share redistributions. More specifically, we construct a new constant-round multiparty Berlekamp-Welch protocol to satisfy the constant round requirement.

We remark that our techniques can be used to also construct a proactive, *statistically* UC-secure MPC scheme with similar communication and computational complexity that tolerates up to $(1/2-\epsilon)n$ corruptions between refreshments. See Appendix E for further discussion.

1.4 Contributions

The main contribution of this paper is a new proactive MPC (PMPC) protocol with communication complexity on par with the most efficient non-proactive MPC protocol in the literature today [DIK10]. The constructed PMPC protocol greatly improves on the communication complexity of the only other known PMPC protocol in [OY91]. The newly constructed PMPC protocol has communication complexity $O(C\log^2(C)\text{polylog}(n) + D\,\text{poly}(n)\log^2(C))$ with perfect UC-security in the synchronous model against an adversary that can corrupt up to $(1/3-\epsilon)n$ parties between refreshments for any constant $\epsilon > 0$. The PMPC protocol can also be modified to be statistically UC-secure and tolerate up to $(1/2-\epsilon)n$ corruptions between refreshments with similar communication and computational complexity and as in the perfectly secure case.

A second contribution of this paper is a "packed" proactive secret sharing (PPSS) scheme that is instrumental in achieving the low communication complexity in the new PMPC protocol. The PPSS scheme has an amortized communication complexity of $O(1)$ per shared secret and is perfectly UC-secure in the synchronous model against an adversary that can corrupt up to $(1/3-\epsilon)$n parties per fixed period of time for any constant $\epsilon > 0$. The PPSS protocol can also be modified to be statistically UC-secure and tolerate up to $(1/2-\epsilon)n$ corruptions between refreshments. We believe that our PPSS protocol may be of independent interest.

2 Definitions and Preliminaries

This section outlines the main techniques required for constructing the proactive MPC protocol. It also contains a brief discussion of the proactive security model under the UC framework (for further details, see Appendix A).

2.1 The Proactive Model under the UC Framework

Security of PMPC is proven in the Universal Composability (UC) framework introduced in [Can01] and revised in [Can05]. Specifically, the proactive UC model considered in this paper considers parties that can perform erasures; in the proactive model, parties must be able to erase their states so that when they are compromised, an adversary only learns their current state and not all their previous ones.

Parties communicate synchronously and have access to secure point-to-point channels and a broadcast channel; how this would be implemented is beyond the scope of this paper.[3] Similar to the definition of proactive UC security in [ADN06], the execution of a proactive protocol, π, proceeds in communication rounds, denoted by $r_{i,l}$, and the initial round is round $r_{0,0}$. A *proactive protocol* proceeds in *phases*. A phase, denoted ph consists of a number of consecutive rounds $\mathsf{r}_{i,l}, ..., \mathsf{r}_{i+j,l}$, and every round $\mathsf{r}_{j,l}$ belongs to exactly one phase ph_l. Each phase of π is either a *refreshment* or an *operation* phase. The phases of π alternate between *refreshment* and *operation* phases. Each *refreshment* phase ph_l consists of rounds $\mathsf{r}_{i,l}, ..., \mathsf{r}_{i+j,l}$, such that there exists a k, $0 \leq k < j$ where rounds $\mathsf{r}_{i,l}, ..., \mathsf{r}_{i+k,l}$ are denoted the *closing* period of refreshment phase ph_l while $\mathsf{r}_{i+k+1,l}, ..., \mathsf{r}_{i+j,l}$ denote the *opening* period of refreshment phase ph_l. Finally, a *stage* st (starting at stage 0) consists of an *opening* refreshment period, an operation phase and then a *closing* refreshment period, therefore including a full (operation) phase and two sequences of two refreshment stages; each refreshment is the closing of one stage and the opening of the other. An adversary in the proactive model is limited to corrupting a certain threshold of parties per *stage*. At the end of each stage (i.e., at the refreshment phase) corrupted parties are rebooted to a pristine state and sent the shares to continue the computation. We note that, as in [ADN06], a party corrupted during the refreshment phase is considered to be corrupted in both stages associated with that phase. We refer the reader to Section 1 for a discussion of rebooting. Appendix A contains the exact details of the the proactive UC model as well as definitions of security.

2.2 Preliminaries

Packed Secret Sharing. Our protocol relies on a generalization of Shamir's polynomial-based secret sharing scheme [Sha79]. In particular, we start with a variant of the *packed secret sharing scheme* due to [FY92] which was utilized in an MPC protocol by [DIK$^+$08, DIK10]. The scheme works as follows: for n parties and $d \in \mathbb{N}$, fix a finite field $\mathbb{Z}_q$ for $q > 2n$ as well as a generator $\alpha \in \mathbb{Z}_q^*$. Set $\beta = \alpha^{-1}$; the field $\mathbb{Z}_q$ is large enough so that $\beta^1, ..., \beta^d, \alpha, ..., \alpha^n$ are distinct for $d \leq n$. A vector $(x_1, ..., x_\ell)$ of secrets is shared simultaneously as a *block* by extending the vector to $(x_1, ..., x_\ell, r_{\ell+1}, ..., r_{d+1})$ for $r_{\ell+1}, .., r_{d+1}$ chosen uniformly and independently at random and constructing the unique degree d polynomial p such that $p(\beta^i) = x_i$ for $1 \leq i \leq \ell$ and $p(\beta^j) = r_j$ for $\ell + 1 \leq j \leq d + 1$. Each party P_i receives the share $p(\alpha^i)$. We denote the sharing in this fashion of $x = (x_1, ..., x_\ell)$ as $[x]_d$, where $[x]_d$ is the ordered n-length vector consisting of the respective shares for each P_i. The reader can verify that $c[x]_d = [cx]_d$ for any $c \in \mathbb{F}_p$, $[x]_d + [y]_d = [x + y]_d$, and $[x]_d[y]_d = [xy]_{2d}$, where addition and multiplication here are element-wise. We refer the reader to [DIK$^+$08, DIK10] for further details.

In what follows, we will require that ℓ is the highest power of 2 not greater than $n/4$, that the maximal corruption rate is $t = n/8$ and[4] that is $d = t + \ell - 1$. We will rely on the constant-round protocol $\mathsf{RobustShare}$ in [DIK$^+$08] to implement this sharing; it is perfectly UC-secure with the above parameters. We stress, however, that the techniques above have never been adopted to the proactive setting before; in particular, one of the main contributions of this paper is to perform a new proactive packed secret share redistribution with low amortized overhead.

Hyper-Invertible Matrices. Many of the subprotocols that we use require a publicly agreed upon hyper-invertible matrix [BTH08]. A hyper-invertible matrix is a matrix where any square sub-matrix formed by removing rows and columns is invertible. It is shown in [BTH08] that one can construct an $a \times b$ hyper-invertible matrix M as follows: Pick $a+b$ distinct field elements $\theta_1, \ldots, \theta_a, \phi_1, \ldots, \phi_b \in \mathbb{F}$, and let M be the matrix be such that if $(y_1, \ldots, y_a)^T = M(x_1, \ldots, x_b)^T$; then the points $(\theta_1, y_1), \ldots, (\theta_a, y_a)$ lie on the polynomial of degree $\leq b-1$ which evaluates to x_j at ϕ_j for each $j = 1, \ldots, b$. (In other words, M interpolates the points $\theta_1, \ldots, \theta_a$ on a polynomial given the points $\phi_1, \ldots, \phi_b$ on that polynomial.)

[3] This could be implemented using proactive PKI or by using a TPM to perform encryption/decryption.

[4] Our MPC protocol without additional party virtualization will therefore have a maximal per-stage corruption rate of $t = n/8$. Only after using Bracha committee-based [Bra87] techniques will our protocol have a corruption rate of $t = (1/3 - \epsilon)n$ for perfect security and $t = (1/2 - \epsilon)n$ for statistical security.

Many of the sub-protocols assume the existence of a publicly known hyper-invertible matrix, and these may be efficiently constructed during pre-processing.

Polynomial Interpolation. One algorithm relied upon for our proactive share redistribution protocol is the classic Berlekamp-Welch algorithm [Ber84]. If a party is given points on a polynomial (such as shares of a block of secrets) and some of the points have been corrupted (such as when corrupted parties alter their shares), the Berlekamp-Welch algorithm allows correct interpolation of the polynomial despite the corrupted points.

Recall that the basic outline of the Berlekamp-Welch algorithm is as follows: A party P_k receives a vector of shares $(y_1, \ldots, y_n)$ where each honest P_i sent $y_i = p(\alpha_i)$ for the degree d polynomial p that P_k is trying to interpolate. Denote the set of all i such that $y_i \neq p(\alpha_i)$ by I. We define a polynomial $g(x) = \prod_{i \in I}(x - \alpha_i)$, and define another polynomial $h = p \cdot g$. Note that the relation $h(\alpha_i) = y_i g(\alpha_i)$ holds for all $i = 1, \ldots, n$. These n relations are used to construct a matrix equation for the coefficients of h and g which the party solves and computes p by dividing h by g. This matrix equation can be solved efficiently using Fast Fourier Transform (FFT).

3 PMPC Protocol Details

This section describes the details of our efficient proactively secure MPC protocol. We first give a more detailed overview of the protocol construction and the techniques and subprotocols required. We then construct an efficient packed proactive secret sharing (PPSS) scheme. Next, we discuss the circuit transformations that are required to perform required operations on packed secret shares, with full details given in Appendix C. Finally, we construct the full protocol secure with per-stage corruption threshold $n/8$. In order to obtain a corruption threshold of $(1/3-\epsilon)n$ for perfect security or $(1/2-\epsilon)n$ for statistical security, we use a Bracha committee construction; due to lack of space, we discuss the construction in Appendix E.

3.1 Required Subprotocols

To perform basic operations such as secret sharing, generating random sharings, and multiplying shared secrets, we use three protocols from [DIK+08] (RobustShare, RanDouSha, and Reco) and three from [DIK10] (RandomPairs, PermuteWithinBlocks, and Multiply). We refer the reader to those works for protocol specifications and corresponding ideal functionalities. Each of these protocols is constant-round and is proven secure in their respective papers, though in the *standard*, non-proactive model with a corruption threshold of at least $n/8$. The sharings used in these protocols are all block sharings as described in Section 2.2.

- RobustShare(d): Allows a set of parties to verifiably share $\Theta(n)$ secrets in blocks with polynomials of degree d.
- RanDouSha(d): Generates random sharings of blocks of secrets $r^{(i)}$ and shares each of them with a degree d sharing $[r^{(i)}]_d$ and a degree $2d$ sharing $[r^{(i)}]_{2d}$.
- RandomPairs(L,π,d): For a permutation π, this protocol generates L pairs of random block-sharings $([r^{(i)}]_d, [\pi(r^{(i)})]_d)$ for $1 \leq i \leq L$ such that the secret stored in location j in the sharing $[r^{(i)}]_d$ is stored in location $\pi(j)$ in $[\pi(r^{(i)})]_d$.
- PermuteWithinBlocks: Using random masks $([r]_d, [\pi(r)]_d)$ generated with RandomPairs, this protocol applies a permutation to a block $[x]_d$ of already-shared secrets, resulting in a sharing $[\pi(x)]_d$.
- Multiply($[x]_d, [y]_d, ([r]_d, [r]_{2d})$): Multiplies two blocks of secrets $[x]_d$ and $[y]_d$ element-wise using the random pair $([r]_d, [r]_{2d})$. In the output sharing, the secret in location j is the product of the secret in location j in $[x]_d$ and the secret in location j in $[y]_d$.
- Reco(P_i, d): Reveals an already-shared block of secrets $[x]_d$ to a party P_i.

3.2 PMPC Protocol Overview

We first provide an overview of how we construct the PMPC protocol. As discussed in Section 3.1, we rely upon several subprotocols already in the MPC literature.

We first construct our efficient PPSS scheme. We do so using the packed secret sharing subprotocol RobustShare, which shares packed secrets in the non-proactive model [DIK$^+$08, DIK10]. We construct the subprotocol Block-Restribute to perform proactive secret redistribution to complete the PPSS scheme. See Section 3.3 for details.

We stress that each of the subprotocols that we use, including Block-Redistribute, are proven secure individually in the *non-proactive* security model. However, by composing them properly, namely by applying Block-Redistribute after each layer of the computation, we obtain proactive security. This is because by construction at each stage, the number of corrupted parties is at most the stage corruption threshold; executing Block-Redistribute ensures that corruptions in one stage cannot be carried over to break security of another future stage.

During initialization of the PMPC protocol, the circuit is transformed so that all addition and multiplication gates have only two inputs and either one or two outputs and are arranged in such a way that each layer consists of only addition gates or multiplication gates. This increases the circuit size by a constant factor, and increases circuit depth by a $\log \mathcal{C}$ multiplicative factor.

In the first step of the protocol, parties share their inputs using RobustShare. Over the course of the protocol, whenever random shares are needed, either by RanDouSha or RandomPairs, they are generated within a constant number of circuit layers from use; this "dynamic" preprocessing is so that Block-Redistribute does not have to be executed unnecessarily to maintain these shares, thereby increasing the asymptotic communication complexity.

Before each layer of the circuit is computed, the secrets are permuted (or re-arranged) to facilitate the process of computation. This is because the secrets are shared in blocks and they must be rearranged to compute on them per circuit specification. This rearrangement is performed by decomposing the required permutation into sub-permutations and then performing each of the sub-permutations in succession; this adds another $\log \mathcal{C}$ multiplicative factor to the communication complexity. See Section 3.4 and Appendix C for details.

After the permutation has been executed, addition gates are evaluated via locally adding shares, while multiplication gates are evaluated by creating pairs of random sharings using RanDouSha (through dynamic preprocessing) and then executing Multiply.

After each layer of the circuit is evaluated, where we now include each sub-permutation execution to constitute a layer, the parties run Block-Redistribute to re-randomize all stored secrets, thereby preserving proactive privacy of all stored values.

Once a sharing for an output gate has been computed, the parties invoke Reco to reveal it to the intended recipient. Once all the outputs have been revealed, the protocol is complete.

3.3 Proactive Share Redistribution

The share redistribution protocol Block-Redistribute forms the core of our PPSS scheme, and forms the redistribution phase of PMPC. It is constructed to redistribute shares for W secrets shared among the parties $\mathcal{P}$ (some of whom may be recently rebooted and do not actually have any shares). The subprotocol consists of three consecutive phases. First, the parties *rerandomize* their shares so that secret shares in the next proactive stage are independently distributed from previous stages. The parties then perform *verifiable double sharing* in order to correctly and privately distribute party share information to all parties without actually revealing the shares. Finally, the parties perform *share reconstruction* to restore the current state of the computation to newly rebooted parties who do not have shares to continue the computation. We will give an overview of each phase in Section 3.3.1 and then specify Block-Redistribute in Section 3.3.2. The ideal functionality $\mathcal{F}_{BR}$ that the protocol is designed to emulate is described in

Figure 3 in Appendix B, where the security proof for Block-Redistribute is also given.

3.3.1 Outline of Block-Redistribute

We refer the reader to Section 2 for the notation used in this section. For simplicity, we assume that the number of secrets, W, to be used as input for Block-Redistribute is a multiple of $\ell^2(n-3t)$ (e.g., $W = B\ell^2(n-3t)$ for some B), where secrets are shared[5] in blocks of size ℓ. We can then arrange the polynomials $\widehat{H}$ corresponding to these shares as

$$\left\{\widehat{H}_a^{(k,m)}\right\}_{\substack{m=1,\ldots,B\\ k=1,\ldots,n-3t\\ a=1,\ldots,\ell}}, \tag{1}$$

where $W = B\ell^2(n-3t)$. We think of B as the number of groupings of secret shares, and operations will be performed on each group in parallel throughout the protocol. We first provide some intuition for this arrangement of polynomials as well as for the parameters a, k, and m, before further outlining Block-Redistribute.

