A Novel Secure and Efficient Data Aggregation Scheme for IoT

Ruinian Li, Carl Sturtivant, Jiguo Yu, and Xiuzhen Cheng

***Abstract*—We define the following problem termed $n \times$ 1-out-of-n oblivious transfer ($n \times$ 1-out-of-n OT): in a system with one server and n clients, how to securely and efficiently assign n secrets to n clients by the server, with each client getting a unique secret from the server, and the server and clients remain unknown of how the secrets are distributed? This is a novel problem that is fundamentally different than 1-out-of-n OT repeated n times, and is different than k-out-of-n OT as well. Nevertheless, the proposed OT has many practical applications such as privacy-preserving data aggregation in smart grids. It can also be employed to design crypto protocols for anonymous communications and group signatures. In this paper, we propose the first algorithm to efficiently and effectively implement the $n \times$ 1-out-of-n OT. We construct hidden permutation circuits to obliviously assign n secrets to n clients by the server within $O(\lg(n))$ time. A rigorous theoretical analysis is also carried out to investigate the security strength and performance of the protocol.**

***Index Terms*—Internet of Things (IoT), oblivious transfer, privacy preservation, privacy-preserving data aggregation.**

I. Introduction

In THIS paper, we define a novel problem termed $n \times 1$-out-of-n oblivious transfer ($n \times 1$-out-of-n OT) for a system with one server and n clients: how to securely and efficiently assign n secrets to n clients by the server, with each client getting a unique secret from the server, and the server and clients remain unknown of how the secrets are distributed? $n \times 1$-out-of-n OT is different from the k-out-of-n OT problem because in the latter one client requests k secrets to be obliviously transferred from the server while in the former (our problem) each of the n clients gets one unique secret from the server, not multiple secrets. It is neither the 1-out-of-n problem repeated n times as we require that each client gets a unique secret from the server without duplication, while repeating n times of the 1-out-of-n obviously may render two or more clients receive the same secret. To our best knowledge, $n \times$ 1-out-of-n OT has never been investigated; nevertheless, it is a fundamental problem with many applications in practice, as outlined in sequel.

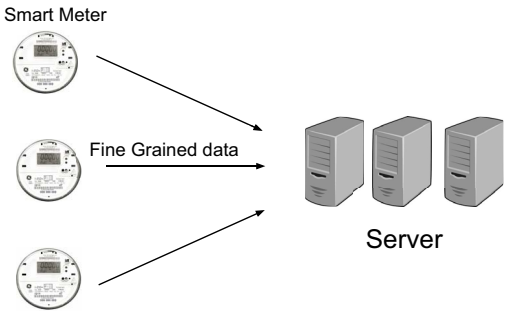

Fig. 1. Smart meter data aggregation.

Our $n \times$ 1-out-of-n OT problem definition is motivated by the study of privacy-preserving data aggregation in smart grids. Smart grids is an Internet of Things [1]–[3] application that delivers electricity from the power plant to your home or business. As shown in Fig. 1, each smart meter reports its fine-grained utility data to the server; and the server needs to compute the aggregation of the data while the privacy of the data must be protected. Many methods have been proposed but they are mainly based on modern cryptographic advances such as differential privacy, homomorphic encryption, and secret sharing [4]–[10]. Unfortunately, the computation overhead at the smart meter side is prohibitively high rendering them inapplicable for data aggregation in smart grids. Using masking values to obscure the true data is the most efficient method as it has simple encryption operations at smart meter side (just one modular addition operation) [5]–[7] but the existing mechanisms for constructing a secret masking value for each smart meter that can cancel out at the server when added together involves high communication and computation overheads. Motivated by this observation, we propose the $n \times 1$-out-of-n OT problem to help carry out privacy-preserving data aggregation in smart grids as follows: each smart meter receives one secret obliviously from the server based on the $n \times$ 1-out-of-n OT protocol while the sum of the n secrets is known only to the server, such that the secrets can be used as masking values by the smart meters to hide their true data via a simple addition operation and the server can retrieve the aggregated result easily by subtracting the sum of the secrets. An additional salient feature brought by our data aggregation

Manuscript received January 19, 2018; revised May 20, 2018; accepted June 5, 2018. Date of publication June 19, 2018; date of current version May 8, 2019. This work was supported by the U.S. National Science Foundation under Grant CNS-1704397 and Grant IIS-1741279 and by the National Natural Science Foundation of China under Grant 61771289, Grant 61672321, and Grant 61373027. *(Corresponding author: Jiguo Yu.)*

R. Li is with the Department of Computer Science, Bowling Green State University, Bowling Green, OH 43403 USA (e-mail: lir@bgsu.ed).

C. Sturtivant is with the Department of Computer Science and Engineering, University of Minnesota, Minneapolis, MN 55455 USA (e-mail: carl@cs.umn.edu).

J. Yu is with the Qilu University of Technology, Shandong Academy of Sciences, Jinan 250353, China, also with the National Supercomputer Center, Shandong Computer Science Center, Jinan 250014, China, and also with the School of Information Science and Engineering, Qufu Normal University, Rizhao 276826, China (e-mail: jiguoyu@sina.com).

X. Cheng is with the Department of Computer Science, George Washington University, Washington, DC 20052 USA (e-mail: cheng@gwu.edu).

Digital Object Identifier 10.1109/JIOT.2018.2848962

approach based on $n \times 1$-out-of-n OT is that the secret at each smart meter is reusable via a simple one-way hash algorithm, which could be public, to guarantee that the masking values at each round are different to avoid side-channel attacks that can be easily launched as the smart meter readings are correlated in time domain. The masking values generated by crypto methods in [5]–[7] cannot be easily changed from round to round, as the sum must remain unchanged, and regenerating a new set of masking values at each round implies that the corresponding data aggregation protocol involves extremely high communication and computational overheads, making them inapplicable in smart grids.

One may argue that we can let each client generate an independent secret, and send the secret to the server through anonymous communications. However, current anonymous communication protocols such as Crowds, mix nets, onion routing, etc., have their limitations [11]–[15]. Crowds [15] and onion routing [13], [14] are vulnerable to a global attacker who is able to watch the traffic; mix net [11], [12] is resistant to global attackers while it needs many trusted relays and traffic in presence to help obscure the traffic for secret delivery.

We propose to utilize hidden permutations that can obscure the communications among all the clients. By this way the secret sent from the server to a client does not necessarily be the one that is intended for this client; this secret, in an encrypted form, can be forwarded through the hidden permutation to the intended client, which is the only client able to decrypt it. The proposed hidden permutation can also be used for designing anonymous communication protocols, as it allows the clients to send secrets to the server anonymously. Our main contributions are briefly summarized as follows.

1) We propose and formalize a novel problem termed $n \times$ 1-out-of-n OT.
2) We develop a practical protocol to solve this problem efficiently, which novelly combines modern cryptography and permutation circuits.
3) Our proposed hidden permutation can be used to implement an anonymous communication system, and our full protocol can be applied to achieve privacy-preserving data aggregation for many applications such as smart grids, wireless sensor networks, and mobile health.
4) We theoretically prove that our proposed protocol is secure and can counter various attacks.
5) We evaluate the performance of our proposed protocol in terms of both computational complexity and communication cost.

