Perfectly Secure Multiparty Computation and the Computational Overhead of Cryptography

Ivan Damgård[1], Yuval Ishai[2]*, and Mikkel Krøigaard[3]**

[1] University of Aarhus, Denmark. Email: ivan@cs.au.dk
[2] Technion and UCLA. Email: yuvali@cs.technion.ac.il
[3] Eindhoven University of Technology. Email: m.kroigaard@tue.nl

Abstract. We study the following two related questions:

- What are the minimal computational resources required for general secure multiparty computation in the presence of an honest majority?
- What are the minimal resources required for two-party primitives such as zero-knowledge proofs and general secure two-party computation?

We obtain a nearly tight answer to the first question by presenting a *perfectly* secure protocol which allows n players to evaluate an arithmetic circuit of size s by performing a total of $\mathcal{O}(s \log s \log^2 n)$ arithmetic operations, plus an additive term which depends (polynomially) on n and the circuit depth, but only logarithmically on s. Thus, for typical large-scale computations whose circuit width is much bigger than their depth and the number of players, the amortized overhead is just polylogarithmic in n and s. The protocol provides perfect security with guaranteed output delivery in the presence of an active, adaptive adversary corrupting a $(1/3 - \varepsilon)$ fraction of the players, for an arbitrary constant $\varepsilon > 0$ and sufficiently large n. The best previous protocols in this setting could only offer *computational* security with a computational overhead of $\text{poly}(k, \log n, \log s)$, where k is a computational security parameter, or perfect security with a computational overhead of $\mathcal{O}(n \log n)$.

We then apply the above result towards making progress on the second question. Concretely, under standard cryptographic assumptions, we obtain zero-knowledge proofs for circuit satisfiability with 2^{-k} soundness error in which the amortized computational overhead per gate is only *polylogarithmic* in k, improving over the $\omega(k)$ overhead of the best previous protocols. Under stronger cryptographic assumptions, we obtain similar results for general secure two-party computation.

1 Introduction

This work studies two different but closely related questions: the complexity of *secure multiparty computation* (MPC) in the presence of an honest majority,

* Supported by BSF grant 2008411, ISF grant 1310/06, and NSF grants 0830803, 0716835, 0627781.

** Most of the work done at the University of Aarhus.

and the complexity of *two-party* cryptographic primitives such as zero-knowledge proofs and secure two-party computation.

1.1 The Complexity of MPC

We consider the question of MPC over secure point-to-point channels in the presence of an active (malicious) adversary, who may corrupt up to some constant fraction δ of the n players. In this work we focus on the case of an honest majority, where $\delta < 1/2$. Unlike the case of MPC with no honest majority, in this case it is possible to guarantee output delivery and provide unconditional security. Following the initial feasibility results of [16, 3, 8, 26], a long sequence of works, initiated by [13, 14, 18, 10], attempted to minimize the communication and computation resources required for general MPC in this setting.

To make the question cleaner and less sensitive to variations in the model, we adopt the following standard conventions. First, to measure the growth of complexity with the number of players, we consider n as a parameter which tends to infinity. A large value of n captures not only computations which combine inputs from many players, but also "cloud computing" scenarios in which a large number n of untrusted or unreliable *servers* are used to distribute computations on inputs that originate from a small number of *clients* or even from just a single client. Second, to eliminate from consideration an *additive* overhead which depends (polynomially) on n and a security parameter[4] but does not grow with the complexity of the functionality f, we assume the circuit complexity of f to be much bigger than n. This is in line with most typical MPC application scenarios, and may capture both complex computations on small inputs and simple computations on massive inputs.

More concretely, we consider the task of securely evaluating a function f represented by a boolean circuit C whose inputs and outputs are arbitrarily partitioned between the n players. We let k denote a security parameter, such that the simulation error of the protocol is bounded by 2^{-k}. (This should hold for computationally unbounded adversaries in the case of statistical security and for 2^k-bounded adversaries in the case of computational security; the parameter k can be ignored in the case of perfect security.) We say that a general MPC protocol has *computational overhead* $c(n, k, s)$ if for all positive integers n, k, s, and circuit C of size s, the total number of bit operations[5] performed by all n players together is at most $s \cdot c(n, k, s) + \text{poly}(n, k, \log s)$. The computational overhead can be thought of as the amortized multiplicative price for achieving security: the ratio between the cost of securely distributing an expensive task between n players and the cost of a centralized (insecure) solution for the same task.

[4] Such an overhead is very sensitive to the underlying network and MPC model, and is required in our settings even for performing the simple MPC task of broadcasting a single bit.

[5] Our count of bit operations includes both local computations and point-to-point communication.

Note that the computational overhead of a protocol implies a similar bound on its *communication* overhead with respect to the circuit size. However, in light of Gentry's recent candidate for a fully homomorphic encryption scheme [15], the circuit size should no longer be generally seen as a barrier for the communication complexity of MPC. This notion still looks meaningful in the setting of unconditional security or for circuits whose input or output length are comparable to their size. See Section 8 for further discussion.

The computation and communication overhead of the first general MPC protocols [16, 3, 8] were large polynomials in n, k (e.g., $\mathcal{O}(n^8)$ for a naive implementation of the perfectly secure BGW protocol over a point-to-point network [3, 18]). Following a long sequence of works (see [12] for a survey) the current state of the art can be summarized as follows. For simplicity, we do not state the resilience level of each protocol. Using a general protocol composition technique from [6, 17, 12], all protocols can be made nearly optimally resilient with the same asymptotic overhead.

In the setting of computational security, an overhead of $c(n, k, s) = \text{poly}(k, \log n, \log s)$ was achieved in [12]. This protocol can be realized with a constant number of rounds under standard cryptographic assumptions.

In the case of unconditional security, all efficient MPC protocols from the literature require the round complexity to grow with the circuit depth d. Since all players in these protocols are active in each round, we redefine computational overhead for the unconditional case to allow an additive term of $\text{poly}(n, k, d, \log s)$ (the exponent of d should be extremely low here, or the term can become dominant). The computational overhead of the best *perfectly* secure protocol prior to this work [2] was $n \cdot \text{polylog}(n)$. This protocol has a similar communication overhead. In the case of *statistical* security and protocols which take inputs from and deliver outputs to only a *constant* number of clients (but still distribute the computation among n servers) a variant of the protocol from [11] based on algebraic geometric secret-sharing [9] (see [19, 21]) has computation overhead of $k \cdot \text{polylog}(n)$ and communication overhead of $\mathcal{O}(1)$.

This state of the art leaves open several natural questions:

- Can the computational overhead be simultaneously sublinear in both n and k in *any* MPC model? This question turns out to be relevant for the applications discussed in Section 1.3 below.
- Can the computational overhead be sublinear in n with *perfect* security, or alternatively with statistical security even when inputs can originate from all players (as opposed to a constant number of clients as in [11, 21])? These questions are open even for the easier case of *communication* overhead.

1.2 Our Results

We present a *perfectly* secure general MPC protocol whose computational overhead is *polylogarithmic* in n, answering the above questions affirmatively.

More concretely, the protocol can tolerate an active, adaptive adversary corrupting up to a $1/3 - \varepsilon$ fraction[6] of the players, for an arbitrary constant $\varepsilon > 0$ and all sufficiently large n. The computational (and communication) complexity required for evaluating a boolean circuit C of size s and depth d is $\mathrm{polylog}(n) \log s \cdot s + d^2 \cdot \mathrm{poly}(n, \log s)$. If C is an *arithmetic* circuit over a finite field of size bigger than n, the total computational work involves $\mathcal{O}(\log^2 n \log s \cdot s) + d^2 \cdot \mathrm{poly}(n, \log s)$ arithmetic operations and the communication includes $\mathcal{O}(\log n \log s \cdot s) + d^2 \cdot \mathrm{poly}(n, \log s)$ field elements.

Alternatively, in the case where d^2 is too large, we provide an option to increase the circuit size by a factor $\log d$ while decreasing the d^2 factor to $d \log d$. The intuition is that the first factor on the second term is dX, where X is defined as follows. Dividing the circuit into layers in the natural way, we define the number X to be the maximal number of layers reachable by one wire from any given layer. In general, $X = \mathcal{O}(d)$ and so the factor is d^2. With our alternative approach, $X = \mathcal{O}(\log d)$ and so the factor is $d \log d$. The real calculation is a bit more involved, but this is the basic idea.

Thus, with the above alternative, the computational complexity for an arithmetic circuit becomes $\mathcal{O}(\log^2 n \log s \log d \cdot s) + d \log d \cdot \mathrm{poly}(n, \log s)$, and similarly for the other complexities.

Since the modification of the circuit increases its size by a factor $\log d$, it is not always the best solution. Only for circuits with a large depth is the alternative a good choice. Furthermore, the d^2 factor is the result of a somewhat pessimistic worst-case analysis, and for most typical circuits the additive term grows only linearly with d.

As a final remark about our protocol, it seems "lean" enough to be implemented in practice. This should be contrasted with the previous best protocol from [12], which involves a distributed evaluation of a pseudorandom function for every gate in the circuit.

Techniques. Our protocol employs several techniques that were used in previous works along this line, including the share-packing technique from [13], allowing to secret-share a block of secrets with a low amortize cost, and the efficient verifiable secret sharing protocol from [2, 12]. The main technical challenge is to perform "non-homogenous" computations on pairs of blocks, i.e., ones that are different from coordinate-wise addition or multiplication of blocks. We address this challenge by embedding the computation in a special form of a universal circuit based on the so-called Beneš network [5, 28]. The high level idea is that the structure of the circuit reduces the computation in a given layer of the circuit to an arbitrary permutation *between* blocks (which can be done locally), homogenous operations, and a logarithmic number of distinct permutations *within* blocks. We propose an efficient procedure for the latter. See Section 4 for a more detailed technical overview.

[6] In our model we assume that only point-to-point channels are available, in which case it is impossible to achieve unconditional security with guaranteed output delivery if at least 1/3 of the players can be corrupted.

An independently interesting contribution is a new methodology for the security analysis of honest-majority MPC protocols. Similarly to most protocols of this type, our protocol is composed from subprotocols that generate auxiliary secret shared values to help in the computation, a subprotocol for sharing the inputs, and finally a "layer-protocol" that performs secure computation corresponding to one layer of the circuit, i.e., it starts with the shares of values going into the layer, consumes some auxiliary shared values, and outputs shares of values coming out of the layer. Our proof of security first proves all subprotocols to be UC secure. We then define a functionality $\mathcal{F}_i$ that takes inputs from the players and outputs shares of the values output by the i'th layer of the circuit (where layer 0 just produces the inputs to the circuit). We then show that $\mathcal{F}_0$ can be implemented by calling the auxiliary subprotocols, and $\mathcal{F}_i$ for $i > 0$ can be (UC-)implemented by calling $\mathcal{F}_{i-1}$ and then executing the layer-protocol.