We will require a particular structure of secrets in order to perform share reconstruction in an efficient fashion. The idea is that we want to be able to allow rebooted parties to reconstruct their shares in such a way that corrupt parties cannot learn any information about honest parties' shares. We accomplish this by having parties distribute shares of their shares. The parameter a is bounded by ℓ because every ℓ shares (each corresponding to ℓ secrets) will form the basis for a single polynomial for the double sharing. One can perform computations on these double sharings using Lagrange coefficients to perform share reconstruction for the rebooted parties.

In order to verify that these double sharings are correct, the parties arrange their shares and double shares in B vectors of length $n-3t$, with an additional t elements chosen at random to further mask the inputs; this is why k is bounded by $n-3t$. A hyper-invertible matrix M will be applied to each of these B vectors to obtain B vectors of length n, corresponding to the n parties. Each party P_i will check the double sharings by each checking the ith entry of the B output vectors; in this fashion, we completely distribute double sharing verification. Because M is hyper-invertible, the outputs that each party checks will be immune from corrupt parties trying to skew the verification. We now explain the three phases of Block-Redistribute in more detail.

Share Rerandomization. This component comprises the *closing period* of the redistribution phase (see Section 2.1). We mask the shares by constructing $\ell(n-3t)B$ polynomials $Q_a^{(k,m)}$ that correspond to 0^ℓ, $[0]_d$, using a slight variant of RanDouSha and then computing $H_a^{(k,m)} \leftarrow \widehat{H}_a^{(k,m)} + Q_a^{(k,m)}$. All parties then erase all their shares for $\widehat{H}_a^{(k,m)}$ and $Q_a^{(k,m)}$. By the additive property of our secret sharing scheme, the new shares correspond to the same secrets but are now distributed uniformly at random from all past shares.

Verifiable Double Sharing. This component together with Share Reconstruction below comprises the *opening period* of the redistribution phase. Let M be a (publicly agreed upon) hyper-invertible matrix with n rows and $n-2t$ columns. As was noted in [DN07], if $\boldsymbol{y} = M\boldsymbol{x}$, then we can also use Berlekamp-Welch to "interpolate" $\boldsymbol{x}$ from $\boldsymbol{y}$ if no more than t coordinates of $\boldsymbol{y}$ are in error (via adversarial corruption in this context).

The first step is to construct random padding in the form of additional polynomials $\left\{H_a^{(k,m)}\right\}_{\substack{m=1,\ldots,B\\ k=n-3t+1,\ldots,n-2t\\ a=1,\ldots,\ell}}$ using RanDouSha. Now all parties hold shares for the set of polynomials $\left\{H_a^{(k,m)}\right\}_{\substack{m=1,\ldots,B\\ k=1,\ldots,n-2t\\ a=1,\ldots,\ell}}$. Each party P_i then shares all of his shares; that is, each P_i constructs polynomials $U^{(i,k,m)}$ for $1 \le k \le n-2t$, $1 \le m \le B$ such that $U^{(i,k,m)}(\beta^a) = H_a^{(k,m)}(\alpha^i)$. Note that for each i

[5] If W is not a multiple of $\ell^2(n-3t)$, we can generate random sharings of blocks to make it so; using RanDouSha, this can be done with poly(n) communication complexity, and since it adds only a poly(n) amount of data to W, this does not asymptotically affect the overall communication complexity of redistributing W secrets.

and m, the last t polynomials of the $U^{(i,k,m)}$ correspond with the t random padding polynomials. Each P_i then secret shares the $U^{(i,k,m)}$ with $\mathcal{P}$. Therefore, every party has their own share of every $U^{(j,k,m)}$.

Next, for each m, each party uses their local shares of the group of polynomials $H_a^{k,m}$ and $U^{(i,k,m)}$ (each of length $n-2t$) to construct shares for new polynomials $\widetilde{H}_a^{(\widetilde{k},m)}$ and $\widetilde{U}^{(i,\widetilde{k},m)}$ for $\widetilde{k}=1,\ldots,n$ by applying M to each group, respectively (see step 2.3 below). Each P_i then sends their share for $\widetilde{H}_a^{(\widetilde{k},m)}$ and $\widetilde{U}^{(i,\widetilde{k},m)}$ to $P_{\widetilde{k}}$, who can interpolate the polynomials via Berlekamp-Welch.

The new polynomials $\widetilde{H}$ and $\widetilde{U}$ perfectly hide the values of H and U used to construct them because of the t random padding inputs used to construct each polynomial via M; in other words, there is a degree of freedom in the random padding for each potential corruption that can occur. Further, since at most t shares of the $\widetilde{H}_a^{(\widetilde{k},m)}$ and $\widetilde{U}^{(i,\widetilde{k},m)}$ can be corrupted, by the error correction properties of hyper-invertible M, $P_{\widetilde{k}}$ is able to reconstruct $\widetilde{H}_a^{(\widetilde{k},m)}$ and $\widetilde{U}^{(i,\widetilde{k},m)}$.

Therefore, $P_{\widetilde{k}}$ can verify that $\widetilde{U}^{(i,\widetilde{k},m)}(\beta^a)=\widetilde{H}_a^{(\widetilde{k},m)}(\alpha^i)$; if not, $P_{\widetilde{k}}$ broadcasts an accusation of P_i and *both* parties are viewed as corrupted. While this strategy lowers the corruption threshold from $n/4$ to $n/8$, it is an efficient way to handle dispute resolution, and increasing the threshold later can be accomplished using Bracha committees (see Appendix E).

Share Reconstruction. Now that each party has verified double shares of all the other parties' shares, share reconstruction for a newly rebooted[6] party P_j can be accomplished by applying the appropriate Lagrange coefficients $\lambda_{j,i}$ to the double sharings $U^{(i,k,m)}$ and then having P_j apply Berlekamp-Welch to interpolate her own share. This is because, for indices $z_1,...,z_{n-2t}$ of parties with correct double shares, $\lambda_{j,1}U^{(z_1,k,m)}(\beta^a)+\cdots+\lambda_{j,n-2t}U^{(z_{n-2t},k,m)}(\beta^a)=\lambda_{j,1}H_a^{(k,m)}(\alpha^{z_1})+\cdots+\lambda_{j,n-2t}H_a^{(k,m)}(\alpha^{z_{n-2t}})=H_a^{(k,m)}(\alpha^j)$.

3.3.2 Specification of Block-Redistribute

Our protocol requires a slightly altered version of RanDouSha for the first step. In [DIK+08], RanDouSha calls on a subprotocol SemiRobustShare, and in that protocol, step 2(a) (see page 6 in [DIK+08]) is altered so that the parties check that the polynomials evaluate to zero at β^j for $j=1,\ldots,\ell$, and an accusation is broadcast if they do not. Security for this slightly modified protocol easily follows.

Block-Redistribute$\left(\left\{\widehat{H}_a^{(k,m)}\right\}_{\substack{m=1,\ldots,B\\ k=1,\ldots,n-3t\\ a=1,\ldots,\ell}}\right)$

We assume that the secrets have been stored in blocks of size ℓ (as described in Section 2) using polynomials $\widehat{H}_a^{(k,m)}$.

1. *Share Rerandomization*
   - *1.1* The parties in $\mathcal{P}$ invoke RanDouSha to generate polynomials $Q_a^{(k,m)}$ for $1\le a\le l$, $1\le k\le n-3t$ and $1\le m\le B$, satisfying $Q_a^{(k,m)}(\beta^j)=0$ for $j=1,\ldots,\ell$.
   - *1.2* The parties locally compute $H_a^{(k,m)}\leftarrow\widehat{H}_a^{(k,m)}+Q_a^{(k,m)}$.
   - *1.3* All parties erase their shares of each $\widehat{H}_a^{(k,m)}$ and $Q_a^{(k,m)}$.
2. *Verifiable Double Sharing*
   - *2.1* The parties use RanDouSha to generate polynomials $H_a^{(k,m)}$ for $1\le a\le l$, $1\le m\le B$ and $k=n-3t+1,\ldots,n-2t$.
   - *2.2* Each P_i selects polynomials $U^{(i,1,m)},\ldots,U^{(i,(n-2t),m)}$ of degree $\le d$ such that $U^{(i,k,m)}(\beta^a)=H_a^{(k,m)}(\alpha^i)$ for $a=1,\ldots,\ell$, $k=1,\ldots,n-2t$, and $m=1,\ldots,B$ and shares them via RobustShare.

[6] Asymptotically, there is no cost to executing share redistribution for all, rather than rebooted, parties.

2.3 Define $\widetilde{H}_a^{(\widetilde{k},m)}$ and $\widetilde{U}^{(i,\widetilde{k},m)}$ for $\widetilde{k} = 1, \ldots, n$ by

$$\left(\widetilde{H}_a^{(1,m)}, \ldots, \widetilde{H}_a^{(n,m)}\right)^T = M\left(H_a^{(1,m)}, \ldots, H_a^{(n-2t,m)}\right)^T$$

and

$$\left(\widetilde{U}^{(i,1,m)}, \ldots, \widetilde{U}^{(i,n,m)}\right)^T = M\left(U^{(i,1,m)}, \ldots, U^{(i,n-2t,m)}\right)^T.$$

Each party in $\mathcal{P}$ locally computes their shares of these polynomials.

2.4 Each party in $\mathcal{P}$ sends *all* their shares of $\widetilde{H}_a^{(\widetilde{k},m)}$ and $\widetilde{U}^{(i,\widetilde{k},m)}$ to party $P_{\widetilde{k}}$ for each a, i, and m.

2.5 Each $P_{\widetilde{k}}$ uses Berlekamp-Welch on the shares of each $\widetilde{U}^{(i,\widetilde{k},m)}$ to interpolate $\widetilde{U}^{(i,\widetilde{k},m)}(\beta^a)$ for each $a = 1, \ldots, \ell$.

2.6 Each $P_{\widetilde{k}}$ uses Berlekamp-Welch on the shares of each $\widetilde{H}_a^{(\widetilde{k},m)}$ to interpolate $\widetilde{H}^{(\widetilde{k},m)}(\alpha^i)$ for each $i = 1, \ldots, n$.

2.7 Each $P_{\widetilde{k}}$ checks if $\widetilde{U}^{(i,\widetilde{k},m)}(\beta^a) = \widetilde{H}_a^{(\widetilde{k},m)}(\alpha^i)$ for each $a = 1, \ldots, \ell$. If not for some $\widetilde{U}^{(i,\widetilde{k},m)}$, then $P_{\widetilde{k}}$ broadcasts $(P_{\widetilde{k}}, \texttt{J'accuse}, P_i)$. All parties add P_i and $P_{\widetilde{k}}$ to $\mathcal{C}orr$. (After a party is added to set of nodes marked as corrupted, $\mathcal{C}orr$, any further accusations from that party are ignored.)

2.8 Each party erases all their shares of each $\widetilde{H}_a^{(\widetilde{k},m)}$ and $\widetilde{U}^{(i,\widetilde{k},m)}$ for $\widetilde{k} = 1, \ldots, n$ and $H_a^{(k,m)}$ and $U^{(i,k,m)}$ for $k = n - 3t + 1, \ldots, n - 2t$.

3. Share Redistribution

3.1 P_j: Define G to be the set of the first $n - 2t$ parties in $\mathcal{P} - \mathcal{C}orr$. Let $\{z_1, \ldots, z_{n-2t}\}$ denote the set of indices of parties in G. Let $\lambda_{j,i}$ denote the Lagrange coefficients for interpolating P_j's share of a secret from the shares of parties in G (i.e. for a polynomial f of degree $\leq d$, $f(\alpha^j) = \lambda_{j,1}f(\alpha^{z_1}) + \cdots + \lambda_{j,n-2t}f(\alpha^{z_{n-2t}})$.)

3.2 For each $k = 1, \ldots, n - 3t$, each $m = 1, \ldots, B$, and each $j = 1, \ldots, n$, each party in G sends his share of $\lambda_{j,1}U^{(z_1,k,m)} + \cdots + \lambda_{j,n-2t}U^{(z_{n-2t},k,m)}$ to P_j.

3.3 Each P_j uses Berlekamp-Welch to interpolate the polynomials received in the previous step to obtain all $H_a^{(k,m)}(\alpha_j)$.

3.4 Each party erases all their shares of each $U^{(i,k,m)}$ (retaining the shares of $H_a^{(k,m)}$).

The protocol Block-Redistribute has communication complexity $O(W + \text{poly}(n))$ for redistributing W secrets.

Theorem 1 *Assuming perfectly secure point-to-point channels and a secure broadcast channel, protocol* Block-Redistribute *perfectly UC-realizes $\mathcal{F}_{BR}$ in the synchronous, adaptive corruption model with corruption threshold $n/8$.*

We note that this theorem is in the standard, rather than proactive, security model. For the proof of Theorem 1, see Appendix B.

3.4 Circuit Transformation and Share Permutation

Since the circuit computes on individual values but secrets are shared in blocks, care must be taken to actually implement the circuit on secret-shared blocks of data because addition and multiplication of blocks occurs element-wise according to position within the blocks, which therefore must be rearranged to perform arbitrary addition and multiplication of the shares per the circuit instructions. To remedy this, we first transform the circuit itself in the same manner as Appendices A.5 and A.6 of the full version of [DIK10]. Again, this increases the circuit size by a constant factor, and increases circuit depth by a $\log \mathcal{C}$ multiplicative factor. Then, at each layer of computation of the circuit we will perform a permutation

on *all* the secrets to make sure that the secrets are arranged in the correct order for whatever arithmetic operations that layer requires.

In order to perform the permutations, we use Beneš networks [Ben64], also known as Waksman networks [Wak68]. These networks were used in the context of multiparty computation in [DIK10], and also in the context of fully homomorphic encryption in [GHS12]. Let the layer of the circuit have width w, where each gate has fan-in 2. The main idea is that an arbitrary permutation on the $O(w/\ell)$ blocks of secrets needed to compute the layer can be performed using $O(\log(w))$ constant-round subprotocols that execute the decomposition of the permutation into $O(\log(w))$ sub-permutations. We denote this protocol PermuteLayer, which has as inputs the permutation π as well as all the secret shares of each of the parties. We defer the construction of PermuteLayer to the Appendix due to lack of space; see Appendix C.

3.5 The Full PMPC Protocol

Figure 1 outlines the protocol that uses all the sub-protocols described above to securely compute the circuit. We note that step 4.1 below is abbreviated; in the full implementation, we invoke RanDouSha to pad to a multiple of $\ell^2(n-3t)$ sharings.

PMPC:

1. The circuit is transformed so that each layer of computation contains only one type of gate (addition or multiplication) and each gate has fan-in 2 and fan-out 1 or 2.
2. Each party shares their inputs using RobustShare.
3. Execute Block-Redistribute to redistribute all the currently stored secrets.
4. For each layer of the circuit, the following steps are performed.
   - *4.1* Suppose we have W input secrets for this layer where W is a multiple of $\ell^2(n-3t)$. Let σ denote the permutation to be performed on the first W secrets before the computation for this layer. Execute PermuteLayer using σ on the W secrets.
   - *4.2* If this is an addition layer, then compute the additions locally. If this is a multiplication layer, then the parties invoke RanDouSha to generate random sharings, and then use these random sharings to invoke Multiply for each block to multiply.
   - *4.3* All secrets no longer needed for the rest of the computation are erased.
   - *4.4* Invoke Block-Redistribute to redistribute all W secrets.
5. For the output layer, invoke Reco to reconstruct all the outputs toward the intended recipients.