Section II summarizes the most related work. Section III presents our problem formulation, system model, and security model. In Section IV, we outline a generic solution and present a protocol for our $n \times 1$-out-of-n OT problem. Section V details how the data is transmitted through a hidden permutation, and Section VI introduces an efficient permutation circuit. Protocol analysis is provided in Section VII and we conclude this paper with a future research discussion in Section VIII.

II. Related Work

Oblivious transfer is a type of protocol in which a client gets secret values from a server while the server remains unknown of which secret each client gets. Our problem belongs to the family of oblivious transfer as we require the server and clients remain unknown of how the secrets are distributed, i.e., each client knows only its own secret obtained from the server while the server does not know the secret received by any client. The first oblivious transfer protocol was proposed by Rabin [16]. In Rabin's [16] method, a sender sends a message to a receiver with a probability of $1/2$, without knowing whether the receiver can receive the message. Lots of work have been done to extend Rabin's work [17]–[26]. Oblivious transfer can be divided into two categories of 1-out-of-n OT and k-out-of-n OT. In 1-out-of-n OT [18]–[20], [23], [24], a client is able to get one secret from the server, and the server does not know what secret the client gets. In k-out-of-n OT [21], [22], [25], [26], a client can get k secrets from the server, while the server does not know what secrets the client gets. Our problem is different from these two cases. On one hand, we require each client to get only one secret from the server, which is different from k-out-of-n OT; on the other hand, our problem is not simply a 1-out-of-n OT repeated n times, which cannot guarantee that each client gets a unique secret.

The $n \times$ 1-out-of-n problem is motivated by our investigation on privacy-preserving data aggregation in smart grids. We notice that utilizing masking values for privacy-preserving data aggregation has a few attractive features making it applicable in resource-constrained environments: such an approach does not rely on a trusted third party and the involved encryption and decryption operations are extremely simple. Nevertheless, constructing the masking values at the client side that can sum to a value known to the server is nontrivial. Kursawe *et al.* [5] proposed four approaches to generate the masking values based on secret sharing, Diffie–Hellman, and bilinear mapping. In [6], Paillier homomorphic encryption was applied to produce the masking values and in [7], a distributed Laplacian perturbation algorithm was presented in which the clients jointly create a Laplacian noise. These approaches allow the masking values to be generated by the clients, which involves high communication and computational overheads. Nevertheless, the masking values should be updated frequently, preferably a unique mask value is used by each data item of a client for countering the side-channel attacks. Our protocol for the novel $n \times 1$-out-of-n problem proposed in this paper intends to tackle the challenges mentioned above.

Random permutation is an important primitive for cryptosystems that aims at randomly ordering a set of objects [27]–[31]. Rackoff and Simon [31] demonstrated that almost every switching network of polylogarithmic depth almost randomly shuffles any sequence of $n/2$ "0"s and $n/2$ "1"s. Following Rackoff and Simon [31] and Czumaj [29] proposed a novel construction that shuffles an arbitrary number of 0s and 1s. In this paper, we take advantage of random permutations to shuffle n clients. Each client chooses to swap or not to swap a message with another client to construct a hidden permutation, which hides the communications among the clients. We give a construction P_n and prove that it can be used to implement any permutation.

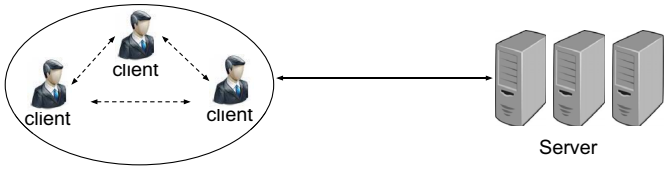

Fig. 2. Communication architecture.

III. Problem Formulation and Models

A. Problem Definition

A server S would like to share a set X of n secrets to n clients, where $X = \{x_0, x_1, \ldots, x_{n-1}\}$. The clients are numbered as $\{0, 1, \ldots, n-1\}$. The secret assignment must satisfy the following two requirements.

1) The assignment of secrets to the clients remains entirely unknown to the server.
2) Each client's knowledge is confined to the value of its own secret.

B. System Model

As shown in Fig. 2, our system model consists of two major entities: 1) server and 2) clients. Let p_i be the public key of client i and p_S be the public key of the server. The corresponding private keys are denoted by s_i and s_S, respectively. We denote the encryption to ciphertext c of data d with public key p by $c = E_p(d)$, and decryption of ciphertext c with private key s by $d = D_s(c)$. For secure symmetric encryption and decryption with the key k, the encryption and decryption processes are denoted analogously by $c = E_k(d)$ and $d = D_k(c)$, respectively. We also denote the signing and signature verifying of a message m as $\text{Sign}_s(m)$ and $\text{Ver}_p(m)$, respectively. The server is responsible for generating the set of n secrets X to be obliviously transferred to the n clients.

C. Security Model

We adopt a semihonest mode, where the participants follow the protocols honestly, but they attempt to learn more information of other entities. We do not assume a secure channel in our model, and we further insist that an adversary can be a global attacker, which implies that all traffic can be seen by the adversary. We also assume that the server is able to observe all the messages transferred among the clients. Therefore we require that all communications are confidential (protected by encryption) to protect against such attacks.

IV. Protocol Outlook

A. Overview of the Protocol

To obliviously distribute n secrets to n clients, there must exist some ways to obscure the communications between the clients and the server. Our idea is to let the clients cooperatively work to communicate with the server, so that the client who directly talks with the server is not the final receiver of a secret x_i, but a relay who helps to deliver the secret to the receiver. A random permutation can be utilized to hide the relationship of the clients such that when the server sends a secret x_i to a client, it has no knowledge of where the secret is forwarded. In the meantime, the relay who passes the secret to the final receiver should not be able to obtain the true value of the secret, which implies the adoption of some cryptographic mechanisms.

To build such a protocol, there are many problems to solve. For example, how to securely achieve a permutation of the clients so that they can cooperatively work together? How to hide the permutation from an outsider attacker, or even a participant in the protocol? How to protect the confidentiality of the secrets so that only the final receiver is able to obtain the true value of the secrets? To answer these questions, we illustrate how a generic solution protocol is developed in detail.

B. Development of Generic Solution Protocol

It is clear that after the protocol has executed, S should be able to infer that any secret x_i could have arrived at any client j while remaining entirely ignorant of the actual permutation σ mapping secrets to clients. If the server initially sends x_i to client i for each i in parallel for efficiency, then presumably the clients must send data back and forth among themselves, while S is not aware of how the secrets are permuted among the clients. During this process the permutation σ is implemented so that x_i arrives at client $j = \sigma(i)$. We call this kind of obfuscated distributed implementation of a permutation *scrambling*.

To ensure that only the ultimate destination of x_i namely client $j = \sigma(i)$ is privy to the value x_i, we must ensure that x_i is encrypted by S so that the only client in possession of the correct key to reveal it is client j. To make this paper, S must possess n keys and yet not know which one corresponds to which client. This superficially appears to be exactly the problem we are trying to solve! However, there is an important difference: each key may be randomly created in the corresponding client, whereas we impose no constraints on the values of the secrets and insist they are produced by the server.

Proceeding in this fashion, let k_j be the key for the agreed symmetric cryptosystem randomly generated by client j. Provided there is a way to get k_j to S without revealing its value to any other client, and without revealing which client it comes from, then the necessary encryption of the secrets described above can be performed. So scrambling is needed for this stage of the protocol too.

