Efficient Cloud-based Secret Shuffling via Homomorphic Encryption

Kilian Becher
Chair of Privacy and Data Security
TU Dresden
Dresden, Germany
kilian.becher@tu-dresden.de

Thorsten Strufe
Chair of IT Security
Karlsruhe Institute of Technology
Karlsruhe, Germany
thorsten.strufe@kit.edu

***Abstract*—When working with joint collections of confidential data from multiple sources, e.g., in cloud-based multi-party computation scenarios, the ownership relation between data providers and their inputs itself is confidential information. Protecting data providers' privacy desires a function for secretly shuffling the data collection. We present the first efficient secure multi-party computation protocol for secret shuffling in scenarios with a central server. Based on a novel approach to random index distribution, our solution enables the randomization of the order of a sequence of encrypted data such that no observer can map between elements of the original sequence and the shuffled sequence with probability better than guessing. It allows for shuffling data encrypted under an additively homomorphic cryptosystem with constant round complexity and linear computational complexity. Being a general-purpose protocol, it is of relevance for a variety of practical use cases.**

***Index Terms*—Privacy-preserving computation, secure multi-party computation, homomorphic encryption, secret shuffling**

I. Introduction

In an industrial context, security against semi-honest adversaries [26] is a valid assumption as companies typically have a financial and legal interest in the correct execution of processes. Proactive misbehaviour or negligent data handling could lead to a loss of reputation or legal consequences, such as those imposed by the European Union's General Data Protection Regulation (GDPR) [30].

To make well-informed business decisions, companies need to determine their strengths and weaknesses. One widely-used measure is cross-company benchmarking. In cross-company benchmarking, companies compare their key performance indicators (KPI), e.g., return on investment, to those of other companies of the same industry. As results, they obtain statistical measures, such as quartiles and mean. To compute rank-based statistical measures like quartiles, sorting KPIs typically is an important aspect of benchmarking. However, as the companies' performances are confidential, no company should learn another company's KPIs. Instead, benchmark results should only help them determine how they perform relatively to their overall peer group. To ensure that, benchmarks typically are performed by trusted third parties (TTP), neutral companies that take the companies' KPIs in plaintext and centrally compute the statistical measures. However, using a TTP requires trust. On the one hand, companies need to trust that the TTP does not proactively abuse the companies' private KPIs. As described above, this is a valid assumption as the neutral party has a financial and legal interest in honest behavior. However, on the other hand, they need to trust that the TTP implements sufficient security measures that prevent data breaches. This is an important drawback of the TTP approach as data breaches might cost companies their competitive advantage or reputation.

Alternatively, benchmarking could be performed via secure multi-party computation (MPC) [15]. An MPC protocol emulates a TTP by having the parties, e.g., companies, jointly evaluate some public function, e.g., quartile computation, over their inputs. Most importantly, those inputs are kept private, e.g., processed in encrypted form. Such a protocol is secure in the sense that parties only learn their own inputs, their outputs, and what can be inferred from that. Hence, confidential KPIs are protected from any internal and external observer, enabling privacy-preserving benchmarking. We restrict our considerations to MPC scenarios where n parties each contribute confidential inputs and jointly evaluate the target function with a service provider. We refer to the data providers as players and require the service provider to be a single, central instance (see Fig. 1).

As the core of a benchmarking MPC protocol, encrypted KPIs need to be sorted according to their underlying plaintexts in a privacy-preserving fashion. Such oblivious sorting can be done via sorting networks in $n \log n$ comparisons orchestrated by the service provider as described in [21]. However, this would cause the service provider to learn the order of the confidential KPIs, that is, how a particular company performs relatively to another particular company. Even if the service provider is assumed to not misuse this information proactively, a data breach could leak this confidential performance information.

To reduce the risk of benchmarks leaking confidential data and relative performance information, an efficient privacy-preserving benchmarking protocol based on MPC should ensure anonymity in the sense that no observer can infer ownership relations between companies and their encrypted KPIs. This can be done by secretly shuffling the encrypted

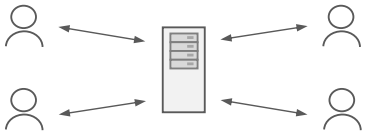


Fig. 1: Network of One Service Provider and Four Players

KPIs prior to benchmarking. We refer to a secret shuffle as a function that randomizes the order of a sequence of encrypted inputs such that no observer can map elements in the original sequence to their corresponding elements in the shuffled sequence with probability better than guessing. Preventing such a mapping also implies a need for modifying the ciphertexts without affecting the underlying plaintexts. Using shuffled KPI sequences as basis for benchmarking prevents leakage of ownership relations during KPI sorting later in the benchmarking process.

Besides privacy-preserving benchmarking, our protocol can be applied as a pre-processing step to any scenario where n players send encrypted inputs to a central service provider, e.g., cloud service, but do not want it to learn which player provided which input. This includes anonymous surveys, polls, and voting, where shuffling hides ownership relations by randomly rearranging encrypted inputs. Before we present our shuffling protocol in Section IV, we introduce required definitions and preliminaries and give an overview of related work. We conclude this paper with approaches to future work.