Figure 1: PMPC Protocol Outline

To analyze the communication complexity of PMPC, note that the communication complexity of performing multiplications on a layer, performing additions on a layer, permuting a layer, and redistributing a layer are all the same, namely $O(W + \text{poly}(n))$ if the circuit width is W. In the worst case, $W = C$, so it is $O(C + \text{poly}(n))$. The number of layers is initially D, the depth of the circuit. After the circuit is transformed so that no gate has fan-in or fan-out more than 2, the depth is $D\log(C)$. Furthermore, if there are C secrets to permute at each layer, than this would take $\log(C)$ permutations. So if we think of the operation of permutation as a layer in the circuit, the final depth of the circuit would be $D\log^2(C)$. Thus the communication complexity is $O((C + \text{poly}(n)) \cdot D\log^2(C))$, or $O(DC\log^2(C) + D\text{poly}(n)\log^2(C))$. The same reasoning shows that the computational complexity is $O(DC\log^2(C)\text{polylog}(n) + D\text{poly}(n)\log^2(C))$.

For circuits that are "layered" in such a way that each output from a gate is used in the next layer and no other layers, we no longer assume that $W = C$; instead, we denote the widths of layers 1 through $D' := D\log^2(C)$ by $W_1, \ldots, W_{D'}$, and we have $C = \sum_{i=1}^{D'} W_i$. The communication complexity

is therefore $\sum_{i=1}^{D'} W_i + \text{poly}(n) = O(C \log^2 C + D\text{poly}(n) \log^2 C)$ with a computational complexity of $O(C \log^2 C \text{polylog}(n) + D\text{poly}(n) \log^2 C)$.

In terms of broadcast complexity, our protocol uses $O(1)$ broadcasts per dispute. Since the number of disputes that can arise between secret redistributions is at most t, the total number of broadcasts is $O(Dn)$.

Theorem 2 *For n parties and an arithmetic circuit C that is at least $\Omega(n)$ gates wide, the protocol* PMPC *realizes $\mathcal{F}_C$ with perfect security in the proactive UC model against an active and adaptive adversary corrupting up to $t < n/8$ parties per stage.*

Using the party virtualization techniques of [Bra87] (see Appendix E), we can increase the corruption threshold as to $t < (1/3 - \epsilon)n$. We denote PMPC-pv as the protocol that executes PMPC using the techniques in Appendix E. The communication complexity of PMPC-pv is $O(C \log^2 C \text{polylog} n + D\text{poly}(n) \log^2 C)$, which is sum of the communication complexity and the computation complexity of PMPC. However, the communication complexity of Block-Redistribute is *not* affected by player virtualization.

Theorem 3 *For any $0 < \delta < 1/3$, for n parties and an arithmetic circuit C that is at least $\Omega(n)$ gates wide, the protocol* PMPC-pv *realizes $\mathcal{F}_C$ with perfect security in the proactive UC model against an active and adaptive adversary corrupting up to $t < \delta n$ parties per stage.*

Theorem 3 can also be extended so that, with statistical security, $0 < \delta < 1/2$; see Appendix E for further details.

Acknowledgements

We would like to thank Ivan Damgård, Yuval Ishai and Jonathan Katz for helpful discussions about this work.

References

[ADN06] Jesús F. Almansa, Ivan Damgård, and Jesper Buus Nielsen. Simplified threshold rsa with adaptive and proactive security. In *Proceedings of the 24th annual international conference on The Theory and Applications of Cryptographic Techniques*, EUROCRYPT'06, pages 593–611, Berlin, Heidelberg, 2006. Springer-Verlag.

[AHU74] Alfred V. Aho, John E. Hopcroft, and J. D. Ullman. *The Design and Analysis of Computer Algorithms*. Addison-Wesley, 1974.

[Ben64] Václav E. Beneš. Optimal rearrangeable multistage connecting networks. *The Bell System Technical Journal*, 43(4):1641–1656, July, 1964.

[Ber84] Elwyn R. Berlekamp. *Algebraic Coding Theory*. Aegean Park Press, 1984.

[BFO12] Eli Ben-Sasson, Serge Fehr, and Rafail Ostrovsky. Near-linear unconditionally-secure multiparty computation with a dishonest minority. In *CRYPTO*, pages 663–680, 2012.

[BOGW88] Michael Ben-Or, Shafi Goldwasser, and Avi Wigderson. Completeness theorems for non-cryptographic fault-tolerant distributed computation (extended abstract). In *STOC*, pages 1–10, 1988.

[Bol03] Alexandra Boldyreva. Threshold signatures, multisignatures and blind signatures based on the gap-diffie-hellman-group signature scheme. In *Proceedings of the 6th International Workshop on Theory and Practice in Public Key Cryptography: Public Key Cryptography*, PKC '03, pages 31–46, London, UK, UK, 2003. Springer-Verlag.

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

[BTH08] Zuzana Beerliová-Trubíniová and Martin Hirt. Perfectly-secure mpc with linear communication complexity. In *TCC*, pages 213–230, 2008.

[Can01] Ran Canetti. Universally composable security: A new paradigm for cryptographic protocols. In *FOCS*, pages 136–145, 2001.

[Can05] Ran Canetti. Universally composable security: A new paradigm for cryptographic protocols. Cryptology ePrint Archive, Report 2000/067, 2005.

[CCD88] David Chaum, Claude Crépeau, and Ivan Damgard. Multiparty unconditionally secure protocols. In *Proceedings of the twentieth annual ACM symposium on Theory of computing*, STOC '88, pages 11–19, New York, NY, USA, 1988. ACM.

[CGJ+99] Ran Canetti, Rosario Gennaro, Stanislaw Jarecki, Hugo Krawczyk, and Tal Rabin. Adaptive security for threshold cryptosystems. In *Proceedings of the 19th Annual International Cryptology Conference on Advances in Cryptology*, CRYPTO '99, pages 98–115, London, UK, UK, 1999. Springer-Verlag.

[CKLS02] Christian Cachin, Klaus Kursawe, Anna Lysyanskaya, and Reto Strobl. Asynchronous verifiable secret sharing and proactive cryptosystems. In *ACM Conference on Computer and Communications Security*, pages 88–97, 2002.

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

[DIK10] Ivan Damgård, Yuval Ishai, and Mikkel Krøigaard. Perfectly secure multiparty computation and the computational overhead of cryptography. In *EUROCRYPT*, pages 445–465, 2010.

[DN07] Ivan Damgård and Jesper Buus Nielsen. Scalable and unconditionally secure multiparty computation. In *CRYPTO*, pages 572–590, 2007.

[FGMY97a] Y. Frankel, P. Gemmell, P. D. MacKenzie, and Moti Yung. Optimal-resilience proactive public-key cryptosystems. In *Proceedings of the 38th Annual Symposium on Foundations of Computer Science*, FOCS '97, pages 384–, Washington, DC, USA, 1997. IEEE Computer Society.

[FGMY97b] Yair Frankel, Peter Gemmell, Philip D. MacKenzie, and Moti Yung. Proactive rsa. In *Proceedings of the 17th Annual International Cryptology Conference on Advances in Cryptology*, CRYPTO '97, pages 440–454, London, UK, UK, 1997. Springer-Verlag.

[FL82] Michael J. Fischer and Nancy A. Lynch. A lower bound for the time to assure interactive consistency. *Inf. Process. Lett.*, 14(4):183–186, 1982.

[FMY01] Yair Frankel, Philip D. MacKenzie, and Moti Yung. Adaptive security for the additive-sharing based proactive rsa. In *Proceedings of the 4th International Workshop on Practice and Theory in Public Key Cryptography: Public Key Cryptography*, PKC '01, pages 240–263, London, UK, UK, 2001. Springer-Verlag.

[FY92] Matthew K. Franklin and Moti Yung. Communication complexity of secure computation (extended abstract). In *STOC*, pages 699–710, 1992.

[Gao02] Shuhong Gao. A new algorithm for decoding reed-solomon codes. In *Communications, Information and Network Security, V.Bhargava, H.V.Poor, V.Tarokh, and S.Yoon*, pages 55–68. Kluwer, 2002.

[GHS12] Craig Gentry, Shai Halevi, and Nigel P. Smart. Fully homomorphic encryption with polylog overhead. In *EUROCRYPT*, pages 465–482, 2012.

[GMW87] O. Goldreich, S. Micali, and A. Wigderson. How to play any mental game. In *Proceedings of the Nineteenth Annual ACM Symposium on Theory of Computing*, STOC '87, pages 218–229, New York, NY, USA, 1987. ACM.

[HJKY95] Amir Herzberg, Stanislaw Jarecki, Hugo Krawczyk, and Moti Yung. Proactive secret sharing or: How to cope with perpetual leakage. In *CRYPTO*, pages 339–352, 1995.

[HMQ05] Dennis Hofheinz and Jorn Muller-Quade. A synchronous model for multi-party computation and the incompleteness of oblivious transfer. Cryptology ePrint Archive, Report 2004/016, 2005.

[IKOS08] Yuval Ishai, Eyal Kushilevitz, Rafail Ostrovsky, and Amit Sahai. Cryptography with constant computational overhead. In *STOC*, pages 433–442, 2008.

[JO08] Stanislaw Jarecki and Josh Olsen. Financial cryptography and data security. chapter Proactive RSA with Non-interactive Signing, pages 215–230. Springer-Verlag, Berlin, Heidelberg, 2008.

[JS05] Stanislaw Jarecki and Nitesh Saxena. Further simplifications in proactive rsa signatures. In *Proceedings of the Second international conference on Theory of Cryptography*, TCC'05, pages 510–528, Berlin, Heidelberg, 2005. Springer-Verlag.

[KMTZ13] Jonathan Katz, Ueli Maurer, Bjoern Tackmann, and Vassilis Zikas. Universally composable synchronous computation. In *TCC*, 2013.

[Lei92] Frank Thomson Leighton. *Introduction to parallel algorithms and architectures: arrays, trees, hypercubes*. Morgan Kaufmann, 1992.

[Nie03] Jesper Buus Nielsen. On protocol security in the cryptographic model. PhD Thesis. University of Aarhus., 2003.

[OY91] Rafail Ostrovsky and Moti Yung. How to withstand mobile virus attacks (extended abstract). In *PODC*, pages 51–59, 1991.

[Rab98] Tal Rabin. A simplified approach to threshold and proactive rsa. In *Proceedings of the 18th Annual International Cryptology Conference on Advances in Cryptology*, CRYPTO '98, pages 89–104, London, UK, UK, 1998. Springer-Verlag.

[RB89] T. Rabin and M. Ben-Or. Verifiable secret sharing and multiparty protocols with honest majority. In *Proceedings of the twenty-first annual ACM symposium on Theory of computing*, STOC '89, pages 73–85, New York, NY, USA, 1989. ACM.

[Sch07] David Schultz. *Mobile Proactive Secret Sharing*. PhD thesis, Massachusetts Institute of Technology, 2007.

[Sha79] Adi Shamir. How to share a secret. *Commun. ACM*, 22(11):612–613, 1979.

[Wak68] Abraham Waksman. A permutation network. *J. ACM*, 15(1):159–163, 1968.

[WWW02] Theodore M. Wong, Chenxi Wang, and Jeannette M. Wing. Verifiable secret redistribution for archive system. In *IEEE Security in Storage Workshop*, pages 94–106, 2002.

[Yao82] Andrew C. Yao. Protocols for secure computations. In *Proceedings of the 23rd Annual Symposium on Foundations of Computer Science*, FOCS '82, pages 160–164, Washington, DC, USA, 1982. IEEE Computer Society.

[ZSvR05] Lidong Zhou, Fred B. Schneider, and Robbert van Renesse. Apss: proactive secret sharing in asynchronous systems. *ACM Trans. Inf. Syst. Secur.*, 8(3):259–286, 2005.

A The Proactive Security Model

Security of the developed protocols is proven using the Universal Composability (UC) framework introduced in [Can01] and revised in [Can05]. There are two major challenges that face casting the proactive security model considered in this paper into the UC framework. First, this paper considers a synchronous communication model, while communication in the original UC framework is inherently asynchronous. Second, the proactive security model requires parties to be able to securely erase their memory; such erasure capability guarantees that once parties are compromised only their current state is revealed to the adversary.

Synchronous UC: The latest revision of the UC framework in [Can05] describes an ideal functionality ($\mathcal{F}_{SYN}$) that models synchronous communication. The authors in [Nie03, HMQ05] also propose modifications to the UC framework that allow it to model synchronous communication. The most recent proposal to model synchronous communication in the UC framework is that of [KMTZ13], where two new ideal functionalities are proposed (one capturing a bounded delay channel, $\mathcal{F}_{BD}$, and one capturing loose synchronization, $\mathcal{F}_{CLOCK}$). [KMTZ13] demonstrates that these two ideal functionalities are enough to capture synchronous communication and also model previous attempts in [Can05] and [Nie03]. For concreteness, this paper uses the synchronous UC model using $\mathcal{F}_{BD}$ and $\mathcal{F}_{CLOCK}$ from [KMTZ13].

UC with Erasures: The latest revision of the UC framework allows modeling of *erasures*, i.e., allowing parties to only leak current internal states when corrupted by an adversary. Capturing erasures is essential to model proactive security as parties are periodically rebooted. As shown in [Can05] composability still holds when allowing erasures in the UC framework.

A.1 UC Computation Model

All entities (parties, ideal functionalities, adversary, and environment) in the considered UC framework are interactive Turing machines (ITM). A protocol π executed between n parties in the $\mathcal{F}'$-hybrid model consists of a set of n ITMs. The identities of the n ITMs, i.e., $P_1, ..., P_n$, are unique. In addition, the parties have access to an ideal functionality $\mathcal{F}'$. Note that, in general, the ideal functionality $\mathcal{F}' = (\mathcal{F}'_1, \mathcal{F}'_2, ..., \mathcal{F}'_m), 1 < m$, may consist of several ideal functionalities. The protocol in this paper uses the following ideal functionalities: $\mathcal{F}'_{\texttt{SMT}}$ for secure message transmission (from [Can05]), $\mathcal{F}'_{\texttt{auth}}$ for authenticated broadcast (from [Can05]), $\mathcal{F}'_{\texttt{BD}}$ and $\mathcal{F}'_{\texttt{CLOCK}}$ to capture bounded delay channels and loosely synchronized parties which are required to model synchronous communication (both from [KMTZ13]).

A.2 Real-World Execution of a Proactive Protocol

A protocol in the UC framework runs while interacting with an adversary, an ITM denoted by $\mathcal{A}$, and an environment, another ITM denoted by $\mathcal{Z}$. The execution is initiated by $\mathcal{Z}$, which provides inputs to and obtains outputs from parties involved in the protocol and also communicates with $\mathcal{A}$. $\mathcal{A}$ has access to the ideal functionalities in the hybrid model and also functions as a network between machines

of different parties. During execution, the ITMs are activated sequentially where the exact order of activation depends on the considered model.

