Secret-Shared Shuffle

Melissa Chase*, Esha Ghosh**, and Oxana Poburinnaya***

Abstract. Generating secret shares of a *shuffled* dataset - such that neither party knows the order in which it is permuted - is a fundamental building block in many protocols, such as secure collaborative filtering, oblivious sorting, and secure function evaluation on set intersection. Traditional approaches to this problem either involve expensive public-key based crypto or using symmetric crypto on permutation networks. While public-key based solutions are bandwidth efficient, they are computation-bound. On the other hand, permutation network based constructions are communication-bound, especially when the elements are long, for example feature vectors in an ML context.

We design a new 2-party protocol for this task of computing secret shares of shuffled data, which we refer to as secret-shared shuffle. Our protocol is secure against static semi-honest adversary.

At the heart of our approach is a new method of obtaining two sets of pseudorandom shares which are "correlated via the permutation", which can be implemented with low communication using GGM puncturable PRFs. This gives a new protocol for secure shuffle which is concretely more efficient than the existing techniques in the literature. In particular, we are three orders of magnitude faster than public key based approach and one order of magnitude faster compared to the best known symmetric-key cryptography approach based on permutation network when the elements are moderately large.

Keywords: secure shuffle, secure function evaluation, puncturable PRF

1 Introduction

Machine Learning algorithms are data-hungry: more data leads to better understanding of accuracy of models. On the other hand, privacy of data is becoming exceedingly important, for social, business reasons and policy compliance such as GDPR. There has been decades of groundbreaking work in the academic literature that developed cryptographic technology for developing collaborative computation. But it still has some significant bottlenecks in terms of wide-scale adoption. Although theoretical results demonstrate the possibility of generic secure computation, they are not efficient enough to be adopted, both in terms of computation and communication size. For instance, Google cited network cost as a major hindrance is adopting cryptographic secure computation solution [11].

* Microsoft Research. Email: melissac@microsoft.com.

** Microsoft Research. Email: Esha.Ghosh@microsoft.com.

*** Simons Institute for the Theory of Computing. The work was partially done while doing internship at Microsoft Research. Email: oxanapob@bu.edu.

Secret-shared shuffle. In this work, we focus on computation and communication efficiency of a fundamental building block used in a wide range of secure computation protocols, which we call "secret-shared shuffle". Secret-shared shuffle is a protocol which allows two parties to jointly shuffle data and obtain secret shares of the result - without any party learning the permutation corresponding to the shuffle.

Motivation. To see the importance of this primitive, consider the task of securely evaluating some function on an intersection of sets belonging to two parties; in particular, the intersection itself should also remain secret. Ideally we would use a private set intersection protocol which outputs an intersection in some "encrypted" form - e.g. by encrypting or secret sharing elements in the intersection. However, currently known efficient private set intersection protocols do not output "encrypted" intersection: instead they output an encrypted vector of bits indicating if each element is in the intersection or not [3]. The difference is that in the former case one would directly run secure function evaluation (SFE) on the encrypted intersection, whereas in the latter case SFE has to be run on the whole database. Needless to say, this incurs unnecessary overhead, especially in cases where the intersection is relatively small compared to the initial sets.

In other words, ideally we would want to get rid of non-intersection elements before running SFE. A natural way to do this without compromising security is to shuffle the elements together with the indicator vector and give parties secret-shared result. Then parties can reveal the indicator vector and discard elements which are not in the intersection. Note that it is crucial that neither party learns how exactly the elements were permuted; otherwise this party could learn whether some of its elements are in the intersection or not. Also note that the requirement on the secrecy of the permutation implies that the result of the shuffle has to be in some encrypted or secret-shared form, in order to prevent linking original and shuffled elements.

Known techniques and their limitations. It is instructive to look at "a half" of a secret-shared shuffle, which we call Permute+Share : in this protocol P_0 holds a permutation π and P_1 holds the database $\boldsymbol{x}$, and they would like to learn secret shares of permuted database[1]. While this problem can be solved by any generic SFE, to the best of our knowledge, there are two specialized solutions for this problem, which differ in how exactly the permuting happens. One approach is to give P_0's shares of $\boldsymbol{x}$ to P_1 in some encrypted form, let P_1 permute them according to π under the encryption, rerandomize them, and return them to P_0. This is a folklore solution that uses rerandomizable additively homomorphic public-key encryption. This approach is compute heavy. We elaborately describe this solution in Section 6.1. The other approach is to start with secret-shared $\boldsymbol{x}$ and jointly compute atomic swaps, until all elements arrive to their target location. To prevent linking, each atomic swap should also rerandomize the shares.

[1] Note that one can get secret-shared shuffle by combining two instance of Permute+Share .

This approach is taken by [21, 14], who let parties jointly apply a permutation network to the shares, where each atomic swap is implemented using OT in [21] and garbled circuit in [14]. The downside of this approach is its communication complexity which is proportional to $N \log N \cdot \ell$, where N is the number of elements in the database and ℓ is the size of each element. This overhead seems to be inherent in approaches based on joint computation of atomic swaps, since each element has to be fully fed into at least $\log N$ swaps.

Our Contribution We design a protocol for Permute+Share (and therefore secret-shared shuffle) which follows a novel approach. At a very high level, we show how parties can use puncturable PRFs to generate two sets of pseudorandom values - one per party - with a special permutation-related dependency between them; then each party uses its set to compute shares of permuted database. Importantly, we show how these sets can be generated with communication only proportional to $N \log N \cdot \lambda$ (in addition to $N \cdot \ell$ which is inherent), where λ is security parameter. Note that the size ℓ of the element could be very long (e.g. each element could be a feature vector in ML algorithm), and thus it could be a significant improvement in communication over permutation network-based approach.

It should be noted that in our protocol the permuting itself happens within the generation of the two mentioned sets. In particular, in our protocol parties do not permute encrypted shares and thus do not require public-key operations (except in base OTs), nor do we perform atomic swaps, which enables saving on the communication.

Our protocol uses lightweight crypto primitives (XORs and PRGs) which is optimal for large data elements (or data elements with long associated data). Our protocol is secure in the semi-honest model. We measure the concrete cost of our protocol and simulate its performance over a typical WAN. We see a a three orders of magnitude improvement over the best known public key based approach and an order of magnitude improvement over the best known symmetric key approach. The details of our experiment are in Section 6.

1.1 Applications

Collaborative Filtering One immediate application of our shuffle protocol is to allow two parties who hold shares of a set of elements to filter out elements that satisfy a certain criterion. This could include removing poorly formed or outlier elements. Or it could be used after e.g. a PSI protocol [23, 24, 3] or in database join [20] to remove elements that were not matched. If we are willing to reveal the number of elements meeting this criterion, we can use a shuffle to securely remove these elements so that subsequent operations can be evaluated only on the resulting smaller set, which is particularly valuable if the subsequent computation is expensive (e.g. a machine learning task [19]). To do this, we first shuffle the set, then apply a 2PC to each element to evaluate the criterion, revealing the result bit in the clear, and finally remove those items whose result is 1.

Sorting under MPC Our secret shared shuffle protocol can also be used to build efficient protocols for other fundamental operations. For example to sort a list of secret shared elements and output resulting secret shares we can use the shuffle-and-reveal approach proposed by [13]. The idea is to first shuffle the data, and then run a sorting algorithm. At this point we can use MPC to evaluate comparisons and reveal the results of each comparison in the clear. This yields more efficient results than the standard oblivious sorting protocol based on sorting networks; those protocols either have huge constants [1] or require $O(N \log^2 N)$ running time (using Bitonic Sorting network), where N is the number of elements in the database. Note that in many cases we want to sort not just a set of elements, but also some associated data for each element.

Sort, in addition to being a fundamental operation, can be used to find the top k results in a list, to evaluate the median or quantiles, to find outliers, etc.

Secure Computation for RAM programs There has been a line of work starting with [10, 8, 17, 16, 18, 27, 25, 7] that looks at secure computation for RAM programs (as opposed to circuits). The primary building block in these constructions is oblvious RAM (ORAM), which is a technique for transforming a RAM program to be oblivious in that the memory accesses do not reveal anything about the computation (in particular they don't reveal which RAM entries are being accessed). When used in secure computation, generally each party holds a share of the transformed memory, and the two parties jointly convert logical RAM access into a series of random looking memory accesses, which they each perform locally to retrieve the corresponding share. One challenge in these schemes is to initialize the ORAM to store the parties' inputs. A naive solution simply performs an ORAM write operation for each input item, but the concrete costs on this are very high. [16, 27] show that this can be made much more efficient using a shuffle: the parties simply permute their entries using a random secret shared permutation, and then they can direclty store them as the ORAM memory. [27] achieve significant improvements by using garbled circuits to implement a permutation network; as we will see in section 6 our solution far outperforms this approach, so we should get significant performance improvements for this application. Note that in ORAM it is often beneficial to have somewhat large block size (the cost of retrieving a block is generally $O(\log N)$ and the cost of shuffling is $O(N \log N)$, where N is the number of blocks, although once a block is retrieved the 2PC will have to scan linearly over the block to find the particular entry desired. We leave it to future work to find the optimal point in this trade-off, but note that our more efficient shuffle makes it more advantageous to use larger blocks.

1.2 Technical overview.

Notation. By bold letters $\boldsymbol{x}, \boldsymbol{a}, \boldsymbol{b}, \boldsymbol{r}, \boldsymbol{\Delta}$ we denote vectors of N elements, and by $\boldsymbol{x}[j]$ we denote the j-th element of $\boldsymbol{x}$. By $\pi(\boldsymbol{x})$, where π is a permutation, we denote the permuted vector $(\boldsymbol{x}[\pi(1)], \ldots, \boldsymbol{x}[\pi(N)])$.

Secret-Shared Shuffle. Recall that the goal of the secret-shared shuffle is to let parties learn secret shares of a shuffled data. More concretely, consider parties P_0, P_1, where P_1 owns database $\boldsymbol{x}$. Our goal is to build a protocol which allows P_0 to learn $\boldsymbol{r}$ and P_1 to learn $\boldsymbol{r} \oplus \pi(\boldsymbol{x})$, but nothing more; here $\boldsymbol{r}$ is a random vector of the same size as the database, and π is a random permutation of appropriate size. Our protocol also works for the case when $\boldsymbol{x}$ was secret shared between P_0 and P_1 to begin with (instead of being an input of one party).

Secret-shared shuffle can be easily built given its variant where one of the parties *chooses* the permutation; we call this protocol Permute+Share . That is, in this protocol P_0 holds π and P_1 holds $\boldsymbol{x}$, and as before, they would like to learn $\boldsymbol{r}$ and $\boldsymbol{r} \oplus \pi(\boldsymbol{x})$, respectively. Indeed, secret-shared shuffle can be obtained by executing Permute+Share twice, where first P_0 and then P_1 chooses the permutation (note that in the second execution the database is itself already secret shared). Thus, in the rest of the introduction we describe how to build Permute+Share . The details of how to obtain secret-shared shuffle from Permute + Share are in Section 5.4.

Our construction proceeds in three steps: first we explain how to build Permute+Share using another protocol called Share Translation protocol, then we build the latter using Oblivious Punctured Vector protocol, and finally we explain how to design OPV protocol with low communication using Oblivious Transfer and Pseudorandom Functions.

Note that we are going to describe our protocols using $\oplus$ operation for simplicity, however, in the main body we instead use a more general syntax with addition and subtraction, to allow our protocols to work in different groups.

Building simplified Permute+Share *from Share Translation protocol.* We first describe a simplified and inefficient version of Permute+Share protocol; the running time of this protocol is proportional to the square of the size of the database. Later in the introduction we explain how we exploit the structure of Benes permutation network [2] to achieve our final protocol.

As a starting point, consider the following idea: P_1 chooses random masks $\boldsymbol{a} = (\boldsymbol{a}[1], \ldots, \boldsymbol{a}[N])$ and sends its masked data $\boldsymbol{x} \oplus \boldsymbol{a}$ to P_0. Now P_0 and P_1 together hold a secret-shared $\boldsymbol{x}$, albeit not permuted. Note that P_0 knows the permutation π and could easily locally rearrange its shares in order of $\pi(\boldsymbol{x} \oplus \boldsymbol{a})$. However, P_1 doesn't know π and thus cannot rearrange $\boldsymbol{a}$ into $\pi(\boldsymbol{a})$. Further, any protocol which allows P_1 to learn $\pi(\boldsymbol{a})$ would immediately reveal π to P_1, since P_1 also knows $\boldsymbol{a}$.

Therefore, instead of choosing a single set of masks, P_1 should choose two different and independent sets of masks, $\boldsymbol{a}$ and $\boldsymbol{b}$, where $\boldsymbol{a}$, as before, is used to hide $\boldsymbol{x}$ from P_0, and $\boldsymbol{b}$ will become the final P_1's share of $\pi(\boldsymbol{x})$. However, now P_0 has a problem: since P_1's share is $\boldsymbol{b}$, P_0's share should be $\pi(\boldsymbol{x}) \oplus \boldsymbol{b}$; however, P_0 only receives $\boldsymbol{x} \oplus \boldsymbol{a}$ from P_1, and has no way of "translating" it into $\pi(\boldsymbol{x}) \oplus \boldsymbol{b}$. Thus we additionally let parties execute a Share Translation protocol to allow P_0 obtain a "translation function" $\boldsymbol{\Delta} = \pi(\boldsymbol{a}) \oplus \boldsymbol{b}$, as we explain next in more detail:

Share Translation protocol takes as input permutation π from P_0 and outputs vectors $\boldsymbol{\Delta}$ to P_0 and $\boldsymbol{a}, \boldsymbol{b}$ to P_1, such that $\boldsymbol{\Delta} = \pi(\boldsymbol{a}) \oplus \boldsymbol{b}$, and, roughly speaking, $\boldsymbol{a}, \boldsymbol{b}$ look random[2]. Permute+Share can be obtained from Share Translation as follows:

1. P_0 and P_1 execute a Share Translation protocol, where P_0 holds input π, receives output $\boldsymbol{\Delta}$, and P_1 receives output $\boldsymbol{a}, \boldsymbol{b}$.
2. P_1 sends $\boldsymbol{x} \oplus \boldsymbol{a}$ to P_0 and sets its final share to $\boldsymbol{b}$.
3. P_0 sets its share to $\pi(\boldsymbol{x} \oplus \boldsymbol{a}) \oplus \boldsymbol{\Delta}$. Note that this is equal to $\pi(\boldsymbol{x}) \oplus \pi(\boldsymbol{a}) \oplus \pi(\boldsymbol{a}) \oplus \boldsymbol{b} = \pi(\boldsymbol{x}) \oplus \boldsymbol{b}$, and therefore the parties indeed obtain secret-shared $\pi(\boldsymbol{x})$.

In other words, share translation function $\boldsymbol{\Delta}$ allows P_0 to translate "shares of x under $\boldsymbol{a}$" into "shares of permuted x under $\boldsymbol{b}$"; hence the name.

Note that the Share Translation protocol can be viewed as a variant of Permute+Share protocol, with a difference that the "data" which is being permuted and shared is pseudorandom and out of parties' control (i.e. it is chosen by the protocol): indeed, in Share Translation protocol P_1 receives the "pseudorandom data" $\boldsymbol{a}$, and in addition P_0 and P_1 receive $\boldsymbol{\Delta} = \pi(\boldsymbol{a}) \oplus \boldsymbol{b}$ and $\boldsymbol{b}$, respectively, which can be thought of as shares of $\pi(\boldsymbol{a})$ using mask $\boldsymbol{b}$. In other words, we reduced the problem of permuting the fixed data $\boldsymbol{x}$ to the problem of permuting some pseudorandom, out-of-control data $\boldsymbol{a}$. In the following paragraphs we explain how we can exploit pseudorandomness of $\boldsymbol{a}$ and $\boldsymbol{b}$ to build Share Translation protocol with reduced communication complexity.

Building Share Translation from Oblivious Punctured Vector. We start with defining Oblivious Punctured Vector protocol (OPV): this protocol, on input $j \in [N]$ from P_0, allows parties to jointly generate vector $\boldsymbol{v}$ with random-looking elements such that:

- P_0 learns all vector elements except for its j-th element $\boldsymbol{v}[j]$;
- P_1 learns the whole vector $\boldsymbol{v}$ (but doesn't learn index j)[3].

This protocol can be used to build Share Translation protocol as follows: the parties are going to run N executions of OPV protocol to generate N vectors $\boldsymbol{v}_1, \ldots, \boldsymbol{v}_N$, where P_0's input in execution i is $\pi(i)$. Consider an $N \times N$ matrix $\{\boldsymbol{v}_i[j]\}_{i,j \in N^2}$. By the properties of OPV protocol, P_1 learns the whole matrix, and P_0 learns the matrix except for elements corresponding to the permutation, i.e. it learns nothing about $\boldsymbol{v}_1[\pi(1)], \ldots, \boldsymbol{v}_N[\pi(N)]$ (see fig. 1).