To ensure that only S and client j know k_j, the first thing client j does is to encrypt k_j with the server's public key producing $\kappa_j = E_{p_S}(k_j)$, and it is the value κ_j that will be scrambled among the clients via S and then for efficiency passed in parallel, one per client to S and thereupon decrypted by S using $k_j = D_{s_S}(\kappa_j)$. Suppose for each j that κ_j is sent to S from client i defining a hidden permutation $j = \sigma(i)$. If S uses the key obtained from client i to encrypt secret x_i and sends it to client i for each i in parallel for efficiency as discussed above, then the value sent to client i is $E_{k_j}(x_i)$ where $j = \sigma(i)$. The only client able to decrypt this is client $j = \sigma(i)$. So our protocol must move $E_{k_j}(x_i)$ to client $\sigma(i)$ for each i using scrambling to guarantee that the server does not know anything of σ.

In summary, each client j produces its own random symmetric key k_j and encrypts it to $\kappa_j = E_{p_S}(k_j)$. Scrambling

implements the permutation σ^{-1} which maps $j = \sigma(i)$ to i placing κ_j at client i. Then in parallel each client i sends κ_j to S which decrypts each one using its private key, effectively obtaining $k_j = D_{s_S}(\kappa_j)$ from client i for each i. We may imagine that S has an array K for storing these values as they arrive, so $K[i] = k_j$ where $j = \sigma(i)$ but S does not know the value of j. Now in parallel S encrypts each secret with the corresponding key and sends it back to client i. Scrambling now implements the permutation σ so that the message from the server arrives at client $j = \sigma(i)$. Since the message is encrypted by client j's public key, only client j is able to decrypt it.

The generic protocol specified above arranges that client $j = \sigma(i)$ has secret x_i in accordance with the hidden permutation σ. This is tantamount to a reduction of the main problem to the implementation of *scrambling* as informally described above, i.e., to the problem of randomly obtaining and securely implementing a hidden permutation of data σ and its inverse σ^{-1} among the clients. We defer effective implementation of scrambling until later.

One possible problem here is that by colluding with some compromised client, the server is able to able to decrypt some secrets in the transmission. To prevent the server from such a traffic analysis, we add another layer of encryption. If κ_j is first encrypted with client i's public key, then S can no longer recover k_j transferred between the clients. When the encrypted message from client j arrives at client i, it is decrypted using client i's private key to recover κ_j. In order for the client j to perform the encryption, it needs to get the authenticated public key p_i. Using scrambling to move the value i to client j is not adequate as it leaves a trail in any client indicating where it starts from, which is an information leak about the hidden permutation σ. Since p_i uniquely identifies i to S, moving p_i is just as bad as moving i.

It is convenient to regard this problem as an instance of a general one, that ideally scrambling to implement the hidden permutation σ should have end-to-end encryption, i.e., when a value at client i is scrambled to client $j = \sigma(i)$, it should first be encrypted with the public key of the destination p_j so that it is meaningless *en route* inside clients other than i and j, and a similar condition should hold when implementing σ^{-1} with scrambling. In this way scrambling is relieved of the responsibility of moving meaningful data without giving away information about the permutation implemented.

To achieve this objective, for all pairs of clients (i, j) where $i = \sigma(j)$, we need to solve the following problems. How could client i and j verifiably and secretly obtain each other's identity? A solution is for each client i to create a *new* random public and private key pair $(\widetilde{p}_i, \widetilde{s}_i)$ and send $\widetilde{p}_i$ via scrambling to client $j = \sigma(i)$. Then each client j computes a signed message containing a random salt and its index j using its authenticated private key $\text{sign}_j = D_{s_j}(\text{salt}, j)$ and using $\widetilde{p}_i$ encrypts it to $e\text{sign}_j = E_{\widetilde{p}_i}(j, \text{sign}_j)$, which is scrambled to client i where it is decrypted using $\widetilde{s}_i$ to produce j and sign_j.

Now client i can use j to obtain the authenticated p_j and can confirm the value of j by using p_j to decrypt sign_j, which will produce j if it is genuine. At this point client i has a validated $j = \sigma(i)$ and the validated public key p_j belonging to client j. A reciprocal communication with client j of the same form encrypted this time with p_j can put client j in the same situation in regard to client i. Note that unlike p_i, $\widetilde{p}_i$ is not made public, such that no one is able to match $\widetilde{p}_i$ to its owner i.

C. Proposed Protocol

Putting all the above discussions together, we present our protocol with three phases: 1) initialization; 2) clients to server; and 3) server to clients.

1) Initialization: To start the protocol, each client generates two pairs of public/private keys: 1) (p_{i_1}, s_{i_1}) and 2) (p_{i_2}, s_{i_2}). The server also generates two pairs of public/private keys, (p_{S_1}, s_{S_1}) and (p_{S_2}, s_{S_2}). The first pair is used for encryption, while the second pair is used for signing. As mentioned before, we assume that given any client or the server, an entity can obtain the public key p_i of client i or the public key p_S of the server.

In this phase, the clients establish their connections with each other, resulting in $n/2$ pairs of clients. We call i as j's *matching client* if i and j are in a pair. i and j knows each other and their communication is built through the help of the other users. It is noted that the initialization phase is used to build connections among the clients, and it is no longer needed after the clients are matched with each other. The detailed steps are listed as follows.

1) Scrambling sets up a random hidden permutation σ among the clients.
2) For $0 \leq i < n$ in parallel client i generates a new pair of public/private key pair $(\widetilde{p}_i, \widetilde{s}_i)$.
3) For $0 \leq i < n$ scrambling is used to move $\widetilde{p}_i$ to client $j = \sigma(i)$, which checks that a single value is received.
4) For $0 \leq j < n$ in parallel client j computes $\text{sign}_j = \text{Sig}_{s_{j_2}}(\text{salt}, j)$ with a random salt.[1]
5) For $0 \leq j < n$ in parallel client j computes $e\text{sign}_j = E_{\widetilde{p}_i}(j, \text{sign}_j)$.
6) For $0 \leq j < n$ scrambling is used to move $e\text{sign}_j$ to client $i = \sigma^{-1}(j)$, which checks that a single value is received.
7) For $0 \leq i < n$ in parallel client i computes $(j, \text{sign}_j) = D_{\widetilde{s}_i}(e\text{sign}_j)$.
8) For $0 \leq i < n$ in parallel client i uses j to get the authenticated public key p_j.
9) For $0 \leq i < n$ in parallel client i computes $\text{Ver}_{p_{j_2}}(\text{sign}_j)$. If the signature verification succeeds, move to the next step; otherwise the message will be discarded.

The above steps successfully build the scrambling, and $n/2$ pair of matching clients are created.

2) Clients to Server: After the initialization, each client j needs to send a secret key k_j to the server through the help of other users. Here k_j is encrypted using both server's and j's matching client's public key for confidentiality. Upon receiving k_j, the server uses it to encrypt the secret x_j and sends the secret to the corresponding client. The detailed steps are listed as follows.

1) For $0 \leq j < n$ in parallel client j randomly creates k_j and computes $\kappa_j = E_{p_{S_1}}(k_j)$.

[1]Each salt used in this protocol is generated separately to guarantee the messages look random and different.