We believe this may be the first example of a general honest-majority MPC protocol with a fully modularized proof of security. The main challenge is that it is non-trivial to define functionalities for the subprotocols such that 1) the subprotocol actually realizes the functionality and 2) the functionality provides what is needed in the larger context. It is well known that even for a simple task such as digital signatures, defining the "right" functionality is not easy.

In our case, the main idea turn out to be that a functionality that is supposed to output shares of some secrets, should not simply choose those shares on its own and send them to the players, although that may seem like the most natural approach. Instead, our functionalities ask the adversary which shares it wants the corrupted players to get, and the functionality then chooses shares for the honest players conditioned on the shares obtained from the adversary and the secret. In a sense, this models the fact that we do not care about the distribution of shares the adversary sees, as long as the secret is safe.

1.3 The Computational Overhead of Cryptography

A somewhat unexpected motivation for this work comes from the recent applications of honest-majority MPC to two-party primitives such as zero-knowledge proofs and general secure two-party computation [19, 21]. We note that these general tasks can be used as building blocks for more specialized two-party tasks such as identification or different flavors of signatures.

The computation and communication overhead of standard two-party cryptographic primitives can be defined similarly to the overhead of MPC as defined above, except that here n is viewed as a constant and s corresponds to work required for an insecure implementation (e.g., length of message in case of encryption, or size of witness verification circuit in the case of zero-knowledge). For instance, typical implementations of encryption have a constant communication overhead, but a poly(k) computation overhead.[7] In contrast, for typical

[7] Since for the purpose of concreteness we consider attackers that run in time 2^k, this requires to assume that the underlying hardness assumption is 2^{n^ε}-strong for some $\varepsilon > 0$.

implementations of zero-knowledge proofs or secure two-party computation protocols from the literature, both the communication and computation overhead are poly(k).

In [20] it was shown that, under plausible assumptions, various primitives including encryption, signatures, and secure two-party computation in the *semi-honest* model can be implemented with a constant computational overhead. For primitives such as encryption, commitment, hashing, and signatures, constructions with polylog(k) overhead relying on lattice-based assumptions or error-correcting codes were given in [25, 23, 1].

Obtaining similar results for zero-knowledge proofs and secure two-party computation against malicious parties is one of the main questions left open in [20]. Combining our main result with general transformations from [19, 21], we can make progress on the this question. Concretely, under standard cryptographic assumptions (e.g., assuming 2^{n^ε}-hardness of decoding random linear codes [1]), our main result yields zero-knowledge proofs for circuit satisfiability with 2^{-k} soundness error and simulation error, in which the amortized computational overhead per gate is only *polylogarithmic* in k, improving over the $\omega(k)$ overhead of the best previous protocols under *any* assumptions. Under stronger cryptographic assumptions, we obtain similar results for general secure two-party computation with simulation error 2^{-k}. Both types of protocols are unconditionally secure when implemented in the natural hybrid model (i.e., using ideal commitments in the case of zero-knowledge, or oblivious transfer in the case of secure computation). This implies that all "cryptographic" computations can be done during a preprocessing stage, before the actual inputs are known. See Section 7 for more details.

2 The Model

We consider the standard setting of *perfectly* UC-secure MPC [7], with guaranteed output delivery, over a synchronous network of secure point-to-point channels. Our protocols also employ a *broadcast* primitive, but since the number of broadcasts will be small they can be simulated over point-to-point channels without affecting the amortized overhead.

The players in our protocol are divided into three categories: *input clients* who contribute inputs, *output clients* who receive outputs, and n *servers* who help distribute the computation. To simplify the asymptotic complexity expressions, the number of clients is assumed to be $\mathcal{O}(n)$. Note that a player in the protocol is permitted to have one or more roles, and therefore this client-server model generalizes the usual model where every player has all three roles. The adversary is unbounded, active and adaptive, may corrupt up to t servers and any number of clients, where t is some constant fraction of n. (Concretely, one can use $t = n/8$ in the basic version of our protocol.)

We assume that the functionality f computed by the protocol is described by an arithmetic circuit C over a finite field $\mathbb{Z}_p$, where $p > 2n$. (In the case of boolean circuits, we can use the least p which satisfies this requirement. This results in an additional logarithmic communication overhead and polylogarithmic computation overhead.) The inputs and outputs of C may be arbitrarily partitioned between the input clients and the outputs clients, respectively.

It will be convenient to partition the gates into *layers*, such that each layer gets its input only from the previous layers and provides output to subsequent layers. This can be done by partitioning the gates according to the length of a longest path from an input. The *size* of the circuit C is written as $|C|$, and it is defined to be the number of gates plus the number of wires. Its *depth* is the length of the longest path from an input to an output, which is equal to the number of layers in the case of layered circuits.

Finally, since our efficiency goals are impossible to meet if each server needs to read an entire description of C, we separate the *protocol compilation* from the *protocol execution.* The protocol compiler takes a description of an arithmetic circuit C (whose inputs and outputs are partitioned between the clients) and a number of servers n and generates the "code" of each player in the protocol. When analyzing the complexity of the protocol we count only the cost of the protocol execution (combined over all players), but note that the protocol compilation can be performed with the same asymptotic computational cost as executing the protocol.

3 Packed Secret-Sharing

We will use the packed secret-sharing technique introduced by Franklin and Yung [13]. This is similar to standard Shamir secret-sharing [27] over $\mathbb{Z}_p$, but where a block of l different values $(x_1, .., x_l)$ are shared at once using a polynomial that evaluates to $x_1, ..., x_l$ in l distinct points. For privacy if t players are corrupted, the polynomial must be random of degree at most $d = t + l - 1$. We need that, from a set of n shares, one from each player, where at most t are incorrect, the correct block of secrets can be efficiently determined, even if the polynomial has degree up to $2d$. This will be the case if we set $t = n/8$ and $l = n/4$. Also, to have enough distinct evaluation points, we need that $p > 2n$. This is the same variant of packed secret sharing as was used in [12], which we refer to for further details.

Denote by $[x]_d$ a packed secret-sharing of the block x using a polynomial of degree at most d. Any vector of shares $\{s_1, \ldots, s_n\}$ among n servers is called *d-consistent* if the shares correctly match a degree at most d polynomial in the n first points and therefore uniquely defines a block of secrets.

Throughout the paper we will need many different protocols dealing with block sharings. Most notably we need verifiable secret-sharing for the input and reconstruction with error correction for the output. In Section 5 on page 9 we describe the known protocols that we will use.

4 Overview of the Protocol

Using packed secret sharing, it is straightforward to do secure addition or multiplication on l values in parallel, at the price of what a single operation would cost using normal secret sharing. This was already observed in [13] and can be used to compute the circuit C securely and efficiently if we arrange it such that every layer contains only one type of gates, and if we can produce sets of shared blocks $S_1, S_2, ..$ such that blocks in S_i contain the i'th input bit to the gates in a given layer, in some fixed order. We will call this a *correct line-up* for the given layer.

Demanding correct line-up is a problem, however: It implies that the values in the computation will have to be permuted between layers in arbitrary ways that depend on the concrete circuit. This is not easy to implement efficiently using packed secret sharing. We solve this problem by first constructing from C a new circuit C' that computes the same function but is more well-behaved. More precisely, we have

Lemma 1. *Given an arithmetic circuit C that is at least l gates wide, there is an efficient algorithm to transform it into another circuit C' with the following properties:*

1. *$C'(x) = C(x)$ for all inputs x.*
2. *Every layer contains only one type of gate.*
3. *If all values are stored in blocks using packed secret sharing where the block size l is a 2-power, the action between any two layers to achieve correct line-up is to permute the blocks and then in some blocks permute the elements within the block, where the same permutation applies to all blocks in the layer*[8]*. In the entire circuit, only $\log l$ different permutations are needed to handle permutations within blocks.*
4. *$|C'| = \mathcal{O}(|C| \log |C| + \text{depth}(C)^2 n \log^3 |C|)$, $\text{depth}(C') = \mathcal{O}(\log^2 |C| \text{depth}(C))$.*

The restriction on the width of the circuit is fairly insignificant, since n is generally small compared to the circuit size. Some of the layers in C' will not be a block wide, but since those layers also do not require a permutation, it will cause no problems.

We show in Appendix A how this construction works in detail. The basic idea is to handle the arbitrary permutations needed in C by inserting a small piece of circuitry that permutes the values as desired. This subcircuit can be made very regular using permutation networks as described by Waksman [28]. These are based on Beneš networks [5]. It follows from the construction that C' only contains addition, multiplication and H-gates, where H swaps two input values x, y or leaves them alone, depending on a control-bit c: $\text{H}(x, y, c) = (cx + (1 - c)y, cy + (1 - c)x)$.

Now, given the input arithmetic circuit C, we first transform it into C' as described in the lemma. We begin our actual computation by secret-sharing the

[8] In some cases, it may additionally be necessary to discard some blocks.

input values in blocks of size $l = \Theta(n)$, where l is a 2-power, and we then go through C' layer by layer, computing at each stage the output values from the layer in packed secret-shared form. Once we have the output from the last layer, shares of these are sent to the output clients for reconstruction.

Going into each layer we permute the shared blocks we have so far as needed to get correct line-up for the layer, and then do the computation required. The only non-trivial issue is how to permute the elements inside a shared block, i.e., how to compute $[\pi(x)]_d$ from $[x]_d$ for a permutation π. The idea is to first precompute pairs of the form $[r]_d, [\pi(r)]_d$ for random blocks r. We show below how to generate many such pairs using the same π at a small amortized cost per pair. This is sufficient, since by the above lemma, we only need a small number of different permutations. The idea then is to reveal $x + r$ to a single server, who then locally computes $\pi(x + r)$ and secret-shares it, proving in the process that $[\pi(x+r)]_d$ was correctly formed. This can be done efficiently if we do many blocks in parallel. Then, given $[\pi(x + r)]_d = [\pi(x) + \pi(r)]_d$ and $[\pi(r)]_d$, players subtract shares locally to get $[\pi(x)]_d$.

5 Subprotocols

In the previous sections, we have covered how to evaluate a circuit C by transforming it into C' and computing layer by layer. We begin this section by listing known protocols that we will be using for this. Subsequently we cover new protocols we propose.