Similar to previous definitions of proactive protocols [ADN06], the execution of a proactive protocol, π, proceeds in communication rounds, denoted by $r_{i,l}$, and the initial round is round $r_{0,0}$. A *proactive protocol* proceeds in *phases*. A phase, denoted ph consists of a number of consecutive rounds $\mathsf{r}_{i,l}, ..., \mathsf{r}_{i+j,l}$, and every round $\mathsf{r}_{j,l}$ belongs to exactly one phase ph_l. Each phase of π is either a *refreshment* or an *operation* phase. The phases of π alternate between *refreshment* and *operation* phases. Each *refreshment* phase ph_l consists of rounds $\mathsf{r}_{i,l}, ..., \mathsf{r}_{i+j,l}$, such that there exists a k, $0 \leq k < j$ where rounds $\mathsf{r}_{i,l}, ..., \mathsf{r}_{i+k,l}$ are denoted the *closing* period of refreshment phase ph_l while $\mathsf{r}_{i+k+1,l}, ..., \mathsf{r}_{i+j,l}$ denote the *opening* period of refreshment phase ph_l. Finally, a *stage* st (starting at stage 0) consists of an *opening* refreshment period, an operation phase and then a *closing* refreshment period, therefore including a full (operation) phase and two sequences of two refreshment stages; each refreshment is the closing of one stage and the opening of the other.

Stages are executed consecutively, and the number of stages is equal to the number of operation phases, which corresponds to the depth of a circuit. One exception to the alternating configuration of refreshment and operational stages is that the first stage starts with an operation phase, and the last stage ends with an operation phase. The intuition is that during the operation phases, the protocol computes the functionality (layers of the arithmetic circuit) that it was designed for, whereas refreshment phases are used to rerandomize the data.

The environment $\mathcal{Z}$ decides when a new stage st_i begins by sending a special command $\mathsf{refresh}_i$ to each party. Refreshment ends when all honest parties have output a special symbol ζ_i indicating end of stage st_i. Further, $\mathcal{A}$ may corrupt parties adaptively throughout the protocol, subject to the limitation that no more than t parties can be simultaneously corrupted during any stage. In particular, this means that if a party is corrupt during a refreshment phase (either opening or closing period), she is considered to be corrupt in both of the two stages to which the phase belongs. After corruption, $\mathcal{A}$ acts on behalf of the corrupted party. Corruption may be either passive (where the adversary only learns a party's current state) or active (where adversary may arbitrarily control behavior of a party).

If party P_i is corrupted during an operation phase of stage st_j, $\mathcal{A}$ is given the view of P_i starting from her state at the beginning of the current stage. This models the assumption that all randomness and data used in the previous refreshment phase is erased except for the information that the protocol specifies should be used afterwards.

If the corruption of P_i is made during a refreshment phase which belongs to stages st_j and st_{j+1}, $\mathcal{A}$ receives the view of P_i starting from her state at the beginning of stage st_j, and P_i is assumed to be corrupt for stage st_{j+1}.

If P_i is corrupt during the closing period of a refreshment phase in stage st_j, $\mathcal{A}$ may decide to *leave* her, which may allow $\mathcal{A}$ to corrupt new parties, subject to the bound on t corruptions per stage. In this case, we say P_i is *decorrupted*. The intuition for this is that so long is a party is not corrupted by the opening period of a refreshment stage, she can then receive a new version of her shares and participate in subsequent stages.

A decorrupted party immediately starts taking part in the protocol as an honest party. In the passive corruption case, she starts from the correct state specified by the protocol at this point. In the active corruption case, she starts from a default state after round r. This state is application-dependent in general.

A.3 Ideal World Model for PMPC

In Figure A.3, we specify the ideal functionality $\mathcal{F}_{PMPC}$, for PMPC. In the specification, we use the following notation:

- $\mathcal{Z}$: Environment

- $\mathcal{A}$: Adversary
- $\mathcal{F}_{PMPC}$: Ideal functionality
- n: The number of parties in the protocol
- t: Threshold for corruption. Note that the adversary may corrupt no more than t parties at any stage.
- $\mathcal{H} \subseteq [n]$: Set of Honest Parties
- $\mathcal{R} \subset [n]$: Set of Corrupted Parties
- P_i: Party with index i, where for honest parties $i \in \mathcal{H}$ and for corrupted parties $i \in \mathcal{R}$
- z: Auxiliary input to the adversary
- f: The function to be computed by $\mathcal{F}_{PMPC}$ on n inputs

We note that the ideal world specification for $\mathcal{F}_{PMPC}$ is very similar to the ideal world specification for honest-majority MPC except that it allows for decorruption and recorruption up to the specified threshold.

A.4 UC Security

Security is defined by comparing protocol π's real world execution with that of the *ideal world execution*. In the ideal world an ideal functionality $\mathcal{F}$ is used to specify the desired input and output behavior of π. $\mathcal{F}$ also specifies the information allowed to be leaked from π to $\mathcal{A}$ and $\mathcal{Z}$. Informally, security is demonstrated by showing that whatever $\mathcal{A}$ can achieve by attacking π in the real world, can also be achieved by an ITM acting as a simulator, $\mathcal{S}$, interacting with $\mathcal{F}$ in the ideal world. The goal of the $\mathcal{S}$ is to *simulate* $\mathcal{A}$'s view of π, based only on the information $\mathcal{F}$ exchanges with $\mathcal{A}$. π is considered secure in the $\mathcal{F}'$-hybrid model if no $\mathcal{Z}$ can distinguish interactions with π from those with $\mathcal{F}$ and $\mathcal{S}$. More formally: $\mathcal{Z}$ at the end of an execution of π outputs a bit guessing whether it had interacted with $\mathcal{A}$ and π (real world) or $\mathcal{F}$ and $\mathcal{S}$ (ideal world).

View in the Real World: When $\mathcal{Z}$ interacts with $\mathcal{A}$ and π in the $\mathcal{F}'$-hybrid model, on security parameter k, auxiliary input z to $\mathcal{Z}$, and when the random coins of all machines are uniformly chosen, the content of the output tape of $\mathcal{Z}$ is a random variable denoted $\mathsf{HYB}^{\mathcal{F}'}_{\pi,\mathcal{Z},\mathcal{A}}(k,z)$. $\mathsf{HYB}^{\mathcal{F}'}_{\pi,\mathcal{Z},\mathcal{A}}$ denotes the ensemble $\{\mathsf{HYB}^{\mathcal{F}'}_{\pi,\mathcal{Z},\mathcal{A}}(k,z)\}_{k\in\mathbb{N},z\in\{0,1\}^*}$.

View in the Ideal World: When $\mathcal{Z}$ interacts with $\mathcal{S}$ and $\mathcal{F}$ in the ideal world model, on security parameter k, auxiliary input z to $\mathcal{Z}$, and when the random coins of all machines are uniformly chosen, the content of the output tape of $\mathcal{Z}$ is a random variable denoted $\mathsf{IDEAL}_{\mathcal{F},\mathcal{S},\mathcal{Z}}(k,z)$. $\mathsf{IDEAL}_{\mathcal{F},\mathcal{S},\mathcal{Z}}$ denotes the ensemble $\{\mathsf{IDEAL}_{\mathcal{F},\mathcal{S},\mathcal{Z}}(k,z)\}_{k\in\mathbb{N},z\in\{0,1\}^*}$.

Definition 4 (UC Security) *A protocol π proactively t-realizes a functionality $\mathcal{F}$ in the $\mathcal{F}'$-hybrid model if for all adversaries $\mathcal{A}$ corrupting at most t parties per stage there exists a simulator $\mathcal{S}$ such that for every environment $\mathcal{Z}$ the $\mathsf{IDEAL}_{\mathcal{F},\mathcal{S},\mathcal{Z}}$ and $\mathsf{HYB}^{\mathcal{F}'}_{\pi,\mathcal{Z},\mathcal{A}}$ ensembles are indistinguishable, i.e., $\mathsf{IDEAL}_{\mathcal{F},\mathcal{S},\mathcal{Z}} \overset{c}{\approx} \mathsf{HYB}^{\mathcal{F}'}_{\pi,\mathcal{Z},\mathcal{A}}$.*

B Security of Block-Redistribute (Proof of Theorem 1)

We present a proof of Theorem 1 by carefully specifying a simulator $\mathcal{S}$ for Block-Redistribute and demonstrating how the views of the adversary in the real and ideal worlds are independently distributed. First, recall the theorem; then we specify the ideal functionality before giving the full proof.

1. **Input Phase:**
   (a) The environment $\mathcal{Z}$ invokes the adversary, $\mathcal{A}$, with auxiliary input z.
   (b) $\mathcal{Z}$ invokes each of the n parties P_i with input x_i, such that (without loss of generality) $|x_i| = |x_j|$ for $i, j \in \{1, \ldots, n\}$.
   (c) Each P_i sends x_i to $\mathcal{F}_{PMPC}$.
2. **Initial Corruption Phase:**
   (a) $\mathcal{A}$ sends $(\texttt{corr?}, P_i)$ to $\mathcal{Z}$
   (b) $\mathcal{Z}$ sends either `yes` or `no`; $\mathcal{Z}$ never sends `yes` when $|\mathcal{R}| \geq t$. If $\mathcal{Z}$ sends `no` to $\mathcal{A}$, proceed to step 2f. If $\mathcal{Z}$ sends `yes`, $\mathcal{Z}$ updates $\mathcal{R}$ to include P_i.
   (c) $\mathcal{A}$ sends $(\texttt{corr}, P_i)$ to $\mathcal{F}_{PMPC}$.
   (d) $\mathcal{F}_{PMPC}$ sends x_i to $\mathcal{A}$ and adds P_i to $\mathcal{R}$.
   (e) $\mathcal{A}$ sends $(\texttt{input}, P_i, x_i')$ to $\mathcal{F}_{PMPC}$.
   (f) $\mathcal{A}$ either returns to step 2a or sends `proceed` to $\mathcal{F}_{PMPC}$ and proceeds to step 3a.
3. **Computation Phase:**
   (a) $\mathcal{F}_{PMPC}$ computes f on inputs either x_i if $P_i \in \mathcal{H}$ or x_i' if $P_i \in \mathcal{R}$. Denote this input vector $\vec{x}$.
4. **Decorruption Phase:**
   (a) $\mathcal{A}$ sends $(\texttt{decorr}, P_i)$ to $\mathcal{Z}$ for any $P_i \in \mathcal{R}$, in parallel; $\mathcal{Z}$ removes the P_i from $\mathcal{R}$.
   (b) $\mathcal{A}$ sends $(\texttt{decorr}, P_i)$ to $\mathcal{F}_{PMPC}$ for the same P_i as in the above step; $\mathcal{F}_{PMPC}$ removes the P_i from $\mathcal{R}$.
5. **Output Corruption Phase:**
   (a) $\mathcal{A}$ sends $(\texttt{corr}?, P_i)$ to $\mathcal{Z}$.
   (b) $\mathcal{Z}$ sends either `yes` or `no`; $\mathcal{Z}$ never sends `yes` when $|\mathcal{R}| \geq t$. If $\mathcal{Z}$ sends `no` to $\mathcal{A}$ proceed to step 5e.
   (c) If $\mathcal{Z}$ sends *yes*, $\mathcal{A}$ sends $(\texttt{corr}, P_i)$ to $\mathcal{F}_{PMPC}$ and updates $\mathcal{R}$ to include P_i.
   (d) $\mathcal{F}_{PMPC}$ sends $f_i(\vec{x})$ to $\mathcal{A}$ and add P_i to $\mathcal{R}$.
   (e) $\mathcal{A}$ either returns to step 5a or sends `proceed` to $\mathcal{F}_{PMPC}$ and proceeds to step 6a.
6. **Output Phase:**
   (a) For every $P_j \in \mathcal{H}$, $\mathcal{F}_{PMPC}$ sends $f_j(\vec{x})$ to P_j.

Figure 2: **The $\mathcal{F}_{PMPC}$ functionality**

Theorem 5 *Assuming perfectly secure point to point channels and a secure broadcast channel, protocol* Block-Redistribute *perfectly UC-realizes $\mathcal{F}_{BR}$ in the synchronous, adaptive corruption model with corruption threshold $n/8$.*

We now describe the simulator $\mathcal{S}$ for Block-Redistribute. We use lowercase lettered polynomials to denote the simulated version of real-world execution polynomials, which are denoted by the corresponding uppercase letter.

Correctness. Correctness of share rerandomization follows from the correctness of RanDouSha and by construction. Correctness of verifiable double sharing follows from the construction of the $U^{(i,k,m)}$ polynomials as well as the error correction properties of the hyper-invertible matrix M. Correctness of

Let $\mathcal{R}$ be the set of corrupted parties, initially $\emptyset$. $\mathcal{F}_{BR}$ receives t, d, and n as input (via the parties, who received from $\mathcal{Z}$).

1. $\mathcal{Z}$ instantiates each party with their shares of each of m polynomials $\widehat{H}_1, \ldots, \widehat{H}_m$. $\mathcal{Z}$ instantiates $\mathcal{A}$ with auxiliary input z.
2. All parties send their shares to $\mathcal{F}_{BR}$. If parties sent different numbers of shares, then $\mathcal{F}_{BR}$ outputs `abort` and aborts.
3. Input Corruption Phase
   - 3.1 $\mathcal{A}$ sends $(\texttt{corr?}, P_i)$ to $\mathcal{Z}$
   - 3.2 $\mathcal{Z}$ sends either `yes` or `no`; $\mathcal{Z}$ never sends `yes` when $|\mathcal{R}| \geq t$. If $\mathcal{Z}$ sends `no` to $\mathcal{A}$, proceed to step 3.6. If $\mathcal{Z}$ sends `yes`, $\mathcal{Z}$ updates $\mathcal{R}$ to include P_i.
   - 3.3 $\mathcal{A}$ sends $(\texttt{corr}, P_i)$ to $\mathcal{F}_{PMPC}$.
   - 3.4 $\mathcal{F}_{BR}$ sends P_i's shares of $\widehat{H}_1, \ldots, \widehat{H}_m$ to $\mathcal{A}$ and adds P_i to $\mathcal{R}$.
   - 3.5 $\mathcal{A}$ provides new inputs for P_i to F_{BR}.
   - 3.6 $\mathcal{A}$ either returns to step 3.1 or sends `proceed` to $\mathcal{F}_{PMPC}$ and proceeds to step 4.
4. $\mathcal{F}_{BR}$ interpolates the secrets $\widehat{H}_k(\beta^j)$ for $j = 1, \ldots, \ell$ and $k = 1, \ldots, m$ from the received shares.
5. $\mathcal{F}_{BR}$ sends m and (`Request Shares`) to the adversary.
6. The adversary sends shares $H_k(\alpha^i)$ for $k = 1, \ldots, m$ and $P_i \in \mathcal{R}$ to $\mathcal{F}_{BR}$.
7. For each $k = 1, \ldots, m$, $\mathcal{F}_{BR}$ chooses a polynomial H_k of degree $\leq d$ that corresponds to the shares broadcast in the previous step and satisfies $H_k(\beta^j) = \widehat{H}_k(\beta^j)$ for $j = 1, \ldots, \ell$. $\mathcal{F}_{BR}$ selects the points $H_k(\beta^{\ell+j})$ uniformly and independently at randomly for $j = 1, \ldots, t - |\mathcal{R}|$.
8. Output Corruption Phase
   - 8.1 $\mathcal{A}$ sends $(\texttt{corr?}, P_i)$ to $\mathcal{Z}$
   - 8.2 $\mathcal{Z}$ sends either `yes` or `no`; $\mathcal{Z}$ never sends `yes` when $|\mathcal{R}| \geq t$. If $\mathcal{Z}$ sends `no` to $\mathcal{A}$, proceed to step 8.6. If $\mathcal{Z}$ sends `yes`, $\mathcal{Z}$ updates $\mathcal{R}$ to include P_i.
   - 8.3 $\mathcal{A}$ sends $(\texttt{corr}, P_i)$ to $\mathcal{F}_{BR}$.
   - 8.4 $\mathcal{F}_{BR}$ sends $H_k(\alpha^i)$ to $\mathcal{A}$ for each $k = 1, \ldots, m$ and adds P_i to $\mathcal{R}$.
   - 8.5 $\mathcal{A}$ provides new inputs for P_i to F_{BR}.
   - 8.6 $\mathcal{A}$ either returns to step 8.1 or sends `proceed` to $\mathcal{F}_{BR}$ and proceeds to step 9.
9. $\mathcal{F}_{BR}$ sends $H_k(\alpha^i)$ to each P_i for each $k = 1, \ldots, m$.