[2] More precisely, P_1 shouldn't learn anything about π, and P_0 shouldn't learn $\boldsymbol{a}, \boldsymbol{b}$, except for what is revealed by π and $\boldsymbol{\Delta}$ (note that it still learns, e.g., $\boldsymbol{a}_{\pi(1)} \oplus \boldsymbol{b}_1$).

[3] Note that this is very similar to 1-out of-N OT - except that j specifies which element P_0 *doesn't* learn - and in fact is almost the same as $N-1$-out of-N OT. The difference is that in our primitive vector $\boldsymbol{v}$ is pseudorandom and given by the protocol to the parties (rather than chosen by the sender as in standard OT). We use this fact to save on communication.

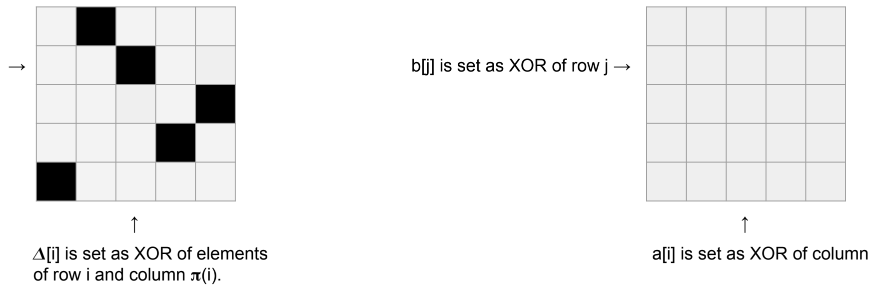

Fig. 1. (left) P_0 receives a "punctured" matrix, which is missing elements at positions $(i, \pi(i))$. Note that the missing elements are not needed to compute $\boldsymbol{\Delta}$. (right) P_1 receives the full matrix and uses it to compute masks $\boldsymbol{a}, \boldsymbol{b}$.

Then P_1 sets elements of $\boldsymbol{a}, \boldsymbol{b}$ to be column- and row-wise sums of the matrix elements, i.e. for all $i \in N$ it sets $\boldsymbol{a}[i] \leftarrow \bigoplus_j \boldsymbol{v}_j[i]$, and for all $j \in N$ it sets $\boldsymbol{b}[j] \leftarrow \bigoplus_i \boldsymbol{v}_j[i]$. P_0 computes $\boldsymbol{\Delta}[i]$ by taking the the sum of column $\pi(i)$ (except the element $\boldsymbol{v}_i[\pi(i)]$ which it doesn't know) and adding the sum of row i (again, except the element $\boldsymbol{v}_i[\pi(i)]$ which it doesn't know), i.e. it sets $\boldsymbol{\Delta}[i] \leftarrow \left(\bigoplus_{j \neq i} \boldsymbol{v}_j[\pi(i)]\right) \oplus \left(\bigoplus_{j \neq \pi(i)} \boldsymbol{v}_i[j]\right)$.

Correctness of this protocol can be immediately verified: indeed, each $\boldsymbol{\Delta}[i] = \boldsymbol{a}[\pi(i)] \oplus \boldsymbol{b}[i]$, since the missing value $\boldsymbol{v}_i[\pi(i)]$ participates in the sum $\boldsymbol{a}[\pi(i)] \oplus \boldsymbol{b}[i]$ twice and therefore doesn't influence the result. For security, note that P_0 doesn't learn anything about $\boldsymbol{a}, \boldsymbol{b}$ (except for $\boldsymbol{\Delta}$), since it is missing exactly one element from each row and column of the matrix; the missing element acts as a one-time pad and hides each $\boldsymbol{a}[i], \boldsymbol{b}[j]$ from P_0. P_1 doesn't learn anything about the permutation π due to index hiding of the OPV protocol.

Note that this protocol has running time proportional to N^2.

Building Oblivious Punctured Vector from OT and PRFs. While OPV could be readily implemented using $N-1$-out of-N OT, we will make use of the fact that the vector $\boldsymbol{v}$ is pseudorandom to reduce communication complexity to $\log N$ 1-out of-2 OTs.

In the beginning of the protocol P_1 computes $\boldsymbol{v}$ by choosing key for GGM PRF at random, denoted seed_ϵ, and setting each $\boldsymbol{v}[i] \leftarrow PRF(\mathsf{seed}_\epsilon; i)$, $i \in [N]$. Recall that in GGM construction the key is treated as a prg seed, which implicitly defines a binary tree with leaves containing PRF evaluations $F(1), F(2), \ldots, F(N)$. In other words, we set vector $\boldsymbol{v}$ to contain values at the leaves of the tree.

Let P_0's input in the OPV protocol be j. This means that P_0 should learn leaves $F(i), i \neq j$, as a result of the protocol. This can be done as follows. Let us denote internal seeds in the tree by $\{\mathsf{seed}_\gamma\}$, where γ is a string describing the position of the note in the tree (in particular, at the root $\gamma = \epsilon$, an empty string). Let's assume for concreteness that the first bit of j is 1. The parties are going to run 1-out of-2 OT protocol, where P_0's input is the complement of the first bit of j, i.e. 0, and P_1's inputs are seed_0, seed_1. This allows P_0 to recover seed_0 and therefore to locally compute the left half of the tree, i.e. all values $F(1), \ldots, F(N/2)$, and corresponding intermediate seeds.

Next, assume the second bit of j is 0. Note that the parties could run 1-out of-4 OT to let P_0 learn seed_{11} and therefore locally compute the right quarter of the tree $F(3N/4), \ldots, F(N)$, then run 1-out of-8 OT and so on. However, this approach would require eventually sending 1-out of N OT, which defeats our initial purpose of having $\log N$ 1-out of-2 OTs only.

Instead, we let P_0 learn seed_{11} in a different way: we let P_1 send only *two* values, via 1-out of 2-OT: the first value is the sum of seeds which are left children, i.e. $\mathsf{seed}_{00} \oplus \mathsf{seed}_{10}$, and the second value is the sum of seeds which are right children, i.e. $\mathsf{seed}_{01} \oplus \mathsf{seed}_{11}$. Since P_0 already knows the whole left subtree and in particular seed_{00} and seed_{01}, it can receive $\mathsf{seed}_{01} \oplus \mathsf{seed}_{11}$ from the OT protocol and add seed_{01} to it to obtain seed_{11}. We note that this idea of sending the sums of left and right children was inspired by a similar technique by Doerner and Shelat [4] in the context of optimizing function secret sharing.

More generally, the parties execute $\log N$ 1-out of-2 OTs - one for each level of the tree - where at each level k the first input to OT is the sum of all odd seeds at that level, and the second input to OT is the sum of all even seeds at that level. It can be seen that each sum contains exactly one term which P_0 doesn't know yet, and therefore it can receive the appropriate sum (depending on the k-th bit of j) and subtract other seeds from it to learn the next seed of the subtree. Note that these OT's can be executed in parallel.

Note that the running time of the parties is proportional to the vector size, but their communication size only depends on its logarithm.

Achieving simulation-based definition. We note that the protocols we described so far only achieve indistinguishability-based definition, but not simulation-based definition. To see where the problem lies, assume our Permute+Share protocol is used as a subroutine in a larger protocol, and let's try to simulate this execution. Suppose the simulator of the larger protocol came up with simulated shares $\boldsymbol{y}, \boldsymbol{z}$, and now we need to simulate the internal state of parties in Permute+Share , given $\boldsymbol{y}, \boldsymbol{z}$ as the output of the protocol. This task however is problematic: indeed, recall that each element $\boldsymbol{z}[i]$ is a sum of pseudorandom values, which are the leaves of the GGM PRF tree. Since P_1 knows the whole tree, including its root, this means that the simulator, given some element $\boldsymbol{z}[i]$, has to come up with a root of the GGM tree such that its leaves sum up to $\boldsymbol{z}[i]$, in order to simulate P_1's state. However, finding such a root is hard by security of the PRF, even if it exists.

To achieve simulation-based definition, we slightly modify the original Permute+Share protocol as follows: we additionally instruct P_1 to sample random string $\boldsymbol{w}$ of the size of the database and send it to P_0, together with $\boldsymbol{x} \oplus \boldsymbol{a}$. Then

P_0 should set its share to be $\pi(\boldsymbol{x} \oplus \boldsymbol{a}) \oplus \boldsymbol{\Delta} \oplus \boldsymbol{w}$, and P_1 should set its share to be $\boldsymbol{b} \oplus \boldsymbol{w}$. In other words, P_1 should additionally secret-share its vector $\boldsymbol{b}$ using random $\boldsymbol{w}$. Such a protocol can be simulated by a simulator who executes Share Translation protocol honestly (obtaining some $\boldsymbol{a}', \boldsymbol{b}', \boldsymbol{\Delta}'$) and then sets simulated $\boldsymbol{w}$ to be $\boldsymbol{z} \oplus \boldsymbol{b}'$ (where $\boldsymbol{z}$ is the output of Permute+Share protocol simulated by an external simulator)

Our final protocol. Recall that, while communication complexity in our protocol is low, computation complexity is proportional to the size of the database squared, and thus can be prohibitively high for large database size. To deal with this issue, we consider a "merge" of previously described Permute+Share protocol and permutation-network based approach. The idea is to split the permutation π into a composition of multiple permutations $\pi_1 \circ \ldots \circ \pi_d$, such that each π_i is itself a composition of several disjoint permutations, each acting on T elements, for some parameter T. Such a decomposition can be found using a special structure of Bene's permutation network. For instance, for $T = 4$ and 8-element network, note that in the first layer x_{000} and x_{100} may get swapped, as well as x_{010} and x_{110}, and that in the second layer x_{000} and x_{010} may get swapped, as well as x_{100} and x_{110}; this means that in the first two layers a 4-element permutation is applied to elements x_{000}, x_{010}, x_{100}, x_{110} (fig. 1.2). Note that, this is an illustrative example that is instructive to build the intuition, the actual decomposition is shown in Section 5.3.

With such a decomposition in place, parties can run parallel executions of Share Translation protocols, each acting on domain of size T. Note that, since the running time of a single Share Translation is proportional to the domain size squared, it is better to run N/T protocols of size T each, rather than a single protocol on domain size N. Concretely, our experiments show that the best efficiency is achieved for $T = \sqrt{N}$. Note that setting $T = N$ corresponds to our simplified Permute+Share protocol described before, and setting $T = 2$ results in essentially computing the permutation network, where each swap is implemented in a somewhat-complicated way, using Share Translation protocol. Thus, this scheme can be thought of as a golden middle between the two approaches.

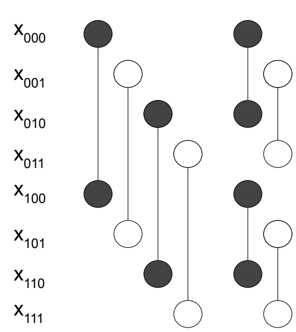

Fig. 2. The initial part of the Benes permutation network for 8 elements. Note that the first two layers could be replaced by two 4-elements disjoint permutations: one acting on white elements and the other acting on black elements.

It remains to note that parties can run all executions of Share Translation protocol in parallel (as opposed to taking multiple rounds, following the layered structure of the permutation network). To achieve this, in all execution except for the first ones, P_1 instead of sending initial masked data $\boldsymbol{x} \oplus \boldsymbol{a}$ should send correction vector $\boldsymbol{a}^{new} \oplus \boldsymbol{b}^{old}$, which can be added to the shares of P_0 in order to obtain $\boldsymbol{x} \oplus \boldsymbol{a}^{new}$. We refer the reader to Section 5.3 for more details.

2 Notations

We denote the security parameter as λ. the bit length of each element in the input set is ℓ, $\ell = \mathsf{poly}(\lambda)$. We denote and upper bound on the size of the database as N. Ideal functionality is denoted as $\mathcal{F}$. We will denote vectors with bold fonts and individual elements with indices. For example, $\boldsymbol{v}$ is a vector of N elements where each individual element is denoted at v_i. $\stackrel{\$}{\leftarrow}$ denotes selected uniformly at random from a domain. By S_N we denote the group of all permutations on N elements.

We also make use of the following notation:

Exec: Let Π be a two-party protocol. By $(\mathsf{output}_0, \mathsf{output}_1) \leftarrow \mathsf{exec}^{\Pi}(\lambda; x_0, x_1; r_0, r_1)$ we denote the concatenated outputs of all parties after the execution of the protocol Π with security parameter λ on inputs x_0, x_1 using randomness r_0, r_1.

View: Let Π be a two-party protocol. By $\mathsf{view}_b^{\Pi}(\lambda; x_0, x_1; r_0, r_1)$ we denote the view of party b when parties P_0 and P_1 run the protocol Π with security parameter λ on inputs x_0, x_1 using randomness r_0, r_1. The view of each party includes its inputs, random coins, all messages it receives, and its outputs. When the context is clear, we also write view_b for short.

Honest-but-curious security for a 2PC: Honest-but-curious security for a 2PC protocol Π evaluating function $\mathcal{F}$ is defined in terms of the following two experiments:

$IDEAL^{\mathcal{F}}_{\mathsf{sim},b}(\lambda, x_0, x_1)$ evaluates $\mathcal{F}(x_0, x_1)$ to obtain output (y_0, y_1) runs the stateful simulator $\mathsf{sim}(1^\lambda, b, x_b, y_b)$ which produces a simulated view view_b for party P_b. The output of the experiment is $(\mathsf{view}_b, y_{1-b})$.

$REAL^{\Pi}_b(\lambda, x_0, x_1)$ runs the protocol with security parameter λ between honest parties P_0 with input x_0 and P_1 with input x_1 who obtain outputs y_0, y_1 respectively. It outputs $(\mathsf{view}_b, y_{1-b})$.

Definition 1. *Protocol Π realizes $\mathcal{F}$ in the honest-but-curious setting if there exists a simulator* sim *such that for all inputs x_0, x_1, and corrupt parties $b \in \{0,1\}$ the two experiments are indistinguishable.*

Pseudo Random Generator Let $\mathsf{G} : \{0,1\}^m \to \{0,1\}^l$, $l \geq m$ be a PRG. The security definition of a PRG is the following. G is a PRG if the following distributions are computationally indistinguishable:

$$\mathcal{D}_1 = \{\mathsf{s} \leftarrow \{0,1\}^m : \mathsf{G}(\mathsf{s})\}, \mathcal{D}_2 = \{x \leftarrow \{0,1\}^l : x\}$$

When $l = 2m$, we call this a length doubling PRG.

Oblivious Transfer (OT*)* OT is a secure 2-party protocol that realizes the functionality $\mathcal{F}_{\mathsf{OT}} : ((\mathsf{str}_0, \mathsf{str}_1), b) = (\perp, \mathsf{str}_b)$ where $\mathsf{str}_0, \mathsf{str}_1 \in \{0,1\}^k, b \in \{0,1\}$.

3 Oblivious Punctured Vector (OPV)

3.1 Definition and Security Properties

An Oblivious Punctured Vector (OPV) for domain $\mathbb{D}$ is an interactive protocol between two parties, P_0 and P_1, where parties' inputs are $((1^\lambda, \mathsf{n}), (1^\lambda, \mathsf{n}, i))$ and their outputs are $(\boldsymbol{v}_0, \boldsymbol{v}_1)$, respectively. Here λ is the security parameter that determines the running time of the protocol, $\boldsymbol{v}_b, b \in \{0,1\}$ are vectors of length n, $i \in [\mathsf{n}]$ and $\boldsymbol{v}_b \in [\mathbb{D}]^{\mathsf{n}}$.

This protocol lets the two parties jointly generate vector $\boldsymbol{v}$ with random-looking elements such that: 1) P_0 learns the whole vector $\boldsymbol{v}$ but doesn't learn index i. 2) P_1 learns all vector elements except for its i-th element $\boldsymbol{v}[i]$. So we define the protocol to be *correct* if $\boldsymbol{v}_1[j] = \boldsymbol{v}_0[j]\ \forall j \neq i$.

To capture the first property, we want to say that an adversarial P_0, who is given two distinct indies $i, i' \in [\mathsf{n}]$, $i \neq i'$ and participates in two executions of the protocol, one where party P_1 holds i, and the other, where P_1 holds i', cannot tell the two executions apart. We call this property *Position hiding*. To capture the second property, we want to say that an adversarial P_1, who, in addition to its view in the protocol execution, receives the vector $\boldsymbol{v}_0$, cannot differentiate between the two cases: when $\boldsymbol{v}_0$ is generated according to exec and when $\boldsymbol{v}_0$ is generated according to exec, then $\boldsymbol{v}_0[i]$ is replaced a random string from the domain. We call this security property *Value hiding*. We define the properties formally below.