II. Preliminaries

We restrict our considerations to asymmetric cryptosystems, i.e., a tuple $\mathcal{CS} = (G, E, D)$ consisting of three polynomial-time algorithms. The probabilistic key-generation algorithm G takes as input a security parameter κ and outputs a key pair (pk, sk) consisting of a public encryption key pk and a secret decryption key sk. The probabilistic encryption algorithm E takes as input a plaintext $x \in \mathcal{M}$ and pk and outputs the ciphertext $y = E(x, pk) \in \mathcal{C}$. $\mathcal{M}$ and $\mathcal{C}$ denote the plaintext and ciphertext space, respectively. The decryption algorithm D takes as input a ciphertext $y \in \mathcal{C}$ and sk and outputs the plaintext $x = D(y, sk) \in \mathcal{M}$. For simplification, we denote the encryption of $x \in \mathcal{M}_i$ under a cryptosystem $\mathcal{CS}_i = (G_i, E_i, D_i)$ for pk_i by $y = E_i(x)$ and the decryption of $y \in \mathcal{C}_i$ for sk_i by $x = D_i(y)$.

Homomorphic cryptosystems, such as RSA [31], Paillier's [28], and BGV [5], allow for computations on ciphertexts. A cryptosystem $\mathcal{CS}$ is homomorphic if applying an operation $\circ$ to ciphertexts $E(x_1)$ and $E(x_2)$ yields the ciphertext $E(y)$ of the result $y = x_1 * x_2$ of a corresponding homomorphic operation $*$ applied to the plaintexts x_1 and x_2 [22]. That is, $E(x_1) \circ E(x_2) = E(x_1 * x_2)$. We restrict our considerations to cryptosystems with an additive homomorphism enabling addition of the underlying plaintexts as depicted in (1) and (2), such as Paillier's cryptosystem [28].

$$D(E(x_1) \cdot E(x_2)) = x_1 + x_2 \tag{1}$$

$$D(E(x_1)^{x_2}) = x_1 \cdot x_2 \tag{2}$$

That is, multiplication of ciphertexts encrypted under the same key pk yields an encryption of the sum of the underlying plaintexts, encrypted under pk. This also enables multiplication of an encrypted value by a plaintext value via exponentiation.

Paillier's cryptosystem allows for rerandomization [28]. Given pk and a ciphertext $E(x)$ of a plaintext x, rerandomization is an operation that computes a valid ciphertext $E'(x)$ without decryption. With high probability, $E(x) \neq E'(x)$ is ensured such that the output distributions of rerandomization and encryption are computationally indistinguishable [14]. For Paillier's cryptosystem, it can be performed by multiplication with the encrypted identity element 0 as depicted in (3) [28].

$$E'(x) = E(x) \cdot E(0) \tag{3}$$

A hash function $h(\cdot)$ is a function that, for arbitrarily long inputs x, computes outputs $h(x)$ of fixed length [20]. It is easy to compute $h(x)$, which we refer to as a hash. A hash function is cryptographic if it provides collision-resistance, meaning that it is computationally infeasible to find two hashes $h(x) = h(x')$ such that $x \neq x'$ [20]. This also guarantees that it is computationally infeasible to compute x given only $h(x)$ [20].

We define a sequence S as an enumeration of elements s_i that are arranged in a particular order. Multiple s_i can have the same value. The number of s_i in S is referred to as its length. We only use finite sequences of fixed length n and denote them either by $S = (s_1, ..., s_n)$ or by $S = (..., s_i, ...)$ depending on whether we want to emphasize the elements' order or their form. Given $S = (s_1, ..., s_n)$, a random permutation $\pi : S \to S'$ yields the permuted sequence $S' = (..., s_i, ... | s_i \in S)$ containing the same n elements as S but arranged in a random order. We denote the position of s_i in S' permuted via π by $\pi(s_i)$. See [20] for details on (pseudo)random permutations.

A function $\varphi(m)$ is negligible in m if for every positive polynomial $p(m)$ there is an $m_0 \in \mathbb{N}$ such that for every $m > m_0$, $\varphi(m) < \frac{1}{p(m)}$ applies [20]. Let $\left\{X_m^1\right\}_{m\in\mathbb{N}}$ and $\left\{X_m^2\right\}_{m\in\mathbb{N}}$ be two sets of random variables. If for a probabilistic polynomial-time algorithm A the advantage

$$\epsilon = \left| Pr\left[A\left(X_m^1, 1^m\right) = 1\right] - Pr\left[A\left(X_m^2, 1^m\right) = 1\right] \right| \tag{4}$$

is negligible in m, the two sets are computationally indistinguishable [20].

In [18], a shuffle of a sequence of ciphertexts is defined as a sequence of different ciphertexts of the same plaintexts, arranged in a permuted order. We additionally require the permutation to be secret and define a secret shuffle as follows.