2) For $0 \le j < n$ in parallel client j signs κ_j by computing $\text{sign}_{\kappa_j} = \text{Sig}_{s_{j_2}}(\text{salt}, \kappa_j)$.
3) For $0 \le j < n$ in parallel client j encrypts sign_{κ_j} by computing $e\text{sign}_{\kappa_j} = E_{p_{i_1}}(\kappa_j, \text{sign}_{\kappa_j})$.
4) For $0 \le j < n$ scrambling is used to move $e\text{sign}_{\kappa_j}$ to client $i = \sigma^{-1}(j)$.
5) For $0 \le i < n$ in parallel client i computes $(\kappa_j, \text{sign}_{\kappa_j}) = D_{s_{i_1}}(e\text{sign}_{\kappa_j})$.
6) For $0 \le i < n$ in parallel client i computes $\text{Ver}_{p_{j_2}}(\text{sign}_{\kappa_j})$. If the signature verification succeeds, move to the next step; otherwise the message is discarded.
7) For $0 \le i < n$ in parallel client i signs κ_j by computing $\text{sign}'_{\kappa_j} = \text{Sig}_{s_{i_2}}(\text{salt}, \kappa_j)$, and sends it to the server.

The above steps illustrate how a client's key k_j is secretly delivered to the server. This key is used for the server to encrypt the final secret that client j is about to receive.

3) Server to Clients: In this phase, the server obtains the secret keys k_j in parallel from the clients, uses them to encrypt each secret in $\{x_0, x_1, \ldots, x_{n-1}\}$, then sends them back to each client correspondingly.

1) For $0 \le i < n$ in parallel S computes $\text{Ver}_{p_{i_2}}(\text{sign}'_{\kappa_j})$. If the signature verification succeeds, move to the next step; otherwise the message is discarded.
2) For $0 \le i < n$ in parallel S computes $k_j = D_{s_{S_1}}(\kappa_j)$, and place k_j in $K[i]$.
3) For $0 \le i < n$ in parallel S signs the secret by computing $\lambda_i = \text{Sig}_{s_{S_2}}(\text{salt}, x_i)$.
4) For $0 \le i < n$ in parallel S computes $\chi_i = E_{K[i]}(x_i, \lambda_i)$, and sends it to client i.
5) Scrambling is used to move χ_i to client $j = \sigma(i)$.
6) For $0 \le j < n$ in parallel client j computes $(x_i, \lambda_i) = D_{k_j}(\chi_i)$.
7) For $0 \le j < n$ in parallel client j computes $\text{Ver}_{p_{S_2}}(\lambda_i)$. The successful signature verification means the client i finally gets the secret x_i from the server.

The above steps illustrate how the server deliver secrets to the each client. During this process, only the final receiver is able to decrypt his message from the server and obtain the final secret, because the message is encrypted with a secret key that only the final receiver is aware of.

V. Implementation of Scrambling

A. Basic Scrambling

An actual solution protocol may be obtained from our generic solution protocol by providing an implementation of *scrambling*. We introduce a basic scheme for scrambling in this section. In the next section, we show that it can be made secure. Later, we provide an efficient concrete construction that implements this scheme. For security, our implementation relies upon information about which permutation σ has been randomly generated. In fact, it requires $\lceil \lg n! \rceil \sim n \lg(n)$ bits of information to single out one permutation from all $n!$ of them. In our construction each of the n clients has only a logarithmic number of bits of information about which permutation has been generated.

It is well known and obvious that any permutation may be generated from a series of transpositions. In our context a transposition consists of two clients communicating and swapping the single data item they currently hold. Our scrambling scheme can implement any permutation σ of data among the clients using such transpositions. Of course different permutations can be made from different compositions of transpositions; so naively a traffic analysis by S may reveal something of which permutation is actually being implemented. To escape this, we need the implementation of any permutation as a composition of transpositions to involve identical traffic between clients no matter which permutation has been realized.

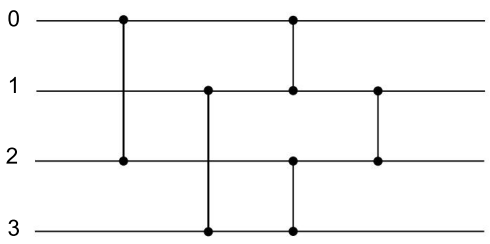

Fig. 3. Simple sorting network.

An interesting property of a transposition is that a traffic analysis does not reveal whether or not it has actually been implemented when it appears that it has, because each of the two clients may ignore the data sent to it by the other and simply keep its own data item, instead of replacing each with the ones communicated. We call a possible transposition an *exchange*. In an exchange two clients send each other their respective data items, and depending upon a shared bit of information, they either transpose or do nothing, by each, respectively, deleting their own data item or deleting the one sent by the other client. An exchange keeps a permutation between n clients concealed when $n = 2$.

By making a fixed organized composition of exchanges for a given n, we show how any permutation may be realized just by changing the shared bits associated with each of the exchanges. The permutation implemented is effectively represented by those hidden and distributed bits. Then a traffic analysis by S can reveal exactly the same pattern of communications no matter which hidden permutation is actually implemented.

Such an organized composition of exchanges is in effect a circuit with pairs of clients engaged in exchanges that either swap their data values or not according to a hidden binary switch. Such a *permutation circuit* may be illustrated in the same fashion as a sorting network as shown in Fig. 3,[2] bearing in mind our distributed interpretation of the computation.

Here each horizontal line means a client with parallel time increasing from left to right. A vertical line is an exchange with its secret bit shared between the proximate pair of clients. In this case, exchanges between clients 0 and 2 and between clients 1 and 3 may occur in parallel, followed by exchanges between clients 0 and 1 and between clients 2 and 3 in parallel, followed by an exchange between clients 1 and 2. Depending upon whether an exchange is a swap or does nothing many

[2][Online]. Available: https://en.wikipedia.org/wiki/Sorting_network

different permutations may be implemented with exactly the same communication traffic.

The depth of such a circuit, 3 in this case, is a measure of the communication traffic that we insist to be $O(\lg(n))$. Such a permutation circuit may be represented in client i as an *exchange sequence* τ_i. For example client 2 has exchanges with clients (0, 3, 1) in that order in the above circuit, and that is therefore τ_2. In a logarithmic depth permutation circuit an exchange sequence has at most logarithmic length as needed to meet our efficiency bounds.

We exhibit later a permutation circuit of depth about $2\lg(n)$ capable of implementing any permutation, as well as having a very fast algorithm to compute the exchange sequence for any client. Henceforth the term *permutation circuit* can imply that any permutation can be implemented. A random permutation may then be obtained by each pair of clients involved in an exchange by jointly and randomly choosing a single bit. In this way only two clients know whether a given exchange is a swap or not, and information about the overall permutation is distributed sparingly.[3]

B. Secure Scrambling

To ensure that a permutation network be useful when employed to implement scrambling in the generic solution protocol, exchanges must be encrypted against S because S might eavesdrop on the traffic. All communicated data must appear random to S which must not see the same data twice and thus infer the movement of the same data item. A client also needs to be able to check upon receipt that the data does come from the expected client.