Correctness For any sufficiently large security parameter $\lambda \in \mathbb{N}$, for any $\mathsf{n} \in \mathbb{N}, i \in [\mathsf{n}]$, if $(\boldsymbol{v}_0, \boldsymbol{v}_1) \leftarrow \mathsf{exec}^{\mathsf{OPV}}((\lambda \mathsf{n}), (\lambda, \mathsf{n}, i))$ and $\boldsymbol{v}_b \in [\mathbb{D}]^{\mathsf{n}}, b \in \{0,1\}$, then $\boldsymbol{v}_1[j] = \boldsymbol{v}_0[j]\ \forall j \neq i$.

Position hiding For any any sufficiently large security parameter $\lambda \in \mathbb{N}$, $\mathsf{n} \in \mathbb{N}, i, i' \in [\mathsf{n}]$, the following distributions are computationally indistinguishable:

$$\mathcal{D}_1 = \{(\boldsymbol{v}_0, \boldsymbol{v}_1) \leftarrow \mathsf{exec}^{\mathsf{OPV}}((1^\lambda, \mathsf{n}), (1^\lambda, \mathsf{n}, i)) : (1^\lambda, \mathsf{n}, i, i', \mathsf{view}_0)\}$$
$$\mathcal{D}_2 = \{(\boldsymbol{v}_0, \boldsymbol{v}_1) \leftarrow \mathsf{exec}^{\mathsf{OPV}}((1^\lambda, \mathsf{n}), (1^\lambda, \mathsf{n}, i')) : (1^\lambda, \mathsf{n}, i, i', \mathsf{view}_0)\}$$

Value hiding For any any sufficiently large security parameter $\lambda \in \mathbb{N}$, for any $\mathsf{n} \in \mathbb{N}, i \in [\mathsf{n}]$, the following distributions are computationally indistinguishable:

$$\mathcal{D}_1 = \{(\boldsymbol{v}_0, \boldsymbol{v}_1) \leftarrow \mathsf{exec}^{\mathsf{OPV}}((1^\lambda, \mathsf{n}), (1^\lambda, \mathsf{n}, i)) : (1^\lambda, \mathsf{n}, i, \boldsymbol{v}_0, \mathsf{view}_1)\}$$
$$\mathcal{D}_2 = \{((\boldsymbol{v}_0, \boldsymbol{v}_1) \leftarrow \mathsf{exec}^{\mathsf{OPV}}((1^\lambda, \mathsf{n}), (1^\lambda, \mathsf{n}, i)), \boldsymbol{v}_0[i] := r \text{ where } r \xleftarrow{\$} \mathbb{D} : (1^\lambda, \mathsf{n}, i, \boldsymbol{v}_0, \mathsf{view}_1)\}\}$$

3.2 Construction

To implement a OPV protocol for a domain $\mathbb{D}$, we first define two algorithms as follows.

$\mathsf{Setup}(1^\lambda, \mathsf{n}, i) \to (\mathsf{s}_0, \mathsf{s}_1)$**:** Setup is a PPT algorithm, that, given a security parameter λ, a vector length n and an index $i \in [\mathsf{n}]$, outputs a pair of seeds $(\mathsf{s}_0, \mathsf{s}_1)$, where $\mathsf{s}_0, \mathsf{s}_1 \in \{0,1\}^{\mathsf{poly}(\lambda)}$ and s_1 includes i.

$\mathsf{Expand}(b, \mathsf{s}_b) \to (\boldsymbol{v}_b)$**:** Expand is a polynomial time algorithm that, given a party index $b \in \{0,1\}$ and a seed s_b, outputs a vector $\boldsymbol{v}_b$ of length n, $\boldsymbol{v}_b \in [\mathbb{D}]^{\mathsf{n}}$.

We implement the algorithms as follows. First we give a 2 party protocol $\mathsf{OblivSetup}$ that realizes the functionality $\mathcal{F}_{[\mathbb{D}]}((\lambda, \mathsf{n}), (\lambda, \mathsf{n}, i)) = \mathsf{Setup}(1^\lambda, \mathsf{n}, i)$. We fix our domain $\mathbb{D}$ to strings of length λ, i.e., $\{0,1\}^\lambda$. Then we give the construction for Expand which P_0 and P_1 run non-interactively.

Given an OPV for $\mathbb{D}$ of strings of length λ, we can build an OPV for domain $\mathbb{D}'$, where $\mathbb{D}'$ is strings of length $l \geq \lambda$, in a blackbox way. We give this construction in Section 3.3.

G	Length doubling PRG
$i = \sigma_1\sigma_2 \dots \sigma_{\log \mathsf{n}}$	binary representation of input index i
$l = k_1 k_2, \dots k_j$	j-bit binary representation of l
$x^{j,l}$	l^{th} node from the left at level j in the tree, where $l \in [0, 2^j - 1], j \in [1, \log \mathsf{n}]$

Table 1. Notations

Setup:

- Pick $\mathsf{seed}_\epsilon \leftarrow \{0,1\}^m$. Let $\mathsf{seed}_0 \circ \mathsf{seed}_1 \leftarrow \mathsf{G}(\mathsf{seed}_\epsilon)$.
- For $l = 1, \dots, \log \mathsf{n} - 1$: $\mathsf{seed}_{\sigma_1 \dots \sigma_l 0} \circ \mathsf{seed}_{\sigma_1 \dots \sigma_l 1} \leftarrow \mathsf{G}(\mathsf{seed}_{\sigma_1 \dots \sigma_l})$.
- Set $\mathsf{s}_0 := (\mathsf{n}, \mathsf{seed}_\epsilon), \mathsf{s}_1 := (\mathsf{n}, i, \mathsf{seed}_{\overline{\sigma_1}}, \mathsf{seed}_{\sigma_1\overline{\sigma_2}}, \dots, \mathsf{seed}_{\sigma_1\sigma_2\dots\overline{\sigma_{\log \mathsf{n}}}})$ and output $(\mathsf{s}_0, \mathsf{s}_1)$.

OblivSetup: Let us assume both parties hold an implicit full binary tree and the levels of the tree are numbered as follows: root is at level 0 and leaves are at level $\log \mathsf{n}$. The protocol proceeds as follows:

1. Party P_0 picks $\mathsf{seed}_\epsilon \leftarrow \{0,1\}^\lambda$.

2. For $j = 1, \ldots, \log \mathsf{n}$: do the following:
$$\{x^{j,2l} \circ x^{j,2l+1}\}_{l\in[0,2^{j-1}-1]} \leftarrow \{\mathsf{G}(x^{j-1,l})\}_{l\in[0,2^{j-1}-1]}$$
$$\mathsf{str}^{j,0} \longleftarrow \bigoplus_{l\in[0,2^{j-1}-1]} x^{j,2l}, \quad \mathsf{str}^{j,1} \longleftarrow \bigoplus_{l\in[1,2^{j-1}-1]} x^{j,2l+1}$$
Note that $x^{j,l} = \mathsf{seed}_{k_1k_2,\ldots k_j}$.
3. For $j = 1, \ldots, \log \mathsf{n}$: P_0 and P_1 run $\mathsf{OT} : ((\mathsf{str}^{j,0}, \mathsf{str}^{j,1}), \sigma_j) = (\perp, \mathsf{str}^{j,\overline{\sigma_j}})$.
4. At the end of the OT phase P_1 locally expands the strings it received through OT to compute $\mathsf{seed}_{\overline{\sigma_1}}, \mathsf{seed}_{\sigma_1\overline{\sigma_2}}, \ldots, \mathsf{seed}_{\sigma_1\sigma_2\ldots\overline{\sigma_{\log \mathsf{n}}}}$. The expansion works as follow. For $j = 1, \ldots, \log \mathsf{n}$: P_1 has received, through the OT, $\mathsf{str}^{j,\overline{\sigma_j}}$. Note that $\mathsf{str}^{j,\overline{\sigma_j}}$ contains $\mathsf{seed}_{\sigma_1\sigma_2\ldots\overline{\sigma_{\log j}}}$ and P_1 can take off the extra terms by expanding the $2^{j-1} - 1$ seeds from the previous levels. More concretely,
$$\mathsf{seed}_{\sigma_1\sigma_2\ldots\overline{\sigma_j}} \leftarrow \mathsf{str}^{j,\overline{\sigma_j}} \bigoplus_{k_1,k_2,\ldots,k_{j-1}\in\{0,1\},k_j=\overline{\sigma_j}\wedge k_1k_2\ldots k_{j-1}\neq\sigma_1\sigma_2\ldots\sigma_{j-1}} \mathsf{seed}_{k_1k_2\ldots k_j}$$
5. At the end of this step, P_1 outputs $\mathsf{s}_0 := (\mathsf{n}, \mathsf{seed}_\epsilon)$ and P_1 outputs $\mathsf{s}_1 := (\mathsf{n}, i, \mathsf{seed}_{\overline{\sigma_1}}, \mathsf{seed}_{\sigma_1\overline{\sigma_2}}, \ldots, \mathsf{seed}_{\sigma_1\sigma_2\ldots\overline{\sigma_{\log \mathsf{n}}}})$.

Expand: For party b, construct $\boldsymbol{v}_b$ as follows.

$b = 0$**:** Parse s_0 as $(\mathsf{n}, \mathsf{seed}_\epsilon)$. Compute $\mathsf{seed}_0 \circ \mathsf{seed}_1 \leftarrow \mathsf{G}(\mathsf{seed}_\epsilon)$.
For $j = 1, \ldots, \log \mathsf{n}$: do the following: $\mathsf{seed}_{k_1k_2\ldots k_{j-1}k_j} \circ \mathsf{seed}_{k_1k_2\ldots k_{j-1}\overline{k_j}} \leftarrow \mathsf{G}(\mathsf{seed}_{k_1k_2\ldots k_{j-1}})$ for $k_1, \ldots, k_j \in \{0, 1\}$.
For $t \in [1, \mathsf{n}]$, set $\boldsymbol{v}_0[t] := \mathsf{seed}_{k_1k_2\ldots k_{\log \mathsf{n}}}$ where $t = k_1k_2 \ldots k_{\log \mathsf{n}}$, i.e., the binary representation of t.
$b = 1$**:** Parse s_1 as $(\mathsf{n}, i = \sigma_1 \ldots \sigma_{\log \mathsf{n}}, \mathsf{seed}_{\overline{\sigma_1}}, \mathsf{seed}_{\sigma_1\overline{\sigma_2}}, \ldots, \mathsf{seed}_{\sigma_1\sigma_2\ldots\overline{\sigma_{\log \mathsf{n}}}})$.
For $j = 2 \ldots, \log \mathsf{n}$, expand each of the seeds as follows:
$\mathsf{seed}_{k_1k_2\ldots k_{j-1}k_j} \circ \mathsf{seed}_{k_1k_2\ldots k_{j-1}\overline{k_j}} \leftarrow \mathsf{G}(\mathsf{seed}_{k_1k_2\ldots k_{j-1}})$ for $k_1, \ldots, k_j \in \{0, 1\} \wedge k_1k_2 \ldots k_{j-1} \neq \sigma_1\sigma_2 \ldots \sigma_{j-1}$.
For $t \in [1, \mathsf{n}] \wedge t \neq i$, set $\boldsymbol{v}_1[t] := \mathsf{seed}_{k_1k_2\ldots k_{\log \mathsf{n}}}$ where $t = k_1k_2 \ldots k_{\log \mathsf{n}}$, i.e., the binary representation of t. Set $\boldsymbol{v}_1[i] = \perp$.

Security Proof Correctness OPV according to Def 3.1 follows from the correctness of OT protocols. Now we will prove that our construction satisfies both position and value hiding. In order to prove that, we first prove some helper theorems.

Theorem 1. $\mathsf{OblivSetup}$ *securely realizes the ideal functionality* $\mathcal{F}_{[\mathbb{D}]}((\mathsf{n}),(\mathsf{n},i)) = \mathsf{Setup}(1^\lambda,\mathsf{n},i) = (\mathsf{s}_0,\mathsf{s}_1)$ *as per Definition 1.*

Proof. We first construct a simulator that works as follows:

If $b=0$ (i.e. P_0 is corrupt): $\mathsf{sim}(1^\lambda,0,\mathsf{n},\mathsf{s}_0)$ will parse s_0 as $\mathsf{n},\mathsf{seed}_\epsilon$. Then it will run the protocol steps to generate $\mathsf{str}^{j,0},\mathsf{str}^{j,1}$ for $j=1,\ldots,\log\mathsf{n}$ and simulate the view from the OTs with $\mathsf{sim}^{\mathsf{OT}}(1^\lambda,0,(\mathsf{str}^{j,0},\mathsf{str}^{j,1}),\perp)$.

If $b=1$ (i.e. P_1 is corrupt): $\mathsf{sim}(1^\lambda,1,(\mathsf{n},i),\mathsf{s}_1)$ will parse i as $i=\sigma_1\sigma_2\ldots\sigma_{\log\mathsf{n}}$ and s_1 as $(\mathsf{n},i,\mathsf{seed}_{\overline{\sigma_1}},\mathsf{seed}_{\sigma_1\overline{\sigma_2}},\ldots,\mathsf{seed}_{\sigma_1\sigma_2\ldots\overline{\sigma_{\log\mathsf{n}}}})$. It will simulate the view from the OTs with $\mathsf{sim}^{\mathsf{OT}}(1^\lambda,1,\sigma_j,\mathsf{str}^{j,\overline{\sigma_j}})$, where it generates $\mathsf{str}^{j,\overline{\sigma_j}}$ as follows:

$$\mathsf{seed}_{k_1k_2\ldots k_{j-1}k_j}\circ\mathsf{seed}_{k_1k_2\ldots k_{j-1}\overline{k_j}}\leftarrow\mathsf{G}(\mathsf{seed}_{k_1k_2\ldots k_{j-1}})$$

for $k_1,\ldots,k_j\in\{0,1\}\wedge k_1k_2\ldots k_{j-1}\neq\sigma_1\sigma_2\ldots\sigma_{j-1}$.

$$\mathsf{str}^{j,\overline{\sigma_j}}\leftarrow\mathsf{seed}_{\sigma_1\sigma_2\ldots\overline{\sigma_j}}$$

$$\bigoplus_{k_1,k_2,\ldots,k_{j-1}\in\{0,1\},k_j=\overline{\sigma_j}\wedge k_1k_2\ldots k_{j-1}\neq\sigma_1\sigma_2\ldots\sigma_{j-1}}\mathsf{seed}_{k_1k_2\ldots k_j}$$

We show that this simulator produces an ideal experiment that is indistinguishable from the real experiment. We start with the case where $b=0$ and show this through a series of games:

We define Game k as the following: In Game k, run the OT simulator $\mathsf{sim}^{\mathsf{OT}}(1^\lambda,0,(\mathsf{str}^{j,0},\mathsf{str}^{j,1}),\perp)$ for $j=0,\ldots,k$ and for $j=k+1,\ldots,\log\mathsf{n}$, run the OT protocol. Notice that Game 0 is identical to the real experiment and Game $\log\mathsf{n}$ is identical to the ideal experiment. Now, Games k and $k+1$ are computationally indistinguishable by the security of the OT protocol. Therefore for $b=0$ the simulator produces an ideal experiment that is computationally indistinguishable from the real experiment.

Now we look at the case where $b=1$ and proceed though a series of games as before. In Game k, run the OT simulator $\mathsf{sim}^{\mathsf{OT}}(1^\lambda,1,\sigma_j,\mathsf{str}^{j,\overline{\sigma_j}})$ for $j=0,\ldots,k$ and for $j=k+1,\ldots,\log\mathsf{n}$, run the OT protocol. Notice that Game 0 is identical to the real experiment and Game $\log\mathsf{n}$ is identical to the ideal experiment. Games k and $k+1$ are computationally indistinguishable by the security of the OT protocol. Therefore for $b=1$ the simulator produces an ideal experiment that is computationally indistinguishable from the real experiment.

Theorem 2. *Our scheme satisfies the following property: for any* $\mathsf{n}\in\mathbb{N},i,i'\in[\mathsf{n}]$*, the following distributions are computationally indistinguishable:*

$$\mathcal{D}_1=\{(\mathsf{s}_0,\mathsf{s}_1)\leftarrow\mathsf{Setup}(1^\lambda,\mathsf{n},i):(1^\lambda,\mathsf{n},i,i',\mathsf{s}_0)\}$$
$$\mathcal{D}_2=\{(\mathsf{s}_0,\mathsf{s}_1)\leftarrow\mathsf{Setup}(1^\lambda,\mathsf{n},i'):(1^\lambda,\mathsf{n},i,i',\mathsf{s}_0)\}$$

Proof. Since the seed $\mathsf{s}_0 = (\mathsf{n}, \mathsf{seed}_\epsilon \xleftarrow{\$} \{0,1\}^\lambda)$, it does not depend on i. Hence the two distributions are identical.