Definition 1 (Secret Shuffle)**.** *Given a sequence of ciphertexts $X = (..., E(x_i), ...)$ with $1 \leq i \leq n$. A secret shuffle $\mathcal{S}(\cdot)$ is a function that, for input X, yields a sequence $\mathcal{X} = (..., E'(x_{\pi(i)}), ...)$ such that the ciphertexts $E'(x_1) \neq E(x_1), ..., E'(x_n) \neq E(x_n)$ encrypt pairwise equal plaintexts $x_1, ..., x_n$. The order of the elements in $\mathcal{X}$ is randomly permuted via a random permutation π. No participant can learn more than negligibly much information about π.*

III. Related Work

A. Approaches with Additional (Neutral) Instances

In [7], Chaum introduces mix networks, a protocol that enables anonymity and unlinkability of messages to their senders at the cost of additional computational overhead. Mix networks involve a sequence of servers, called mixes, which receive a set of messages, shuffle, and forward them to the next mix [18]. Unlinkability is guaranteed if at least one mix is honest [18]. There are two kinds of shuffles: decryption and re-encryption shuffles [1]. In decryption shuffles, the messages are layered ciphertexts. Each mix removes one layer of encryption from each message and sorts the resulting plaintexts. In re-encryption shuffles, the mixes rerandomize and permute the messages via a randomly chosen permutation. A re-encryption mix network that ensures simplified key management based on universal re-encryption is given in [17]. In [32], the first mix network that is universally composable and efficient independent of the number of mixes is presented. The first efficient non-interactive zero-knowledge proof for proving that a mix shuffled correctly is proposed in [19]. A description of how the permutation used by a mix can be constructed by multiple parties is given in [10].

Due to several drawbacks, mix networks do not properly ensure unlinkability between players and their inputs in our target scenario. Primarily, mixes need to be provided by different, independent parties [29]. This cannot be guaranteed in scenarios with a single, central service provider. The same applies to Riffle [24], an alternative to mix networks.

B. Approaches Based on Trusted Hardware

Alternatively, unlinkability can be achieved by shuffling in trusted hardware, e.g., Intel Software Guard Extensions (SGX) [8]. Generating and applying the random permutation as well as rerandomization can be done inside trusted hardware on the service-provider side. Such an approach is described in [13] for database access pattern obfuscation. In [11], an approach with a trusted unit that performs shuffling of encrypted data with limited private memory is given. An architecture for privacy-preserving online analysis of client data based on trusted hardware is presented in [3]. In setups with a trusted CPU but no trusted memory, oblivious RAM (ORAM) can ensure that untrustworthy RAM does not leak confidential information [16]. However, these approaches imply different trust assumptions and relations such as trust in the manufacturing of the trusted hardware. Therefore, they are not suitable for our scenario with distrusting participants.

C. Approaches Based on Secure Multi-party Computation

One approach to secure multi-party computation is secret sharing. In [25], three shuffling MPC protocols are proposed for the Sharemind secure computation platform, focusing on low communication and round complexity. In Sharemind, computation is done by three independent parties [4]. This does not fit our scenario with a single, central service provider. Another shuffling protocol based on secret sharing is presented in [27]. However, it is designed for decentralized settings.

Secure multi-party computation can also be based on homomorphic encryption. In [6], such an MPC protocol for shuffling data in a setting of multiple data providers and one data miner is proposed. It emulates a mix network in the sense that each data provider itself acts as a mix. Hence, it does not require independent mix servers. With its quadratic computational and linear round complexity, it does not scale well.

A shuffling protocol that combines both secret sharing and homomorphic encryption is proposed in [23]. It is used as a subprotocol to anonymizes players' inputs prior to decentralized sorting and benchmarking. The ownership relation is concealed in a multi-round protocol where mix networks are used to ensure anonymity. Hence, it has drawbacks similar to those of mix networks. A constant-round benchmarking protocol for centralized scenarios based on homomorphic encryption is presented in [21]. Instead of sorting the full list of encrypted KPIs, it computes in a privacy-preserving fashion for each input the number of inputs that are smaller, such that no participant learns any KPI's rank. Even though this approach does not require shuffling to prevent leaking KPIs' ranks, it comes at the cost of quadratic computational and communication complexity, which implies poor scalability.

IV. Secret Shuffling Protocol

A. Adversary Model

We design our protocol to be secure against any semi-honest adversary $\mathcal{A}$ [26] that corrupts either an arbitrary number of players or the service provider. That is, we exclude collusion between any player and the service provider, like the related work. Our shuffling protocol ensures input privacy. Hence, $\mathcal{A}$ does not learn anything about non-corrupted players' secret inputs. Most importantly, we ensure that no such $\mathcal{A}$ is able to map non-corrupted players' inputs to their equivalents in the shuffled sequence generated by the shuffling protocol. In summary, an adversary corrupting either any subset of the players or the service provider can neither determine the ownership relation between non-corrupted players and their inputs nor learn their secret inputs.