To this end, when a data item q is sent from client i to client j, it is encrypted using client j's public key as follows. First a random salt is generated and the sequence number $\text{snum}_i[j]$ for client i's communications with client j is incremented, and then the following is computed:

$$\delta = E_{p_j}\big(\text{salt}, q, \text{snum}_i\big[j\big], i, j\big).$$

Then the value (δ, j) is sent to S which forwards δ to client j; client j decrypts it with its private key, giving

$$\big(\text{salt}, q, \text{snum}_i\big[j\big], i, j\big) = D_{s_j}(\delta)$$

which it can then use to filter communications into separate queues, one for each client of origin i, maintained in order of sequence number for client i, while checking that all communications are intended for itself, client j. The parts of the protocol that wait to receive data from a client can then block until the communication from the appropriate client with the next sequence number is available. This mechanism is assumed in what follows.

1) Choosing Random Permutation: To set up a permutation circuit ready to implement a random permutation, each client first computes its exchange sequence. Then it iterates through that sequence sending a random bit to each client in it, as well as collecting the random bits sent to it from each such client. By ex-oring the bit sent to a client with its own random bit, a random bit is jointly agreed with the client on the other end of an exchange, and this is stored in a sequence parallel to the exchange sequence, the *decision sequence* d_i. Here is the distributed algorithm to choose one such bit for each exchange.

1) For $0 \le i < n$ in parallel, client i does the following.
   a) Compute the exchange sequence τ_i from (i, n) for the permutation circuit being used.
   b) Set up a dictionary snum_i with $\text{snum}_i[i'] = 0$ for each client i' in τ_i.
   c) For each $0 \le k < |\tau_i|$ do:
      i) get the authenticated public key $p_{i'}$ of client $i' = \tau_i[k]$;
      ii) increment $\text{snum}_i[i']$ where $i' = \tau_i[k]$;
      iii) generate a random bit b_k and a random salt, and encrypt the message
      $$\beta_k = E_{p_{i'}}\big(\text{salt}, b_k, \text{snum}_i\big[i'\big], i, i'\big)$$
      and send β_k to client $i' = \tau_i[k]$;
      iv) block until the next communication from client $i' = \tau_i[k]$ is received and decrypted with $D(s_i, \cdot)$. Before decryption the message should be
      $$\beta'_k = E_{p_i}\big(\text{salt}, b'_k, \text{snum}_j[i], i', i\big) \quad (1)$$
      and so client i''s random bit b'_k is obtained;
      v) assign $d_i[k]$ the value $b_k \oplus b'_k$.

In summary, after a random permutation has been set up, client i has an exchange sequence τ_i and a decision sequence d_i both consistent with a permutation circuit, defined as follows:

$$\begin{aligned} \tau_i[k] = &\text{ the } k\text{th client for client } i \\ &\text{to execute an exchange with} \\ d_i[k] = &\text{ whether that exchange with } \tau_i[k] \\ &\text{is a swap or not.} \end{aligned}$$

The exchange sequences of all the clients define the permutation circuit being used, and the decision sequences of all the clients define the random permutation σ that circuit implements, by defining which exchanges are real transpositions and which do nothing.

Scrambling implements both hidden permutation σ and its inverse σ^{-1}. The latter is implemented by running the hidden permutation circuit from right to left, undoing each transposition that left to right evaluation would have performed in the reverse order, corresponding to using the exchange and decision sequences in reverse. Distributed algorithms to perform σ and σ^{-1} may therefore be the same apart from this reversal.

2) Implementing the Hidden Permutation: The generic protocol uses scrambling to move data q_i at client i to client $j = \sigma(i)$. The following distributed algorithm uses the exchange and decision sequences defining σ to effect this, assuming that the algorithm to set up σ at random has been executed, and the public keys and dictionaries of sequence numbers found and used there have been retained.

1) For $0 \le i < n$ in parallel client i does the following.
   a) Assign the data item to be scrambled to client $j = \sigma(i)$ encrypted with a random salt using the

[3] A random permutation may also be obtained by using a sorting network to sort random values, one per client, which elegantly gives exactly an equal probability to each possible permutation. However, this is too inefficient in practice to meet our parallel communication and time constraints.

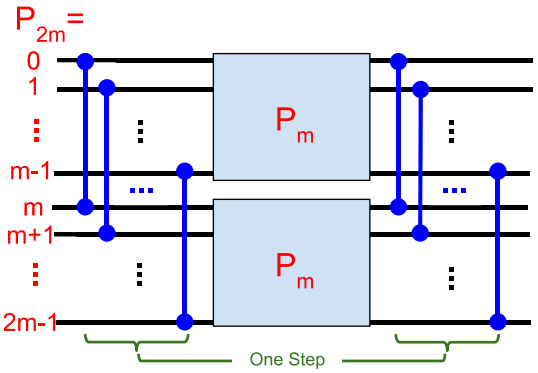

Fig. 4. Permutation circuit when n is an even number.

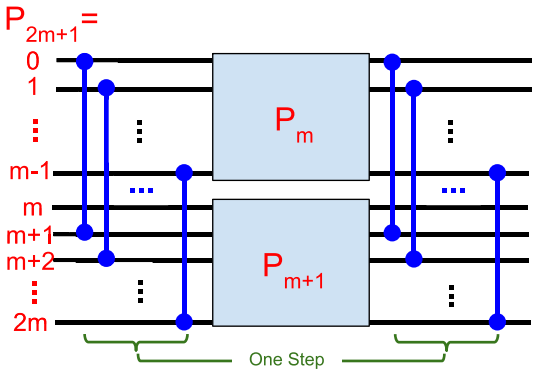

Fig. 5. Permutation circuit when n is an odd number.

destination's public key

$$Q_i := E_{p_j}(\text{salt}, q_i).$$

b) For each $0 \le k < |\tau_i|$ do:
   i) increment $\text{snum}_i[i']$, where $i' = \tau_i[k]$;
   ii) encrypt Q_i and a random salt

$$\delta_k = E_{p_{i'}}\left(\text{salt}, Q_i, \text{snum}_i\left[i'\right], i, i'\right)$$

   and send δ_k to client $i' = \tau_i[k]$;
   iii) block until the next communication from client $i' = \tau_i[k]$ is received and decrypted with $D(s_i, \cdot)$. Before decryption that should be

$$\delta'_k = E_{p_i}\left(\text{salt}, Q_{i'}, \text{snum}_{i'}[i], i', i\right)$$

   so on decryption $Q_{i'}$ is obtained;
   iv) if $d_i = 1$ then assign $Q_i := Q_{i'}$, i.e., perform the swap.

2) For $0 \le j < n$ in parallel client j does the following.
   a) Compute $D_{s_j}(Q_j)$ which contains q_i, where $j = \sigma(i)$.
   b) Assign $q_j := q_i$.

The algorithm to implement σ^{-1} is identical, apart from using the reverse of τ_i and d_i so as to evaluate the permutation circuit from right to left instead of left to right.

VI. Efficient Permutation Circuit

A. Construction of the Circuit

We give a recursive construction of a permutation circuit satisfying all aforementioned requirements. Then we give an efficient algorithm for each client to compute its exchange sequence, and a proof that it does indeed implement all permutations of n, and finally we give a proof that the probability distribution of permutations generated randomly from our circuit is approximately uniform.