Theorem 3. *Our construction satisfies the following property: for any* $\mathsf{n} \in \mathbb{N}, i \in [\mathsf{n}]$*, the following distributions are computationally indistinguishable:*

$$\mathcal{D}_1 = \{(\mathsf{s}_0, \mathsf{s}_1) \leftarrow \mathsf{Setup}(1^\lambda, \mathsf{n}, i), \boldsymbol{v}_0 \leftarrow \mathsf{Expand}(0, \mathsf{s}_0) : (1^\lambda, \mathsf{n}, i, \boldsymbol{v}_0, \mathsf{s}_1)\}$$
$$\mathcal{D}_2 = \{(\mathsf{s}_0, \mathsf{s}_1) \leftarrow \mathsf{Setup}(1^\lambda, \mathsf{n}, i), \boldsymbol{v}_1 \leftarrow \mathsf{Expand}(1, \mathsf{s}_1),$$
$$\boldsymbol{v}_0[j] := \boldsymbol{v}_1[j]\ \forall j \neq i, \boldsymbol{v}_0[i] := r \text{ where } r \xleftarrow{\$} \mathbb{D} : (1^\lambda, \mathsf{n}, i, \boldsymbol{v}_0, \mathsf{s}_1)\}$$

Proof. We show that the two distributions are computationally indistinguishable through a series of distributions defined as follows:

H_0: $\mathcal{D}_1 = \{(\mathsf{s}_0, \mathsf{s}_1) \leftarrow \mathsf{Setup}(1^\lambda, \mathsf{n}, i), \boldsymbol{v}_0 \leftarrow \mathsf{Expand}(0, \mathsf{s}_0) : (1^\lambda, \mathsf{n}, i, \boldsymbol{v}_0, \mathsf{s}_1)\}$

H_1: Identical to the previous distribution except the following: In Setup, instead of generating seed_ϵ, set $\mathsf{seed}_{\sigma_1}, \mathsf{seed}_{\overline{\sigma_1}} \xleftarrow{\$} \{0,1\}^\lambda$. Run the rest of the protocol steps to generate all the leaves, set $\boldsymbol{v}_0$ and s_1.

H_k: Identical to the previous distribution except the following: In setup, set $\mathsf{seed}_{\sigma_1 \ldots \sigma_k}, \mathsf{seed}_{\sigma_1 \ldots \overline{\sigma_k}} \xleftarrow{\$} \{0,1\}^\lambda$ for $k = 2, \ldots, \log \mathsf{n}$. Run the rest of the protocol steps to generate all the leaves, set $\boldsymbol{v}_0$ and s_1.

$H'_{\log \mathsf{n}}$: Identical to $H_{\log \mathsf{n}}$ except the following. Instead of generating $\boldsymbol{v}_0$, run $\mathsf{Expand}(1, \mathsf{s}_1)$ to generate $\boldsymbol{v}_1$, set $\boldsymbol{v}_0[i] \xleftarrow{\$} \{0,1\}^\lambda$.

By the security of PRG, distributions H_k, H_{k+1} are identical for $k = 1, \ldots, \log \mathsf{n}$. Finally, distributions $H_{\log \mathsf{n}}$ and $H'_{\log \mathsf{n}}$ are identical.

Now we define another series of hybrid distributions as follows:

$G_{\log \mathsf{n}}$: This distribution is identical to $H'_{\log \mathsf{n}}$ except the following: compute $\mathsf{seed}_{\sigma_1 \ldots \sigma_{\log \mathsf{n} - 1}} \xleftarrow{\$} \{0,1\}^\lambda$

$$\mathsf{seed}_{\sigma_1 \ldots \sigma_{\log \mathsf{n}}} \circ \mathsf{seed}_{\sigma_1 \ldots \overline{\log \mathsf{n}}} \leftarrow \mathsf{G}(\mathsf{seed}_{\sigma_1 \ldots \sigma_{\log \mathsf{n} - 1}})$$

. Then replace $\mathsf{seed}_{\sigma_1 \ldots \sigma_{\log \mathsf{n}}} \xleftarrow{\$} \{0,1\}^\lambda$.

G_k: This distribution is identical to the previous, except the following: For $k = \log \mathsf{n} - 1, \ldots, 1$: Instead of setting $\mathsf{seed}_{\sigma_1 \ldots \sigma_k}, \mathsf{seed}_{\sigma_1 \ldots \overline{k}} \xleftarrow{\$} \{0,1\}^\lambda$, compute $\mathsf{seed}_{\sigma_1 \ldots \sigma_{k-1}} \xleftarrow{\$} \{0,1\}^\lambda$

$$\mathsf{seed}_{\sigma_1 \ldots \sigma_k} \circ \mathsf{seed}_{\sigma_1 \ldots \overline{k}} \leftarrow \mathsf{G}(\mathsf{seed}_{\sigma_1 \ldots \sigma_{k-1}})$$

G_0: This distribution is identical to the previous, except the following: Instead of generating $\mathsf{seed}_{\sigma_1}, \mathsf{seed}_{\overline{\sigma_1}} \xleftarrow{\$} \{0,1\}^\lambda$, generate $\mathsf{seed}_\epsilon \xleftarrow{\$} \{0,1\}^\lambda$ and set

$$\mathsf{seed}_{\sigma_1} \circ \mathsf{seed}_{\overline{\sigma_1}} \leftarrow \mathsf{G}(\mathsf{seed}_\epsilon)$$

This distribution is identical to $\mathcal{D}_2$.

For $k = 0, \ldots, \log \mathsf{n}$, distributions G_k and G_{k+1} are computationally indistinguishable from the security of PRG. It remains to show that $H'_{\log \mathsf{n}}$ and $G_{\log \mathsf{n}}$ are computationally indistinguishable as well.

To show this, we show that if there is a PPT distinguisher D that distinguishes $H'_{\log \mathsf{n}}$ and $G_{\log \mathsf{n}}$ with non-negligible probability, then we can use D to build a PPT distinguisher $\mathcal{A}$ that breaks PRG security with same advantage. $\mathcal{A}$ does the following: on input $w_1 \circ w_2 \in \{0,1\}^{2\lambda}$, chooses $w'_1 \xleftarrow{\$} \{0,1\}^{\lambda}$ and runs D with $w'_1 \circ w_2$. If $w_1 \circ w_2 \xleftarrow{\$} \{0,1\}^{2\lambda}$, then D exactly simulates game $H'_{\log \mathsf{n}}$, otherwise it simulates game $G_{\log \mathsf{n}}$. Now if D can distinguish $H'_{\log \mathsf{n}}$ and $G_{\log \mathsf{n}}$, then $\mathcal{A}$ can distinguish whether $w_1 \circ w_2$ is the output of a PRG or a truly random string immediately with the same advantage as D. Hence, $H'_{\log \mathsf{n}}$ and $G_{\log \mathsf{n}}$ are computationally indistinguishable.

Now we are ready to prove the main theorem.

Theorem 4. *Our construction satisfies position and value hiding as defined in Definition 3.1.*

Proof. Since our protocol satisfies Theorem 1 and Theorem 2, it implies that our construction satisfies position hiding. Since our protocol satisfies Theorem 1 and Theorem 3, it implies that our construction satisfies value hiding.

3.3 OPV construction for longer strings

Let $\mathsf{OPV}_{\mathbb{D}}$ denote the interactive protocol between two parties, P_0 and P_1, where parties' inputs are $((1^\lambda, \mathsf{n}), (1^\lambda, \mathsf{n}, i))$ and their outputs are $(\boldsymbol{v}_0, \boldsymbol{v}_1)$, where $\boldsymbol{v}_b \in [\mathbb{D}]^{\mathsf{n}}$ and $\mathbb{D}$ is strings of length λ. We construct $\mathsf{OPV}_{\mathbb{D}'}$ where $\mathbb{D}'$ is strings of length $l \geq \lambda$ using $\mathsf{OPV}_{\mathbb{D}}$ and a PRG $\mathsf{G} : \{0,1\}^\lambda \rightarrow \{0,1\}^l$ as follows.

- Run $(\boldsymbol{v}_0, \boldsymbol{v}_1) \leftarrow \mathsf{exec}^{\mathsf{OPV}_{\mathbb{D}}}((1^\lambda, \mathsf{n}), (1^\lambda, \mathsf{n}, i)$
- Party P_b, $b \in \{0,1\}$ does the following: for each $\boldsymbol{v}_b[j], j \in [1, \mathsf{n}]$, expand it to a l-bit string using $\mathsf{G}(\boldsymbol{v}_b[j])$, i.e., $\boldsymbol{v}'_b[j] \leftarrow \mathsf{G}(\boldsymbol{v}_b[j])$. P_b's output is $\boldsymbol{v}'_b$.

Theorem 5. *If $\mathsf{OPV}_{\mathbb{D}}$ satisfies correctness, position and value hiding as defined in Definition 3.1, and G is a secure PRG, then our construction for $\mathsf{OPV}'_{\mathbb{D}}$ satisfies correctness, position and value hiding as well.*

Proof. Correctness: By the correctness of $\mathsf{OPV}_{\mathbb{D}}$, $\boldsymbol{v}_0[j] = \boldsymbol{v}_1[j]$, $\forall j \neq i$. Therefore, by our construction, $\boldsymbol{v}'_0[j] = \boldsymbol{v}'_1[j]$, $\forall j \neq i$.

Position hiding: For the sake of contradiction, suppose not. Then, there exists a distinguisher D that breaks the position hiding property of $\mathsf{OPV}_{\mathbb{D}'}$. We use D to build a distinguisher $\mathcal{A}$ that breaks the position hiding property of $\mathsf{OPV}_{\mathbb{D}}$ as follows. $\mathcal{A}$ receives $(1^\lambda, \mathsf{n}, i, i', \mathsf{view}_0^{\mathsf{OPV}_{\mathbb{D}}})$ as input, where $\mathsf{view}_0^{\mathsf{OPV}_{\mathbb{D}}}$ contains $\boldsymbol{v}_0$. For every $\boldsymbol{v}_0[j], j \in [1, \mathsf{n}]$, $\mathcal{A}$ computes $\boldsymbol{v}'_0[j] = \mathsf{G}(\boldsymbol{v}_0[j])$. Then it constructs $\mathsf{view}_0^{\mathsf{OPV}_{\mathbb{D}'}}$, which is $\mathsf{view}_0^{\mathsf{OPV}_{\mathbb{D}}}$, augmented with $\boldsymbol{v}'_0[j]$. $\mathcal{A}$ forwards $(1^\lambda, \mathsf{n}, i, i', \mathsf{view}_0^{\mathsf{OPV}_{\mathbb{D}'}})$ to D. Thus, $\mathcal{A}$ directly inherits the success probability D.

Value hiding: Recall that we are trying to prove the following two distributions are computationally indistinguishable.

$$\mathcal{D}_1 = \{(\boldsymbol{v}_0', \boldsymbol{v}_1') \leftarrow \mathsf{exec}^{\mathsf{OPV}_{\mathbb{D}'}}((1^\lambda, \mathsf{n}), (1^\lambda, \mathsf{n}, i)) : (1^\lambda, \mathsf{n}, i, \boldsymbol{v}_0', \mathsf{view}_1^{\mathsf{OPV}_{\mathbb{D}'}})\}$$
$$\mathcal{D}_2 = \{((\boldsymbol{v}_0', \boldsymbol{v}_1') \leftarrow \mathsf{exec}^{\mathsf{OPV}_{\mathbb{D}'}}((1^\lambda, \mathsf{n}), (1^\lambda, \mathsf{n}, i)), \boldsymbol{v}_0'[i] := r \text{ where } r \xleftarrow{\$} \mathbb{D}' : (1^\lambda, \mathsf{n}, i, \boldsymbol{v}_0', \mathsf{view}_1^{\mathsf{OPV}_{\mathbb{D}'}})\}$$

The proof will proceed through a series of hybrid steps, as in the proof of Theorem 3. We define a series of distributions as follows.

H_0**:** $\mathcal{D}_1 = \{(\boldsymbol{v}_0', \boldsymbol{v}_1') \leftarrow \mathsf{exec}^{\mathsf{OPV}_{\mathbb{D}'}}((1^\lambda, \mathsf{n}), (1^\lambda, \mathsf{n}, i)) : (1^\lambda, \mathsf{n}, i, \boldsymbol{v}_0', \mathsf{view}_1^{\mathsf{OPV}_{\mathbb{D}'}})\}$

H_1**:** Identical to the previous distribution except the following: generate $(\boldsymbol{v}_0, \boldsymbol{v}_1) \leftarrow \mathsf{exec}^{\mathsf{OPV}_{\mathbb{D}}}((1^\lambda, \mathsf{n}), (1^\lambda, \mathsf{n}, i))$, then set $\boldsymbol{v}_0[i] := r$ where $r \xleftarrow{\$} \mathbb{D}$ and set $\boldsymbol{v}_0'[i] \leftarrow \mathsf{G}(\boldsymbol{v}_0'[i])$. By the value-hiding property of $\mathsf{OPV}_{\mathbb{D}}$, H_0, H_1 are identical.

H_2**:** Identical to the previous distribution except the following: instead of computing $\boldsymbol{v}_0'[i] \leftarrow \mathsf{G}(\boldsymbol{v}_0'[i])$, set $\boldsymbol{v}_0'[i] := r'$ where $r' \xleftarrow{\$} \mathbb{D}'$. By the security property of PRG, H_1, H_2 are identical. Note that distribution H_2 is identical to $\mathcal{D}_2$. So this concludes the proof of value hiding. □

4 Share Translation Protocol

4.1 Definition

Share Translation (ST) protocol with parameters (N, ℓ) is an interactive protocol between two parties, P_0 and P_1, where parties' inputs are $(\pi, \perp)$ and their outputs are $(\boldsymbol{\Delta}, (\boldsymbol{a}, \boldsymbol{b}))$, respectively. Here π is a permutation on N elements, and $\Delta, \boldsymbol{a}, \boldsymbol{b}$ are all vectors of N elements, where each element has size ℓ. The protocol should satisfy the following correctness and security guarantees:

Correctness: For each sufficiently large security parameter λ, for each $\pi \in S_N$, and for each r_0, r_1 of appropriate length, let $(\boldsymbol{\Delta}, (\boldsymbol{a}, \boldsymbol{b})) \leftarrow \mathsf{exec}^{\mathsf{ST}}(\lambda; \pi, \perp; r_0, r_1)$. Then it should hold that $\boldsymbol{\Delta} = \boldsymbol{b} - \pi(\boldsymbol{a})$.

This definition can be modified in a straightforward way for statistical or computational correctness.

Permutation hiding: For all sufficiently large λ it should hold that for all $\pi, \pi' \in S_N$,

$$\mathsf{view}_1^{\mathsf{ST}}(\lambda; \pi, \perp; r_0, r_1) \approx \mathsf{view}_1^{\mathsf{ST}}(\lambda; \pi', \perp; r_0, r_1),$$

where indistinguishability holds over uniformly chosen r_0, r_1.

Share hiding: For all sufficiently large λ it should hold that for any $\pi \in S_N$,

$$(\boldsymbol{a}, \boldsymbol{b}, \mathsf{view}_0^{\mathsf{ST}}(\lambda; \pi, \perp; r_0, r_1)) \approx (\boldsymbol{a}', \boldsymbol{b}', \mathsf{view}_0^{\mathsf{ST}}(\lambda; \pi, \perp; r_0, r_1)),$$

where $(\boldsymbol{\Delta}, \boldsymbol{a}, \boldsymbol{b}) = \mathsf{exec}_{\mathsf{ST}}(\lambda; \pi, \perp; r_0, r_1)$, $\boldsymbol{a}' \xleftarrow{\$} [2^\ell]^N$, $\boldsymbol{b}' = \boldsymbol{\Delta} + \pi(\boldsymbol{a}')$, and indistinguishability holds over uniformly chosen r_0, r_1.