B. Prerequisites

In the description of our protocol SHUFFLE, we use the indices $1 \leq i, j \leq n$ for players P_i and P_j, respectively, as well as their inputs x_i (x_j), random values r_i (r_j), etc. We denote concatenation by "||".

We assume two instances $\mathcal{CS}_1$ and $\mathcal{CS}_2$ of the Damgård-Jurik cryptosystem [9], like Paillier's [28]. The public keys pk_1 and pk_2 are known to the service provider P_S and the players P_i. The secret key sk_1 is known only to the players and could be generated and distributed via Diffie-Hellman key exchange [12]. The secret key sk_2 is only known to P_S. We require the plaintext space $\mathcal{M}_2$ of $\mathcal{CS}_2$ to be a subset of the plaintext space $\mathcal{M}_1$ of $\mathcal{CS}_1$, i.e.,

$$\mathcal{M}_2 \subseteq \mathcal{M}_1. \tag{5}$$

This ensures that any message that can be encrypted with pk_2 can also be encrypted with pk_1.

TABLE I: Secret Shuffling Protocol with Step Labels and Computations

Step	Computation
1.1	$\underline{P_i \to P_S}$: $E_1(x_i)$
1.2	$E_1(r_{1_i})$
2.1	$\underline{P_S \to P_i}$: $R'_1 = (..., E_1(r'_{1_i}) = E_1(r_{1_{\pi_1(i)}}), ...)$
2.2	$X' = (..., E_1(x'_i + r_{2_i}) = E_1(x_{\pi_2(i)} + r_{2_i}) = E_1(x_{\pi_2(i)}) \cdot E_1(r_{2_i}), ...)$
2.3	$R_2 = (..., E_2(r_{2_i}), ...)$
2.4	r_{1_S}
2.5	$\underline{P_i}$: $H = (..., h_j = h(r'_{1_j} \| r_{1_S}) = h(D_1(E_1(r'_{1_j})) \| r_{1_S}), ...)$
2.6	$\rho_i = position(H' = sort(H), h_i)$
2.7	$\underline{P_i \to P_S}$: $E'_1(x'_{\rho_i} + r_{2_{\rho_i}}) = E_1(x'_{\rho_i} + r_{2_{\rho_i}} + 0) = E_1(x'_{\rho_i} + r_{2_{\rho_i}}) \cdot E_1(0)$
2.8	$E_2(r_{2_{\rho_i}} + r_{3_i}) = E_2(r_{2_{\rho_i}}) \cdot E_2(r_{3_i})$
2.9	$E_1(r_{3_i})$
2.10	$\underline{P_S}$: $\mathcal{X} = (..., E'_1(x'_{\rho_i}) = E'_1(x'_{\rho_i} + r_{2_{\rho_i}}) \cdot E_1((-1) \cdot D_2(E_2(r_{2_{\rho_i}} + r_{3_i}))) \cdot E_1(r_{3_i}), ...)$

We require two random permutations π_1 and π_2, a cryptographic hash function $h(\cdot)$, and two functions $sort(S)$ and $position(S, s_i)$. The permutations π_1 and π_2 are both chosen by and only known to P_S. The hashes of $h(\cdot)$ are assumed to be uniformly distributed among the domain $dom(h(\cdot))$. Given a sequence $S = (s_1, ..., s_n)$, $sort(S)$ outputs a sequence S' that contains $s_1, ..., s_n$ in ascending order, i.e., $S' = (s'_1 \leq s'_2 \leq \cdots \leq s'_n \mid s'_i \in S)$. The function $position(S, s_i)$ outputs the position of s_i in S.

Moreover, we assume pairwise secure, i.e., secret and authentic, channels between each player and the service provider, for instance established via Transport Layer Security (TLS).

C. Protocol Specification

Our protocol is given in Table I. It runs in two communication rounds: one that collects the players' inputs (steps 1.1-1.2) and one that shuffles them (steps 2.1-2.10). As in Definition 1, for a protocol to secretly shuffle a sequence, it has to permute the order of the entries by a random permutation π. Furthermore, it has to modify the ciphertexts of the secret inputs such that π cannot be reconstructed by simply looking at the rearranged ciphertexts. Permutation is achieved by a novel approach to random index distribution where each player randomly but uniquely selects some player's encrypted input from the full list of input ciphertexts. Ciphertext modification is done by rerandomizing the selected ciphertext (see Equation (3)). In detail, this is done as follows.

In step 1.1, each player P_i takes its private input x_i that is supposed to be shuffled, encrypts it under $\mathcal{CS}_1$, and sends it to the service provider P_S. Then, in step 1.2, each player chooses a (presumably unique) random value $r_{1_i} \in \mathcal{M}_1$ that will be used for random index distribution, encrypts it under $\mathcal{CS}_1$, and sends it to P_S. Hence, P_S receives $2 \cdot n$ ciphertexts that it cannot decrypt, two from each player.