Known protocols. From [12] we borrow the following protocols:

- `Share`(D, d): A dealer D computes shares of a block of l secrets using a degree d polynomial and sends a share to each player. Communication is $\mathcal{O}(n)$ and computation is $\mathcal{O}(n \log n)$.
- `Reco`(R, d): Assumes a block has been shared using a polynomial of degree at most d. All players send their shares of the block to R, who uses standard error correction techniques to reconstruct the block. Communication is $\mathcal{O}(n)$ and computation is $\mathcal{O}(n \log n)$.
- `RobustShare`(d): This protocol basically implements verifiable secret-sharing for one or more dealers who want to secret-share $\Theta(n)$ blocks each using polynomials of degree d. The functionality it implements, $\mathcal{F}_{RobustShare}$, is shown in Figure 1 on the next page.
- `RanDouSha`(d): Generates a vector of random blocks and a degree d and a degree $2d$ sharing of each block. More precisely, it implements the functionality shown in Figure 2 on page 11.
- `RobustReshare`(d, d'): Takes as input a number of secret shared blocks. For each input $[x]_d$ it outputs a new sharing $[x]_{d'}$. However, it does not keep x secret.
- `SemiRobustShare`(d): Same as `RobustShare`(d), but the adversary can cause some of the honest dealers to fail. However, during the entire global protocol, he can only make up to t honest dealers fail.

For every protocol above except for the first two, the communication complexity is $\mathcal{O}(\beta n^2)$, and the computational complexity is $\mathcal{O}(\beta n^2 \log n)$, for handling β groups of $\Theta(n)$ blocks. In both cases we must additionally pay $\mathcal{O}(n^2)$ per complaint. Complaints are handled as in our protocol `RandomPairs` in Figure 4 on page 12. Since each complaint results in at least one corrupted player being eliminated from the protocol, at most t complaints can occur in total.

Furthermore, there is a minimal cost for these protocols, since they are built to handle groups of blocks and not just single blocks at a time. `RobustShare` for example always costs at least as much as for $\beta = n$. For a protocol like `SemiRobustShare`, it is possible to handle $\beta = 1$ efficiently, but then we need to add $\mathcal{O}(n^3)$ for n broadcasts. However, as we will show later, these cases make no difference in our final complexity; for this we do not care about how well our protocols handle a small number of elements, we care about how they scale.

In [12] there is a proof of perfect privacy and correctness for each of the protocols above, but it was not proved there that `RanDouSha` and `RobustShare` implement the corresponding functionalities. A proof of this follows quite easily from correctness and privacy in the same way as in the proof for the protocol `RandomPairs`, which we present in detail below.

We define functionalities only for some of the protocols above. The rest are mentioned because we use them as parts of other protocols. The final UC proof in Appendix C only requires these parts to have perfect privacy and correctness.

1. Receive from all honest players the identities of the dealers and the number of blocks they want to share. Abort if the input is inconsistent. Receive also a set of input blocks to share from each honest dealer.
2. Send "Shares?" to the adversary together with the identities of the dealers and the number of blocks they want to share.
3. Receive from the adversary, for each block to be shared by an honest dealer, one share for each corrupted player (this should be thought of as the shares the adversary wants the corrupted players to receive). For each corrupt dealer, receive a polynomial of degree at most d.
4. For each block to be shared by an honest dealer, choose a random polynomial of degree at most d that is consistent with the block and the shares the adversary chose for the corrupted players. Compute and send the resulting shares to the honest players, and send the entire polynomial to the dealer.
5. For each block to be shared by a corrupt dealer, if the adversary sent a polynomial of correct degree, compute shares using this polynomial and send them to the players, otherwise tell all players that the dealer failed.

Fig. 1: The functionality $\mathcal{F}_{RobustShare}$

5.1 Permuting Elements within a Block

The basic idea behind our protocols for permuting the set of elements within each block for a vector of blocks was already explained in Section 4. To use this idea, we need to be able to produce pairs of sharings $[r]_d$, $[\pi(r)]_d$ for random r's, and a server needs to be able to secret-share blocks while showing that they were correctly permuted. First we present the protocol `RandomPairs` for producing the required permuted pairs. The protocol for resharing and proving is simpler and

1. Each honest player sends a natural number r to $\mathcal{F}_{double}$. If the honest players sent different values for r, $\mathcal{F}_{double}$ halts and outputs ABORT. Otherwise, send r and message "Shares?" to the adversary.
2. The adversary chooses $2r$ sets of shares for the corrupted players.
3. $\mathcal{F}_{double}$ chooses r random blocks $(x_1, \ldots, x_r)$ and creates random sharings $([x_1]_d, \ldots, [x_r]_d)$ and $([x_1]_{2d}, \ldots, [x_r]_{2d})$ such that they are consistent with the shares submitted by the adversary.
4. $\mathcal{F}_{double}$ outputs the resulting shares to the players.

Fig. 2: The functionality $\mathcal{F}_{double}$

yet very similar, and for that case we provide only a sketch. The protocol makes use of *hyperinvertible matrices*. A matrix is hyperinvertible if any intersection between k rows and k columns of the matrix is invertible. In [2], it is described how such a matrix can be constructed. We refer to [2] for the details, but it is important to note, as was also done in [12], that we may use the $\mathcal{O}(n \log n)$ FFT algorithms to multiply our hyperinvertible matrices onto vectors.

Creating Permuted Pairs The functionality $\mathcal{F}_{pairs}$ shown in Figure 3 details our requirements for the creation of permuted pairs. It works almost exactly like $\mathcal{F}_{double}$.

1. Each honest player sends a natural number r and a permutation π to $\mathcal{F}_{pairs}$. If the honest players sent different values for r or π, $\mathcal{F}_{pairs}$ halts and outputs ABORT. Otherwise, send r and message "Shares?" to the adversary
2. The adversary chooses $2r$ sets of shares for the corrupted players.
3. $\mathcal{F}_{pairs}$ chooses r random blocks $(x_1, \ldots, x_r)$ and chooses random sharings $([x_1], \ldots, [x_r])$ and $([\pi(x_1)], \ldots, [\pi(x_r)])$ such that they are consistent with the shares submitted by the adversary.
4. $\mathcal{F}_{pairs}$ outputs the chosen shares to the players.

Fig. 3: The functionality $\mathcal{F}_{pairs}$

An observation is needed before we present the protocol. Say we have some permutation π on l different elements, a vector of random blocks $(x_1, \ldots, x_n)$, and a vector of $y_i = \pi(x_i)$. Now suppose we apply some m by n matrix M and get the resulting vectors $(x'_1, \ldots, x'_m)$ and $(y'_1, \ldots, y'_m)$

Applying M to a vector of blocks corresponds to applying M to l different vectors at once. Permuting all blocks and then applying M clearly has the same result as applying M and then permuting the resulting blocks. More precisely, after applying M, $\pi(x'_i) = y'_i$.

We now present the protocol `RandomPairs`. It is run in parallel for all of the players with the restriction that $n - 3t = \Omega(n)$. The matrix M is hyperinvertible of dimension n by $n - 2t$, and X is hyperinvertible of dimension $n - 2t$ by $n - 2t$. The protocol is shown in Figure 4 on the next page.

Proposition 1. *The protocol `RandomPairs` securely realizes $\mathcal{F}_{pairs}$ in the UC model with perfect security against an active and adaptive adversary corrupting at most t players, where $n - 3t = \Omega(n)$. `RandomPairs` creates $\Theta(n^2)$ permuted*

1. **Sharing**
For each player D acting as dealer, and each group g of pairs to make, run the following in parallel:
   (a) D picks random blocks $(x_1, \dots, x_{n-2t})$ and $(y_1, \dots, y_{n-2t}) = (\pi(x_1), \dots, \pi(x_{n-2t}))$.
   (b) D shares the x_i and the y_i using protocol `Share`.
   (c) All players calculate
$$([x'_1], \dots, [x'_n]) = M([x_1], \dots, [x_{n-2t}])$$
$$([y'_1], \dots, [y'_n]) = M([y_1], \dots, [y_{n-2t}]).$$
   (d) For all i, all players P_j send their shares of $[x'_i]$ and $[y'_i]$ to P_i.
   (e) For all i, the dealer D sends all shares of $[x'_i]$ and $[y'_i]$ to P_i.
2. **Checking**
Initialize $\mathcal{C} = \emptyset$. This set will contain sets of conflicting players. Now for each player P_i in parallel:
   (a) P_i checks that the sharings received for x'_i and y'_i by all D for all groups are consistent, and that $y'_i = \pi(x'_i)$. For any pair (P_j, D) where this check went well, P_i also checks that he received the same shares from all pairs of dealers D and P_j. If all goes well, he broadcasts a 1, and a 0 is broadcast if one or more checks fail.
   (b) If P_i broadcast a 0, he now proceeds to broadcast the number of complaints he intends to make. The complaints are then handled as described in the following. If at any point P_i broadcasts badly formatted complaints or the same complaint more than once, P_i is immediately eliminated and ignored.
   (c) If a dealer D dealt inconsistent shares or the pairs were not correctly permuted, P_i broadcasts $(\textsc{conflict}, P_i, D)$. All players include the set $\{P_i, D\}$ in $\mathcal{C}$.
   (d) Otherwise, if P_i sees that it has received different shares from some P_j and D for a group g, it broadcasts $(\textsc{conflict}, D, P_j, g, share_D, share_{P_j}, w)$, where w indicates whether it is a conflict with shares of $[x'_i]$ or $[y'_i]$. Such conflicts are sent out for any relevant cases, but at most one conflict is sent out for any specific pair (D, P_j).
      i. If D finds that $share_D$ does not match what he sent to P_i, he broadcasts $(\textsc{conflict}, D, P_i)$, and it is recorded in $\mathcal{C}$.
      ii. If P_j finds that $share_{P_j}$ does not match what he sent to P_i, he broadcasts $(\textsc{conflict}, P_j, P_i)$. This is recorded in $\mathcal{C}$.
      iii. If neither D nor P_j broadcasts a conflict, the conflicting set $\{D, P_j\}$ is included in $\mathcal{C}$.
3. **Elimination**
All players now locally run the following elimination algorithm:
   (a) If there is a pair $\{P_i, P_j\} \in \mathcal{C}$ such that neither player has been eliminated so far, eliminate both players by removing them from the set $\mathcal{S}$ of player.
   (b) Keep all pairs $([x_i], [y_i])$ shared by non-eliminated players, throw away the rest.
4. **Postprocessing phase**
   (a) Reorder the players such that 1 through $n - 2t$ are non-eliminated.
   (b) (x_i^j, y_i^j) is the i'th pair of blocks known to the j'th player, for all non-eliminated j, and for each group.
   (c) Every player calculates
$$([a_i^1], \dots, [a_i^{n-2t}]) = X^{-1}([x_i^1], \dots, [x_i^{n-2t}])$$
$$([b_i^1], \dots, [b_i^{n-2t}]) = X^{-1}([y_i^1], \dots, [y_i^{n-2t}]).$$
   for all $i \in \{1, \dots, n - 3t\}$, and for each group.
   (d) For each group, the output is given by the pairs $([a_i^j], [b_i^j])$ for $i, j \in \{1, \dots, n - 3t\}$.