4.2 Construction

We build Share Translation protocol out of Oblivious Punctured Vector (OPV) protocol. Let π be P_0's input in Share Translation protocol. The protocol proceeds as follows:

1. P_0 and P_1 run N executions of OPV protocol in parallel, where P_0 uses $\pi(i)$ as its input in execution i, for $i \in [N]$. Denote $\boldsymbol{v}'_i, \boldsymbol{v}_i$ to be the outputs of the OPV protocol in execution i, for parties P_0 and P_1, respectively, and denote $\boldsymbol{v}'_i[j], \boldsymbol{v}_i[j]$ to be j-th elements of these vectors.
2. For each $i \in [N]$ P_0 sets $\boldsymbol{\Delta}[i] \leftarrow \sum_{j \neq \pi(i)} \boldsymbol{v}'_i[j] - \sum_{j \neq i} \boldsymbol{v}'_j[\pi(i)]$. It sets its output to be $\boldsymbol{\Delta} = (\boldsymbol{\Delta}[1], \ldots, \boldsymbol{\Delta}[N])$.
3. For each $i \in [N]$ P_1 sets $\boldsymbol{b}_i \leftarrow \sum_j \boldsymbol{v}_i[j]$, $\boldsymbol{a}_i \leftarrow \sum_j \boldsymbol{v}_j[i]$. It sets $(\boldsymbol{a}, \boldsymbol{b})$ as its output, where $\boldsymbol{a} = (\boldsymbol{a}[1], \ldots, \boldsymbol{a}[N])$, $\boldsymbol{b} = (\boldsymbol{b}[1], \ldots, \boldsymbol{b}[N])$.

Theorem 6. *The construction described above satisfies correctness, permutation hiding and share hiding, assuming underlying OPV protocol satisfies correctness, value hiding and position hiding.*

Correctness. For any $i \in [N]$ we have

$$\boldsymbol{\Delta}_i = \sum_{j \neq \pi(i)} \boldsymbol{v}'_i[j] - \sum_{j \neq i} \boldsymbol{v}'_j[\pi(i)] \overset{(1)}{=} \sum_{j \neq \pi(i)} \boldsymbol{v}_i[j] - \sum_{j \neq i} \boldsymbol{v}_j[\pi(i)] \overset{(2)}{=}$$

$$\overset{(2)}{=} \sum_{j \in [N]} \boldsymbol{v}_i[j] - \sum_{j \in [N]} \boldsymbol{v}_j[\pi(i)] = \boldsymbol{b}_i - \boldsymbol{a}_{\pi(i)}.$$

Here (1) follows from correctness of the OPV protocol, and (2) holds since we add and subtract the same value $\boldsymbol{v}_i[\pi(i)]$. Note that computationally (resp., statistically, perfectly) correct OPV protocol results in computationally (resp., statistically, perfectly) correct ST protocol.

Permutation hiding. Recall that we need to show that for all $\pi_1, \pi_2 \in S_N$,

$$\mathsf{view}_1^{\mathsf{ST}}(\lambda; \pi, \perp; r_0, r_1) \approx \mathsf{view}_1^{\mathsf{ST}}(\lambda; \pi', \perp; r_0, r_1).$$

We show this indistinguishability in a sequence of hybrids $H_0, H_1, \ldots, H_N$, where:

- $H_0 = \mathsf{view}_1^{\mathsf{ST}}(\lambda; \pi, \perp; r_0, r_1)$, for uniformly chosen r_0, r_1,
- $H_N = \mathsf{view}_1^{\mathsf{ST}}(\lambda; \pi', \perp; r_0, r_1)$, for uniformly chosen r_0, r_1,
- For $1 \leq i < N$, $H_i = \mathsf{view}_1^{(i)}(\lambda; (\pi, \pi'), \perp; r_0, r_1)$, where $\mathsf{view}_1^{(i)}(\lambda; (\pi, \pi'), \perp; r_0, r_1)$ is a view of P_1 in the modified Share Translation protocol where party P_0 uses $\pi'(j)$ as its input in OPV executions $1 \leq j \leq i$ and $\pi(j)$ as its input in OPV executions $i < j \leq N$. r_0, r_1 are uniformly chosen.

We argue that for each $1 \leq i \leq N$ $H_i \approx H_{i-1}$ due to position-hiding property of the OPV protocol, and therefore $H_0 \approx H_N$.

Indeed, note that the only difference between H_i and H_{i-1} is that in i-th execution of OPV party P_0 uses input $\pi'(i)$ instead of $\pi(i)$. Therefore if some PPT adversary distinguishes between H_i and H_{i-1}, then we break position hiding of OPV as follows. Given the challenge in the OPV position hiding game $(\pi(i), \pi'(i), \mathsf{view}_1^{\mathsf{OPV}}(\lambda; x, \perp; r_0^{\mathsf{OPV}}, r_1^{\mathsf{OPV}}))$, where $r_0^{\mathsf{OPV}}, r_1^{\mathsf{OPV}}$ are uniformly chosen randomness of P_0 and P_1 in the OPV protocol, and $\mathsf{view}_1^{\mathsf{OPV}}$ is a view of P_1 in OPV protocol (which uses randomness $r_0^{\mathsf{OPV}}, r_1^{\mathsf{OPV}}$ and P_0's input x which is either $\pi(i)$ or $\pi'(i)$), we execute the rest $N-1$ OPV protocols honestly using uniform randomness for each party and setting P_0's input to $\pi'(j)$ (for executions $j < i$) and $\pi(j)$ (for executions $j > i$). Let $\boldsymbol{v}_j$, $j = 1, \ldots, N$, be the output of P_1 in j-th execution of OPV.

We give the adversary P_1's view in all N OPV executions (including $\mathsf{view}_1^{\mathsf{OPV}}(\lambda; x, \perp; r_0^{\mathsf{OPV}}, r_1^{\mathsf{OPV}})$ of i-th execution which we received as a challenge). Depending on whether challenge input x was $\pi(i)$ or $\pi'(i)$, the distribution the adversary sees is either H_{i-1} or H_i. Therefore, if the adversary distinguishes between the two distributions, we can break position hiding of OPV protocol with the same success probability.

Share hiding. Recall that we need to show that for any $\pi \in S_N$,

$$(\boldsymbol{a}, \boldsymbol{b}, \mathsf{view}_0^{\mathsf{ST}}(\lambda; \pi, \perp; r_0, r_1)) \approx (\boldsymbol{a}', \boldsymbol{b}', \mathsf{view}_0^{\mathsf{ST}}(\lambda; \pi, \perp; r_0, r_1)),$$

where $\boldsymbol{a}, \boldsymbol{b}$ are true shares produced by the protocol, and $\boldsymbol{a}', \boldsymbol{b}'$ are uniformly random, subject to $\boldsymbol{\Delta} = \boldsymbol{b} - \pi(\boldsymbol{a})$.

We show this indistinguishability in a sequence of hybrids $H_0, H_1, \ldots, H_N$, where:

- $H_0 = (\boldsymbol{a}, \boldsymbol{b}, \mathsf{view}_0^{\mathsf{ST}}(\lambda; \pi, \perp; r_0, r_1))$, for uniformly chosen r_0, r_1,
- $H_N = (\boldsymbol{a}', \boldsymbol{b}', \mathsf{view}_0^{\mathsf{ST}}(\lambda; \pi, \perp; r_0, r_1))$, for uniformly chosen $r_0, r_1, \boldsymbol{a}'$, and $\boldsymbol{b}' = \boldsymbol{\Delta} + \pi(\boldsymbol{a})$, where $(\boldsymbol{\Delta}, \boldsymbol{a}, \boldsymbol{b}) = \mathsf{exec}_{\mathsf{ST}}(\lambda; \pi, \perp; r_0, r_1)$,
- $H_i = (\boldsymbol{a}^{(i)}, \boldsymbol{b}^{(i)}, \mathsf{view}_0^{\mathsf{ST}}(\lambda; \pi, \perp; r_0, r_1))$, where $(\boldsymbol{\Delta}, \boldsymbol{a}, \boldsymbol{b}) = \mathsf{exec}_{\mathsf{ST}}(\lambda; \pi, \perp; r_0, r_1)$ is the output of the Share Translation protocol for random r_1, r_2, $\boldsymbol{a}^{(i)} = (\boldsymbol{a}_1^{(i)}, \ldots, \boldsymbol{a}_N^{(i)})$ is such that $\boldsymbol{a}_j^{(i)}$ is uniformly chosen for $1 \leq j \leq i$, $\boldsymbol{a}_j^{(i)} = \boldsymbol{a}_j$ for $i < j \leq N$, and $\boldsymbol{b}^{(i)} = \Delta + \pi(\boldsymbol{a}^{(i)})$.

We argue that for each $1 \leq i \leq N$ $H_i \approx H_{i-1}$, by reducing it to value hiding of OPV protocol. Indeed, note that the only difference between H_i and H_{i-1} is that $\boldsymbol{a}_i^{(i)}$ is generated uniformly at random, rather then set to the true output of the protocol. Therefore if some PPT adversary distinguishes between H_i and H_{i-1}, then we break security of OPV as follows. Assume we are given the challenge $(\boldsymbol{v}_i, \mathsf{view}_0^{\mathsf{OPV}}(\lambda; \pi(i), \perp; r_0^{\mathsf{OPV}}, r_1^{\mathsf{OPV}}))$, where $r_0^{\mathsf{OPV}}, r_1^{\mathsf{OPV}}$ are uniformly chosen randomness of P_0 and P_1 in the OPV protocol, and $\mathsf{view}_0^{\mathsf{OPV}}$ is a view of P_0 in OPV protocol (which uses randomness $r_0^{\mathsf{OPV}}, r_1^{\mathsf{OPV}}$ and P_0's input $\pi(i)$), and challenge $\boldsymbol{v}_i$ is either the true output of P_1, or the output of P_1 except that $\boldsymbol{v}_i[\pi(i)]$ is set to a uniform value. We execute the rest $N-1$ OPV protocols honestly using uniform randomness for each party and setting P_0's input to $\pi(j)$, for $j \neq i$. Let's denote the outputs of each OPV execution $j \neq i$ as $(\boldsymbol{v}_j, \boldsymbol{v}'_j)$.

Then we compute $\boldsymbol{a}^{(i)}$, $\boldsymbol{b}^{(i)}$ as follows:

- $\boldsymbol{b}^{(i)}[k] \leftarrow \sum_j \boldsymbol{v}_k[j]$, for each $k \in [N]$,
- $\boldsymbol{a}^{(i)}[k] \leftarrow \sum_j \boldsymbol{v}_j[k]$, for each $k \in [N]$,

Then we give the adversary $\boldsymbol{a}^{(i)}, \boldsymbol{b}^{(i)}$, and the views of party P_0 in all N OPV executions (including the challenge view $\mathsf{view}_0^{\mathsf{OPV}}(\lambda; \pi(i), \perp; r_0^{\mathsf{OPV}}, r_1^{\mathsf{OPV}})$ of i-th execution). Depending on whether challenge $\boldsymbol{v}_i[\pi(i)]$ was uniform or not, the distribution the adversary sees is either H_{i-1} or H_i.

5 Permute and Share and Secret Shared Shuffle

Here we will abuse notation a bit and use $\pi(\boldsymbol{x})$ for a permutation π and vector $\boldsymbol{x}$ to mean the permutation which produces $x_{\pi(1)}, ..., x_{\pi(N)}$.

We will use the Share Translation scheme we presented in the previous scheme to construct first a secure computation for permuting and secret sharing elements where one party chooses the permutation and the other the elements, and then a construction for a full secret shared shuffle.

5.1 Definitions

We consider the following functionality, which we call permute and share, in which one party provides as input a permutation, and the other party provides as input a set of elements, and the output is secret shares of the permuted elements:

$$\mathcal{F}_{\mathsf{Permute+Share}[N,\ell]}(\pi, \boldsymbol{x}) = (\boldsymbol{r}, \pi(\boldsymbol{x}) - \boldsymbol{r}), \text{ where } \boldsymbol{r} \xleftarrow{\$} [2^\ell]^N.$$

We can also consider the equivalent functionality when the permutation or the initial database is secret shared as input. (Here we consider a secret sharing of permutation π which consists of two permutations π_0, π_1 such that $\pi = \pi_0 \circ \pi_1$.)

Finally, we define the secret shared shuffle functionality:

$$\mathcal{F}_{\mathsf{SecretSharedShuffle}[N,\ell]}(\boldsymbol{x}_0, \boldsymbol{x}_1) = (\boldsymbol{r}, \pi(\boldsymbol{x}_0 + \boldsymbol{x}_1) - \boldsymbol{r}),$$

where $\boldsymbol{r} \xleftarrow{\$} [2^\ell]^N$ and π is a random permutation over N elements.

5.2 Permutation networks

Before we describe our $\mathsf{Permute+Share}$ construction, we briefly review the Benes permutation network for permutations on $N = 2^n$ elements.

The Benes network has $2\log N - 1$ layers each with $N/2$ 2-element permutations (each is either an identity permutation or a swap). Any permutation on N elements can be represented as a combination of these $N/2 * (2\log N - 1)$ 2 element permutations.

Specifically, if inputs are numbered with index $1...N$ where index i is expressed in binary as $\sigma_1...\sigma_n$, then the ι layer and the $2\log N - \iota$th layer contain permutations of pairs of elements with indices of the form $\sigma_1 \ldots \sigma_{\iota-1} 0 \sigma_{\iota+1}, \ldots \sigma_n$ and $\sigma_1 \ldots \sigma_{\iota-1} 1 \sigma_{\iota+1}, \ldots \sigma_n$.

Larger subpermutations For our application, we note that we can divide this network up into permutations on $T = 2^t$ bits each. The ith layer in this network corresponds to layers $it - (t-1), \ldots, it$ of the Benes network, where each permutation is applied to a group of elements of the form $\sigma_1 \ldots \sigma_{i(t-1)} x \sigma_{it+1} \ldots \sigma_n$ where x includes all t- bit strings. Finally, we note that the center $2t-1$ layers of the Benes network can be seen as a set of N/T permutations on T elements each. Thus, the total number of layers will be $\lceil 2\frac{n-t}{t} \rceil + 1 = 2\lceil \frac{n}{t} \rceil - 1$.

So, given any permutation π, we can reformulate it into choices for each of the 2-element permutations in the switching network, and then segment that into $d = 2\lceil \frac{n}{t} \rceil - 1$ layers of N/T T-element permutations. Call the resulting composite N-element permutation for the ith layer π_i, and call this $\pi_1 \ldots \pi_m$ the T-subpermutation representation of π.

5.3 Permute + Share from Share Translation

Let $\mathsf{ShareTrans}_T$ be a protocol satisfying the definition in Section 4 for permutations on T elements. We construct our permute and share protocol $\mathsf{Permute} + \mathsf{Share}$ using the permutation network described above as follows.

1. P_0 computes the T-subpermutation representation $\pi_1, \ldots, \pi_d$ of its input π.
2. For each layer i, the parties run N/T instances of $\mathsf{ShareTrans}_T$, with P_0 providing as input the N/T permutations making up π_i. (Note that all of these instances and layers can be run in parallel.) For each i, P_1 obtains $\boldsymbol{a}^{(i,1)}, \ldots, \boldsymbol{a}^{(i,N/T)}$ and $\boldsymbol{b}^{(i,1)}, \ldots, \boldsymbol{b}^{(i,N/T)}$. Call the combined vectors $\boldsymbol{a}^{(i)}$ and $\boldsymbol{b}^{(i)}$. Similarly, P_0 obtains $\boldsymbol{\Delta}^{(i,1)}, \ldots, \boldsymbol{\Delta}^{(i,N/T)}$, which we will call $\Delta^{(i)}$.
3. For each i, P_1 computes $\boldsymbol{\delta}^{(\boldsymbol{i})} = \boldsymbol{a}^{(i+1)} - \boldsymbol{b}^{(i)}$ and sends it to P_0. P_1 also sends $\boldsymbol{m} = \boldsymbol{x} + \boldsymbol{a}^{(1)}$, and samples and sends random $\boldsymbol{w}$. P_1 outputs $\boldsymbol{b} = \boldsymbol{w} - \boldsymbol{b}^{(d)}$
4. P_0 computes $\boldsymbol{\Delta} = \boldsymbol{\Delta}^{(d)} + \pi_d(\boldsymbol{\delta}^{(d-1)} + \boldsymbol{\Delta}^{(d-1)} + \pi_{d-1}(\boldsymbol{\delta}^{(d-2)} + \boldsymbol{\Delta}^{(d-2)} + \ldots. + \pi_2(\boldsymbol{\delta}^{(1)} + \boldsymbol{\Delta}^{(1)})$ and outputs $\pi(m) + \boldsymbol{\Delta} - \boldsymbol{w}$.

Theorem 7. *The construction described above is a* $\mathsf{Permute}$+Share *protocol secure against static semi-honest corruptions.*

Correctness By correctness of $\mathsf{ShareTrans}_T$, for all i $\boldsymbol{\Delta}^{(i)} = \boldsymbol{b}^{(i)} - \pi_i(\boldsymbol{a}^{(i)})$. This means that for all i, $\boldsymbol{\delta}^{(i)} + \boldsymbol{\Delta}^{(i)} = \boldsymbol{a}^{(i+1)} - \boldsymbol{b}^{(i)} + \boldsymbol{b}^{(i)} - \pi_i(\boldsymbol{a}^{(i)}) = \boldsymbol{a}^{(i+1)} - \pi_i(\boldsymbol{a}^{(i)})$.