The base of the construction consists of trivial permutation circuits for $n = 1$ and $n = 2$. The first has no exchanges and the second consists of a single exchange. The recursive part of the construction has two cases, even n and odd n. Fig. 4 illustrates the case for even $n = 2m$, which consists of m exchanges in parallel followed by two recursive permutation circuits, each of m elements, followed by the same m exchanges in parallel as before the recursion.

Fig. 5 illustrates the case for odd $n = 2m + 1$, which consists of m exchanges in parallel followed by two recursive permutation circuits, one of $m = \lfloor n/2 \rfloor$ elements and one of $m + 1 = \lceil n/2 \rceil$ elements, followed by the same m exchanges in parallel as before the recursion. Note that client m has no exchanges outside of the recursion.

The depth of this permutation circuit is given by $D(n) = D(\lceil n/2 \rceil) + 2$ where $D(1) = 0$ and $D(2) = 1$, which has solution $D(n) = 2\lceil \lg(n) \rceil - 1 \le 2\lg(n) + 1$, so that its use for scrambling involves at most about $2\lg(n)$ parallel communication steps as promised.

B. Generating Client's Exchange Sequence

In both diagrams $m = \lfloor n/2 \rfloor$, if $i < \lfloor n/2 \rfloor$ then client i is routed through the upper part of the recursion. In the even diagram $\lceil n/2 \rceil = m$ and such a client i has an exchange with client $i + m = i + \lceil n/2 \rceil$ before and after the recursion. In the odd diagram $\lceil n/2 \rceil = m + 1$ and such a client i has an exchange with client $i + m + 1 = i + \lceil n/2 \rceil$. So in both cases if $i < \lfloor n/2 \rfloor$ then i has an exchange with $i + \lceil n/2 \rceil$ followed by recursion into the top circuit of $\lfloor n/2 \rfloor$ inputs followed by an exchange with $i + \lceil n/2 \rceil$.

In the even network if $i \ge m$ then i has reciprocal exchanges to the ones mentioned above, therefore with $i - \lceil n/2 \rceil$. In the odd network if $i \ge m + 1$ then i has a similar exchange with $i - \lceil n/2 \rceil$. So in both cases if $i \ge \lceil n/2 \rceil$ then i has an exchange with $i - \lceil n/2 \rceil$ followed by recursion into the bottom circuit of $\lceil n/2 \rceil$ inputs followed by an exchange with $i - \lceil n/2 \rceil$. If we number the inputs to the recursive circuit from zero then i enters that recursive circuit at position $i - \lfloor n/2 \rfloor$, subtracting off the number of elements in the upper recursion.

Finally if neither $i < \lfloor n/2 \rfloor$ is nor $i \ge \lceil n/2 \rceil$ then we are in the odd diagram with $i = \lfloor n/2 \rfloor = m$, and there is just recursion into the bottom circuit of $\lceil n/2 \rceil$ inputs at local position $i - \lfloor n/2 \rfloor = 0$.

These observations enable the exchange sequence τ_i of client i in a permutation circuit of n inputs up to just before the point where recursion stops to be computed by the following simple loop:

```
j := i  //index of client i inside current recursive circuit
while n > 2 do
    if j < ⌊n/2⌋ then
        print i + ⌈n/2⌉
```

$n \coloneqq \lfloor n/2 \rfloor$ //size of the recursive circuit
else if $j \geq \lceil n/2 \rceil$ then
print $i - \lceil n/2 \rceil$
$j \coloneqq j - \lfloor n/2 \rfloor$ //input position in the recursive circuit
$n \coloneqq \lceil n/2 \rceil$
else //n is odd and $j = \lfloor n/2 \rfloor$
$j \coloneqq 0$
$n \coloneqq \lceil n/2 \rceil$.

After the loop terminates, if $n = 2$ it is clear that there is a single exchange in the middle of the sequence with client $i+1$ when $j = 0$ and client $i - 1$ when $j = 1$ (otherwise $n = 1$ at this point, and there is no exchange at the mid point). This is then followed by the reverse of the sequence produced above by the mirror symmetry of the construction. Thus we have a $O(\lg(n))$ algorithm to produce the exchange sequence of client i in a permutation circuit of n clients as promised.

C. Implementing Any Permutation With P_n

We prove by induction on n that any permutation may be implemented by P_n. The base cases are trivial: when $n = 1$ no exchange is needed to implement the single possible permutation; and when $n = 2$ a single exchange can implement both possible permutations.

Now let $n = 2m$ be even for $n > 2$. Suppose we are given an arbitrary permutation $\sigma : \{0, \ldots, n-1\} \to \{0, \ldots, n-1\}$. We now argue how to set each exchange in P_n so as to implement σ using the inductive hypothesis that P_m can implement any permutation of its inputs.

Start from an arbitrary i entering on the left-hand side of P_n and using the first exchange it encounters to map it to either instance of P_m from which there is a unique exit point if it is to reach its destination $j = \sigma(i)$ on the right. (We assume in sequel that any such mapping through the recursive instances of P_m is realizable by the inductive hypothesis.) When i emerges from the recursive P_m there is a unique setting for the final exchange on the right which we choose so that i reaches position j.

Next we find the companion j' of j in that exchange on the right, which has now been routed going right to left into the other recursive P_m from which there is a unique exit point if it is to reach its source $i' = \sigma^{-1}(j')$ on the left. Now set the exchange it reaches on the left so that it reaches i', and go to the companion in that exchange and route it forward as before, continuing in this manner until eventually the companion of the initial i in its first exchange is encountered. If there are any unrouted values on the left, start again with any one and proceed as before. Eventually the top level exchanges will be entirely set to swap or not, and the permutation will be implemented assuming the maps discovered for the two recursive P_m's can be which is so by the inductive hypothesis.

Now let $n = 2m+1$ be odd for $n > 2$. Suppose we are given an arbitrary permutation $\sigma : \{0, \ldots, n-1\} \to \{0, \ldots, n-1\}$. We extend σ by adding an additional element -1 with $\sigma(-1) = -1$ and we add exchanges to the recursive construction between -1 and m before and after the recursion, and we temporarily modify the upper recursive P_m into a P_{m+1} to account for the extra input in the obvious way.

We argue as in the even case, except with an effective n value of $2m+2$ and two recursive instances of P_{m+1}. The same argument shows that σ may be implemented at the top level with implied maps to be implemented by the two recursive circuits, carefully starting the construction with $i = -1$ on the left, routed into the upper instance of P_{m+1} so that the two new exchanges are not used as swaps. At the end there is an implied map for the upper instance of P_{m+1} that maps -1 to -1 and that part of the map is simply ignored when replacing it by an instance of P_m which will implement the rest of the implied map of the upper half by the inductive hypothesis. The -1 line now does not interact with anything and may be removed, and this completes the proof.

VII. Protocol Analysis

A. Security Analysis

1) Confidentiality: In the initialization phase, scrambling is used for all communications, which hides all the information from the server. The only information a relay m can capture is the temporary public key $\widetilde{p}_i$ of client i on the path where m is a relay. This, however, does not give any valuable information to client m because client m is not able to match $\widetilde{p}_i$ to its owner i.

After the initialization phase, a client encrypts its message using both server's public key p_S and its matching client's public key p_i before sending the message out using scrambling. With scrambling, a message is encrypted once again using its successor's public key. Therefore, no entity is able to decrypt the message without the knowledge of all the private keys.