Fig. 4: Protocol `RandomPairs`

pairs at a time with a communication complexity of $\mathcal{O}(n^3)$, and a computational complexity of $\mathcal{O}(n^3 \log n)$. In both cases, we add $\mathcal{O}(n^2)$ per complaint.

Proof. The proof is divided into three parts. The first two are correctness and simulation, and together they prove security in the UC model. The last part deals with the complexity.

Correctness: To show correctness, we must prove that all generated pairs are consistently shared and correctly permuted. Consider the set of players $\mathcal{P}$. If we denote by $\mathcal{P}'$ the subset of non-eliminated players, we know that by the end of the elimination step, only sharings coming from players in $\mathcal{P}'$ will be used.

We know that for any dealer $D \in \mathcal{P}'$, there are no conflicts $\{P_i, D\} \in \mathcal{C}$ for any $P_i \in \mathcal{P}'$. If there were such conflicts, they would have caused the elimination of either D or P_i in the elimination phase. This means that all honest players in $\mathcal{P}'$ agree that the shares they have received from dealers $D \in \mathcal{P}'$ are consistent and represent correctly permuted pairs, and furthermore these shares agree with all shares received from $P_j \in \mathcal{P}'$.

Now consider all non-eliminated honest players. We know that at least for every two players eliminated, one of the players must have been corrupted. Therefore, we have at least $n - 2t$ honest players in $\mathcal{P}'$. Now select exactly $n - 2t$ of those and form the set H. It can be seen then that

$$([x'_i])_{P_i \in H} = M_H([x_i])_{1 \le i \le n-2t},$$

where M_H is a matrix containing only the rows of M with indices corresponding to the players in H. Since M_H is a square submatrix of a hyperinvertible matrix, it must be invertible. This means that

$$([x_i])_{1 \le i \le n-2t} = M_H^{-1}([x'_i])_{P_i \in H}.$$

The calculations above also hold for the y_i. We know that all pairs (x'_i, y'_i) where $P_i \in H$ are guaranteed to be consistently shared and correctly permuted. Applying the linear transformation M_H^{-1} preserves this property, and so we know that all of the original pairs (x_i, y_i) must be correct as long as the dealer is in $\mathcal{P}'$, but these are exactly the pairs we keep after the elimination phase.

Following the elimination phase, new pairs are created by applying yet another linear transformation. As before, linear transformations preserve the consistency of sharings and the property that pairs are correctly permuted, and thus correctness is ensured.

Simulation: To prove UC security, we must also show that we can construct a simulator $\mathcal{S}$ such that any environment $\mathcal{Z}$ cannot distinguish between the real world where it communicates with the adversary $\mathcal{A}$ and the ideal world where it communicates with $\mathcal{S}$. We do this by first proving perfect privacy (i.e. we prove that the adversary's view is independent of the secrets shared), and then we show how to use this and correctness to build a simulator.

For perfect privacy, all values seen by the adversary should be independent of the secret, which in this case is the set of output pairs. Throughout the protocol, $\mathcal{A}$ learns openings of sharings from honest players, and it knows its own sharings as well. It is these values that should be independent of the output. More specifically, we need only examine sharings by non-eliminated players, since the others are not used to create the output.

First, we prove that the sharings distributed by non-eliminated honest players are independent of the sharings opened towards $\mathcal{A}$. For any honest dealer and any group, let $I = \{1, \ldots, n-3t\}$ be the indices of the initial blocks and R those of the remaining blocks. Now choose a set C of size t that contains all indices of the corrupted players. The corrupted players now know openings of

$$([x'_i])_{i\in C} = M_C^I([x_i])_{i\in I} + M_C^R([x_i])_{i\in R},$$

where M_A^B means the matrix M restricted to rows in A and columns in B. A similar equation holds for the y'_i. Since $|C| = |R|$, there is exactly one choice of blocks in R that matches what the adversary can see for any set of blocks in I. In other words, the blocks opened to $\mathcal{A}$ are independent of the ones dealt by the honest dealers.

The final output blocks are created using the sharings from all non-eliminated servers, possibly including some corrupted servers. Therefore, we must also prove that the final outputs are independent of sharings from non-eliminated corrupt players. For the a_i^j and any group (the proof is the same for the b_i^j), let $I = \{1, \ldots, n-3t\}$ be the set of the initial $n-3t$ indices, R the subsequent t, and C a set of size t containing the indices of all non-eliminated corrupted players (fill the rest of C with other players if there are less than t). The adversary knows x_i^j for all $j \in R$, so the sharings known to $\mathcal{A}$ are

$$([x_i^j])_{j\in C} = X_C^I([a_i^j])_{j\in I} + X_C^R([a_i^j])_{j\in R},$$

for all i. Since $|C| = |R|$, and since X is hyperinvertible, X_C^R is invertible. Therefore, for any set of blocks known to the adversary, there is exactly one choice of blocks $[a_i^j]_{j\in R}$ not output for any set of output blocks. In other words, the blocks dealt by $\mathcal{A}$ are independent of the output blocks. This concludes our proof of privacy.

We can now show how to construct a simulator $\mathcal{S}$. It simply runs dummy versions of the honest players and lets the execute the protocol with $\mathcal{A}$. We know that any values seen by $\mathcal{A}$ during the protocol are independent of the actual secrets shared, so the values generated by $\mathcal{S}$ towards $\mathcal{A}$ must be correctly distributed. When the protocol is done, the shares for corrupted players generated by the simulated run is fed into $\mathcal{F}_{pairs}$. The functionality now chooses the output sharing so to match these values, i.e. the honest players obtain shares that are consistent with a set of correctly distributed secrets and with the shares held by the adversary. By correctness of the protocol, this matches exactly the distribution of the output of a real protocol run.

The very last part of the proof is to deal with adaptive corruptions. First of all, if an honest player is corrupted during the protocol run but before we receive outputs from $\mathcal{F}_{pairs}$, we may simply open up one of the dummy parties to the adversary and continue from there. The only difficult part is if a server is corrupted after the output sharings have been chosen, because in that case the view of a dummy party does not match the output sharings. To adjust the view of a dummy party to the actual output shares of $\mathcal{F}_{pairs}$, we examine how these shares are constructed. We start by adjusting the shares of the $[a_i^j]$ for $j \in I$ (all of the following works in the same way for the b_i^j). The adversary knows the full sharings of

$$([x_i^j])_{j\in C} = X_C^I([a_i^j])_{j\in I} + X_C^R([a_i^j])_{j\in R},$$

so for those we simply pick the correct shares of $[a_i^j]$ for $j \in R$ to match the adjusted shares for $j \in I$. Now calculate $([x_i^j])_j = X([a_i^j])_j$ to find the remaining shares owned by the newly corrupted player. This of course means that the other dummy parties have to adjust their sharings from this point. The last problem is x_i^j created by this player. We can easily adjust its sharing of those values to match what we need, but it also needs to match the values opened to the adversary during the sharing of them. Luckily, we already know that this is simply a matter of adjusting the randomness used in the sharing.

Complexity: We now examine the complexity of the protocol. Going through each step of the protocol and remembering that every server is a dealer, we see that each step has a maximum communication complexity of $\mathcal{O}(n^3)$. Clearly this is also the total communication complexity. The computational complexity is $\mathcal{O}(n^3 \log n)$ plus the cost of each complaint, since in the slowest step, every server must check the consistency of $\Theta(n)$ sharings by interpolation, which can be done by using $\mathcal{O}(n \log n)$ FFT. Every complaint adds $\mathcal{O}(n^2)$ to both complexities for the broadcast.

Permuting Elements within Blocks The next subprotocol `PermuteWithinBlocks`, and it is shown in Figure 5 takes as input the shares of blocks $([x_1], \ldots, [x_n])$, a vector of random pairs $(([s_1], [\pi(s_1)]), \ldots, ([s_n], [\pi(s_n)]))$, and the permutation π. It outputs shares of new sharings $([\pi(x_1)], \ldots, [\pi(x_n)])$. For this protocol, we prove correctness and privacy here, and use these properties in the simulation proof for the main protocol.

1. For every n input blocks, we do the following.
2. The servers locally compute $[x_i + s_i] = [x_i] + [s_i], \quad 1 \le i \le n$.
3. The servers select the non-eliminated server j that has least recently been chosen in this way and invoke `Reco` to reconstruct the $[x_i + s_i]$ to j.
4. Server j locally computes $\pi(x_i)$ for all i.
5. Server i uses protocol `Permuted` to share $[\pi(x_i + s_i)]$ for all i, proving in the process that it has been consistently shared and permuted. If `Permuted` outputs FAIL, return to step 3 (see description of `Permuted` in the text).
6. The players locally compute $[\pi(x_i)] = [\pi(x_i + s_i)] - [\pi(s_i)], \quad 1 \le i \le n$.

Fig. 5: Protocol `PermuteWithinBlocks`

Note that we only run the protocol for n blocks at a time to limit the cost of Permuted failing. For efficiency, we must work on at least n blocks at a time, so this is the natural choice. The protocol Permuted that was mentioned above is an adaptation of RandomPairs: there is only one dealer, server j. Rather than sharing both the x_i's and $\pi(x_i)$'s, the server shares only $\pi(x_i)$, since servers already have shares of the x_i's in question. However some extra random x_i's are added to ensure privacy (recall that RandomPairs requires extra random blocks that will not be output). Otherwise, we do exactly the same as in RandomPairs but if FAIL if server j is eliminated we stop immediately and output fail. The postprocessing phase is omitted, since there is only a single dealer who is allowed to know the (masked) secret.

It is perfectly private and correct by for the same reason that PermutedPairs is. As for the complexities, we consider permuting β groups of $\Theta(n)$ blocks (i.e. we permute $\Theta(\beta n)$ blocks). Ignoring broadcasts for a moment, we see that communication is at its most expensive when initially sharing, which costs $\mathcal{O}(\beta n^2)$. The most expensive computational step is still checking, which costs $\mathcal{O}(\beta n^2 \log n)$. For both computation and communication, we need to add $\mathcal{O}(n^3)$ in broadcast costs in both cases (regardless of the number of groups) and a further $\mathcal{O}(n^2)$ per complaint.