Figure 3: The $\mathcal{F}_{BR}$ functionality.

share redistribution follows from Lagrange interpolation of polynomials and the correct construction of the $U^{(i,k,mz)}$ polynomials.

Security. We prove security for the corresponding protocol Block-Redistribute constructed in the $(\mathcal{F}_{double}, \mathcal{F}_{RobustShare})$-hybrid model, with each call to RanDouSha replaced by a call to $\mathcal{F}_{double}$[7] and each call to RobustShare replaced by a call to $\mathcal{F}_{RobustShare}$. The simulator $\mathcal{S}$ is therefore designed to emulate the adversary $\mathcal{A}$ interacting with these functionalities.

It is worth noting that RanDouSha as described in [DIK+08] generates *pairs* of sharings of random blocks of data, one with degree d and one with degree $2d$. In Block-Redistribute, we only use the degree d sharings, ignoring the $2d$ sharings.

While we use $\mathcal{C}orr$ to denote the set of parties that are known (by all parties in the real-world execution) to be corrupt, we use $\mathcal{E}vil$ to denote the set of parties that are *actually* corrupt.

Finally, without loss of generality, whenever the (internally executed) adversary asks $\mathcal{S}$ to corrupt a party, $\mathcal{S}$ always forwards the message to $\mathcal{Z}$ and forwards $\mathcal{Z}$'s response back to $\mathcal{A}$, who then acts according

[7] $\mathcal{F}_{double}$ is the ideal functionality defined in [DIK10] for the protocol RanDouSha defined in [DIK+08].

to $\mathcal{Z}$'s message (through $\mathcal{S}$).

The Simulator $\mathcal{S}$

1. *Share Rerandomization*
   - *1.1* The simulator internally executes the adversary $\mathcal{A}$ with a uniformly randomly chosen auxiliary input z.
   - *1.2* The simulator generates uniformly random polynomials $\widehat{h}_a^{(k,m)}$ of degree $\leq d$ subject to the constraint that $\widehat{h}_a^{(k,m)}(\alpha^i) = \widehat{H}_a^{(k,m)}(\alpha^i)$ for each $i \in \mathcal{E}vil$.
   - *1.3* The simulator (emulating $\mathcal{F}_{double}$) sends the message (`Shares?`) to $\mathcal{A}$.
   - *1.4* The adversary sends shares $Q_a^{(k,m)}(\alpha^i)$ for $a = 1, \ldots, \ell$, $k = 1, \ldots, n-3t$, and $m = 1, \ldots, B$ for each $i \in \mathcal{E}vil$ to $\mathcal{S}$.
   - *1.5* The simulator generates uniformly random polynomials $q_a^{(k,m)}$ of degree $\leq d$ subject to the constraints that $q_a^{(k,m)}(\alpha^i) = Q_a^{(k,m)}(\alpha^i)$ for each $i \in \mathcal{E}vil$ and $q_a^{(k,m)}(\beta^j) = 0$ for each $j = 1, \ldots, \ell$.
   - *1.6* The simulator sets $h_a^{(k,m)} \leftarrow \widehat{h}_a^{(k,m)} + q_a^{(k,m)}$ for $a = 1, \ldots, \ell$, $k = 1, \ldots, n-3t$, and $m = 1, \ldots, B$.
2. *Verifiable Double Sharing*
   - *2.1* The simulator (emulating $\mathcal{F}_{double}$) sends the message (`Shares?`) to $\mathcal{A}$.
   - *2.2* The adversary sends shares $H_a^{(k,m)}(\alpha^i)$ to $\mathcal{S}$ for each $i \in \mathcal{E}vil$, where a and m range over the same values as before, but $k = n-3t+1, \ldots, n-2t$.
   - *2.3* For $k = n-3t+1, \ldots, n-2t$, the simulator generates uniformly random polynomials $h_a^{(k,m)}$ of degree $\leq d$ subject to the constraint that $h_a^{(k,m)}(\alpha^i) = H_a^{(k,m)}(\alpha^i)$ for each $i \in \mathcal{E}vil$.
   - *2.4* The simulator (emulating $\mathcal{F}_{RobustShare}$) sends the message (`Shares?`) to $\mathcal{A}$, along with the identities of the dealers (all parties act as dealer) and the number of sharings to generate (which is $n(n-2t)B$).
   - *2.5* The adversary sends polynomials $U^{(i,k,m)}$ to $\mathcal{S}$ for each $i \in \mathcal{E}vil$, $k = 1, \ldots, n-2t$, and $m = 1, \ldots, B$.
   - *2.6* The adversary sends $U^{(j,k,m)}(\alpha^i)$ to $\mathcal{S}$ for each $j \notin \mathcal{E}vil$, $i \in \mathcal{E}vil$, $k = 1, \ldots, n-2t$, and $m = 1, \ldots, B$.
   - *2.7* The simulator defines $u^{(i,k,m)} = U^{(i,k,m)}$ for each $i \in \mathcal{E}vil$.
   - *2.8* The simulator generates uniformly random polynomials $u^{(j,k,m)}$ of degree $\leq d$ for each $j \notin \mathcal{E}vil$, subject to the constraint that $u^{(j,k,m)}(\alpha^i) = U^{(j,k,m)}(\alpha^i)$ for each $i \in \mathcal{E}vil$ and $u^{(j,k,m)}(\beta^a) = h_a^{(k,m)}(\alpha^j)$ for all $a = 1, \ldots, \ell$.
   - *2.9* The simulator defines $\widetilde{h}_a^{(\widetilde{k},m)}$ and $\widetilde{u}^{(i,\widetilde{k},m)}$ for $\widetilde{k} = 1, \ldots, n$ by

$$\left(\widetilde{h}_a^{(1,m)}, \ldots, \widetilde{h}_a^{(n,m)}\right)^T = M \left(h_a^{(1,m)}, \ldots, h_a^{(n-2t,m)}\right)^T$$

     and

$$\left(\widetilde{u}^{(i,1,m)}, \ldots, \widetilde{u}^{(i,n,m)}\right)^T = M \left(u^{(i,1,m)}, \ldots, u^{(i,n-2t,m)}\right)^T.$$

   - *2.10* For each corrupt $P_{\widetilde{k}}$, $\mathcal{S}$ sends $\widetilde{h}_a^{(\widetilde{k},m)}(\alpha^j)$ and $\widetilde{u}^{(i,\widetilde{k},m)}(\alpha^j)$ to $\mathcal{A}$ for each $j \notin \mathcal{E}vil$.
   - *2.11* The adversary may at this point send $(\texttt{corrupt}, P_w)$ to $\mathcal{S}$ for some $w \notin \mathcal{E}vil$, corresponding to step 3.3 of $\mathcal{F}_{BR}$.
   - *2.12* If the adversary did not request to corrupt in the previous step, then $\mathcal{S}$ proceeds to step 2.13. Otherwise, the following steps are performed:

(a) The simulator relays the message $(\texttt{corrupt}, P_w)$ to $\mathcal{F}_{BR}$.

(b) The functionality $\mathcal{F}_{BR}$ sends the internal state of P_w (which includes P_w's shares of each $\widehat{H}_a^{(k,m)}$ for $a = 1, \ldots, \ell$, $k = 1, \ldots, n-3t$, and $m = 1, \ldots, B$) to $\mathcal{S}$.

(c) The simulator now updates each $h_a^{(k,m)}$ and $u^{(i,k,m)}$ to reflect the state of P_w while remaining consistent with the shares of $h_a^{(k,m)}$, $u^{(i,k,m)}$, $\widetilde{h}_a^{(\widetilde{k},m)}$, and $\widetilde{u}^{(i,\widetilde{k},m)}$ already sent to $\mathcal{A}$. Precisely how this is done is explained immediately after this specification.

(d) The simulator sends the simulated internal state of P_w to $\mathcal{A}$. This includes all P_w's shares of each $h_a^{(k,m)}$, $u^{(i,k,m)}$, $\widetilde{h}_a^{(\widetilde{k},m)}$, and $\widetilde{u}^{(i,\widetilde{k},m)}$, as well as *all* shares of each $\widetilde{h}_a^{(w,m)}$ and $\widetilde{u}^{(i,w,m)}$.

(e) Return to step 2.11.

2.13 The simulator sends $(\texttt{proceed})$ to $\mathcal{F}_{BR}$.

2.14 The adversary may now instruct the corrupt parties to send (or not send) all of their purported shares of each $\widetilde{h}_a^{(\widetilde{k},m)}$ and $\widetilde{u}^{(i,\widetilde{k},m)}$ to each $\widetilde{k} \notin \mathcal{E}vil$ according to the specification of Block-Redistribute.

2.15 The simulator emulates the honest parties broadcasting accusations in this step if any accusations are needed (according to the specification of Block-Redistribute), and the adversary may instruct corrupt parties to broadcast accusations. Any accusations $\mathcal{S}$ needs to emulate (according to the protocol specification) are sent to $\mathcal{A}$, and $\mathcal{S}$ receives from $\mathcal{A}$ any accusations it wishes to make.

2.16 Based on the accusations sent in the previous step, $\mathcal{S}$ determines which parties (if any) need to be added to $\mathcal{C}orr$ as described in step 2.7 of Block-Redistribute.

3. Share Redistribution

3.1 Upon receiving $(\texttt{Request Shares})$ from $\mathcal{F}_{BR}$, the simulator sends to $\mathcal{F}_{BR}$ the shares $h_a^{(k,m)}(\alpha^j)$ defined by

$$h_a^{(k,m)}(\alpha^j) = \lambda_{j,1} u^{(z_1,k,m)}(\beta^a) + \cdots + \lambda_{j,n-2t} u^{(z_{n-2t},k,m)}(\beta^a)$$

for each $j \in \mathcal{E}vil$, $a = 1, \ldots, \ell$, $k = 1, \ldots, n-3t$, and $m = 1, \ldots, B$.

3.2 Defining z_i and $\lambda_{j,i}$ as in Block-Redistribute, $\mathcal{S}$ sends to $\mathcal{A}$ each honest party's (simulated) share of

$$\lambda_{j,1} u^{(z_1,k,m)} + \cdots + \lambda_{j,n-2t} u^{(z_{n-2t},k,m)}$$

for each $j \in \mathcal{E}vil$, $a = 1, \ldots, \ell$, $k = 1, \ldots, n-3t$, and $m = 1, \ldots, B$.

3.3 The adversary may at this point send $(\texttt{corrupt}, P_w)$ to $\mathcal{S}$ for some $w \notin \mathcal{E}vil$, corresponding to step 8.3 of $\mathcal{F}_{BR}$.

3.4 If the adversary did not request to corrupt in the previous step, then $\mathcal{S}$ proceeds to step 3.5. Otherwise, the following steps are performed:

(a) The simulator relays the message $(\texttt{corrupt}, P_w)$ to $\mathcal{F}_{BR}$.

(b) The functionality $\mathcal{F}_{BR}$ sends the internal state of P_w (which includes P_w's shares of each $\widehat{H}_a^{(k,m)}$ for $a = 1, \ldots, \ell$, $k = 1, \ldots, n-3t$, and $m = 1, \ldots, B$) to $\mathcal{S}$.

(c) The simulator now updates each $h_a^{(k,m)}$ and $u^{(i,k,m)}$ to reflect the state of P_w while remaining consistent with the shares of $h_a^{(k,m)}$, $u^{(i,k,m)}$, $\widetilde{h}_a^{(\widetilde{k},m)}$, and $\widetilde{u}^{(i,\widetilde{k},m)}$ already sent to $\mathcal{A}$. Precisely how this is done is explained immediately after this specification.

(d) The simulator sends the simulated internal state of P_w to $\mathcal{A}$. This includes all P_w's shares of each $h_a^{(k,m)}$ and $u^{(i,k,m)}$.

(e) Return to step 3.3.

3.5 The simulator sends $(\texttt{proceed})$ to $\mathcal{F}_{BR}$.

Description of steps 2.12(c) and 3.4(c) of the simulator and completing the proof. Assuming the ability of the simulator to execute steps 2.12(c) and 3.4(c), statistical independence of the environment's views follows, even for active corruptions (indeed, it is for active corruptions that steps 2.12(c) and 3.4(c) are needed, in order to ensure consistency of the internal states that the adversary obtains by corruption). Therefore, if we can show how the simulator can always execute these steps correctly, the statistically independent outputs (between real and ideal world executions) of these steps yields the proof.

We now describe how $\mathcal{S}$ updates the polynomials in steps 2.12(c) and 3.4(c) to account for the newly corrupted party P_w. The simulator must re-choose the polynomials $\widehat{h}_a^{(k,m)}$ that were constructed in step 1.2 subject to the same constraints, but with the added constraint that $\widehat{h}_a^{(k,m)}(\alpha_w) = \widehat{H}_a^{(k,m)}(\alpha_w)$ for each $a = 1, \ldots, \ell$, $k = 1, \ldots, n-3t$, and $m = 1, \ldots, B$. Then $\mathcal{S}$ re-defines each $h_a^{(k,m)}$ defined in step 1.6 to correspond to each new $\widehat{h}_a^{(k,m)}$, setting $h_a^{(k,m)} \leftarrow \widehat{h}_a^{(k,m)} + q_a^{(k,m)}$ (using the same $q_a^{(k,m)}$).

The simulator must now re-choose $h_a^{(k,m)}$ for $k = n-3t+1, \ldots, n-2t$ in order to correspond to the polynomials $\widetilde{h}_a^{(\widetilde{k},m)}$ already sent to the adversary for $\widetilde{k} \in \mathcal{E}vil$. Let $E \subset \{1, \ldots, n\}$ be a set of size t containing the indices of the parties in $\mathcal{E}vil$ (including w). Let M_E denote the rows of M corresponding to the indices in E (i.e., if $\widetilde{k} \in E$, then the $\widetilde{k}^{\text{th}}$ row of M is in M_E). Let M_E^F denote the first $n-3t$ columns of M_E, and let M_E^L denote the last t columns. If $\widetilde{k}_1, \ldots, \widetilde{k}_t$ is an enumeration of the indices in E, then for each $a = 1, \ldots, \ell$ and $m = 1, \ldots, B$, the real adversary knows

$$\left(\widetilde{h}_a^{(\widetilde{k}_1,m)}, \ldots, \widetilde{h}_a^{(\widetilde{k}_t,m)}\right)^T = M_E^F \left(h_a^{(1,m)}, \ldots, h_a^{(n-3t,m)}\right)^T + M_E^L \left(h_a^{(n-3t+1,m)}, \ldots, h_a^{(n-2t,m)}\right)^T.$$

Since M in hyper-invertible, M_E^L is invertible. So $\mathcal{S}$ computes $h_a^{(k,m)}$ for $k = n-3t+1, \ldots, n-2t$ by setting

$$\begin{aligned}
&\left(h_a^{(n-3t+1,m)}, \ldots, h_a^{(n-2t,m)}\right)^T \\
&\quad = \quad (M_E^L)^{-1} \left[\left(\widetilde{h}_a^{(\widetilde{k}_1,m)}, \ldots, \widetilde{h}_a^{(\widetilde{k}_t,m)}\right)^T - M_E^F \left(h_a^{(1,m)}, \ldots, h_a^{(n-3t,m)}\right)^T\right].
\end{aligned}$$

After computing $h_a^{(n-3t+1,m)}, \ldots, h_a^{(n-2t,m)}$, the simulator re-defines $\widetilde{h}_a^{(\widetilde{k},m)}$ for $\widetilde{k} \in E^C$ (where E^C is the set complement of E) by re-computing the matrix multiplication in step 2.9.