We denote the list of encrypted random values $E_1(r_{1_i})$ by R'_1. In step 2.1, P_S permutes R'_1 by π_1 and sends the permuted list to the players. Permutation prevents the players from learning which r_{1_i} was provided by which player. Similarly, we denote the list of encrypted input values $E_1(x_i)$ by X'. In step 2.2, P_S permutes X' via π_2, hides each plaintext $x_{\pi_2(i)}$ by homomorphically adding a random value $r_{2_i} \in \mathcal{M}_1$ (see Equation (1)), and sends X' to each player. Additive hiding prevents the players from learning other players' secret inputs. In step 2.3, P_S encrypts the random values r_{2_i} under $\mathcal{CS}_2$ and sends the full list, denoted by R_2, to the players. Then, P_S chooses a random value $r_{1_S} \in \mathcal{M}_1$ and sends it to the players in step 2.4. Hence, the players receive the same three lists R'_1, X', R_2 of n ciphertexts and the same random value r_{1_S}.

In step 2.5, each P_i decrypts the ciphertexts $E_1(r'_{1_j}) \in R'_1$, $1 \leq j \leq n$. If the values r'_{1_j} are not pairwise distinct, each player aborts the protocol and notifies P_S. Otherwise, each player concatenates each resulting plaintext r'_{1_j} with the random value r_{1_S} of P_S and computes the n hashes h_j of the list H. Using r_{1_S} as a seed for $h(\cdot)$ prevents players P_i from selecting a specific r_{1_i} in step 1.2 to obtain a desired hash h_i, which would eventually affect the randomness of the index distribution. In step 2.6, each P_i sorts the list of hashes H and obtains the sorted list H'. For the hash $h_i = h(r'_{1_i} \| r_{1_S})$ corresponding to P_i's random value r'_{1_i}, its position ρ_i in H' is the random index of P_i. Hence, each player computes an individual, random index ρ_i that is unknown to P_S and not related to the rank of its input x_i.

Given ρ_i, each player selects the ciphertext $E_1(x'_{\rho_i} + r_{2_{\rho_i}}) \in X'$, rerandomizes it, and sends it to P_S in step 2.7. Rerandomization prevents P_S from learning which ciphertext was selected. In step 2.8, P_i selects the encrypted random value $E_2(r_{2_{\rho_i}}) \in R_2$ of index ρ_i, additively hides the plaintext by homomorphically adding a random value r_{3_i}, and sends the resulting ciphertext to P_S. This random value r_{3_i} is then encrypted under $\mathcal{CS}_1$ and sent to P_S in step 2.9. Hence, the service provider receives three ciphertexts from each player.

In step 2.10, for each $1 \leq i \leq n$, P_S decrypts the ciphertext $E_2(r_{2_{\rho_i}} + r_{3_i})$ received in step 2.8, multiplies the resulting plaintext with -1, and encrypts the product under cryptosystem $\mathcal{CS}_1$. The resulting ciphertext is then multiplied with the ciphertexts $E_1(x'_{\rho_i} + r_{2_{\rho_i}})$ of step 2.7 and $E_1(r_{3_i})$ of step 2.9. Consequently, the random values r_{2_i} and r_{3_i} are eliminated,

TABLE II: Service Provider's Computational and Communication Complexity of the Protocol

Step	Enc	Dec	Mult	Message length
2.1				$n \cdot n \cdot l_{\mathcal{C}_1}$
2.2	n		n	$n \cdot n \cdot l_{\mathcal{C}_1}$
2.3	n			$n \cdot n \cdot l_{\mathcal{C}_2}$
2.4				$n \cdot l_{\mathcal{M}_1}$
2.10	n	n	$2 \cdot n$	
Total	$3 \cdot n$	n	$3 \cdot n$	$2 \cdot n^2 \cdot l_{\mathcal{C}_1} + n^2 \cdot l_{\mathcal{C}_2} + n \cdot l_{\mathcal{M}_1}$

TABLE III: Each Player's Computational and Communication Complexity of the Protocol

Step	Enc	Dec	Mult	Message length
1.1	1			$l_{\mathcal{C}_1}$
1.2	1			$l_{\mathcal{C}_1}$
2.5		n		
2.6				
2.7	1		1	$l_{\mathcal{C}_1}$
2.8	1		1	$l_{\mathcal{C}_2}$
2.9	1			$l_{\mathcal{C}_1}$
Total	5	n	2	$4 \cdot l_{\mathcal{C}_1} + l_{\mathcal{C}_2}$

resulting in rerandomized ciphertexts $\chi_i = E_1(x'_{\rho_i}) \in \mathcal{X}$.