For the protocol PermuteWithinBlocks, it is clear that we still have privacy, since random blocks are added before opening. Correctness is trivial from the construction. As for the complexities, the most expensive step is Permuted. So both computational and communication complexities are as above, with the exception that the cost is multiplied by the number of times we fail and have to rerun Permuted. Since each failure results in at least one corrupt player being eliminated, the worst case is having to rerun t times.

5.2 Multiplications

As explained earlier, our circuit consists of only addition, multiplication and H-gates, where $\text{H}(x, y, c) = (cx + (1 - c)y, cy + (1 - c)x)$. Since addition is trivially done by local computation, it is sufficient to explain how to handle multiplications. In order to do this, we need the protocol RobustReshare; as mentioned above it coverts a vector of blocks from being shared with degree d_1 to shares with degree d_2. In a nutshell, it publicly reconstructs the values and then reshares them. Assume that we are given shared blocks $[x]_d, [y]_d$ with degree d and sharings $[r]_d, [r]_{2d}$ of the same r but with degree d and $2d$. The protocol Multiply then works as shown in Figure 6.

1. For every pair of blocks x, y to multiply, we assume sharings $[r]_d, [r]_{2d}$ are available. The servers locally compute $[xy + r]_{2d} = [x]_d[y]_d + [r]_{2d}$.
2. RobustReshare is run to obtain $[xy + r]_d$ for all x, y.
3. For every x, y the servers locally compute $[xy]_d = [xy + r]_d - [r]_d$.

Fig. 6: Protocol Multiply

The pairs $[r]_d, [r]_{2d}$ we need can be generated using `RanDouSha` mentioned above. Correctness follows from correctness of `RobustReshare`. Privacy follows from privacy of `RanDouSha` since we can then assume the r is uniformly random from the adversary's point of view. The complexity is clearly dominated by `RobustReshare` whose complexity was covered earlier.

6 The Main Protocol

The final protocol is described in Figure 8, while the functionality realized is in Figure 7. This leads to:

1. The input clients send their inputs $(x_1, \ldots, x_r)$ to $\mathcal{F}_C$.
2. F_C distributes $(y_1, \ldots, y_t) = C(x_1, \ldots, x_r)$ to the intended output clients.

Fig. 7: The functionality $\mathcal{F}_C$ for the circuit C

Preprocessing: Transform C into C'.
Step 0**:** Input clients invoke the functionality $\mathcal{F}_{RobustShare}$ to share their inputs to the servers. The servers invoke $\mathcal{F}_{pairs}$ and $\mathcal{F}_{double}$ to create a set P_i of pairs and a set DS_i of double sharings for every layer $1 \leq i \leq d$ of C', where $d = \text{depth}(C')$.
Step i**:** For $1 \leq i \leq d$, we have from the previous layers the set I_i of inputs for this layer as well as pairs and double sharings P_i and DS_i for this layer. Layer i is evaluated on I_i by the servers through local computations and a constant number of calls to `Multiply`.
The outputs of the layer may need to be permuted. If the blocks are to be permuted, they are permuted by local computation. If the elements within the blocks need to be permuted, the servers invoke `PermuteWithinBlocks` on the blocks in question.
Step $d+1$**:** The servers open sharings to the relevant output clients using `Reco`.

Fig. 8: Protocol `EvalCircuit`

Theorem 1. *There exists* $0 < \delta < 1/3$ *such that given* n *servers and an arithmetic circuit* C *that is at least* $\Omega(n)$ *gates wide, the protocol* `EvalCircuit` *realizes* $\mathcal{F}_C$ *with perfect security in the UC model against an active and adaptive adversary corrupting up to* $t < \delta n$ *servers.*

The total communication complexity is

$$\mathcal{O}(\log n \log |C| \cdot |C|) + \text{poly}(n, \log |C|) \cdot \text{depth}(C)^2,$$

while the total computational complexity is

$$\mathcal{O}(\log^2 n \log |C| \cdot |C|) + \text{poly}(n, \log |C|) \cdot \text{depth}(C)^2.$$

The actual threshold in Theorem 1 is quite far from the optimal $n/3$ bound. To improve on this, we may use the *player virtualization* technique by Bracha [6] in the same way it was used in [12], to which we refer for the details of the construction. The basic idea is to construct virtual servers that run our protocol. To simulate each virtual server, a subset of the servers run a less efficient protocol, the inner protocol, that has a high threshold.

The difference from [12] is that here we are interested in perfect security. Therefore we need an inner protocol that also has perfect security. To this end, we can employ the BGW protocol [3]. Since it has threshold $n/3$, the construction from [12] gives us a threshold of $n/3 - \varepsilon$ for sufficiently large n, where $\varepsilon > 0$ may be chosen arbitrarily.

The construction increases both the computational and communication complexities to be the sum of the previous computational and communication complexities. Therefore, the new bound for both will be

$$\mathcal{O}(\log^2 n \log |C| \cdot |C| + \mathrm{poly}(n, \log |C|) \cdot \mathrm{depth}(C)^2).$$

The proof of Theorem 1 on the previous page is given in Appendix C.

In Appendix B we prove Corollary 1, which is a reduction in the complexity in some cases, namely when the depth is large and there are many pairs of layers (i, j) in C such that there is a wire from i to j.

Corollary 1. *With the modification of Appendix B, the complexities of Theorem 1 can be altered to*

$$\mathcal{O}(\log \mathrm{depth}(C) \log n \log |C| \cdot |C|) + \mathrm{poly}(n, \log |C|) \cdot \mathrm{depth}(C) \log \mathrm{depth}(C)$$

for communication and

$$\mathcal{O}(\log \mathrm{depth}(C) \log^2 n \log |C| \cdot |C|) + \mathrm{poly}(n, \log |C|) \cdot \mathrm{depth}(C) \log \mathrm{depth}(C)$$

for computation.

7 Application to Two-Party Cryptography

In this section we sketch the application of our main result to reducing the computational overhead of zero-knowledge proofs and secure two-party computation.

In [19] it is shown how to obtain a zero-knowledge proof for the satisfiability of a circuit C from any MPC protocol for n servers in which one client ("the prover") has an input w and another client ("the verifier") should output $C'(w)$, where C' is a constant-depth circuit of roughly the same size as C which is easily determined by C. If the MPC protocol is adaptively secure against an active adversary who corrupts the prover and a constant fraction of the servers, the resulting zero-knowledge protocol will have soundness error of $2^{-\Omega(n)}$ plus the correctness error of the MPC protocol. The simulation error corresponds to that of the MPC protocol. The efficiency of the zero-knowledge protocol is essentially the same as that of the MPC protocol, excluding the cost of n commitments to strings whose total size is roughly the communication complexity of the MPC protocol.

The above transformation was combined with the MPC techniques from [11, 9] to yield zero-knowledge proofs with a *constant* communication overhead. However, to guarantee soundness error of 2^{-k}, the *computational overhead* of this protocol must be $\Omega(k)$, even if ideal commitments are used. Plugging in our main result, we obtain a perfect zero-knowledge protocol in the *commitment-hybrid model* (i.e., using ideal commitments) in which both the communication and computation overhead are polylogarithmic in k. As a side benefit, the perfect security of our protocol allows for a simpler and more round-efficient transformation into a zero-knowledge proof protocol (see [19], Section 4).

To implement the commitment-hybrid model, we can use the constant overhead constructions from [20] or the polylog-overhead constructions from [1]. The latter have the advantage of relying on fairly standard cryptographic assumptions, related to the intractability of decoding random linear codes or learning with errors.

We note that in the case of zero-knowledge *arguments* (with computational soundness), it is possible to combine the PCP-based approach of [22, 24] for efficient arguments with state of the art PCP constructions [4] and efficient lattice-based constructions of collision-resistant hash functions [25, 23] to get alternative constructions with polylogarithmic computational overhead. However, other than offering only computational soundness, the resulting protocol requires stronger assumptions, inherits the complex and seemingly impractical nature of current PCP constructions, and does not allow to eliminate the need for cryptography using preprocessing.

We finally note that similar results can be obtained in the more general context of secure two-party computation. One approach to obtain these results is to apply the GMW-compiler [16], with the efficient zero-knowledge proofs described above, to a constant-overhead protocol for the semi-honest model from [20]. The latter protocol relies on the existence of a pseudorandom generator stretching n bits to n^2 bits in which each bit of the output depends on just a constant number of input bits — a plausible but nonstandard assumption. Another approach, which can offers *unconditional* security in the OT-hybrid model, is to instantiate the protocol compiler from [21] with our main protocol as the “outer protocol”.

8 On the Relevance of Gentry's Scheme

The recent breakthrough of Gentry [15], suggesting the first plausible candidate for a fully homomorphic encryption scheme, has a great impact on the theoretical efficiency of MPC. By distributing the key generation and decryption of Gentry's scheme between the n players, it is possible to obtain general constant-round MPC protocols whose communication complexity only depends on n and the length of the inputs and outputs of C rather than the size of C. We note, however, that this protocol can only provide *computational* security (under a non-standard assumption) and, perhaps more importantly, its computational overhead involves a large polynomial in the security parameter. The high computational cost seems to make Gentry's scheme, in its current form, too inefficient for practical purposes. Finally, for circuits whose output length is not much smaller than their size (as in the case of performing a large number of simple computations), even the communication overhead of this protocol becomes a large polynomial in k and n. In contrast, our protocol has the same overhead even in this case. In light of the above, it seems fair to conclude that Gentry's result has limited relevance to the results of the present work from both a theoretical and from a practical point of view.