We still need to verify that $\mathcal{A}$'s shares of the polynomials in $\left(h_a^{(n-3t+1,m)}, \ldots, h_a^{(n-2t,m)}\right)^T$ have not changed from what $\mathcal{A}$ specified in step 2.2. However, this follows immediately from the fact that $\mathcal{A}$'s shares of the polynomials in $\left(h_a^{(1,m)}, \ldots, h_a^{(n-3t,m)}\right)^T$ were not changed when $\mathcal{S}$ re-chose them, and $\mathcal{S}$ did not change *any* share of the polynomials in $\left(\widetilde{h}_a^{(\widetilde{k}_1,m)}, \ldots, \widetilde{h}_a^{(\widetilde{k}_t,m)}\right)^T$.

The simulator re-chooses $u^{(w,k,m)}$ for $k = 1, \ldots, n-3t$ and $m = 1, \ldots, B$ as in step 2.8, but using the updated $h_a^{(k,m)}$. Then $u^{(w,k,m)}$ for $k = n-3t+1, \ldots, n-2t$ are defined as were the $h_a^{(k,m)}$ for $k = n-3t+1, \ldots, n-2t$, namely:

$$\begin{aligned}
&\left(u^{(w,n-3t+1,m)}, \ldots, u^{(w,n-2t,m)}\right)^T \\
&\quad = \quad (M_E^L)^{-1} \left[\left(\widetilde{u}^{(w,\widetilde{k}_1,m)}, \ldots, \widetilde{u}^{(w,\widetilde{k}_t,m)}\right)^T - M_E^F \left(u^{(w,1,m)}, \ldots, u^{(w,n-3t,m)}\right)^T\right].
\end{aligned}$$

We still need to verify that $u^{(w,k,m)}(\beta^a) = h_a^{(k,m)}(\alpha_w)$ for $k = n-3t+1, \ldots, n-2t$. However, this follows immediately from the fact that $\widetilde{u}^{(w,\widetilde{k},m)}(\beta^a) = \widetilde{h}_a^{(\widetilde{k},m)}(\alpha_w)$ for $\widetilde{k} \in E$ and $u^{(w,k,m)}(\beta^a) = h_a^{(k,m)}(\alpha_w)$ for $k = 1, \ldots, n-3t$. Also, $\mathcal{A}$'s shares of $u^{(w,k,m)}$ for $k = n-3t+1, \ldots, n-2t$ have not changed (because the same argument that applied to the $h_a^{(k,m)}$ for $k = n-3t+1, \ldots, n-2t$ applies here).

C Permuting Circuit Layers

In this section, we expand upon the manner in which permutations are executed on the block-shared secrets in order to perform the required computations as specified by the circuit. In order to perform the permutations, we use Beneš networks [Ben64], also known as Waksman networks [Wak68].

The following lemma captures the behavior of Beneš networks.[8] The proof can be found in any computer science text that deals with Beneš networks (such as [Lei92]). A more general version of this lemma can be found in [GHS12, Lemma 2].

Lemma 6 *Let σ be a permutation on L elements, where L is a power of 2. Each element is given an index for its location represented as a binary integer of length $\log L$ (where the elements are indexed 0 through $L-1$). Then σ can be decomposed into $\sigma = \pi_1 \circ \pi_2 \circ \cdots \circ \pi_{2\log L - 1}$* sub-permutations *such that π_k only swaps elements whose index differs in bit k for $k \leq \log L$ and only swaps elements whose index differs in bit $2\log L - k$ for $k \geq \log L$.*

In particular, each sub-permutation is a product of disjoint two-cycles[9] where the distance between the two elements in each of the two elements in the two-cycles is the same and, moreover, is a power of 2. More specifically, if π is a sub-permutation as constructed in Lemma 6, we denote by $\mathrm{sh}(\pi)$ the distance between any two elements in one of π's 2-cycles (which will be the same for each 2-cycle). For example, $\mathrm{sh}\left((4\ 6)(5\ 7)\right) = 2$. Further, let L be constructed of consecutive blocks of ℓ elements, where ℓ is also a power of 2. Then if $\mathrm{sh}(\pi_i) < \ell$, then π_i is a permutation within blocks, and if $\mathrm{sh}(\pi_i) \geq \ell$, then π_i is a permutation between blocks.

As an example with $L = 8$, consider the permutation $\sigma = (0\ 6\ 5\ 3\ 4\ 2)$. This can be decomposed into $\sigma = \pi_1 \circ \pi_2 \circ \pi_3 \circ \pi_4 \circ \pi_5$ with $\pi_1 = (2\ 6)(3\ 7)$, $\pi_2 = (4\ 6)(5\ 7)$, $\pi_3 = (6\ 7)$, $\pi_4 = (0\ 2)$, and $\pi_5 = (3\ 7)$. One can see that each permutation satisfies the property stated in the lemma. For instance, if we write π_1 in binary, it becomes $(010\ 110)(011\ 111)$, and it is clear that each 2-cycle only permutes elements that differ in the first (leftmost) bit. In binary, $\pi_2 = (100\ 110)(101\ 111)$, and it is clear that only elements that differ in the second bit are permuted, etc. (note also that $\mathrm{sh}(\pi_1) = 4, \mathrm{sh}(\pi_2) = 2, \mathrm{sh}(\pi_3) = 1, \mathrm{sh}(\pi_4) = 2$, and $\mathrm{sh}(\pi_5) = 4$).

C.1 Implementing Circuit Reformatting Via Beneš Networks

At each layer of the circuit, we will perform some permutation σ to make sure the secrets, in blocks, are in the correct order to compute on. This requires us to assume that the number of secrets, in addition to the number of blocks, is a power of 2, which may require generating extra random secrets (which does not affect the asymptotic complexity of the protocol). We will use Lemma 6 to decompose the permutation. *Because the block size is always a power of 2, each sub-permutation in the decomposition either permutes* between *blocks, such that elements get mapped to the same location in different blocks, or the sub-permutation permutes* within *blocks, such that locations in any one block must be mapped back into that same block.* Using the example permutation above, and assuming the block size $\ell = 4$, then the secrets would be stored in two blocks, (a, b, c, d) and (e, f, g, h), where we say a is in location 0 and h is in location 7. Then π_1 from above switches c with g and d with h; note that each of these pairs are in the same location in their respective blocks. The result is (a, b, g, h) and (e, f, c, d). Then π_2 switches e with c and f with d; note that each of these pairs are in the same block.

As we have seen so far in this section, in order to perform an arbitrary permutation σ on W secrets so that the secrets are correctly arranged for computation at a given layer in the circuit, we first decompose the permutation into sub-permutations, and then perform each sub-permutation individually.

[8] Beneš networks can capture what we wish to accomplish in two different ways: One in which the paths used are edge-disjoint, and one in which the paths are node-disjoint. The lemma is for the node-disjoint version.

[9] Recall that a *two-cycle* is a permutation where exactly two elements are switched with each other; a product of permutations is *disjoint* if each element is affected by at most one of the permutations.

We describe two different protocols for performing permutations: one for permuting between blocks, denoted PermuteBetweenBlocks, and one for permuting within blocks, denoted PermuteIndividualBlocks. Executing these protocols sequentially, per the decomposition in Lemma 6 yields the final protocol PermuteLayer which performs the necessary arbitrary permutation for performing arithmetic operations on block-shared secrets at a given layer of the circuit. We note that the protocols we describe here are not the same as in [DIK10] because their subprotocols do not exactly meet our requirements; rather, our protocols more related to those from [GHS12]. We will prove correctness and perfect privacy for each of these protocols; simulation security will be proved as part of the larger MPC protocol (see the proof of Theorem 2 in Appendix D).

C.2 Permuting Between Blocks

For the protocol PermuteBetweenBlocks, we assume the secrets are stored in polynomials[10] $\{H_m\}_{m=1,\ldots,A}$, where each polynomial holds ℓ secrets and both ℓ and A are powers of 2. We refer to the sub-permutation being used as π.

We will need the following notation: For $I \subseteq \{1,\ldots,\ell\}$, let f_I denote the polynomial of degree $\leq d$ satisfying $f_I(\beta^i) = 1$ for $i \in I$, $f_I(\beta^i) = 0$ for $i \in \overline{I}$, and $f_I(\beta^i) = 0$ for $i = \ell+1,\ldots,\ell+t$. We note again that PermuteBetweenBlocks is executed for sub-permutations. In particular, PermuteBetweenBlocks is only executed when $\mathrm{sh}(\pi) \geq \ell$; since both $\mathrm{sh}(\pi)$ and ℓ are powers of 2, this implies that ℓ divides $\mathrm{sh}(\pi)$.

PermuteBetweenBlocks$(\pi, \{H_m\}_{m=1,\ldots,A})$

1. Invoke RanDouSha to generate $2A$ pairs of random polynomials (r, R), where r is of degree d, R is of degree $2d$, and both polynomials share the same block of secrets.
2. The following is done in parallel for each pair of polynomials $(H_m, H_{m+\mathrm{sh}(\pi)/\ell})$ such that $m-1$ is zero in the $\log(\mathrm{sh}(\pi))$ bit. (We look at the binary expansion of $m-1$ instead of m because the indexing of the integers permuted by π starts at zero, whereas the indexing for m starts at 1.)
   - *2.1* We define a set $I \subseteq \{1,\ldots,\ell\}$ as follows: $i \in I$ if and only if the 2-cycle $((m-1)\ell + (i-1), (m-1)\ell + (i-1) + \mathrm{sh}(\pi))$ is contained in π.
   - *2.2* Using the random polynomials generated in step 1, the parties invoke Multiply and then perform local additions to compute (and relabel)

$$\begin{array}{lcll} H_m & \leftarrow & f_{\overline{I}} \cdot H_m & + \; f_I \cdot H_{m+\mathrm{sh}(\pi)/\ell} \\ H_{m+\mathrm{sh}(\pi)/\ell} & \leftarrow & f_{\overline{I}} \cdot H_{m+\mathrm{sh}(\pi)/\ell} & + \; f_I \cdot H_m. \end{array}$$

The protocol PermuteBetweenBlocks has round complexity $O(1)$ and communication complexity $O(W + \mathrm{poly}(n))$ for permuting W secrets. Correctness follows from Lemma 6 as well as the correctness of RanDouSha and Multiply, and perfect privacy follows directly from the perfect privacy of RanDouSha and Multiply.

C.3 Permuting Within Blocks

Recall that if $\mathrm{sh}(\pi_i) < \ell$, then π_i is a permutation within blocks, and if $\mathrm{sh}(\pi_i) \geq \ell$, then π_i is a permutation between blocks. We will need the following notation: If $\mathrm{sh}(\pi) < \ell$ (i.e., π is a permutation within blocks), we can decompose π into $\pi_1 \circ \ldots \circ \pi_A$, where each π_m permutes *only* the block stored in H_m. We know that π_m is a permutation of ℓ consecutive integers, so we may define π_m^* to be π_m "shifted" so that it permutes the integers 1 through ℓ. More concretely, if π_m is a permutation of the integers x through $x+\ell$, then π_m^* contains the 2-cycle $(q, q+\mathrm{sh}(\pi))$ if and only if π_m contains the 2-cycle $(q+x-1, q+x-1+\mathrm{sh}(\pi))$.

[10] For simplicity, we index the polynomials as H_m rather than $H_a^{(k,m)}$ in the rest of the paper.

PermuteIndividualBlocks$(\pi, \{H_m\}_{m=1,\ldots,A})$

If $\mathrm{sh}(\pi) < \ell$, then the following steps are performed for each H_m in parallel.

1. Invoke RanDouSha to generate $3A$ pairs of random polynomials (r, R), where r is of degree d, R is of degree $2d$, and both polynomials share the same block of secrets.
2. Define sets $I_m, J_m, K_m \subseteq \{1, \ldots, \ell\}$ as follows: For each 2-cycle $(b, b+\mathrm{sh}(\pi))$ contained in π^*_m, $b \in I_m$ and $b + \mathrm{sh}(\pi) \in J_m$, and K_m is the set of fixed points of π^*_m.
3. Define ν_I to be the permutation of $\{1, \ldots, \ell\}$ that shifts each integer to the left by $\mathrm{sh}(\pi)$, and define ν_J to be the inverse of ν_I.
4. Invoke RandomPairs to generate A random pairs of block-sharings $(r, \nu_I(r))$ and A random pairs $(r, \nu_J(r))$.
5. Using the random pairs generated in the previous step, invoke PermuteWithinBlocks to apply ν_I to H_m, and call the result $H_{m,I}$; similarly construct $H_{m,J}$.
6. Using the random polynomials generated in step 1, the parties invoke Multiply and then perform local additions to compute (and relabel)

$$H_m \leftarrow f_{K_m} \cdot H_m + f_{I_m} \cdot H_{m,I} + f_{J_m} \cdot H_{m,J}.$$

The protocol PermuteLayer has round complexity $O(1)$ and communication complexity $O(W+\mathrm{poly}(n))$ for permuting W secrets. Correctness follows from Lemma 6 as well as the correctness of PermuteWithinBlocks, RanDouSha and Multiply, and perfect privacy follows directly from the perfect privacy of PermuteWithinBlocks, RanDouSha and Multiply.

C.4 Permuting the Whole Circuit Layer

Utilizing PermuteBetweenBlocks and PermuteIndividualBlocks, we can now execute an arbitrary permutation π on a sequence of W secrets stored in blocks of size ℓ stored in polynomials $\{H_m\}_{m=1,\ldots,A}$, where W and ℓ are both powers of 2.

PermuteLayer$(\pi, \{H_m\}_{m=1,\ldots,A})$

Let π decompose into sub-permutations $\pi_1 \circ \ldots \circ \pi_{2\log W - 1}$ per Lemma 6.

1. For $i = 2\log W - 1, ..., 1$, do sequentially:
   - *1.1* If $\mathrm{sh}(\pi_i) < \ell$, execute $\{H_m\}_{m=1,\ldots,A} \leftarrow$ PermuteIndividualBlocks$(\pi_i, \{H_m\}_{m=1,\ldots,A})$. Then execute Block-Redistribute.
   - *1.2* Else $\mathrm{sh}(\pi_i) \geq \ell$, and execute $\{H_m\}_{m=1,\ldots,A} \leftarrow$ PermuteBetweenBlocks$(\pi_i, \{H_m\}_{m=1,\ldots,A})$. Then execute Block-Redistribute.

This protocol has round complexity $O(\log W)$ and communication complexity $O(W \log W + (\log W) \cdot \mathrm{poly}(n)$ for permuting W secrets. Correctness follows from Lemma 6 as well as the correctness of PermuteIndividualBlocks and PermuteBetweenBlocks, and perfect privacy follows directly from the perfect privacy of PermuteIndividualBlocks and PermuteBetweenBlocks.