The order of the rerandomized ciphertexts χ_i of the input values x_i is determined by the input order of the values in steps 2.7 to 2.9 as received via network. Every P_i sends some P_j's rerandomized, encrypted input, chosen based on its random index ρ_i. The service provider cannot map between the original input order and the order of $\mathcal{X}$. Therefore, P_S's output is a correctly shuffled list. We proof correctness in the extended version of this paper [2]. Furthermore, in a simulation-based proof, we also show that our shuffling protocol is secure in the semi-honest model by demonstrating that anything an adversary $\mathcal{A}$ can learn during protocol execution can as well be learned given only the inputs and outputs of the protocol [26].

V. Performance Evaluation

The performance evaluation of our protocol SHUFFLE is twofold: We first investigate its asymptotic computational, communication, and round complexity in a theoretical analysis. Then, we examine its performance in an empirical analysis and compare it to the performance of mix networks.

A. Asymptotic Complexity

1) Round Complexity: As depicted in Table I, the protocol consists of two rounds and a total of twelve protocol steps. Both values are independent of the number of players n. Therefore, the round complexity is constant in n, i.e., $\mathcal{O}(1)$.

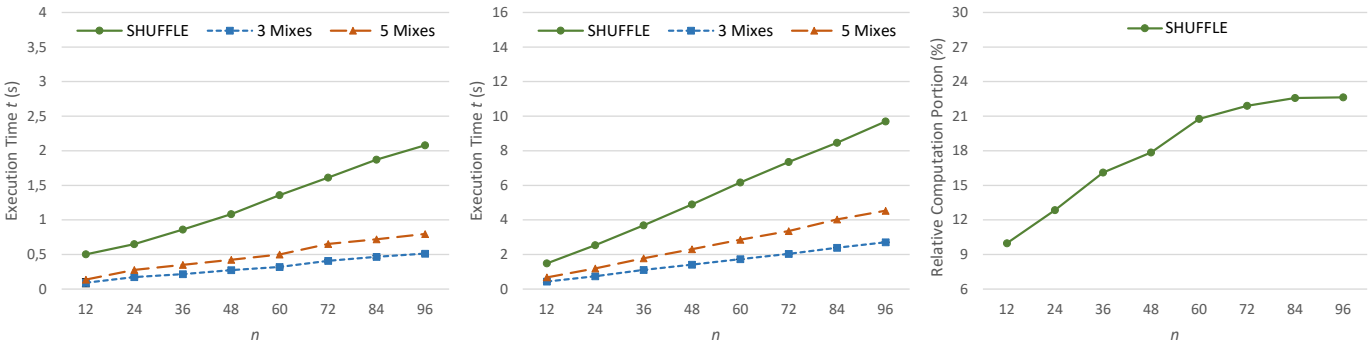
2) Computational Complexity: We investigate the number of operations that need to be carried out by the service provider and each player, respectively. We restrict our considerations to the cryptographic operations encryption, decryption, and ciphertext multiplication as they can be assumed to be the most complex ones. Their numbers are given in the middle columns of Tables II and III. The resulting asymptotic computational complexity is $\mathcal{O}(n)$, i.e., linear in the number of players n, for both the service provider and each player.

3) Communication Complexity: To determine the communication complexity of the protocol, we investigate the length of the messages sent in each step of the protocol by the service provider and each player, respectively. These are given in the rightmost columns of Tables II and III. Here, $l_{\mathcal{M}_i}$ and $l_{\mathcal{C}_i}$ denote the maximum length of plaintexts in $\mathcal{M}_i$ and ciphertexts in $\mathcal{C}_i$, respectively. The total asymptotic communication complexity per player is $\mathcal{O}(n)$, i.e., linear in the number of players n. The service provider's communication complexity is $\mathcal{O}(n^2)$, i.e., quadratic in the number of players n. Compared to related work, such as [7], our protocol has higher asymptotic communication complexity. However, we accept this loss as it helps reduce the computational complexity asymptotically.

B. Empirical Performance

To investigate the practical performance of the protocol, we implemented both the players' and the service provider's part of the protocol and deployed them in a cloud-computing setting. The service provider was implemented as a Java HttpServlet and deployed in a cloud-computing instance with 96 CPUs and 384 GB RAM. To emulate sufficiently large numbers of independent players, we implemented the players' protocol steps in a Java HttpServlet and deployed the players in a Kubernetes cluster based on a cloud-computing instance with 96 CPUs and 384 GB RAM. We instantiated one Kubernetes node per player and provided each node with one CPU and 4 GB RAM, which compares to the minimum requirements on a standard desktop computer. Therefore, we were able to emulate up to 96 players. Service provider and players were deployed in different data centers in two major European cities with a distance of approximately 650 kilometers to ensure a lifelike communication scenario. We used the additively homomorphic Paillier cryptosystem for $\mathcal{CS}_1$ and $\mathcal{CS}_2$.