Therefore, we claim that confidentiality is achieved in our protocol.

2) Integrity: In the proposed protocol, all the messages sent through scrambling are signed with the sender's private keys. The messages sent from the server to the clients are also signed using the server's private key. To break the integrity, an adversary must be able to forge the private keys of the clients and the server, which cannot be done with a non-negligible probability. Furthermore, under our semihonest model, we do not consider the existence of a malicious relay who replaces other clients' messages to corrupt the system. Therefore, the integrity of the messages is well protected.

3) Collusion Attack Resistance: We perform an analysis on collusion attack, since in real world the participants can be inclined to collude with each other to infer others' private information. In this protocol, the clients can collude with each other or collude with the server in order to recover the permutation. We list the two scenarios below.

1) Clients collude with each other. Since the message sent out from a client is encrypted using the server's public key p_S, no client is able to decrypt it without the server's help. Therefore, even all the clients collude with each other, they are not able to decrypt any message.
2) Clients collude with the server. Suppose client m is a user who colludes with the server. Since m is on a logarithmic number of paths between pairs of clients (i, j), where $j = \sigma(i)$, m can obtain messages from $2\lg(n)$ clients. However, these messages are encrypted using

TABLE I
COMPUTATIONAL COST OF $n \times 1$-OUT-OF-n OT

	Server	One Client
Initialization	0	$6\lg(n) * R + 4R$
Clients to Server	0	$2\lg(n) * R + 6R$
Server to Clients	$3n * R + n * C$	$2\lg(n) * R + R + C$
Total	$3n * R + n * C$	$10\lg(n) * R + 11R + C$

both server's public key p_S and matching client i's public key p_i. Therefore, without p_i, the server and client m are not able to decrypt the messages even when they collude.

In fact, secure scrambling sets up a hidden permutation that hides all the traffic flow from the server. Furthermore, another layer of encryption using the public key of the sender's matching client hides the information from both server and other relays. Therefore, collusion attack in our semihonest model is far from practical. To make collusion attacks work, there must exist some malicious users who refuse to follow the protocol correctly at some point of the protocol. We give a situation below for further discussion.

4) MITM Attack Under Malicious Model: Here we consider a particular situation where the server and malicious users collude with each other to gain benefits. Note that this situation is under a malicious model, and we discuss this for an extension of the semihonest model.

Suppose the server colludes with a malicious client m. Client m is able to launch a man-in-the-middle (MITM) attack at the beginning of the initialization phase by replacing $\widetilde{p}_i$ with its public key $\widetilde{p}_m$, then both client i and its matching client j will eventually think they are matched to client m after the initialization has finished. By doing this, client m is able to impersonate the matching client of both i and j. Therefore, after the server distributes a secret x_i to client m, m will forward the secret to client i and tell the server the identity of i. Therefore, the server is able to know which client the secret is forwarded to. Fortunately the problem here is not severe, since with a million clients, there will be only 40 pairs of (i, j) under the MITM attack.

We consider *pooling* to be a practical way to detect such an MITM attack. The idea is illustrated as follows. Suppose i and j are in a pair and m is a relay on the path from i to j. Client i hashes the identity of j and pools the result. If all the participants are honest, then each hash value in the pool should appear exactly once. However, if there is a malicious user m launching an MITM attack by impersonating i and j, the hash of m will occur more than once since both i and j will pool the hash of m. Therefore MITM is detected, and both i and j can see that m is a malicious user.

5) Replay Attack Resistance: Since the permutation circuit is public and the time line is clear, it is hard for an attacker to launch a replay attack. For any client i, a message comes from client i' with a sequence number $\text{snum}_i[i']$ smaller than the one that client i is waiting for will be automatically discarded. In this way, all replayed messages can be identified and discarded.

B. Performance Evaluation

In this section, we evaluate the computational overhead as well as communication overhead. We use M to denote the message size, n to denote the number of clients, C to denote the computational overhead for one symmetric operation, and R to denote the computational overhead for one asymmetric operation. For simplicity, here we treat the computational cost of decryption and encryption the same.

1) *Computational Overhead:* Table I summaries the computational cost for the proposed $n \times 1$-out-of-n OT. We analyze the computational cost for both server and client in each phase of the protocol.

   To start analysis, let us first compute the cost for each client running scrambling once. When secure scrambling is used for data transmissions, each relay needs to decrypt a message from its predecessor, encrypt the message, and send the ciphertext to its successor. Since each relay communicates with $\lg(n)$ other clients, the computational cost is $2\lg(n) * R$.

   a) In the initialization phase, secure scrambling is used three times, resulting in a computational cost of $2\lg(n) * R * 3 = 6\lg(n) * R$ for each client. Meanwhile, each client j sends its identity j to its matching client i, which involves one encryption, one signing operation, one decryption, and one signature verification, resulting in a computational cost of $4R$. Therefore, the total computational cost for each client in the initialization phase is $6\lg(n) * R + 4R$.

   b) In the clients to server phase, each client j encrypts its message using the server's public key p_S, then signs the message and encrypts it using its matching client's public key p_i before sending it out using scrambling, at the cost of $3R$. Then scrambling is used once, resulting in a computational cost of $2\lg(n) * R$ for each client. Finally, as a matching client of the sender, each client decrypts the message, verifies the signature of the sender, signs the message using its own private key, and sends it to the server at the cost of $3R$. Therefore, the total computational cost for each client in the *clients to server* phase is $2\lg(n) * R + 3R + 3R = 2\lg(n) * R + 6R$.

   c) In the server to clients phase, server verifies the signature and decrypts the message from each client at the cost of $2n * R$, signs each secret x_i at the cost of $n * R$, and encrypts each message using the symmetric key k_j at the cost of $n * C$. Hence, the total computational cost for the server is $2n * R + n * C$. After the secrets are sent to the clients from the server, scrambling is used once, resulting in a computational cost of $2\lg(n) * R$ for each client. Finally, each recipient j decrypts the message using the symmetric key k_j and verifies the signature of the server, resulting in a computational cost of $R + C$. Therefore, the total computational cost for each client in the *server to clients* phase is $2\lg(n) * R + R + C$.

TABLE II
COMMUNICATION COST OF $n \times 1$-OUT-OF-n OT

	Communication Overhead
Initialization	$3n\lg(n) * M$
Client to Server	$n\lg(n) * M + n * M$
Server to Client	$n\lg(n) * M + n * M$
Total	$5n\lg(n) * M + 2n * M$

2) *Communication Overhead:*
Table II summaries the communication cost for the proposed $n \times 1$-out-of-n OT. We analyze the communication overhead in each phase of the protocol.
   a) In the initialization phase, secure scrambling is used three times, resulting in a communication cost of $3n\lg(n) * M$.
   b) In the *clients to server* phase, scrambling is used once, resulting in a communication cost of $n\lg(n) * M$. Also, each client needs to send the message received from its matching client to the server, rendering a communication cost of $n*M$. So the total communication cost is $n\lg(n)*M+n*M$.
   c) In the *server to clients* phase, the server sends a message to each of the clients, rendering a communication cost of $n * M$. In the communication among the clients, secure scrambling is used once, which costs $n\lg(n) * M$. Therefore, the total communication cost is $n\lg(n) * M + n * M$.