Thus, the final $\boldsymbol{\Delta}$ produced by P_0 is

$$
\begin{aligned}
&\boldsymbol{\Delta}^{(d)} + \pi_d(\boldsymbol{\delta}^{(d-1)} + \boldsymbol{\Delta}^{(d-1)} + \pi_{d-1}(\boldsymbol{\delta}^{(d-2)} + \boldsymbol{\Delta}^{(d-2)} + \ldots + \pi_2(\boldsymbol{\delta}^{(1)} + \boldsymbol{\Delta}^{(1)}) \\
=&\boldsymbol{\Delta}^{(d)} + \pi_d(\boldsymbol{a}^{(d)} - \pi_{d-1}(\boldsymbol{a}^{(d-1)}) + \pi_{d-1}(\boldsymbol{a}^{(d-1)} - \pi_{d-2}(\boldsymbol{a}^{(d-2)}) + \ldots + \pi_2(\boldsymbol{a}^{(2)} - \pi_1 \boldsymbol{a}^{(1)}))) \\
=&\boldsymbol{\Delta}^{(d)} + \pi_d(\boldsymbol{a}^{(d)} - \pi_{d-1}(\ldots \pi_2(\pi_1 \boldsymbol{a}^{(1)}))) \\
=&\boldsymbol{b}^{(d)} - \pi_d(\boldsymbol{a}^{(d)}) + \pi_d(\boldsymbol{a}^{(d)} - \pi_{d-1}(\ldots \pi_2(\pi_1 \boldsymbol{a}^{(1)}))) \\
=&\boldsymbol{b}^{(d)} - \pi_d(\pi_{d-1}(\ldots \pi_2(\pi_1(\boldsymbol{a}^{(1)})))) \\
=&\boldsymbol{b}^{(d)} - \pi(\boldsymbol{a}^{(1)})
\end{aligned}
$$

The output for P_0, P_1 is:

$$\begin{aligned} &\pi(\boldsymbol{m}) + \boldsymbol{\Delta} - \boldsymbol{w}, && \boldsymbol{w} - \boldsymbol{b}^{(d)} \\ =&\pi(\boldsymbol{x} + \boldsymbol{a}^{(1)}) + \boldsymbol{\Delta} - \boldsymbol{w}, && \boldsymbol{w} - (\boldsymbol{\Delta} + \pi(\boldsymbol{a}^{(1)})) \\ =&\pi(\boldsymbol{x}) + \pi(\boldsymbol{a}^{1)}) + \boldsymbol{\Delta} - \boldsymbol{w}, && -\boldsymbol{\Delta} - \pi(\boldsymbol{a}^{(1)}) + \boldsymbol{w} \end{aligned}$$

If we let $\boldsymbol{r} = \pi(\boldsymbol{x}) + \pi(\boldsymbol{a}^{(1)}) + \boldsymbol{\Delta} - \boldsymbol{w}$, we see that this has the correct distribution.

Security. Our simulator behaves as follows: If $b = 0$ (i.e. P_0 is corrupt): $\mathsf{sim}(1^\lambda, 0, \pi, \boldsymbol{y}_0)$ will first generate the subpermutations for π as described above, and then internally run all of the $\mathsf{ShareTrans}_T$ protocols to obtain simulated view for P_0 and $\boldsymbol{a}^{(1)}, \ldots, \boldsymbol{a}^{(d)}, \boldsymbol{b}^{(1)}, \ldots, \boldsymbol{b}^{(d)}$. Let $\boldsymbol{\Delta}^{(1)}, \ldots, \boldsymbol{\Delta}^{(d)}$ be the corresponding values computed by P_0 in these protocols. Choose random $\boldsymbol{\delta}^{(1)}, \ldots, \boldsymbol{\delta}^{(d-1)}$. It then computes $\boldsymbol{\Delta}$ as in step 4 of the protocol and sets $\boldsymbol{w} = -\boldsymbol{y}_0 + \pi(\boldsymbol{m}) + \boldsymbol{\Delta}$. It outputs the views from the $\mathsf{ShareTrans}_T$ protocols and the messages $\boldsymbol{m}, \boldsymbol{w}, \boldsymbol{\delta}^{(1)}, \ldots, \boldsymbol{\delta}^{(d)}$.

If $b = 1$ (i.e. P_1 is corrupt): $\mathsf{sim}(1^\lambda, 1, \boldsymbol{x}, \boldsymbol{y}_1)$ will pick random π', compute the subpermutations, internally run the $\mathsf{ShareTrans}_T$ protocols with these permutations to obtain the views for P_1, and compute $\boldsymbol{b}^{(d)}$ from these runs as in the real protocol. It will set the random tape $\boldsymbol{w} = \boldsymbol{y}_1 + \boldsymbol{b}^{(d)}$. It outputs the view from the $\mathsf{ShareTrans}_T$ protocols and the random tape $\boldsymbol{w}$.

We show that this simulator produces an ideal experiment that is indistinguishable from the real experiment. We start with the case where $b = 0$ and show this through a series of games:

Real Game : Runs the real experiment. The output is P_0's view (its input the view_0s from the Share Translation protocols and the messages $\boldsymbol{m}$, $\boldsymbol{w}$, and $\boldsymbol{\delta}^{(1)}, \ldots, \boldsymbol{\delta}^{(d-1)}$ it receives), and the honest P_1's input $\boldsymbol{x}$ and output $\boldsymbol{w} - \boldsymbol{b}$.

Game 1: As in the previous game except in step 2, compute $\boldsymbol{\Delta}^{(i)}$ as $\boldsymbol{b}^{(i)} - \pi_i(\boldsymbol{a}^{(i)}$ instead of through the $\mathsf{ShareTrans}_T$ protocols. This is identical by correctness of Share Translation .

Game 2: As in the previous game except after step 2 for each i we sample random $\boldsymbol{a}'^{(i)}$ and compute $\boldsymbol{b}'^{(i)} = \pi_i(\boldsymbol{a}'^{(i)}) + \boldsymbol{\Delta}^{(i)}$, and then use these values in place of $\boldsymbol{a}^{(i)}, \boldsymbol{b}^{(i)}$ in steps 3 and 4.
We can show that this is indistinguishable via a series of hybrids, where in hybrid H_i, we use $\boldsymbol{a}'^{(j)}, \boldsymbol{b}'^{(j)}$ for the output of the first i $\mathsf{ShareTrans}_T$ protocols and $\boldsymbol{a}^{(j)}, \boldsymbol{b}^{(j)}$ for the rest. Then H_i, H_{i+1} are indistinguishable by the share hiding property of $\mathsf{ShareTrans}_T$.

Game 3: As above, but choose random $\boldsymbol{m}, \boldsymbol{\delta}^{(1)}, \ldots, \boldsymbol{\delta}^{(d-1)}$. Set $\boldsymbol{a}'^{(1)} = \boldsymbol{m} - \boldsymbol{x}$. For $i = 1 \ldots d$, compute $\boldsymbol{b}'^{(i)} = \pi_i(\boldsymbol{a}'^{(i)}) + \boldsymbol{\Delta}^{(i)}$ as above, and then set $\boldsymbol{a}'^{(i+1)} = \boldsymbol{\delta}^{(i)} - \boldsymbol{b}^{(i)}$. Note that this is distributed identically to Game 2.

Game Simulated: The only difference between the simulated game and Game 3 is that in Game 3, $\boldsymbol{w}$ is chosen at random, and P_1's output is computed as $\boldsymbol{w} - \boldsymbol{b}'^{(d)}$, while in Game Simulated, P_1's output is random $\boldsymbol{r}$ and $\boldsymbol{w}$ is set to $-\boldsymbol{y}_0 + \pi(\boldsymbol{m}) + \boldsymbol{\Delta} = -(\pi(\boldsymbol{x}) - \boldsymbol{r}) + \pi(\boldsymbol{m}) + \boldsymbol{\Delta} = \pi(\boldsymbol{a}'^{(1)}) + \boldsymbol{r} + \boldsymbol{\Delta} = \boldsymbol{b}'^{(d)} + \boldsymbol{r}$ by construction of $\boldsymbol{\Delta}$. Thus, the two games are identical.

We argue the case when $b = 1$ as follows:

Real Game : Runs the real experiment. The output is P_1's view (it's input $\boldsymbol{x}$, view_1 from the Share Translation protocol and the random string $\boldsymbol{w}$ it chooses) and the honest P_0's input π and output $\pi(\boldsymbol{m}) + \boldsymbol{\Delta} - \boldsymbol{w}$ where $\boldsymbol{\Delta}$ is as computed in step 4 of the protocol.

Game 1: As in the previous game, but P_0's output is $\pi(x)+\boldsymbol{b}^{(d)}-\boldsymbol{w}$. Note that $\pi(x) + \boldsymbol{b}^{(d)} - \boldsymbol{w} = \pi(\boldsymbol{x} + \boldsymbol{a}^{(1)}) + \boldsymbol{b}^{(d)} - \pi(\boldsymbol{a}^{(1)}) - \boldsymbol{w} = \pi(\boldsymbol{m}) + \boldsymbol{\Delta} - \boldsymbol{w}$ where $\boldsymbol{a}^{(1)}, \boldsymbol{b}^{(d)}$ are the values P_1 obtains from the first and last layer $\mathsf{ShareTrans}_t$ protocols.

Game 2: As in the previous game except run the $\mathsf{ShareTrans}_T$ protocols with $\pi'_1, \ldots, \pi'_d$ derived from a random permutation π'.

We can show that this is indistinguishable via a series of hybrids, where in hybrid H_i, we use the subpermutations derived from π' for the first i protocols, and the subpermutations derived from π for the rest. Then H_i, H_{i+1} are indistinguishable by the permutation hiding property of $\mathsf{ShareTrans}_T$.

Game Simulated: As in the previous game except choose random $\boldsymbol{r}$ and set $\boldsymbol{w} = \pi(\boldsymbol{x}) - \boldsymbol{r} + \boldsymbol{b}^{(d)}$. *This is identically distributed to Game 1 and identical to the ideal experiment.*

5.4 Secret Shared Shuffle from Permute+Share

The Secret Shared Shuffle protocol proceeds as follows:

0. P_0 and P_1 each choose a random permutation $\pi_0, \pi_1 \leftarrow S_N$.
1. P_0 and P_1 run the Permute+Share protocol to apply π_0 to $\boldsymbol{x}_1$, resulting in shares $\boldsymbol{x}_0^{(1)}$ for P_0 and $\boldsymbol{x}_1^{(1)}$ for P_1.
2. P_0 computes $\boldsymbol{x}_0^{(2)} = \pi_0(\boldsymbol{x}_0) + \boldsymbol{x}_0^{(1)}$.
3. P_1 and P_0 run the Permute + Share protocol to apply π_1 to $\boldsymbol{x}_0^{(2)}$, resulting in shares $\boldsymbol{x}_1^{(3)}$ for P_1 and $\boldsymbol{x}_0^{(3)}$ for P_0.
4. P_1 computes $\boldsymbol{x}_1^{(4)} = \pi_1(\boldsymbol{x}_1^{(1)}) + \boldsymbol{x}_1^{(3)}$.
5. P_0 outputs $\boldsymbol{x}_0^{(3)}$ and P_1 outputs $\boldsymbol{x}_1^{(4)}$.

Correctness. The output for P_0, P_1 is:

$$
\begin{aligned}
&\boldsymbol{x}_0^{(3)}, & \boldsymbol{x}_1^{(4)} \\
=&\boldsymbol{x}_0^{(3)}, & \pi_1(\boldsymbol{x}_1^{(1)}) + \boldsymbol{x}_1^{(3)} \\
=&\pi_1(\boldsymbol{x}_0^{(2)}) - \boldsymbol{r}^{(3)}, & \pi_1(\boldsymbol{x}_1^{(1)}) + \boldsymbol{r}^{(3)} \\
=&\pi_1(\pi_0(\boldsymbol{x}_0) + \boldsymbol{x}_0^{(1)}) - \boldsymbol{r}^{(3)}, & \pi_1(\boldsymbol{x}_1^{(1)}) + \boldsymbol{r}^{(3)} \\
=&\pi_1(\pi_0(\boldsymbol{x}_0) + \boldsymbol{r}^{(1)}) - \boldsymbol{r}^{(3)}, & \pi_1(\pi_0(\boldsymbol{x}_1) - \boldsymbol{r}^{(1)}) + \boldsymbol{r}^{(3)} \\
=&\pi_1(\pi_0(\boldsymbol{x}_0)) + \pi_1(\boldsymbol{r}^{(1)}) - \boldsymbol{r}^{(3)}, & \pi_1(\pi_0(\boldsymbol{x}_1)) - (\pi_1(\boldsymbol{r}^{(1)}) - \boldsymbol{r}^{(3)})
\end{aligned}
$$

Where $\boldsymbol{r}^{(1)}$ and $\boldsymbol{r}^{(3)}$ are the values generated by the first and second invocations of Permute+Share . If we let $\boldsymbol{r} = \pi_1(\pi_0(\boldsymbol{x}_0)) + \pi_1(\boldsymbol{r}^{(1)}) - \boldsymbol{r}^{(3)}$ and $\pi = \pi_1 \circ \pi_0$ we see that this has the correct distribution.

Security. Our simulator behaves as follows:

If $b = 0$ (i.e. P_0 is corrupt): $\mathsf{sim}(1^\lambda, 0, \boldsymbol{x}_0, \boldsymbol{y}_0)$ will choose random $\pi_0, \boldsymbol{x}_0^{(1)}$, set $\boldsymbol{x}_0^{(2)} = \pi_0(\boldsymbol{x}_0) + \boldsymbol{x}_0^{(1)}$, simulate the view from the first Permute+Share with $\mathsf{sim}^{\mathsf{Permute+Share}}(1^\lambda, 0, \pi_0, \boldsymbol{x}_0^{(1)})$, and simulate the view from the second Permute+Share with $\mathsf{sim}^{\mathsf{Permute+Share}}(1^\lambda, 1, \boldsymbol{x}_0^{(2)}, \boldsymbol{y}_0)$.

If $b = 1$ (i.e. P_1 is corrupt): $\mathsf{sim}(1^\lambda, 1, \boldsymbol{x}_1, \boldsymbol{y}_1)$ will choose random $\pi_1, \boldsymbol{x}_1^{(1)}$, set $\boldsymbol{x}_1^{(3)} = \boldsymbol{y}_1 - \pi_1(\boldsymbol{x}_1^{(1)})$, simulate the view from the first Permute+Share with $\mathsf{sim}^{\mathsf{Permute+Share}}(1^\lambda, 1, \boldsymbol{x}_1, \boldsymbol{x}_1^{(1)})$, and simulate the view from the second Permute+Share with $\mathsf{sim}^{\mathsf{Permute+Share}}(1^\lambda, 0, \pi_1, \boldsymbol{x}_1^{(3)})$.

The analysis showing that this simulator satisfies the security definition is straightforward and is deferred to the supplementary material.

6 Experimental Evaluation

In this section, we compare the solution for our Permute + Share with public key based solution and permutation network based solution. Recall Permute + Share primitive where party P_0 starts with a permutation π and party P_1 starts with a input vector $\boldsymbol{x}$. We define the domain of $\boldsymbol{x}, \boldsymbol{r}$ to be $\{0,1\}^L$ bit strings and $|\boldsymbol{x}| = N$. We first give the various solutions and then compare their performance in terms of communication and computation. Throughout this section, we do not report the cost of doing local XORs and base OTs since they are extremely fast and the cost is negligible compared to the cost of the rest of the protocol.

6.1 Public Key Encryption (PKE) based solution

Permute + Share can be implemented using a additively homomorphic public key encryption scheme such as Paillier [22]. Another alternative to using Paillier encryption, is to use El Gamal encryption [5] which provides multiplicative homomorphism. But using El Gamal encryption will result in multiplicative shares instead of additive and converting them to additive share introduces huge overhead and quickly makes the scheme unfeasible. We discuss both approaches here.

Pailler encryption based solution: Before getting into the the Permute + Share construction, let us recall $\mathsf{Pailler} = (\mathsf{Gen}, \mathsf{Enc}, \mathsf{Dec})$[22].

Key Generation: This algorithm consists of the following:

1. $n = pq$ where p, q are two large primes of equal length.
2. Define $\phi(n) = (p-1)(q-1)$.
3. Set $g = n + 1$ and $\mu = \phi(n)^{-1} \mod n$
4. Set $\mathsf{sk} = (p, q)$ and $\mathsf{pk} = (n, g)$.