For comparison, we implemented a simple yet efficient re-encryption mix network. Its construction is similar to the one described in [17], but instead of the ElGamal cryptosystem with universal re-encryption, we used the standard version of Paillier's cryptosystem. Re-encryption (rerandomization) is performed given the public key of the players, which is a valid approach as the senders, i.e., players, in the shuffling scenario share the same key and the recipient, i.e., service provider, is not supposed to decrypt the received confidential data. We implemented the mixes as Java HttpServlets and deployed them in a similar cloud-computing setting as above, running each mix on an instance with 96 CPUs and 384 GB RAM. In a cascade of mixes, each mix receives all the messages at the same time in one batch, permutes and re-encrypts them, and forwards the full batch to the next mix or the recipient. This matches a communication setting where the service provider has the mix network shuffle all the inputs once it received the full list of inputs from the players.



(a) Execution Time for 1024 Bits (b) Execution Time for 2048 Bits (c) Relative Computation Portion

Fig. 2: Results of the Empirical Performance Analysis

Fig. 2a depicts the execution time t relatively to the number of messages n for 1024-bit keys for our shuffling protocol and for mix networks with cascades of three and five mixes, respectively. Shuffling 96 inputs with our shuffling protocol took 2.08 seconds while the mix networks performed shuffling in 0.51 and 0.80 seconds, respectively. For 2048-bit keys, shuffling 96 inputs took 9.69 seconds with our protocol and 2.71 and 4.53 seconds with mix networks (see Fig. 2b). For both key lengths, the execution time of our shuffling protocol grows linearly in the number of players. Most importantly, the empirical results show that a mix network of five mixes with appropriate key length is only 2.14 times faster than our shuffling protocol. However, recall that to achieve this performance, mix networks require multiple independent servers to perform the mixing whereas our shuffling protocol requires only a single server. Given the linear nature of re-encryption mix networks, one can reasonably assume that our protocol performs similar to a mix network of ten to eleven mixes.

The linear growth of the execution time of our shuffling protocol indicates that the quadratic character of the communication complexity has minor impact on the overall runtime. To further support this assumption, we investigated the ratio of computation time to communication time (see Fig. 2c). For growing n, the portion of the total runtime that is required for computation shows logarithmic trend. Hence, the ratio barely changes for large n. This implies that for large n, the communication time and computation time grow proportionately instead of disproportionately, as one could have expected from their asymptotic complexities.

C. Summary

Our protocol has constant round complexity and linear computational complexity. Our empirical performance analysis shows that the execution time is linear in n. This implies that the quadratic character of the communication complexity has minor impact on the overall runtime. In this analysis, shuffling 96 secret inputs encrypted under a 2048 bits long Paillier key took 9.69 seconds, which proves the practicability of our secret shuffling protocol. Performing shuffling via a mix network of five mixes takes roughly half as long. However, such a mix network requires five independent powerful servers, instead of a single powerful server. This indicates that our shuffling protocol not only provides shuffling in centralized settings in reasonable time but also has lower energy consumption and causes lower cloud-computing costs.

VI. Conclusion

We present an efficient secure multi-party protocol for shuffling encrypted data. It precludes any mapping between ciphertexts in the unshuffled and the shuffled sequence with probability better than guessing. We prove correctness of our shuffling functionality and privacy of the confidential inputs [2]. Key element of our contribution is a novel approach to efficient random index distribution, which provides the random, secret permutation. Our protocol has computational complexity linear in the number of players as well as constant round complexity. It shuffles 96 ciphertexts in 9.69 seconds for 2048 bit long keys. We show that the effect of the communication complexity's quadratic character on the execution time is minor, ensuring good scalability. Our protocol performs asymptotically better than previous MPC-based shuffling approaches that focus on low communication complexity but suffer from higher computational complexity, which has negative impact on scalability. Furthermore, its execution time is only 2.14 times that of a mix network of five mixes but requires no additional, independent servers. This not only enables use cases with centralized communication scenarios but also causes lower cloud-computing costs. As a general-purpose protocol, it can be a building block in a variety of applications such as privacy-preserving benchmarking systems, anonymous surveys, polls, voting, and many more where shuffling enables anonymity by hiding ownership relations.

VII. Future Work

The protocol's applicability could be further improved by reducing its communication complexity. This can be achieved with a more efficient approach to obtaining the input ciphertexts from the service provider and selecting one of unique, random index. Moreover, it could be modified to be secure against malicious adversaries [26]. In more generic scenarios, m encrypted inputs could be present on the service-provider side prior to the protocol execution instead of being

provided by the n players themselves. The players could then shuffle these values. Further security analysis is necessary to investigate the implications of setting $n \ll m$ where players generate multiple random indices and select and rerandomize multiple ciphertexts at once. This would further decrease the communication complexity and improve scalability.

References

[1] B. Adida and D. Wikström. "How to Shuffle in Public". In: *Proceedings of the Theory of Cryptography Conference*. 2007.

[2] K. Becher and T. Strufe. "Efficient Cloud-based Secret Shuffling via Homomorphic Encryption". In: *arXiv e-prints* (2020). arXiv: 2002.05231 [cs.CR].

[3] A. Bittau et al. "Prochlo: Strong Privacy for Analytics in the Crowd". In: *Proceedings of the Symposium on Operating Systems Principles*. 2017.