References

1. Benny Applebaum, David Cash, Chris Peikert, and Amit Sahai. Fast cryptographic primitives and circular-secure encryption based on hard learning problems. In *CRYPTO '09: Proceedings of the 29th Annual International Cryptology Conference on Advances in Cryptology*, pages 595–618, Berlin, Heidelberg, 2009. Springer-Verlag.
2. Zuzana Beerliová-Trubíniová and Martin Hirt. Perfectly-secure MPC with linear communication complexity. In Ran Canetti, editor, *TCC*, volume 4948 of *Lecture Notes in Computer Science*, pages 213–230. Springer, 2008.
3. Michael Ben-Or, Shafi Goldwasser, and Avi Wigderson. Completeness theorems for non-cryptographic fault-tolerant distributed computation (extended abstract). In *STOC*, pages 1–10. ACM, 1988.
4. Eli Ben-Sasson, Oded Goldreich, Prahladh Harsha, Madhu Sudan, and Salil P. Vadhan. Short pcps verifiable in polylogarithmic time. In *IEEE Conference on Computational Complexity*, pages 120–134. IEEE Computer Society, 2005.
5. V.E. Benes. Optimal rearrangable multistage connecting networks. *The Bell System Technical Journal*, 43:1641–1656, 1964.
6. Gabriel Bracha. An O(log n) expected rounds randomized byzantine generals protocol. *J. ACM*, 34(4):910–920, 1987.
7. R. Canetti. Universally composable security: A new paradigm for cryptographic protocols. In *FOCS '01: Proceedings of the 42nd IEEE symposium on Foundations of Computer Science*, pages 136–145, Washington, DC, USA, 2001. IEEE Computer Society.
8. David Chaum, Claude Crépeau, and Ivan Damgard. Multiparty unconditionally secure protocols. In *STOC '88: Proceedings of the twentieth annual ACM symposium on Theory of computing*, pages 11–19, New York, NY, USA, 1988. ACM.
9. Hao Chen and Ronald Cramer. Algebraic geometric secret sharing schemes and secure multi-party computations over small fields. In Cynthia Dwork, editor, *CRYPTO*, volume 4117 of *Lecture Notes in Computer Science*, pages 521–536. Springer, 2006.
10. Ronald Cramer, Ivan Damgård, and Jesper Buus Nielsen. Multiparty computation from threshold homomorphic encryption. In *EUROCRYPT '01: Proceedings of the International Conference on the Theory and Application of Cryptographic Techniques*, pages 280–299, London, UK, 2001. Springer-Verlag.
11. Ivan Damgård and Yuval Ishai. Scalable secure multiparty computation. In Cynthia Dwork, editor, *CRYPTO*, volume 4117 of *Lecture Notes in Computer Science*, pages 501–520. Springer, 2006.
12. Ivan Damgård, Yuval Ishai, Mikkel Krøigaard, Jesper Buus Nielsen, and Adam Smith. Scalable multiparty computation with nearly optimal work and resilience.

In *CRYPTO 2008: Proceedings of the 28th Annual conference on Cryptology*, pages 241–261, Berlin, Heidelberg, 2008. Springer-Verlag.

13. Matthew K. Franklin and Moti Yung. Communication complexity of secure computation (extended abstract). In *STOC*, pages 699–710. ACM, 1992.
14. Rosario Gennaro, Michael O. Rabin, and Tal Rabin. Simplified VSS and fast-track multiparty computations with applications to threshold cryptography. In *PODC '98: Proceedings of the seventeenth annual ACM symposium on Principles of distributed computing*, pages 101–111, New York, NY, USA, 1998. ACM.
15. Craig Gentry. Fully homomorphic encryption using ideal lattices. In *STOC '09: Proceedings of the 41st annual ACM symposium on Theory of computing*, pages 169–178, New York, NY, USA, 2009. ACM.
16. Oded Goldreich, Silvio Micali, and Avi Wigderson. How to play any mental game or a completeness theorem for protocols with honest majority. In *STOC*, pages 218–229. ACM, 1987.
17. Martin Hirt and Ueli M. Maurer. Player simulation and general adversary structures in perfect multiparty computation. *J. Cryptology*, 13(1):31–60, 2000.
18. Martin Hirt, Ueli M. Maurer, and Bartosz Przydatek. Efficient secure multi-party computation. In *ASIACRYPT '00: Proceedings of the 6th International Conference on the Theory and Application of Cryptology and Information Security*, pages 143–161, London, UK, 2000. Springer-Verlag.
19. Yuval Ishai, Eyal Kushilevitz, Rafail Ostrovsky, and Amit Sahai. Zero-knowledge from secure multiparty computation. In *STOC '07: Proceedings of the thirty-ninth annual ACM symposium on Theory of computing*, pages 21–30, New York, NY, USA, 2007. ACM.
20. Yuval Ishai, Eyal Kushilevitz, Rafail Ostrovsky, and Amit Sahai. Cryptography with constant computational overhead. In *STOC '08: Proceedings of the 40th annual ACM symposium on Theory of computing*, pages 433–442, New York, NY, USA, 2008. ACM.
21. Yuval Ishai, Manoj Prabhakaran, and Amit Sahai. Founding cryptography on oblivious transfer — efficiently. In *CRYPTO 2008: Proceedings of the 28th Annual conference on Cryptology*, pages 572–591, Berlin, Heidelberg, 2008. Springer-Verlag.
22. Joe Kilian. A note on efficient zero-knowledge proofs and arguments (extended abstract). In *STOC '92: Proceedings of the twenty-fourth annual ACM symposium on Theory of computing*, pages 723–732, New York, NY, USA, 1992. ACM.
23. Vadim Lyubashevsky and Daniele Micciancio. Generalized compact knapsacks are collision resistant. In Michele Bugliesi, Bart Preneel, Vladimiro Sassone, and Ingo Wegener, editors, *ICALP (2)*, volume 4052 of *Lecture Notes in Computer Science*, pages 144–155. Springer, 2006.
24. Silvio Micali. Computationally sound proofs. *SIAM J. Comput.*, 30(4):1253–1298, 2000.
25. Chris Peikert and Alon Rosen. Efficient collision-resistant hashing from worst-case assumptions on cyclic lattices. In Shai Halevi and Tal Rabin, editors, *TCC*, volume 3876 of *Lecture Notes in Computer Science*, pages 145–166. Springer, 2006.
26. T. Rabin and M. Ben-Or. Verifiable secret sharing and multiparty protocols with honest majority. In *STOC '89: Proceedings of the twenty-first annual ACM symposium on Theory of computing*, pages 73–85, New York, NY, USA, 1989. ACM.
27. Adi Shamir. How to share a secret. *Commun. ACM*, 22(11):612–613, 1979.
28. Abraham Waksman. A permutation network. *J. ACM*, 15(1):159–163, 1968.

A Transforming the Circuit

In this section we show how to transform arbitrary arithmetic circuits into the type we need for our protocol. We start by repeating the lemma that describes the requirements.

Lemma 2. *Given an arithmetic circuit C that is at least l gates wide, there is an efficient algorithm to transform it into another circuit C' with the following properties:*

1. *$C'(x) = C(x)$ for all inputs x.*
2. *Every layer contains only one type of gate.*
3. *If all values are stored in blocks using packed secret sharing where the block size l is a 2-power, the action between any two layers to achieve correct line-up is to permute the blocks and then in some blocks permute the elements within the block, where the same permutation applies to all blocks in the layer[9]. In the entire circuit, only $\log l$ different permutations are needed to handle permutations within blocks.*
4. *$|C'| = \mathcal{O}(|C| \log |C| + \mathrm{depth}(C)^2 n \log^3 |C|)$, $\mathrm{depth}(C') = \mathcal{O}(\log^2 |C| \mathrm{depth}(C))$.*

Before we describe the transformation, we give a description of our main tool, the Beneš network. Note that we are essentially building the same permutation networks as Waksman [28], but we present the construction here in full for completeness.

A.1 Beneš Networks

A Beneš network [5] is a type of graph that can model all permutations π on $\{1, \ldots, m\}$ for $m = 2^w$. Note that we intend to implement such a graph using gates as nodes and wires as edges.

We describe Beneš networks recursively in terms of layers of nodes. The base case ($m = 1$) is just a single node. Now for any $m > 1$, a Beneš network consists of two networks of size $m/2$ on top of each other, yielding a graph with layers of size m. Additionally, there is always an input layer to the left and an output layer to the right.

Each node in an input/output layer is connected to the next/previous layer using two edges – ones goes horizontally and the other goes diagonally either $m/2$ nodes up or $m/2$ nodes down.

Figure 9 on the facing page illustrates the layout of a Beneš network for $m = 8$, and it is easy to see how this generalizes to other values of m. As described above, the middle of the graph is clearly seen to be two networks of size $m = 4$.

In the following, we shall need some facts about Beneš networks, and the most crucial observation is stated in the following proposition:

[9] In some cases, it may additionally be necessary to discard some blocks.

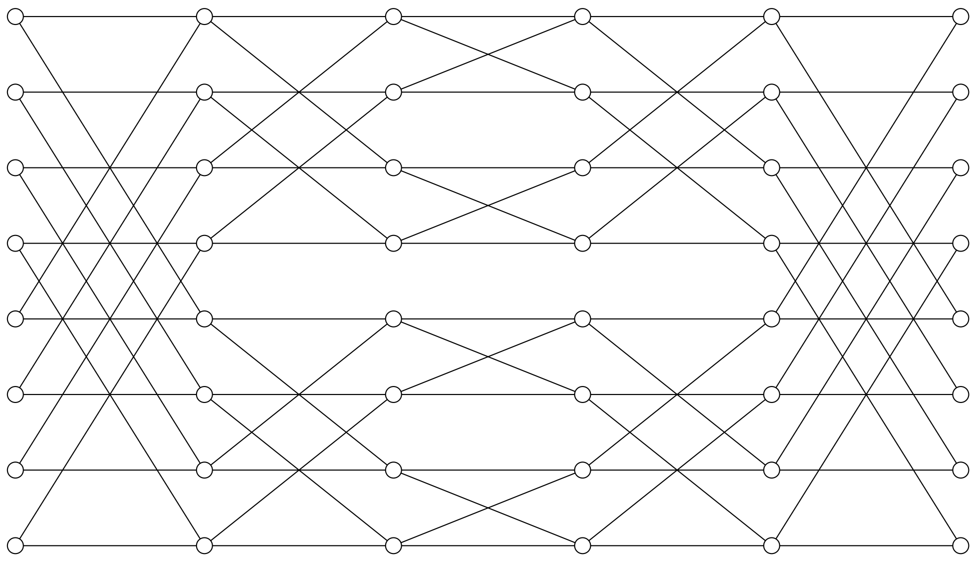

Fig. 9: A Beneš network

Proposition 2. *Given a Beneš network for some size m, we may model any permutation π on $2m$ elements by inputting two elements per node on the left and routing them, one element per edge, through the network to two elements per output node on the right.*

Proof. The proof is along the same lines as the one by Waksman [28]. It is a proof by induction. The basis is trivial. The inductive step assumes that our network consists of the input layer, the output layer, and the two half-size networks in the middle (the upper and lower subnetworks).

We now pick the first element on the upper left, x_1. This is routed to the upper subnetwork, and from there to the relevant node where $\pi(x_1)$ resides. In that node we find another element, $\pi(x_i)$. This is routed through the only unused edge back to the lower subnetwork and back to the node containing x_i. If the node containing x_i also contains another used element, it must be x_1, and in that case we know that since we always return on the lower subnetwork, we took the unused edge to the node. At that point, we have found a loop. If instead the node contains some other x_j, we repeat the procedure as for x_1 until we find a loop back to x_1.