D Security of PMPC (Proof of Theorem 2)

We present the proof of Theorem 2, but first recall the theorem to be proved.

Theorem 7 *For n parties and an arithmetic circuit C that is at least $\Omega(n)$ gates wide, the protocol* PMPC *realizes $\mathcal{F}_{PMPC}$ with perfect security in the proactive UC model against an active and adaptive adversary corrupting up to $t < n/8$ parties per stage.*

Proof of Theorem 2. We note that every subprotocol used in PMPC has been proven correct; it then follows from the construction of PMPC that correctness holds. We also note that the proof here follows in large part from the proof in [DIK10] (see Appendix C there).

We now construct a simulator for PMPC and prove that it satisfies $\mathcal{F}_{PMPC}$. As in [DIK10], we build a simulator by mathematical induction. In particular, since PMPC is constructed for C via the transformed circuit C' by making consecutive calls in parallel to the required subprotocols, circuit layer by circuit layer. Accordingly, we construct a simulator that is built layer-by-layer. We also note that we prove security in the $\mathcal{F}_{BR}$-hybrid model as we have already proven in Appendix B that Block-Redistribute securely realizes $\mathcal{F}_{BR}$.

Let d be the depth of C'. Then $\mathcal{F}_{PMPC}$ consists of $O(d)$ layer-wise calls to individual subprotocols to compute these layers, where we include calls for proactive share redistribution as a new added layer. Including the redistribution calls, denote d' as the number of layers to compute C'.

We construct functionalities $\mathcal{F}_i$, $1 \leq i \leq d'$, as follows: $\mathcal{F}_i$ takes inputs $(x_1, ..., x_n)$ from the n parties and outputs the secret-shared state after computing layer i of C'. For technical reasons, the adversary must input the shares that it wishes to receive for every shared value, and $\mathcal{F}_i$ calculates the sharing of the state such that it is consistent with the adversary's shares. The base case $\mathcal{F}_0$ corresponds to secret-sharing the inputs and the appropriate prepared random pairs and double sharings.

We prove that $\mathcal{F}_{i+1}$ can be securely realized in the $\mathcal{F}_i$-hybrid model for $0 \leq i \leq d'$. Combined with the fact that $\mathcal{F}_0$ can be securely realized, this implies that $\mathcal{F}_{d'}$ can be securely realized. Finally, we show $\mathcal{F}_{d'}$ followed by Reco can be securely realized in the $\mathcal{F}_{d'}$ hybrid model, which implies that $\mathcal{F}_{PMPC}$ can be securely realized and completes the proof. Proactive security follows implicitly by the timing of the share redistribution protocol calls.

Base Case: This case consists of the functionalities $\mathcal{F}_{pairs}$, $\mathcal{F}_{double}$ and $\mathcal{F}_{RobustShare}$ being called; simulation is trivial.

Inductive Step: We assume, by the induction hypothesis, that there exists a simulator $\mathcal{S}_{i-1}$ that securely realizes the functionality $\mathcal{F}_{i-1}$. We now prove that there exists a simulator $\mathcal{S}_i$ that securely realizes $\mathcal{F}_i$ in the $\mathcal{F}_{i-1}$ hybrid model.

The simulator $\mathcal{S}_i$ selected a set of dummy inputs $x'_1, ..., x'_n$ for the parties. $\mathcal{S}_i$ then executes $\mathcal{S}_{i-1}$ on these inputs through the execution of $\mathcal{F}_{i-1}$ with adversary $\mathcal{A}$. At this point, $\mathcal{S}_i$ has computed a set of final shares for corrupted parties; these are sent to $\mathcal{F}_i$ to complete the simulation (because the ideal functionality needs the adversary to tell it which shares to output to corrupted parties).

We note that the simulation is perfect for non-adaptive corruptions because each of the subprotocols used have perfect privacy. However, in the case of adaptive corruptions, we run into two distinct cases that must be handled. The first case is when a party is corrupted before $\mathcal{F}_i$ has output shares. Here, $\mathcal{S}_i$ provides $\mathcal{A}$ the view that $\mathcal{S}_i$ already has from its dummy run of the protocol. Due to perfect privacy, we obtain that this new simulation must also be perfect.

The second and most involved case is when the party is corrupted after the protocol (e.g., the functionality corresponding to computing the circuit up to layer i) has completed. In this case, all parties have received output shares from $\mathcal{F}_i$ and the shares produced for the newly corrupted parties must be consistent with this.

Any output share y' that must be made consistent with the real share y comes from three classes of circuit computations: basic arithmetic (e.g., linear operations and multiplications on the shares and constants), proactive share redistribution (e.g., $\mathcal{F}_{BR}$) or circuit transformation operations (e.g., PermuteBetweenBlocks and PermuteIndividualBlocks).

We handle linear operations first. We note that every output share is the product of some linear equation from the outputs of $\mathcal{F}_{i-1}$, which in aggregate yield a system of linear equations. Such a system must not be over constrained else it would violate the correctness of $\mathcal{F}_i$, which would contradict correctness of $\mathcal{F}_{PMPC}$. Therefore, $\mathcal{S}_i$ is able to pick a random solution for new shares output by $\mathcal{F}_{i-1}$ that in turn

yield the real shares obtained by the adversary. However, some starting shares must now be reset for some of the sharings used by $\mathcal{S}_i$ because they correspond to the actual shares output to the adversary.

In particular, let the input sharings correspond to the polynomials f_1,...,f_m. In order to adjust the appropriate shares to output to the adversary, S_i adds polynomials $g_1, ..., g_m$ to the original polynomials, respectively. Each g_j is of degree at most d such that they evaluate to 0 at all points corresponding to corrupted parties and secret shares.

Moving through the simulation with these adjusted shares/polynomials, we run into no complications until Multiply or the circuit transformation protocols must be invoked, which themselves reduce to an instantiation of Multiply. We note that $\mathcal{F}_{BR}$ does not present a challenge as the functionality rerandomizes shares; in effect, the functionality merely adds a new zero-share to the adjusted shares. Therefore, the challenge is how to account for executing Multiply on the adjusted shares.

The protocol Multiply is constructed by multiplying sharings locally, adding a random block and opening. The sharings are a linear combination of the starting sharings and possibly outputs of previous multiplications. Thus, we would originally be opening the the polynomials

$$p = L_1(f_1, ..., f_m, h_1, ..., h_l) \cdot L_2(f_1, ..., f_m, h_1, ..., h_l) + r_{2d}, \tag{2}$$

where L_1 and L_2 are linear transformations and r_{2d} is the degree $2d$ version of r. However, we now have

$$p' = L_1(f_1 + g_1, ..., f_m + g_m, h_1, ..., h_l) \cdot L_2(f_1 + g_1, ..., f_m + g_m, h_1, ..., h_l) + r_{2d}. \tag{3}$$

In order to fix this inequality, we set

$$r'_{2d} = r + p - p' \tag{4}$$

through their corresponding polynomials, yielding a valid degree $2d$ polynomial. We then use r'_{2d} in equation 2. Note that for any j that became corrupted earlier, $p(j) = p'(j)$ because p' does not differ from p at those points by construction. Therefore, $r'_{2d} = r_{2d}$ in those same points, and we have adjusted the view according to the share adjustments specified above.

It still remains to show how to adjust the final output shares if they come from a multiplication, $\mathcal{F}_{BR}$, or a circuit transformation. However, the reasoning so far shows how one can add adjustment polynomials and "fix" the remaining shares via the Multiply subprotocol, which covers the reasoning necessary for adjusting all the output shares from every protocol but $\mathcal{F}_{BR}$. To see how to adjust the outputs for $\mathcal{F}_{BR}$, we note that the ideal functionality there specifies the output of the ideal functionality to be completely independent from the inputs; $\mathcal{S}_i$ therefore simply selects the real output shares as the output shares of $\mathcal{F}_{BR}$.

Step $d' + 1$: Having proven that $\mathcal{F}_{d'}$ can be securely realized via mathematical induction, it remains to show that $\mathcal{F}_{PMPC}$ can be securely realized. There is only one round in the corresponding protocol here: it consists of calling $\mathcal{F}_{d'}$ to receive sharings and then reconstructing those sharings towards the corresponding parties. Simulation here follows immediately from the UC-security of Reco. ■

E Party Virtualization in a Constant Number of Rounds

The PMPC protocol constructed in Section 3 has a lower per-stage corruption threshold $t < n/8$, than we desire. In order to increase the threshold $(1/3 - \epsilon)n$ for a positive constant ϵ, we use the party virtualization techniques initially described in [Bra87]. At a high level, the parties in the above protocol are replaced with committees of parties in such a way that a greater number of parties can actually be corrupted while still maintaining security. More precisely, n *virtual parties* are constructed from among the n real parties. The virtual parties compute PMPC, which we term the *outer protocol*. The committee making up a virtual party must securely compute the functionalities that the corresponding virtual party

must compute to execute PMPC; we call the committee protocol the *inner protocol*. See [DIK+08, DIK10] for a proof that this composition maintains (perfect UC) security[11] as well as further details.

The result of [Bra87] is non-constructive in that the committees are chosen randomly, which will not work for perfect security. In order to enable perfect security, the technique of [Bra87] was made constructive by [DIK+08].

We call a committee *corrupt* if 1/3 or more of the parties belonging to the committee are corrupt. The following is implicit from [DIK10] using the proof of Lemma 5 in [DIK+08].

Lemma 8 *For any $0 < \epsilon, \delta < 1$, and n parties, there exists a construction of n committees of size $s = O(1/\delta\epsilon^2)$ such that if no more than $(\frac{1}{3} - \epsilon)n$ of the parties are corrupt, then no more than $\delta \cdot n$ committees will be corrupt. The members of the committees can be computed in time $n \cdot \operatorname{polylog} n$.*

Since PMPC is secure with a per-stage corruption threshold of $t < n/8$, we set $\delta = 1/8$. We also set $c = \lceil s/3 \rceil$. Because PMPC-pv must have perfect security, where all subprotocols are constant-round, executions of the the inner protocol must have perfect security and be constant-round. We use the BGW MPC protocol [BOGW88] for the inner protocol, since it satisfies both of these properties for the required functionalities, as we now describe.

There are numerous functionalities that the committees need to be able to compute to carry out PMPC-pv. Communication between two parties is now replaced by communication between two committees. In addition, internal computations by parties in PMPC must now be computed by the committees emulating those parties in PMPC-pv using multiparty computation among the committee. Examining PMPC for all the computations that each party must execute, the types of computations that need to be performed by the committees using MPC throughout the entire protocol are: addition, multiplication of two private values, multiplication of a vector of shares by a publicly known hyper-invertible matrix, and the Berlekamp-Welch algorithm. We now discuss how each of these functionalities is computed by committee.

Broadcasts. In order to reduce the number of broadcasts used in the main protocol, all the broadcasts in the multiplication subprotocol will be implemented with point-to-point channels using a broadcast protocol. The minimum number of rounds to implement a (deterministic) broadcast protocol for a committee with at most s corrupt parties is $s + 1$ (see [FL82]). Since s is a constant that depends on ϵ, our protocol will work in a constant number of rounds. Any broadcast protocol that achieves the $s + 1$ lower bound on the number of rounds and has communication and computational complexity polynomial in the number of committee members will work for our purposes.

Communication between Committees. We describe here how committees emulate secure point-to-point channels for the virtual parties. Suppose one committee wants to send a secret value to another committee. Denote the parties in the sending committee by $p_1^{\text{send}}, \ldots, p_s^{\text{send}}$ and the parties in the receiving committee by $p_1^{\text{rec}}, \ldots, p_s^{\text{rec}}$. Denote the evaluation point of p_j^{send} and p_j^{rec} by γ_j.[12] So each p_j^{send} holds a share $f_0(\gamma_j)$ of some polynomial f_0, where $f_0(0)$ is the secret to be transmitted. The sending parties generate c random polynomials of degree $\leq c$, which we label $f_1, \ldots, f_c$. (The description of how to generate these polynomials is given in [BOGW88].) Then each party in the sending committee sends his share of $f_0 + \gamma_j f_1 + \cdots + \gamma_j^c f_c$ to party p_j^{rec}, who then uses the Berlekamp-Welch algorithm to interpolate the polynomial. The constant term of this polynomial is recorded as p_j^{rec}'s share of the secret. Each party in the receiving committee now holds a share of the polynomial $f_0(0) + x f_1(0) + \cdots + x^c f_c(0)$, and the constant term of this polynomial is the secret. Thus the transmission is complete.

[11] Proactive security is also clearly maintained as the committees jointly execute that subprotocols, with no carry over from one subprotocol to the next that might violate share storage restrictions.

[12] The assumption that they have the same evaluation point is not necessary, but simplifies exposition.

Internal Addition. Additions are performed by adding shares as specified in the BGW protocol.

Multiplication of Two Private Values. The only situation in which a committee needs to multiply two private values is when the committee needs to multiply two of its shares; this only occurs in the Multiply protocol. These multiplications will be handled as specified in the BGW protocol. Since the committee size is constant, this only requires a constant amount of computation per multiplication. Examining the BGW protocol, each multiplication requires 7 communication rounds and 6 broadcast rounds.

Multiplication by Hyper-Invertible Matrices. Matrix multiplication involves both multiplications and additions. Every hyper-invertible matrix in PMPC has dimension $\Theta(n)$ by $\Theta(n)$. Normally, such a computation would require $\Theta(n^2)$ multiplications. However, since the hyper-invertible matrices model polynomial interpolation and evaluation, we can use efficient algorithms from the computer science literature (see, for instance, [AHU74]) to do these computations with only $O(n \operatorname{polylog} n)$ multiplications.

The Berlekamp-Welch Algorithm. Executing the Berlekamp-Welch algorithm in committees as part of the inner protocol requires some care, because a straightforward application of the BGW protocol would lead to non-constant round complexity. The Berlekamp-Welch algorithm can be performed in $O(n \operatorname{polylog}(n))$ time ([Gao02]). This adds a $\operatorname{polylog}(n)$ factor to the communication complexity, which is not problematic. However, it requires $O(\operatorname{polylog}(n))$ rounds of communication, which is not acceptable for our purposes.

We therefore construct a constant-round protocol Committee-BW that each committee jointly computes to perform the Berlekamp-Welch algorithm. This requires generating extra masking randomness. In fact, for each polynomial to interpolate, we must generate an additional c polynomials. Since the committee size is a fixed constant throughout the protocol (as it only depends on the constant ϵ), generating these extra sharings does not affect the asymptotic complexity of the protocol PMPC-pv.

When parties needed to compute the Berlekamp-Welch algorithm in the outer protocol, they execute Committee-BW in the inner protocol. Committee-BW uses RanDouSha as a subprotocol to generate random masking polynomials. The number of polynomials generated and the degrees of the polynomials will be different in different steps. For every polynomial the parties want to interpolate, they generate c masking polynomials of the same degree. Again, this does not add to the overall communication complexity of the protocol since c is constant for constant ϵ. Note that in some instances, a party/committee is not interpolating a polynomial, but rather a vector generated by a hyper-invertible matrix. However, since the hyper-invertible matrices we use model polynomial interpolation, such a vector can be seen as a set of evaluation points on a polynomial.