Encryption: Let m be a message to be encrypted where $0 \leq m < n$. Select a random r where $0 < r < n$ and $r \in Z^*_{n^2}$. Compute ciphertext $c \leftarrow g^m r^n \mod n^2$. Let us denote this as $c = [m]$

Decryption: Given a ciphertext $c < n^2$, compute $m \leftarrow (L(c^{\phi(n)} \mod n^2) * \mu \mod n)$ where $L(u) = \frac{u-1}{n}$ for $u = 1 \mod n$.

We will be using the following properties of Pailler in our construction:

Homomorphism: The product of a ciphertext c with a plaintext m' raising g will decrypt to the sum of the corresponding plaintexts: $\text{Decrypt}([m] \cdot g^{m'} \mod n^2) = m + m' \mod n$.

Ciphertext Randomization: To randomize a ciphertext c, pick a random r' where $0 < r' < n$ and compute $c \cdot {r'}^n \mod n^2$.

Now let us define the $\mathsf{Permute} + \mathsf{Share}$ protocol using $\mathsf{Pailler}$. Let $(\mathsf{sk}, \mathsf{pk})$ be P_1's encryption keys. In the following we denote the component wise Hadamard product of two vectors a, b by $a \odot b$.

1. P_1 sends encrypted vector $\boldsymbol{x}$, denoted as $\boldsymbol{c} = [\boldsymbol{x}]$ to P_0.
2. P_0 picks a vector of random elements $\boldsymbol{r_1}$ where each element $e \in Z^*_{n^2}$ and $0 < e < n$ and randomizes the ciphertexts $\boldsymbol{c'} \leftarrow \boldsymbol{c} \odot \boldsymbol{r_1}^n \mod n^2$.
3. P_0 permutes $\boldsymbol{c'}$ to obtain $\boldsymbol{b} \leftarrow [\pi(\boldsymbol{x})]$
4. Then P_0 picks another vector of random elements $\boldsymbol{r_2}$ where each element $e \in Z^*_{n^2}$ is in $0 < e < n$ and computes $\boldsymbol{b} \cdot \mathbf{g}^{-r_2} \mod n^2$ and sends it back to P_1.
5. P_0's share is $\boldsymbol{r_2}$.
6. P_1 decrypts $[\pi(\boldsymbol{x}) - \boldsymbol{r_2}]$ to receive $\pi(\boldsymbol{x}) - \boldsymbol{r}_2$.

Cost: In this protocol, since every element of $\boldsymbol{x}$ has to be encrypted and the encryption message space in defined to be Z_n, therefore, each element has to be broken into blocks of size n for this protocol. This implies P_0 computes $N * \lceil L/n \rceil$ encryptions and P_1 computes $N * \lceil L/n \rceil$ ciphertext randomizations and ciphertext with plaintext multiplications. The communication for this protocol is $N * \lceil L/n \rceil * 2n$ bits. The protocol is 1 round.

El Gamal encryption based solution: Typically, $\mathsf{Pailler}$ requires 4096 bit primes for the modern standard of security, which is expensive. An alternate solution will be to use El Gamal Encryption [5] which provides multiplicative homomorphism. So, if we were to implement the above scheme using El Gamal encryption, P_0, P_1 will end up with multiplicative shares and will need to run a secure protocol (using Garbled Circuits) to convert from multiplicative to arithmetic shares. El Gamal can be implemented on Elliptic Curves with gives small parameters, typically, 256 bits. But this means, the multiplicative shares are Elliptic Curve point shares; converting EC point shares to arithmetic shares inside a GC is prohibitively expensive. Yet another solution will be to avoid using Elliptic Curves and use large finite fields, but this will require large parameters, typically 2000 bits or more, which will result in multiplicative shares over large finite fields. Converting them to arithmetic shares using a GC is also prohibitively expensive. So we rule out the possibility of using El Gamal.

6.2 Fixed key Block Ciphers

The symmetric key based protocols (ours and the one described in [21]) rely on two fundamental building blocks, namely, Oblivious Transfer extension

(OTe) [15] and GGM PRG [9]. Typically, published OTe protocols are based on a hash function that is modeled as a random oracle. However, in most of the recent implementations, the hash function is instantiated, somewhat haphazardly, using fixed key block ciphers (AES). In a recent work [12], the authors provided a principled way of implementing [15] using fixed key AES and formally proved that it is secure. The authors also propose that the length doubling PRG used in GGM [9] can be implemented using fixed key AES for better efficiency, though they do not prove it. Here, we first prove that it is safe to use this optimized PRG construction [12] and then use it in our experiments. In our experiments, we will also use the fixed-key AES based length extension technique for stretching short messages into longer ones (both for OTe and for OPV message length extension) described in Section 6.1 in [12].

The optimized PRG construction is based on correlation-robust hash (CRH) function [15, 12]. Roughly, the definition of CRH says that H is correlation-robust if the keyed function $f_R(x) = H(x \oplus R)$ is pseudorandom, given R is sufficiently random. Given a CRH H, the length doubling PRG is constructed as follows: $\mathsf{G}(x) = H(1 \oplus x) \circ H(2 \oplus x)$. We give more details in Appendix A.1.

In our experiments, we will use the following concrete instantiation of CRH [12]: $H(x) = \pi(x) \oplus x$ where $\pi(.)$ is a fixed key block cipher, such as AES.

6.3 OT extension costs

In our experiments, we simulate the cost of OT-extensions as follows. The cost is reported in number of fixed-key AES calls for sender and receiver and communication is reported in number of bits. For random OT's on strings of length $l > k = 128$ bits, we use IKNP OT-extension protocol with fixed-key AES optimization [12].The cost for m Random OTs on messages of length l bits are shown in Table 6.3, where the $2ml/k$ for sender and ml/k for receiver is for extending the random messages from k to l bits. We denote this functionality as ROT_l^m. For $l = k$, no message length extension is required (both for ROT and SOT). Fixed message OT's or standard OTs (SOT) are obtained from ROT by using the ROT messages as one-time pads for the actual messages. So SOT_l^m adds an additional $2ml$ bits of communication over ROT_l^m, i.e., the communication cost of SOT_l^m is $m(k+2l)$ bits. There is no additional computation overhead (except some additional XORs, which we ignore).

OT	Sender	Receiver	Communication (bits)
ROT_k^m	$3m$	$3m$	mk
ROT_l^m	$3m + 2ml/k$	$3m + ml/k$	mk
SOT_l^m	$3m + 2ml/k$	$3m + ml/k$	$m(k + 2l)$
SOT_k^m	$3m$	$3m$	$3mk$

6.4 Concrete Efficiency

In this section, we look at the concrete cost of our $\mathsf{Permute} + \mathsf{Share}$ protocol whose construction and compare it with this concrete cost of the $\mathsf{Permute} + \mathsf{Share}$ protocol of [21].

Our protocol: The compute cost of our $\mathsf{Permute} + \mathsf{Share}$ protocol is the compute cost of dN/T $\mathsf{ShareTrans}_T$ protocols, where $d = 2\lceil \log N / \log T \rceil - 1$. The communication cost includes the cost of dN/T $\mathsf{ShareTrans}_T$ protocols + $(d+1)Nl$ bits.

Each $\mathsf{ShareTrans}_T$ protocol requires $\mathsf{SOT}_k^{T \log T}$ and $T^2(2 + l/k)$ local fixed key AES calls (for both parties) which includes PRG calls in the GGM tree and message length extension and for the underlying OPV protocol. There is no additional communication over the cost of $\mathsf{SOT}_k^{T \log T}$.

Protocol from [21]: This $\mathsf{Permute} + \mathsf{Share}$ requires $\mathsf{SOT}_{2l}^{N \log N - N/2}$ and has an additional $2Nl$ bits communication overhead.

Benchmark: We use the *permute_ block* function in prp of [26] to benchmark the cost of a single fixed key AES-ECB 128 on 128 blocks (since we set the security parameter $k = 128$ for our experiments). To get this cost, we run fixed key AES for multiple number of blocks (4096, 8192, 12288) to get the amortized cost of a single AES. We repeat each experiment 100 times and the report the average amortized cost of a single AES call (no significant variance was noticeable). For estimating the cost of a single encryption and a single ciphertext randomization (for the Paillier based protocol in Section 6.1 we use the RSA signing cost for modulus of size 4096. We get this cost using the OpenSSL benchmark [6] by running the command *openssl speed*. The cost we get are the following: *AES-ECB 128:* 3.5 ns, *RSA 4096 signing* 0.17s. All the benchmarks are run a Macbook Pro 2017 with a 3.1 GHz Intel core i-7 processor and 16GB of 2133MHz LPDDR3 RAM.

6.5 Performance Comparison

Now we will simulate the performance of the different constructions described above. For this simulation, we experiment with two different database sizes, $N = 2^{20}$ and $N = 2^{32}$ elements. We vary the lenth of each element in the database from 640 bits to 64000 bits. This range of values is roughly inspired from Machine Learning training applications which has 100s to 1000s of features (with each feature represented by a 64 bit integer). We simulate the total running time on a WAN with bandwidth 9MB/s (we ignore the network latency since all these protocols ate 1-1.5 rounds), WAN with identical bandwidth was considered in [19] for experiments. These performance of our protocols are shown in Figure 3-Figure 4. We see that we are 3 orders of faster compared to Paillier based solution and one order of magnitude faster than [21].

References

1. Ajtai, M., Komlós, J., Szemerédi, E.: An o(n log n) sorting network. In: Proceedings of the 15th Annual ACM Symposium on Theory of Computing, 25-27 April, 1983, Boston, Massachusetts, USA. pp. 1–9 (1983)
2. Benes, V.E.: Optimal rearrangeable multistage connecting networks. The Bell System Technical Journal **43**(4), 1641–1656 (1964)

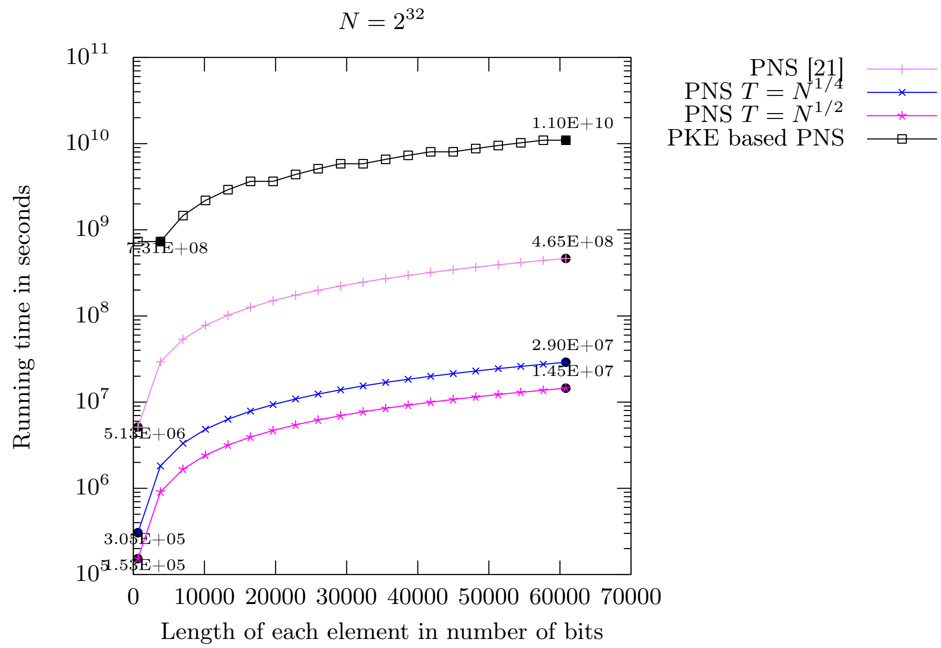

Fig. 3. Total running time for $N = 2^{32}$

3. Ciampi, M., Orlandi, C.: Combining private set-intersection with secure two-party computation. In: Security and Cryptography for Networks - 11th International Conference, SCN 2018, Amalfi, Italy, September 5-7, 2018, Proceedings. pp. 464–482 (2018)
4. Doerner, J., Shelat, A.: Scaling ORAM for secure computation. In: Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, CCS 2017, Dallas, TX, USA, October 30 - November 03, 2017. pp. 523–535 (2017)
5. ElGamal, T.: A public key cryptosystem and a signature scheme based on discrete logarithms. In: Blakley, G.R., Chaum, D. (eds.) Advances in Cryptology. pp. 10–18. Springer Berlin Heidelberg, Berlin, Heidelberg (1985)
6. Foundation, O.S.: OpenSSL. https://www.openssl.org/
7. Garg, S., Gupta, D., Miao, P., Pandey, O.: Secure multiparty RAM computation in constant rounds. In: Theory of Cryptography - 14th International Conference, TCC 2016-B, Beijing, China, October 31 - November 3, 2016, Proceedings, Part I. pp. 491–520 (2016)
8. Gentry, C., Halevi, S., Lu, S., Ostrovsky, R., Raykova, M., Wichs, D.: Garbled RAM revisited. In: Advances in Cryptology - EUROCRYPT 2014 - 33rd Annual International Conference on the Theory and Applications of Cryptographic Techniques, Copenhagen, Denmark, May 11-15, 2014. Proceedings. pp. 405–422 (2014), https://doi.org/10.1007/978-3-642-55220-5_23
9. Goldreich, O., Goldwasser, S., Micali, S.: How to construct random functions. J. ACM **33**(4), 792–807 (Aug 1986). https://doi.org/10.1145/6490.6503,

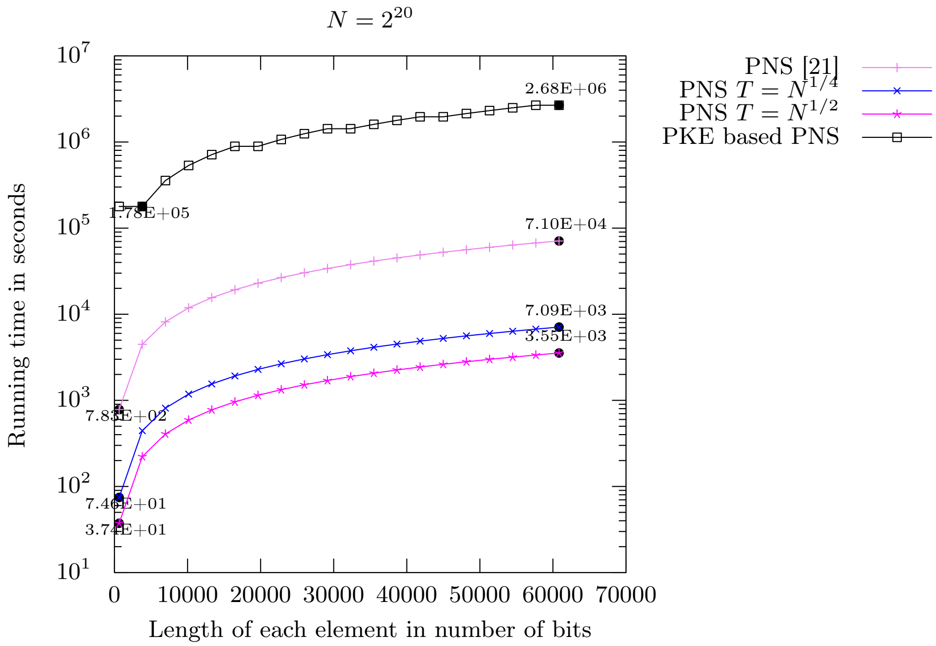

Fig. 4. Total running time for $N = 2^{20}$

http://doi.acm.org/10.1145/6490.6503
10. Gordon, S.D., Katz, J., Kolesnikov, V., Krell, F., Malkin, T., Raykova, M., Vahlis, Y.: Secure two-party computation in sublinear (amortized) time. In: the ACM Conference on Computer and Communications Security, CCS'12, Raleigh, NC, USA, October 16-18, 2012. pp. 513–524 (2012)
11. Group, B.C.: Bristol Cryptography Blog. http://bristolcrypto.blogspot.com/2017/01/rwc-2017-secure-mpc-at-google.html
12. Guo, C., Katz, J., Wang, X., Yu, Y.: Efficient and secure multiparty computation from fixed-key block ciphers. Cryptology ePrint Archive, Report 2019/074 (2019), https://eprint.iacr.org/2019/074
13. Hamada, K., Ikarashi, D., Chida, K., Takahashi, K.: Oblivious radix sort: An efficient sorting algorithm for practical secure multi-party computation. IACR Cryptology ePrint Archive **2014**, 121 (2014), http://eprint.iacr.org/2014/121
14. Huang, Y., Evans, D., Katz, J.: Private set intersection: Are garbled circuits better than custom protocols? In: NDSS (2012)
15. Ishai, Y., Kilian, J., Nissim, K., Petrank, E.: Extending oblivious transfers efficiently. In: Boneh, D. (ed.) Advances in Cryptology - CRYPTO 2003. pp. 145–161. Springer Berlin Heidelberg, Berlin, Heidelberg (2003)
16. Keller, M., Scholl, P.: Efficient, oblivious data structures for MPC. In: Advances in Cryptology - ASIACRYPT 2014 - 20th International Conference on the Theory and Application of Cryptology and Information Security, Kaoshiung, Taiwan, R.O.C., December 7-11, 2014, Proceedings, Part II. pp. 506–525 (2014)