Up to now, we have demonstrated the efficiency of the proposed $n \times 1$-out-of-n OT. The computational cost of a client is $O(\lg(n))$, and the communication cost of the system is $O(n\lg(n))$. The proposed system offers a potential way to achieve an efficient anonymous communication system, where secrets can be transferred from clients to the server anonymously. At this stage, however, we only focus on solving the fundamental $n \times 1$-out-of-n OT problem defined in this paper.

VIII. CONCLUSION

In this paper, we define a novel problem termed $n \times 1$-out-of-n OT and propose a protocol that combines modern cryptography and hidden permutation to efficiently solve the problem. Particularly, the computational complexity of the permutation is $O(\lg(n))$, and the communication overhead is $O(n\lg(n))$. The hidden permutation can be used to implement an anonymous communication system.

Our future research will focus on the following two directions: developing novel mechanisms that can support different oblivious transfer applications such as the case when clients join the system asynchronously; and employing $n \times 1$-out-of-n OT to design more efficient crypto protocols such as group signatures.

REFERENCES

[1] L. Atzori, A. Iera, and G. Morabito, "The Internet of Things: A survey," *Comput. Netw.*, vol. 54, no. 15, pp. 2787–2805, 2010.

[2] R. Li *et al.*, "IoT applications on secure smart shopping system," *IEEE Internet Things J.*, vol. 4, no. 6, pp. 1945–1954, Dec. 2017.

[3] T. Song *et al.*, "A privacy preserving communication protocol for IoT applications in smart homes," *IEEE Internet Things J.*, vol. 4, no. 6, pp. 1844–1852, Dec. 2017.

[4] F. D. Garcia and B. Jacobs, "Privacy-friendly energy-metering via homomorphic encryption," in *Proc. Int. Workshop Security Trust Manag.*, 2010, pp. 226–238.

[5] K. Kursawe, G. Danezis, and M. Kohlweiss, "Privacy-friendly aggregation for the smart-grid," in *Proc. Int. Symp. Privacy Enhancing Technol. Symp.*, 2011, pp. 175–191.

[6] Z. Erkin and G. Tsudik, "Private computation of spatial and temporal power consumption with smart meters," in *Proc. Int. Conf. Appl. Cryptography Netw. Security*, 2012, pp. 561–577.

[7] G. Ács and C. Castelluccia, "I have a DREAM! (Differentially private smart metering)," in *Proc. Int. Workshop Inf. Hiding*, 2011, pp. 118–132.

[8] C. Dwork, K. Kenthapadi, F. McSherry, I. Mironov, and M. Naor, "Our data, ourselves: Privacy via distributed noise generation," in *Proc. Annu. Int. Conf. Theory Appl. Cryptograph. Techn.*, 2006, pp. 486–503.

[9] V. Rastogi and S. Nath, "Differentially private aggregation of distributed time-series with transformation and encryption," in *Proc. ACM SIGMOD Int. Conf. Manag.*, 2010, pp. 735–746.

[10] E. Shi, H. Chan, E. Rieffel, R. Chow, and D. Song, "Privacy-preserving aggregation of time-series data," in *Proc. Annu. Netw. Distrib. Syst. Security Symp. (NDSS)*, 2011, pp. 193–213.

[11] D. L. Chaum, "Untraceable electronic mail, return addresses, and digital pseudonyms," *Commun. ACM*, vol. 24, no. 2, pp. 84–90, 1981.

[12] K. Sampigethaya and R. Poovendran, "A survey on mix networks and their secure applications," *Proc. IEEE*, vol. 94, no. 12, pp. 2142–2181, Dec. 2006.

[13] D. Goldschlag, M. Reed, and P. Syverson, "Onion routing," *Commun. ACM*, vol. 42, no. 2, pp. 39–41, 1999.

[14] M. G. Reed, P. F. Syverson, and D. M. Goldschlag, "Anonymous connections and onion routing," *IEEE J. Sel. Areas Commun.*, vol. 16, no. 4, pp. 482–494, May 1998.

[15] M. K. Reiter and A. D. Rubin, "Crowds: Anonymity for Web transactions," *ACM Trans. Inf. Syst. Security*, vol. 1, no. 1, pp. 66–92, 1998.

[16] M. O. Rabin, "How to exchange secrets by oblivious transfer," Aiken Comput. Lab., Harvard Univ., Cambridge, MA, USA, Rep. TR-81, 1981.

[17] S. Even, O. Goldreich, and A. Lempel, "A randomized protocol for signing contracts," *Commun. ACM*, vol. 28, no. 6, pp. 637–647, 1985.

[18] M. Naor and B. Pinkas, "Oblivious polynomial evaluation," *SIAM J. Comput.*, vol. 35, no. 5, pp. 1254–1281, 2006.

[19] B. Aiello, Y. Ishai, and O. Reingold, "Priced oblivious transfer: How to sell digital goods," in *Proc. Int. Conf. Theory Appl. Cryptograph. Techn.*, 2001, pp. 119–135.

[20] S. Laur and H. Lipmaa, "A new protocol for conditional disclosure of secrets and its applications," in *Applied Cryptography and Network Security*. Berlin, Germany: Springer, 2007, pp. 207–225.

[21] G. Brassard, C. Crépeau, and J.-M. Robert, "All-or-nothing disclosure of secrets," in *Proc. Conf. Theory Appl. Cryptograph. Techn.*, 1986, pp. 234–238.

[22] M. Naor and B. Pinkas, "Oblivious transfer with adaptive queries," in *Proc. Annu. Int. Cryptol. Conf.*, 1999, pp. 573–590.

[23] T. Chou and C. Orlandi, "The simplest protocol for oblivious transfer," in *Proc. Int. Conf. Cryptol. Inf. Security Lat. America*, 2015, pp. 40–58.

[24] W.-G. Tzeng, "Efficient 1-out-of-n oblivious transfer schemes with universally usable parameters," *IEEE Trans. Comput.*, vol. 53, no. 2, pp. 232–240, Feb. 2004.

[25] A. Jain and C. Hari, "A new efficient k-out-of-n oblivious transfer protocol," *arXiv preprint arXiv:0909.2852*, Sep. 2009. [Online]. Available: https://arxiv.org/pdf/0909.2852.pdf

[26] J.-S. Chou, "A novel k-out-of-n oblivious transfer protocol from bilinear pairing," *Adv. Multimedia*, vol. 2012, Jan. 2012.

[27] J. Arndt, "Generating random permutations," M.S. thesis, Dept. Math., Aust. Nat. Univ., Canberra, ACT, Australia, 2010.

[28] P. Diaconis and M. Shahshahani, "Generating a random permutation with random transpositions," *Probab. Theory Related Fields*, vol. 57, no. 2, pp. 159–179, 1981.

[29] A. Czumaj, "Random permutations using switching networks," in *Proc. 47th Annu. ACM Symp. Theory Comput.*, 2015, pp. 703–712.

[30] M. Naor and O. Reingold, "On the construction of pseudorandom permutations: Luby–Rackoff revisited," *J. Cryptol.*, vol. 12, no. 1, pp. 29–66, 1999.

[31] C. Rackoff and D. R. Simon, "Cryptographic defense against traffic analysis," in *Proc. 25th Annu. ACM Symp. Theory Comput.*, 1993, pp. 672–681.

Authors' photographs and biographies not available at the time of publication.