The protocol Committee-BW implements a committee performing Berlekamp-Welch in a constant number of rounds. The ideal functionality that the protocol is designed to emulate is described in Figure 4. We use $\mathcal{P} = \{P_i\}_{i=1}^n$ to denote the set of committees and $Com = \{p_j\}_{j=1}^s$ to denote the committee that is to perform Berlekamp-Welch. The evaluation point of P_i is α_i, and the evaluation point of p_j is γ_j. We assume that the polynomial f to be interpolated has already been sent to the committee. This means that each share $f(\alpha_i)$ is Shamir-shared among the committee as a polynomial f_{α_i} of degree no more than c such that $f_{\alpha_i}(0) = f(\alpha_i)$. Furthermore, the committee holds an additional c polynomials $r^{(1)}, \ldots, r^{(c)}$, shared with polynomials $r_{\alpha_i}^{(k)}$ of degree no more than c such that $r_{\alpha_i}^{(k)}(0) = r^{(k)}(\alpha_i)$. The protocol uses an s by $c+1$ hyper-invertible matrix M', which is publicly known and fixed throughout the PMPC protocol.

Committee-BW$(c, Com, \{f_{\alpha_i}\}_{i=1}^n)$

1. The committees invoke RanDouSha to generate random polynomials $r^{(k)}$ for $k = 1, \ldots, c$. (In the actual PMPC protocol, this will be parallelized for efficiency.)

Let CC be the set of indices of corrupt committees. Initialize $S = \emptyset$.

1. $\mathcal{Z}$ instantiates parties with the same set of points $\xi_1, \ldots, \xi_m$, and each p_j receives shares $f_{\alpha_i}(\gamma_j)$. $\mathcal{Z}$ instantiates $\mathcal{A}$ with auxiliary input z.
2. Each party p_j sends $\xi_1, \ldots, \xi_m$ to $\mathcal{F}_{CBW}$ as well as all shares $f_{\alpha_i}(\gamma_j)$. If different parties send different sets of points ξ_i, then $\mathcal{F}_{CBW}$ broadcasts (`abort`) and aborts.
3. Initial Corruption Phase:
   - *3.1* If $\mathcal{A}$ wishes to corrupt parties p_j, $\mathcal{A}$ sends $(\texttt{corr?}, p_j)$ to $\mathcal{Z}$. $\mathcal{Z}$ responds either `yes` or `no`. $\mathcal{Z}$ never responds `yes` when the number of corrupt parties in the committee is greater than 1/3 (or, in the statistical case, 1/2).
   - *3.2* For all parties p_j that $\mathcal{Z}$ responded `yes` to, $\mathcal{A}$ sends $(\texttt{corr}, p_j)$ to $\mathcal{F}_{CBW}$, who sends p_j's inputs to $\mathcal{A}$.
   - *3.3* $\mathcal{F}_{CBW}$ sends $\mathcal{A}$ all inputs for all corrupted p_j.
   - *3.4* $\mathcal{A}$ may continue to corrupt parties by returning to step 3.1; else, $\mathcal{A}$ sends `proceed` to $\mathcal{F}_{CBW}$.
4. Computation Phase
   - *4.1* $\mathcal{F}_{CBW}$ uses all uncorrupted parties' shares to interpolate $f_{\alpha_i}(0) = f(\alpha_i)$ for each $i = 1, \ldots, n$.
   - *4.2* $\mathcal{F}_{CBW}$ interpolates f, noting which shares are inconsistent. $\mathcal{F}_{CBW}$ updates S by adding to it the indices i of shares $f(\alpha_i)$ that were inconsistent.
   - *4.3* Set $I = \{1, \ldots, n\} \backslash S$. Let $\{\lambda_i^{(k)}\}_{i \in I}$ denote the Lagrange coefficients for interpolating the point ξ_k on a polynomial using the points $\{\alpha_i\}_{i \in I}$. For each $k = 1, \ldots, m$, $\mathcal{F}_{CBW}$ computes
   $$f_{\xi_k}(\gamma_j) = \sum_{i \in I} \lambda_i^{(k)} f_{\alpha_i}(\gamma_j).$$
5. Output Corruption Phase:
   - *5.1* If $\mathcal{A}$ wishes to corrupt parties p_j, $\mathcal{A}$ sends $(\texttt{corr?}, p_j)$ to $\mathcal{Z}$. $\mathcal{Z}$ responds either `yes` or `no`. $\mathcal{Z}$ never responds `yes` when the number of corrupt parties in the committee is greater than 1/3 (or, in the statistical case, 1/2).
   - *5.2* For all parties p_j that $\mathcal{Z}$ responded `yes` to, $\mathcal{A}$ sends $(\texttt{corr}, p_j)$ to $\mathcal{F}_{CBW}$, who sends p_j's inputs to $\mathcal{A}$.
   - *5.3* $\mathcal{F}_{CBW}$ sends $f_{\xi_k}(\gamma_j)$ and S for all corrupted p_j to $\mathcal{A}$.
   - *5.4* $\mathcal{A}$ may continue to corrupt parties by returning to step 5.1; else, $\mathcal{A}$ sends `proceed` to $\mathcal{F}_{CBW}$.
6. $\mathcal{F}_{CBW}$ sends S to each party, and for each $k = 1, \ldots, m$, $\mathcal{F}_{CBW}$ sends $f_{\xi_k}(\gamma_j)$ to each honest p_j.

Figure 4: The $\mathcal{F}_{CBW}$ functionality.

2. Each committee sends its share of each $r^{(k)}$ to Com. Define $r^{(k)}_{\alpha_i}$ as described above.
3. We define polynomials $u^{(1)}, \ldots, u^{(s)}$ by $(u^{(1)}, \ldots, u^{(s)})^T = M'(f, r^{(1)}, \ldots, r^{(c)})^T$. We similarly define $(u^{(1)}_{\alpha_i}, \ldots, u^{(s)}_{\alpha_i})^T = M'(f_{\alpha_i}, r^{(1)}_{\alpha_i}, \ldots, r^{(c)}_{\alpha_i})^T$ for each P_i. Each p_j locally computes his share of each $u^{(k)}_{\alpha_i}$.
4. Each p_j sends his share of $u^{(k)}_{\alpha_i}$ to p_k for each P_i.
5. Each p_j uses Berlekamp-Welch to interpolate $u^{(k)}_{\alpha_i}$ (and hence $u^{(k)}(\alpha_i)$) from the shares received in the previous step.
6. Each p_j uses Berlekamp-Welch to interpolate $u^{(k)}$, noting which shares he believes to be incorrect.
7. Each p_j sends to each member of Com the index of each committee P_i which he believes to have sent an incorrect share (these are called "negative votes").
8. For each α_i that received more than c negative votes in the previous step, the committee Com concludes that committee P_i is corrupt, and his share of f is unneeded (even if the value of $f(\alpha_i)$ sent by P_i was correct). Let I be the set of all i such that P_i was *not* deemed to be corrupt
9. Suppose the committee wants to interpolate a set of points $\xi_1, \ldots, \xi_m$. Let $\{\lambda^{(k)}_i\}_{i \in I}$ denote the Lagrange coefficients for interpolating the point ξ_k on a polynomial using the points $\{\alpha_i\}_{i \in I}$. Each p_j locally computes his share of $f(\xi_k)$ for each $k = 1, \ldots, m$ by

$$f_{\xi_k}(\gamma_j) = \sum_{i \in I} \lambda^{(k)}_i f_{\alpha_i}(\gamma_j).$$

Invoking Committee-BW W/n times in parallel has communication complexity $O(W + \text{poly}(n))$ (assuming the committee wants to interpolate $O(n)$ points per invocation, which will always be the case in the execution of the protocol). It requires $O(1)$ rounds of communication. We stress implementing Berlekamp-Welch in a committee the "straightforward" way would lead to $O(\text{polylog}(n))$ rounds of communication, as opposed to the constant-round protocol here.

In the lemma below, we use "inconsistent" to mean "inconsistent with the shares of the honest committees." Since there are enough honest committees to perform Berlekamp-Welch, there is no ambiguity as to what this means.

We now sketch the security of Committee-BW; we leave a full proof to the full version of this paper.

Lemma 9 *In* Committee-BW*, if the share of at least one of the polynomials $f, r^{(1)}, \ldots, r^{(c)}$ sent to Com is inconsistent at point α_i, then at least $s - c$ of the polynomials $u^{(1)}, \ldots, u^{(s)}$ will have an inconsistent share at point α_i.*

Proof: Suppose by way of contradiction that at least one of the polynomials $f, r^{(1)}, \ldots, r^{(c)}$ is inconsistent at point α_i and that there are $c+1$ polynomials $u^{(k)}$ such that the share at point α_i is consistent. Then since M' is hyper-invertible, one could construct the polynomials $f, r^{(1)}, \ldots, r^{(c)}$ as a linear combination of the $c+1$ polynomials $u^{(k)}$. But then the share at α_i would be consistent for all of $f, r^{(1)}, \ldots, r^{(c)}$, a contradiction. ■

Lemma 10 *Assuming that no more than c members of Com are corrupt and no more than t committees in $\mathcal{P}$ are corrupt in the execution of* Committee-BW*:*

1. *The set of committees that are deemed corrupt by the honest parties in Com in step 8 is correct, in that any committee that sent an inconsistent share is deemed corrupt, and any committee that sent only consistent shares is deemed honest.*
2. *The honest parties correctly interpolate the intended points on f.*
3. *The adversary does not gain any information on f.*

Proof:

1. A party needs to receive at least $c+1$ negative votes to deem a committee corrupt. By lemma 9, if committee P_i sent at least one incorrect share, then at least $s - c > c$ honest parties will send out a negative vote for P_i, and all honest parties will deem P_i corrupt. If P_i sent all correct shares, then P_i's share of each $u^{(k)}$ will be correct, and only corrupt parties might cast a negative vote for P_i; since there are at most c corrupt parties, there will not be enough negative votes to deem P_i corrupt.

2. It follows from the previous point that the shares of f used for interpolation in step 9 of Committee-BW are correct shares, so the interpolation using these points must be correct.

3. Consider all the data that the adversary receives in the execution of Committee-BW:

   - $f_{\alpha_i}(\gamma_j)$ and $r^{(k)}_{\alpha_i}(\gamma_j)$ for each corrupt p_j, each $i = 1, \ldots, n$, and each $k = 1, \ldots, c$.
   - *All* shares of $u^{(k)}_{\alpha_i}$ for each $i = 1, \ldots, n$ and each corrupt p_k.
   - All the negative votes sent by the honest parties in step 7.

   First, with respect to the negative votes, the adversary already knew before the execution of the protocol which shares of the polynomials f and $r^{(k)}$ were corrupt, and since the honest parties only cast negative votes for corrupt shares, seeing the negative votes of the honest parties does not give the adversary any additional information.

   With respect to the shares of polynomials the adversary learns, it suffices to show that for any possible collection of values of these shares, any choice of f is equally likely to correspond to those values. In particular, we want to show that for any collection of values of the adversary's shares, there is precisely one selection polynomials $r^{(k)}$ and $r^{(k)}_{\alpha_i}$ and f_{α_i} that corresponds to a given f. The proof proceeds as follows: We show that given the adversary's shares of each f_{α_i} and the (hidden) shares of f, there can only be one choice of the polynomials f_{α_i}; that given f and $u^{(k)}_{\alpha_i}$ for each corrupt p_k, there can only be one choice of polynomials $r^{(k)}$; and then that given each $r^{(k)}$ and the adversary's shares of $r^{(k)}_{\alpha_i}$, there can only be one choice of polynomials $r^{(k)}_{\alpha_i}$.

   Since $f(\alpha_i) = f_{\alpha_i}(0)$ for each α_i, and since each f_{α_i} has degree no more than c, it follows that the adversary's shares of each f_{α_i} together with the shares $f(\alpha_i)$ are enough to uniquely determine the polynomials f_{α_i}.

   The adversary knows all shares of $u^{(k)}_{\alpha_i}$ for each corrupt p_k. In particular, the adversary knows $u^{(k)}$ for each corrupt p_k. Let M'_C denote the matrix formed from all the rows of M' corresponding to corrupt parties. Let $M'_{C,1}$ denote the first column of M'_C, and let $M'_{C,2}$ denote the matrix formed from all but the first column of $M'_{C,2}$. Since M' is hyper-invertible, it follows that $M'_{C,2}$ is an invertible $c \times c$ matrix. Now by definition, $(u^{(1)}, \ldots, u^{(s)})^T = M'(f, r^{(1)}, \ldots, r^{(c)})^T$, so the polynomials $u^{(k)}$ for corrupt p_k are computed by $M'_{C,1} f + M'_{C,2}(r^{(1)}, \ldots, r^{(c)})^T$. Since $M'_{C,2}$ is invertible, it follows that there is only one choice of polynomials $r^{(1)}, \ldots, r^{(c)}$ corresponding to the given f and $u^{(k)}$ for corrupt p_k.

   Since each r_{α_i} has degree no more than c, and since $r^{(k)}(\alpha_i) = r_{\alpha_i}(0)$ for each α_i, it follows that the adversary's shares of each r_{α_i} together with the shares $r^{(k)}(\alpha_i)$ are enough to uniquely determine the polynomials r_{α_i}. ■

E.1 PMPC with statistical security

We remark that the techniques above can be used to construct a proactively secure protocol with *statistical* security and a per-stage corruption threshold of $(1/2 - \epsilon)n$. We outline briefly how such a protocol is possible.

Using the results in [DIK+08], we call a committee *corrupt* if 1/2 or more of the parties belonging to the committee are corrupt. The following is Lemma 5 in [DIK+08].

Lemma 11 *For any $0 < \epsilon, \delta < 1$, and n parties, there exists a construction of n committees of size $s = O(1/\delta\epsilon^2)$ such that if no more than $(\frac{1}{2} - \epsilon)n$ of the parties are corrupt, then no more than $\delta \cdot n$ committees will be corrupt. The members of the committees can be computed in time $n \cdot \operatorname{polylog} n$.*

For the inner protocol, instead of using the perfectly secure protocol of [BOGW88] with 1/3 threshold, we use the statistically secure protocol of [RB89] with 1/2 threshold. Communication and computation occurs largely as above; the main subtlety is dealing with the authentication tags (e.g., information checking), which are shared across each committee, that [RB89] requires to ensure share correctness. In particular, committees must have a way of securely sending information with correct shared authentication tags to other committees.

Such a functionality can be accomplished as follows: Suppose the parties in one committee, consisting of parties $A_1, ..., A_s$, must send information with tags to another committee consisting of parties $B_1, ..., B_s$. Using [RB89], all $2s$ of these parties compute a functionality which transfers the data. There is one input gate for each A_j and one output gate for each B_i. The input from A_j is her share and authentication/verification tags for the piece of data that the committee $\{A_1, \ldots, A_s\}$ holds. The output for B_j is her (newly generated) share and authentication/verification tags for this same piece of data, now held by the committee $\{B_1, \ldots, B_s\}$. This is done such that the new sharing is independent of the initial sharing. Because the committee sizes are constant, this does not asymptotically increase the communication or computational complexity.

The protocol Committee-BW must also be modified to work when [RB89] is being used as the protocol within committees. The only change that needs to take place is in step 5. The Berlekamp-Welch algorithm cannot be used here since the threshold of corruption (within the committee) is now one half the committee size. So instead of using Berlekamp-Welch to correctly open that sharing, the parties simply open the VSS sharing as specified in [RB89].

Denote PMPC-pv' as the protocol using [RB89] as the inner protocol as discussed. We can therefore obtain the following theorem.

Theorem 12 *For any $0 < \delta < 1/2$, for n parties and an arithmetic circuit C that is at least $\Omega(n)$ gates wide, the protocol* PMPC-pv' *realizes $\mathcal{F}_C$ with statistical security in the proactive UC model against an active and adaptive adversary corrupting up to $t < \delta n$ parties per stage.*