This procedure is repeated until all elements have been routed. If we start a loop on the lower half, we just mirror the situation above and go through the lower subnetwork on the way to the right and the upper when returning.

The proposition tells us that we may model any permutation with a correctly sized Beneš network. Before moving on, we also state a short result on the size of a Beneš network.

Proposition 3. *A Beneš network with m starting and ending nodes has exactly $2\log m$ layers and $2m\log m$ nodes.*

Proof. This follows trivially from the construction.

A.2 Permutation Circuits

Assume that the block size is l and that we are given B blocks of elements in one vector $X = (x_1, \ldots, x_{Bl})$. We now wish to permute this vector according to some permutation π. This is needed to put the outputs from a layer into the correct order for the other layers. However, since the permutation is intended to be computed using packed secret-sharing, where the input vector is divided between multiple shared blocks, we want to break the computation of $\pi(X)$ into simpler parts.

The idea is to make a circuit that takes X as input and outputs $\pi(X)$. This circuit has the same width everywhere, is layered (i.e. outputs go directly from one layer to the subsequent one), and the outputs of a layer must be permuted before being given to the next one. However, this time the permutation is not arbitrary, and we explain in detail how to compute the circuit in Section A.3.

To construct a permutation circuit for $2m$ elements, we first construct a Beneš network of width 2^m. Then we transform it into a circuit by replacing every edge with a wire and every node with a special gate we call H. Obviously H must have at least 2 inputs and 2 outputs for the wires to fit. However, it is also given a third input that is a control bit. If the control bit is c, we define H as follows:

$$\text{H}(x, y, c) = (cx + (1-c)y, cy + (1-c)x).$$

In other words, if the control bit is 0, the inputs are swapped, otherwise they are left unchanged.

The circuit may be programmed to follow the correct routing for any permutation merely by setting the control bits correctly. As for how this circuit is used, the inputs to permute are simply connected to the input gates in the first (left) layer, and the permuted outputs are taken from the last (right) layer. The control bits are publicly known inputs.

A.3 Evaluating Permutation Circuits

Permutation circuits are quite different from the overall circuit. The action for each layer is as always to compute a single type of gates on a vector of blocks, however the actions between the layers are of a completely different nature. Although the actions to perform between layers are not entirely trivial, in some sense they are simpler and more elegant than before, because the input to a layer is always a very simple function of the output from the previous layer and that layer only. In the following we will show how to compute permutation circuits in detail.

Denote the input blocks for layer i by $I_{i,j,b}$. j is the gate input number and b the block number. So $I_{i,2,4}$ would be the fourth block of second inputs for the gates in layer i. We define $O_{i,j,b}$ in the same way, only for outputs from layer i. Note that the control bits $I_{i,3,b}$ are given as default sharings of publicly known values.

We now consider the i'th layer of the circuit (not the final layer). Assume without loss of generality that the first wire out of each gate is always directed horizontally to the next layer. In other words:

$$I_{i+1,1,b} = O_{i,1,b}.$$

The second wires are always connected diagonally with the next layer. Keeping in mind the construction of Beneš networks and the fact that the block size l is a power of two, a block for one of the second channels can be in two situations: either the whole block is connected diagonally, or the block consists of two halves that switch places. Again due to the nature of Beneš networks, we can clearly only have one of the two situations for the whole layer.

In the first situation where whole blocks are switched over diagonally, there is some permutation π such that

$$(I_{i+1,2,b})_{1 \leq b \leq B} = \pi_i((O_{i,2,b})_{1 \leq b \leq B}),$$

where B is the maximal block number. In the second situation, there is a permutation π_i such that

$$I_{i+1,2,b} = \pi_i(O_{i,2,b}).$$

Note that because we use one permutation per layer of the circuit, and there are at most $\log n$ layers, we can get at most $\log n$ different permutations.

This also explains the $\log l$ bound on the total number of different permutations that can happen within blocks. Because l is a power of 2, we reach a subnetwork size of exactly l at some point. When we do, we need to permute within blocks. Because the size of halved for each subnetwork, there are only $\log l$ ways we can permute within blocks.

As mentioned earlier, we still need to perform permutations to evaluate a permutation circuit. However, the permutations are clearly either permutations of entire blocks or they are permutations of elements within a single block.

A.4 Transforming the Circuit

At this point we have all the tools to completely describe the transformation of C into C'. We shall deal with this in three steps, and at each of them we examine the impact on circuit size and depth.

Note before we begin that it is likely that the number of inputs or outputs for a layer is not a multiple of the block size. We simply insert dummy gates to fill out the blocks. Because the circuit is at least a block wide, this does not affect the asymptotic circuit dimensions.

A.5 One Gate Type per Layer

The first requirement is that any layer contains only one type of gates. We ignore fanout/fanin problem for now and divide gates into two types, MUL and ADD. Dividing the original circuit C into a minimal amount of layers first, we now split each layer into two layers; one layer consists of all the MUL gates, and the other of all the ADD gates. Every layer now only consists of one type of gates, but the depth has doubled. We call the circuit in this state C_1. Before we go any further, we write down explicitly what has happened to the circuit dimensions. The size remains the same:

$$|C_1| = |C|,$$

and the depth is doubled:

$$\text{depth}(C_1) = 2 \cdot \text{depth}(C).$$

Note that although this may make layers less than a block wide, for every layer that is too narrow, there is another that is at last half a block wide, which is enough to pay for the narrow layer.

A.6 Handling Variable Fanin and Fanout

In C_1, the gates in each layer may have the same type, but they do not necessarily have the same fanout/fanin and can therefore not be computed easily in parallel. To address this problem, we limit our choice of gates to two-input, one-output ADD and MUL gates, as well as a two-input, two-output ADD gate. Now consider any layer in C_1. It is clear that by adding layers of the two-output ADD gates, we may simulate any fanout greater than 1. Note that we copy a number by adding it and 0.

As for the fanin, we address this problem in a similar way by adding extra layers of ADD or MUL gates before the layer in question. In case we have an odd number of inputs, we may need to provide an extra dummy input of either 0 or 1, depending on whether we are dealing with multiplication or addition.

In both the case of fanin and fanout, we are adding at most a linear number of extra gates; that is, it is linear in the fanin/fanout to simulate. However, per layer we are adding a number of layers that is at most the logarithm of the (largest) fanin/fanout size. Thus, if we denote by C_2 the circuit after this stage of transformation, the size is given by:

$$|C_2| = \mathcal{O}(|C_1|) = \mathcal{O}(|C|),$$

while the depth is now:

$$\text{depth}(C_2) = \mathcal{O}(\log |C_1| \cdot \text{depth}(C_1)) = \mathcal{O}(\log |C| \cdot \text{depth}(C)).$$

Thus, asymptotically we have increased the depth of the circuit by a logarithmic factor in the circuit size.

We note that inserting these fanin and fanout trees creates layers that may have less than a block's width. However, for fanin, the order of the inputs does not matter in any way. For fanout, we just make sure that the public (default inputs) are placed correctly, and the order of the other inputs do not matter (as they are all the same). This shows that within fanin and fanout trees, no permutations are needed. Fortunately, this also means that we can allow ourselves a width that is as small as we want.

A.7 Inserting Permutation Circuits

The only problem left to solve with C_2 to make sure it can be computed by only having to permute blocks, permute within blocks, or doing nothing between two layers.

We first consider the output from a layer in C_2. This output can be used in many subsequent layers, and we do not want outputs for different layers in the same block. To avoid this, we insert a permutation circuit that permutes all of the outputs into different sections of blocks, one for each of the following layers. These sections are maintained for every layer that produces outputs, by having it permute its output into the sections as above.

When we reach a layer of C_2, its input is given by a section of blocks, most likely with the elements in the wrong order. We insert a permutation circuit that permutes the inputs back into the correct order.

This construction ensures that we have exactly the property we wanted about the action between layers. Every permutation is handled by a permutation circuit, and those require only permutations of blocks or permutations within some blocks. Furthermore, we get the nice property that outputs from a layer to another does not need to be copied through every layer between those two. It is just stored on the side.

To conclude our section on transformations, we need only calculate the new size and depth of the final transformed circuit $C' = C_3$. We know that the depth of a permutation circuit is logarithmic in the number of elements involved, and that to permute n elements, we need a circuit of size $n \log n$. The circuits need to be inserted for every layer in C_2. The number of elements to permute is the width of the layer plus at most $\mathcal{O}(Xn)$, where X is the maximal number of layers $j > i$ reachable from a layer i. In the worst case, there is a single element going to X other layers, and since we can only pass on blocks of size $\Theta(n)$, this means Xn extra elements to permute. Note that $Xn = \mathcal{O}(|C|)$.

Let $d = \text{depth}(C_2)$ and let w_i be the width of layer i. The size of the final circuit C' is then given by

$$\begin{aligned}
|C'| &= \mathcal{O}(|C_2| + \sum_{i=1}^{d}(w_i + Xn)\log(w_i + Xn)) \\
&= \mathcal{O}(|C| + \log|C|\sum_{i=1}^{d}(w_i + Xn)) \\
&= \mathcal{O}(|C|\log|C| + dXn\log|C|) \\
&= \mathcal{O}(|C|\log|C| + \text{depth}(C)^2 n\log^3|C|),
\end{aligned}$$

where we use the fact that for general circuits $X = \mathcal{O}(d)$. The depth is given by

$$\text{depth}(C') = \mathcal{O}(\log|C|\text{depth}(C_2)) = \mathcal{O}(\log^2|C|\text{depth}(C)).$$

B Reducing the Complexity

In this section we attempt to reduce the complexity in Theorem 1 on page 17. To understand how to do this, we look back to Appendix A.7, where the circuit size of the final transformed circuit is calculated.

The goal is to reduce X, the maximal number of layers reachable from any single layer. Note that the wires containing public default inputs are not counted in this number.

One way to reduce X is as follows. Take the original circuit C (i.e. before any transformation has taken place) and replace any wire (i, j) with a path of length logarithmic in the depth. First find the bit composition of $j - i$:

$$j - i = b_m \dots b_0.$$

Now we add wires corresponding to that representation. Add a wire $(i, i+2^{b_m})$ ($b_m = 1$ always). Then from any intermediate layer $i + \sum_{i=k}^{m} b_i 2^{b_i}$, we simply add $2^{b_{k'}}$ for the next non-zero bit position k' to find the next wire. Clearly this eventually takes us to j, and the length of such a path is logarithmic in the depth. But the point of the trick is that now every layer i has only wires to other layers of the form $(i, i+2^k)$, where k is at most the depth. In other words:

$$X = \mathcal{O}(\log\text{depth}(C)),$$

and the size of the new circuit, which we denote $\overline{C}$ is now:

$$|\overline{C}| = \mathcal{O}(\log\text{depth}(C)|C|).$$

The other circuit transformations are now performed as usual on this new base circuit. When separating additions and multiplications, the asymptotic statement about X clearly remains true (either an output goes 2^k or $2^k + 1$ steps ahead now). Likewise, when adding fanin and fanout circuits, the statement remains true, since the added intermediate tree-like circuits are completely layered (i.e. their layers take input from the previous layer and provide output only for the subsequent layer).