17. Liu, C., Huang, Y., Shi, E., Katz, J., Hicks, M.W.: Automating efficient ram-model secure computation. In: 2014 IEEE Symposium on Security and Privacy, SP 2014, Berkeley, CA, USA, May 18-21, 2014. pp. 623–638 (2014)
18. Liu, C., Wang, X.S., Nayak, K., Huang, Y., Shi, E.: Oblivm: A programming framework for secure computation. In: 2015 IEEE Symposium on Security and Privacy, SP 2015, San Jose, CA, USA, May 17-21, 2015. pp. 359–376 (2015)
19. Mohassel, P., Zhang, Y.: Secureml: A system for scalable privacy-preserving machine learning. In: 2017 IEEE Symposium on Security and Privacy (SP). pp. 19–38 (May 2017). https://doi.org/10.1109/SP.2017.12
20. Mohassel, P., Rindal, P., Rosulek, M.: Fast database joins for secret shared data. Cryptology ePrint Archive, Report 2019/518 (2019), https://eprint.iacr.org/2019/518
21. Mohassel, P., Sadeghian, S.: How to hide circuits in mpc an efficient framework for private function evaluation. In: Johansson, T., Nguyen, P.Q. (eds.) Advances in Cryptology – EUROCRYPT 2013. pp. 557–574. Springer Berlin Heidelberg, Berlin, Heidelberg (2013)
22. Paillier, P.: Public-key cryptosystems based on composite degree residuosity classes. In: Stern, J. (ed.) Advances in Cryptology — EUROCRYPT '99. pp. 223–238. Springer Berlin Heidelberg, Berlin, Heidelberg (1999)
23. Pinkas, B., Schneider, T., Tkachenko, O., Yanai, A.: Efficient circuit-based psi with linear communication. Cryptology ePrint Archive, Report 2019/241 (2019), https://eprint.iacr.org/2019/241
24. Pinkas, B., Schneider, T., Weinert, C., Wieder, U.: Efficient circuit-based PSI via cuckoo hashing. In: Advances in Cryptology - EUROCRYPT 2018 - 37th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Tel Aviv, Israel, April 29 - May 3, 2018 Proceedings, Part III. pp. 125–157 (2018). https://doi.org/10.1007/978-3-319-78372-7_5, https://doi.org/10.1007/978-3-319-78372-7_5
25. Wang, X., Gordon, S.D., McIntosh, A., Katz, J.: Secure computation of MIPS machine code. In: Computer Security - ESORICS 2016 - 21st European Symposium on Research in Computer Security, Heraklion, Greece, September 26-30, 2016, Proceedings, Part II. pp. 99–117 (2016)
26. Wang, X., Malozemoff, A.J., Katz, J.: EMP-toolkit: Efficient MultiParty computation toolkit. https://github.com/emp-toolkit (2016)
27. Zahur, S., Wang, X., Raykova, M., Gascón, A., Doerner, J., Evans, D., Katz, J.: Revisiting square-root ORAM: efficient random access in multi-party computation. In: IEEE Symposium on Security and Privacy, SP 2016, San Jose, CA, USA, May 22-26, 2016. pp. 218–234 (2016)

A Appendix

Here we show security of our secret shared shuffle protocol from section 5.4.

Security. Our simulator behaves as follows:

If $b = 0$ (i.e. P_0 is corrupt): $\mathsf{sim}(1^\lambda, 0, \boldsymbol{x}_0, \boldsymbol{y}_0)$ will choose random $\pi_0, \boldsymbol{x}_0^{(1)}$, set $\boldsymbol{x}_0^{(2)} = \pi_0(\boldsymbol{x}_0) + \boldsymbol{x}_0^{(1)}$, simulate the view from the first Permute+Share with $\mathsf{sim}^{\mathsf{Permute+Share}}(1^\lambda, 0, \pi_0, \boldsymbol{x}_0^{(1)})$, and simulate the view from the second Permute+Share with $\mathsf{sim}^{\mathsf{Permute+Share}}(1^\lambda, 1, \boldsymbol{x}_0^{(2)}, \boldsymbol{y}_0)$.

If $b = 1$ (i.e. P_1 is corrupt): $\mathsf{sim}(1^\lambda, 1, \boldsymbol{x}_1, \boldsymbol{y}_1)$ will choose random $\pi_1, \boldsymbol{x}_1^{(1)}$, set $\boldsymbol{x}_1^{(3)} = \boldsymbol{y}_1 - \pi_1(\boldsymbol{x}_1^{(1)})$, simulate the view from the first $\mathsf{Permute{+}Share}$ with $\mathsf{sim}^{\mathsf{Permute+Share}}(1^\lambda, 1, \boldsymbol{x}_1, \boldsymbol{x}_1^{(1)})$, and simulate the view from the second $\mathsf{Permute{+}Share}$ with $\mathsf{sim}^{\mathsf{Permute+Share}}(1^\lambda, 0, \pi_1, \boldsymbol{x}_1^{(3)})$.

We show that this simulator produces an ideal experiment that is indistinguishable from the real experiment. We start with the case where $b = 0$ and show this through a series of games:

Real Game : Runs the real experiment.

The output is P_0's view (its input $\boldsymbol{x}_0$, $\mathsf{view}_0^{(1)}, \mathsf{view}_0^{(2)}$ from the two $\mathsf{Permute{+}Share}$ protocols including the outputs $\boldsymbol{x}_0^{(1)}, \boldsymbol{x}_0^{(3)}$, and the honest P_1's input $\boldsymbol{x}_1$ and output $\boldsymbol{x}_1^{(4)} = \pi_1(\boldsymbol{x}_1^{(1)}) + \boldsymbol{x}_1^{(3)}$

Game 1 : In step 1, first compute $\mathcal{F}_{\mathsf{Permute+Share}}(\pi_0, \boldsymbol{x}_1)$, i.e. choose random $\boldsymbol{r}^{(1)}$, and set $\boldsymbol{x}_0^{(1)} = \boldsymbol{r}^{(1)}$ and $\boldsymbol{x}_1^{(1)} = \pi_0(\boldsymbol{x}_1) - \boldsymbol{r}^{(1)}$. Then run the $\mathsf{Permute{+}Share}$ simulator to generate the view $\mathsf{view}_0^{(1)\prime}$ for the first $\mathsf{Permute{+}Share}$.

The output is P_0's view (its input $\boldsymbol{x}_0$, $\mathsf{view}_0^{(1)\prime}, \mathsf{view}_0^{(2)}$ from the two $\mathsf{Permute{+}Share}$ protocols including its outputs from those protocols $\boldsymbol{x}_0^{(1)} = \boldsymbol{r}^{(1)}$ and $\boldsymbol{x}_0^{(3)}$), and the honest P_1's input $\boldsymbol{x}_1$ and output $\boldsymbol{x}_1^{(4)} = \pi_1(\boldsymbol{x}_1^{(1)}) + \boldsymbol{x}_1^{(3)} = \pi_1(\pi_0(\boldsymbol{x}_1) - \boldsymbol{r}^{(1)}) + \boldsymbol{x}_1^{(3)}$.

This is indistinguishable by security of the $\mathsf{Permute{+}Share}$ *protocol.*

Game 2 : In step 3, first compute $\mathcal{F}_{\mathsf{Permute+Share}}(\pi_1, \boldsymbol{x}_0^{(2)})$, i.e. choose random $\boldsymbol{r}^{(3)}$ and set $\boldsymbol{x}_1^{(3)} = \boldsymbol{r}^{(3)}$ and $\boldsymbol{x}_0^{(3)} = \pi_1(\boldsymbol{x}_0^{(2)}) - \boldsymbol{r}^{(3)}$. Then run the $\mathsf{Permute{+}Share}$ simulator to generate the view $\mathsf{view}_0^{(2)}$ for the second $\mathsf{Permute{+}Share}$.

The output is P_0's view (its input $\boldsymbol{x}_0$, $\mathsf{view}_0^{(1)\prime}, \mathsf{view}_0^{(2)\prime}$ from the two $\mathsf{Permute{+}Share}$ protocols including its outputs from those protocols $\boldsymbol{x}_0^{(1)} = \boldsymbol{r}^{(1)}$ and $\boldsymbol{x}_0^{(3)} = \pi_1(\boldsymbol{x}_0^{(2)}) - \boldsymbol{r}^{(3)}$), and the honest P_1's input $\boldsymbol{x}_1$ and output $\boldsymbol{x}_1^{(4)} = \pi_1(\pi_0(\boldsymbol{x}_1) - \boldsymbol{r}^{(1)}) + \boldsymbol{x}_1^{(3)} = \pi_1(\pi_0(\boldsymbol{x}_1) - \boldsymbol{r}^{(1)}) + \boldsymbol{r}^{(3)}$.

This is again indistinguishable by security of the $\mathsf{Permute{+}Share}$ *protocol.*

Game 3 : Choose random $\pi, \boldsymbol{r}, \boldsymbol{x}_0^{(1)}$. Set $\pi_1 = \pi \circ \pi_0^{-1}$, $\boldsymbol{r}^{(1)} = \boldsymbol{x}_0^{(1)}$ and $\boldsymbol{r}^{(3)} = \pi_1(\pi_0(\boldsymbol{x}_0)) + \pi_1(\boldsymbol{r}^{(1)}) - \boldsymbol{r}$. Other than that, proceed as in Game 2.

The output is P_0's view (its input $\boldsymbol{x}_0$, $\mathsf{view}_0^{(1)\prime}, \mathsf{view}_0^{(2)\prime}$ from the two $\mathsf{Permute{+}Share}$ protocols including its outputs from those protocols $\boldsymbol{x}_0^{(1)} = \boldsymbol{r}^{(1)}$ and $\boldsymbol{x}_0^{(3)}$), and the honest P_1's input $\boldsymbol{x}_1$ and output $\boldsymbol{x}^{(4)}$).

This is identically distributed to Game 2. Note also that P_1's output in this game is

$$\begin{aligned}
\boldsymbol{x}_1^{(4)} =&\pi_1(\boldsymbol{x}_1^{(1)}) + \boldsymbol{x}_1^{(3)}\\
=&\pi_1(\boldsymbol{x}_1^{(1)}) + \pi_1(\boldsymbol{x}_0^{(2)}) - \boldsymbol{x}_0^{(3)}\\
=&\pi_1(\boldsymbol{x}_1^{(1)}) + \pi_1(\pi_0(\boldsymbol{x}_0) + \boldsymbol{x}_0^{(1)}) - \boldsymbol{x}_0^{(3)}\\
=&\pi_1(\pi_0(\boldsymbol{x}_1) - \boldsymbol{x}_0^{(1)}) + \pi_1(\pi_0(\boldsymbol{x}_0) + \boldsymbol{x}_0^{(1)}) - \boldsymbol{x}_0^{(3)}\\
=&\pi_1(\pi_0(\boldsymbol{x}_1 + \boldsymbol{x}_0)) - \boldsymbol{x}_0^{(3)}\\
=&\pi(\boldsymbol{x}_1 + \boldsymbol{x}_0) - \boldsymbol{x}_0^{(3)}
\end{aligned}$$

Thus, this is identical to the ideal experiment.

Next, we turn to the case where $b = 1$.

Real Game : Runs the real experiment

Game 1 : In step 1, first compute $\mathcal{F}_{\mathsf{Permute+Share}}(\pi_0, \boldsymbol{x}_1)$, i.e. choose random $\boldsymbol{x}_0^{(1)}$, and then compute $\boldsymbol{x}_1^{(1)} = \pi_0(\boldsymbol{x}_1) - \boldsymbol{x}_0^{(1)}$. Then run the Permute+Share simulator to generate the view for the first Permute+Share . *This is indistinguishable by security of the* Permute+Share *protocol.*

Game 2 : In step 3, first compute $\mathcal{F}_{\mathsf{Permute+Share}}(\pi_1, \boldsymbol{x}_0^{(2)})$, i.e. choose random $\boldsymbol{x}_1^{(3)}$, and then compute $\boldsymbol{x}_0^{(3)} = \pi_1(\boldsymbol{x}_0^{(2)}) - \boldsymbol{x}_1^{(3)}$. Then run the Permute+Share simulator to generate the view for the second Permute+Share . *This is again indistinguishable by security of the* Permute+Share *protocol.*

Game 3 : Choose random $\boldsymbol{x}_0^{(3)}$. Set $\boldsymbol{x}_1^{(3)} = \pi_1(\boldsymbol{x}_0^{(2)}) - \boldsymbol{x}_0^{(3)}$. Other than that, proceed as in Game 2. *This is identically distributed to Game 2.*

Game 4: Choose random π, set $\pi_0 = \pi_1^{-1} \circ \pi$ and set $\boldsymbol{x}_1^{(3)} = \pi(\boldsymbol{x}_0 + \boldsymbol{x}_1) - \pi_1(\boldsymbol{x}_1^{(1)}) - \boldsymbol{x}_0^{(3)}$. *Note that this means $\boldsymbol{x}_1^{(4)} = \pi(\boldsymbol{x}_0 + \boldsymbol{x}_1) - \boldsymbol{x}_0^{(3)}$ so this is distributed identically to the ideal experiment. Note also that this is distributed identically to Game 3, because:*

$$\begin{aligned}
&\pi_1(\boldsymbol{x}_0^{(2)}) - \boldsymbol{x}_0^{(3)}\\
=&\pi_1(\pi_0(\boldsymbol{x}_0) + \boldsymbol{x}_0^{(1)}) - \boldsymbol{x}_0^{(3)}\\
=&\pi_1(\pi_0(\boldsymbol{x}_0) + \pi_0(\boldsymbol{x}_1) - \boldsymbol{x}_1^{(1)}) - \boldsymbol{x}_0^{(3)}\\
=&\pi_1(\pi_0(\boldsymbol{x}_0 + \boldsymbol{x}_1)) - \pi_1(\boldsymbol{x}_1^{(1)}) - \boldsymbol{x}_0^{(3)}\\
=&\pi(\boldsymbol{x}_0 + \boldsymbol{x}_1) - \pi_1(\boldsymbol{x}_1^{(1)}) - \boldsymbol{x}_0^{(3)}
\end{aligned}$$

A.1 Fixed-key blockcipher

In this section we give more details about the definition and security of primitives from section 6.2.

Definition 2. *[12] Let $H : \{0,1\}^\lambda \to \{0,1\}^\lambda$ be a function and $R \in \{0,1\}^\lambda$. Define $\mathcal{O}_R(x) = H(x \oplus R)$. Let F_λ denote the set of all functions from $\{0,1\}^\lambda \to \{0,1\}^\lambda$ and f be randomly picked from F_λ. For a distinguisher D and for any sufficiently large $\lambda \in \mathbb{N}$, let*

$$Adv_{H,\mathcal{R}}(D) = |Pr_{R \leftarrow \{0,1\}^\lambda}[D^{\mathcal{O}_R(.)}(1^\lambda) = 1] - Pr_{f \leftarrow \mathsf{F_k}}[D^{f(.)}(1^\lambda) = 1]|$$

H is CRH *if, for any PPT D making at most q queries to $\mathcal{O}_R(.)$, there exists a negligible function* negl *such that $Adv_{H,\mathcal{R}}(D) \leq \mathsf{negl}(\lambda)$ where q is polynomial in λ.*

We note that, [12] defined a more general definition where R is picked from a distribution with sufficient min-entropy (at least λ), but this definition suffices for our purpose. Now, we are ready to prove the following theorem.

Theorem 8. *if H is a correlation-robust hash function (*CRH*,) then $\mathsf{G}(x)$ defined as $\mathsf{G}(x) = H(1 \oplus x) \circ H(2 \oplus x)$ is a length doubling PRG.*

Proof. For the sake of contradiction, suppose not. Then there exists a PPT distinguisher D that can break the PRG security game with overwhelming advantage. We will use D to construct a distinguisher D' that can win the CRH game in Definition 2 with the same advantage. D' functions as follows. It invokes its own oracle with messages 1 and 2 to get strings w_1, w_2 respectively. Then it constructs $w_1 \circ w_2$ and sends it to D as the PRG challenge. It outputs D's guess bit as its output, thereby inheriting its success probability. Note that, in this reduction, D' implicitly uses the fixed R as the PRG seed, even though it does not know it. This concludes the proof. □