[4] D. Bogdanov. "Sharemind: programmable secure computations with practical applications". PhD thesis. University of Tartu, 2013.

[5] Z. Brakerski, C. Gentry, and V. Vaikuntanathan. "(Leveled) Fully Homomorphic Encryption Without Bootstrapping". In: *ACM Transactions on Computation Theory* (2014).

[6] J. Brickell and V. Shmatikov. "Efficient Anonymity-preserving Data Collection". In: *Proceedings of the International Conference on Knowledge Discovery and Data Mining*. 2006.

[7] D. Chaum. "Untraceable Electronic Mail, Return Addresses, and Digital Pseudonyms". In: *Communications of the ACM* (1981).

[8] V. Costan and S. Devadas. *Intel SGX Explained*. IACR Cryptology ePrint Archive. 2016.

[9] I. Damgård and M. Jurik. "A Generalisation, a Simplification and some Applications of Paillier's Probabilistic Public-Key System". In: *Proceedings of the International Workshop on Public Key Cryptography*. 2001.

[10] I. Damgård et al. "Compact Zero-Knowledge Proofs of Small Hamming Weight". In: *Proceedings of the International Workshop on Public Key Cryptography*. 2018.

[11] H. Dang et al. "Privacy-Preserving Computation with Trusted Computing via Scramble-then-Compute". In: *Proceedings on Privacy Enhancing Technologies* (2017).

[12] W. Diffie and M. Hellman. "New directions in cryptography". In: *IEEE Transactions on Information Theory* (1976).

[13] X. Ding, Y. Yang, and R. Deng. "Database Access Pattern Protection Without Full-Shuffles". In: *IEEE Transactions on Information Forensics and Security* (2011).

[14] H. Galteland and K. Gjøsteen. "Malware, Encryption, and Rerandomization – Everything Is Under Attack". In: *Proceedings of the Paradigms in Cryptology — Mycrypt*. 2017.

[15] O. Goldreich. *Secure Multi-Party Computation*. Tech. rep. 1.4. 2002.

[16] O. Goldreich and R. Ostrovsky. "Software Protection and Simulation on Oblivious RAMs". In: *Journal of the ACM* (1996).

[17] P. Golle et al. "Universal Re-encryption for Mixnets". In: *The Cryptographers' Track at the RSA Conference*. 2004.

[18] J. Groth. "A Verifiable Secret Shuffle of Homomorphic Encryptions". In: *Journal of Cryptology* (2010).

[19] J. Groth and S. Lu. "A Non-interactive Shuffle with Pairing Based Verifiability". In: *Proceedings of the Advances in Cryptology – ASIACRYPT*. 2007.

[20] J. Katz and Y. Lindell. *Introduction to Modern Cryptography*. 2nd ed. Chapman & Hall/CRC, 2014.

[21] F. Kerschbaum. "A privacy-preserving benchmarking platform". PhD thesis. Karlsruhe Institute of Technology, 2010.

[22] F. Kerschbaum. "Practical Privacy-Preserving Benchmarking". In: *Proceedings of the International Information Security Conference*. 2008.

[23] F. Kerschbaum and O. Terzidis. "Filtering for Private Collaborative Benchmarking". In: *Emerging Trends in Information and Communication Security*. 2006.

[24] A. Kwon et al. "Riffle: An Efficient Communication System With Strong Anonymity". In: *Proceedings on Privacy Enhancing Technologies* (2015).

[25] S. Laur, J. Willemson, and B. Zhang. "Round-Efficient Oblivious Database Manipulation". In: *International Conference on Information Security*. 2011.

[26] Y. Lindell. *Tutorials on the Foundations of Cryptography: Dedicated to Oded Goldreich*. 1st ed. Springer, 2017.

[27] M. Movahedi, J. Saia, and M. Zamani. "Secure Multiparty Shuffling". In: *Proceedings of the International Colloquium on Structural Information and Communication Complexity*. 2015.

[28] P. Paillier. "Public-key Cryptosystems Based on Composite Degree Residuosity Classes". In: *Proceedings of the International Conference on Theory and Application of Cryptographic Techniques*. 1999.

[29] U. Parampalli, K. Ramchen, and V. Teague. "Efficiently Shuffling in Public". In: *Proceedings of the International Workshop on Public Key Cryptography*. 2012.

[30] The European Parliament and the Council of the European Union. *Regulation (EU) 2016/679 (General Data Protection Regulation)*. May 2016. URL: https://eur-lex.europa.eu/eli/reg/2016/679/oj.

[31] R. Rivest, A. Shamir, and L. Adleman. "A Method for Obtaining Digital Signatures and Public-key Cryptosystems". In: *Communications of the ACM* (1978).

[32] D. Wikström. "A Sender Verifiable Mix-Net and a New Proof of a Shuffle". In: *Proceedings of the Advances in Cryptology – ASIACRYPT*. 2005.