Furthermore, we observe that the depth increase when inserting fanin and fanout circuits depends on the largest possible fanin and fanout sizes, which have not changed, and thus after this step we still have

$$\mathrm{depth}(C_2) = \mathcal{O}(\log |C| \cdot \mathrm{depth}(C)),$$

and not something depending on $|\overline{C}|$.

Repeating the calculation from Appendix A.7 we get:

$$\begin{aligned}
|C'| &= \mathcal{O}(|C_2| + \sum_{i=1}^{d}(w_i + Xn)\log(w_i + Xn)) \\
&= \mathcal{O}(|C_2| + \log |C_2| \sum_{i=1}^{d}(w_i + Xn)) \\
&= \mathcal{O}(|C| \log |C| \log \mathrm{depth}(C) + \log |C| \cdot dXn) \\
&= \mathcal{O}(|C| \log |C| \log \mathrm{depth}(C) + \mathrm{depth}(C) \log \mathrm{depth}(C) n \log^2 |C|).
\end{aligned}$$

In Appendix C we show that the computation costs $\mathcal{O}(\log^2 n|C'| + \mathrm{poly}(n))$ and the computation $\mathcal{O}(\log n|C'| + \mathrm{poly}(n)$. Now Corollary 1 on page 18 follows trivially.

C Proof of Theorem 6.1

Theorem 2. *There exists $0 < \delta < 1/3$ such that given n servers and an arithmetic circuit C that is at least $\Omega(n)$ gates wide, the protocol* ***`EvalCircuit`*** *realizes $\mathcal{F}_C$ with perfect security in the UC model against an active and adaptive adversary corrupting up to $t < \delta n$ servers.*

The total communication complexity is

$$\mathcal{O}(\log n \log |C| \cdot |C| + \mathrm{poly}(n, \log |C|) \cdot \mathrm{depth}(C)^2),$$

while the total computational complexity is

$$\mathcal{O}(\log^2 n \log |C| \cdot |C| + \mathrm{poly}(n, \log |C|) \cdot \mathrm{depth}(C)^2).$$

Proof. Every subprotocol used has already been proved correct, and therefore it follows from the construction of `EvalCircuit` that it is correct. We need therefore only deal with simulation and complexities.

Simulation: Instead of building one large simulator for everything at once, we follow a more structured approach. The protocol evaluates the transformed circuit C' layer by layer, and our security proof will work in the same way. Define the functionalities $\mathcal{F}_i$ for $0 \leq i \leq \mathrm{depth}(C')$ in the following way:

$\mathcal{F}_i$ takes inputs $(x_1, \ldots, x_r)$ from the input clients and outputs the secret-shared state after computing layer i of C'. For technical reasons, the adversary must input the shares it wishes to receive for every shared value in this state, 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 merely having secret-shared the inputs and prepared pairs and double sharings.

The idea is to realize $\mathcal{F}_{i+1}$ in the hybrid model where we are given $\mathcal{F}_i$, and to show that we can realize $\mathcal{F}_0$ as well. By induction we can realize $\mathcal{F}_d$, which we can then combine with the protocol `Reco` to get the final proof for $\mathcal{F}_C$.

In other words, there is a protocol for each step that invokes the functionality for the previous step (if there is one), and this protocol realizes the functionality for the current step. It is important to understand that the protocol for the very last step is in the hybrid model with $\mathcal{F}_d$, and by going through the whole structure recursively, we see that it is in fact exactly the protocol `EvalCircuit`.

We handle the simulation in three steps. The first two handle the induction proof that gives us the correct behaviour for any F_i. The last one, step $d + 1$, shows how to simulate the output step of the protocol.

Step 0*:* The only thing that happens in this step is that the functionalities $\mathcal{F}_{pairs}$, $\mathcal{F}_{double}$ and $\mathcal{F}_{RobustShare}$ are called. The simulation in this case is therefore trivial.

Step i: The simulator $\mathcal{S}_i$ chooses a set of dummy inputs $x'_1, \ldots, x'_r$ for the input clients. To simulate the adversary in this step, $\mathcal{S}_i$ simply runs the protocol as if it had been run with $\mathcal{A}$ and the dummy values. To clarify, this means inputting $x'_1, \ldots, x'_r$ into $\mathcal{F}_{i-1}$ as well as the shares given by $\mathcal{A}$. The shares are given back to $\mathcal{A}$ as expected, and the rest of the protocol is run as we normally would with the values coming from $\mathcal{A}$ and $\mathcal{F}_{i-1}$. At the end of the run, $\mathcal{S}$ has computed a set of final shares for corrupted players. These are handed to $\mathcal{F}_i$ to complete the simulation.

Because every subprotocol used has perfect privacy, it follows that all values generated and all sharings opened to the adversaries are independent of the actual input values. Therefore, the simulation is perfect.

As for adaptive corruptions, there are three cases. If the player in question is only a client, there is no view to produce in this step and we are already done. If a server is corrupted before $\mathcal{F}_i$ has output shares, we simply provide to $\mathcal{A}$ the view that $\mathcal{S}_i$ already has from its dummy run of the protocol. Because of the privacy, we find again that the simulation for the dummy-version of an honest player must also be perfect.

The last case is when a server is corrupted after the protocol has completed. In this case, the server has received output shares from $\mathcal{F}_i$, and the view we produce for the newly corrupted player must be consistent with this.

Any output share y' that must be made consistent with the real share y either comes from the `PermuteWithinBlocks` protocol, or it is a linear combination of something containing at least the result of one multiplication not used elsewhere, or it is simply the linear combination of constants and shares coming directly from $\mathcal{F}_{i-1}$.

We handle the last case first. Every such output share gives rise to one linear equation, and we end up with a system of equations that is not overdetermined. Therefore we may pick a random solution for new shares coming from $\mathcal{F}_{i-1}$. This adjusts the starting shares for a number of the sharings used by $\mathcal{S}_i$.

Say our starting sharings are given by polynomials $f_1, \ldots, f_m$. We adjust them by adding polynomials $g_1, \ldots, g_m$. All g_j are of degree at most d and have been interpolated from being zero in all points corresponding to corrupted players and in all secret points.

We now work through our simulation once again, and this works correctly as long as we do not run into any multiplications or `PermuteBits`. Let us examine how to repair the situation if we reach a multiplication; the `PermuteBits` case is similar but simpler. Note that we only need to handle the situation with the very first such operation, since we can perfectly adjust the shares at that point.

In multiplication, we multiply sharings locally, add a random block and open. The sharings are linear combinations of the starting sharings and possibly outputs of previous multiplications (note that these have not actually changed). In other words, we would originally be opening the polynomials:

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

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

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

We wish to adjust the r_{2d} polynomial in p' such that the two results are the same, and yet shares for the previously corrupted players do not change. Therefore we set

$$r'_{2d} = r + p - p'.$$

This gives us a perfectly valid degree $2d$ polynomial. For any j that became corrupted earlier, $p(j) = p'(j)$ because p' only differs by terms that are zero in those points. Therefore, $r'_{2d} = r_{2d}$ in those same points, and we have adjusted our view to match what the adversary has already seen.

We still need to explain how to adjust the final output shares if they come from a multiplication (or a linear combination containing a multiplication result, which is equivalent because it contains a nonzero scalar times a random mask block) or a `PermuteBits`. We simply add adjustment polynomials to the randomness, similarly to what was done above.

Step $d+1$: In this step we aim to realize $\mathcal{F}_C$. 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 those sharings towards the right output clients.

To simulate, we take shares from corrupted servers as usual and return them to those servers. Corrupted output clients receive sharings from corrupted servers just by letting $\mathcal{A}$ do what it wants. For sharings opened by honest servers towards corrupted output clients, we receive the correct output value from $\mathcal{F}_C$ and simply pick a random consistent sharing of it to open toward corrupted out clients. Note that this also covers adaptive corruptions, and we are done.

Complexity: For the time being, we ignore the cost of complaints, failures of `Permuted`, and the fact that the subprotocol require a certain amount of elements to be efficient.

The cost of every our operations, for input sharing, computation of the steps, or for output sharing, essentially shares the same bound: the communication for one *element* is $\mathcal{O}(1)$, and the computation is $\mathcal{O}(\log n)$.

Worst-case we need a permuted pair for each n gates of the circuit. Furthermore, these are in the worst case distributed in $\log l = \Theta(\log n)$ groups of different permutations. Therefore, it costs $\mathcal{O}(\log^2 n \cdot |C'|)$ computation and $O(\log n \cdot |C'|)$ communication to construct the pairs. The same costs bound also covers input sharings, the creation of double sharings, and the reconstruction of the output.

For the computation of the $d = \mathrm{depth}(C')$ layers (this includes multiplications and permutations within blocks), we treat each gate only once, and therefore the cost again stays within the same bounds. However, `Permuted` might fail, and this can happen a total of $t = \Theta(n)$ times. Since we only work on n blocks at a time, the total cost of failing the maximum number of times is bounded by $\mathrm{poly}(n)$. The same goes for all of the broadcasts of complaints; there are at most t of them, since each results in at least one corrupt party being eliminated, and therefore the total cost of all complaints is at most $\mathrm{poly}(n)$.

We conclude that the total cost of computation is

$$\mathcal{O}(\log^2 n \cdot |C'| + \mathrm{poly}(n)) = \mathcal{O}(\log^2 n \log |C| \cdot |C| + \mathrm{poly}(n, \log |C|) \cdot \mathrm{depth}(C)^2),$$

while the total cost of communication is

$$\mathcal{O}(\log n \cdot |C'| + \mathrm{poly}(n)) = \mathcal{O}(\log n \log |C| \cdot |C| + \mathrm{poly}(n, \log |C|) \cdot \mathrm{depth}(C)^2).$$

All of this assumes that $|C'| = \Omega(n^3)$, and that every layer processes at least $\Omega(n^3)$ elements. If this is not the case, we pay at each step for n^3 elements anyway. However, the cost of such cases easily stay within the $\mathrm{poly}(n, \log |C|) \cdot \mathrm{depth}(C)^2$ bound, and so the complexities presented above do in fact hold in any case.