Oblivious Assignment with m Slots

G. ATENIESE[‡], R. BALDONI[†], S.BONOMI[†], G. A. DI LUNA[†]

[†] Dipartimento di Ingegneria Informatica, Automatica e Gestionale Antonio Ruberti
Universitá degli Studi di Roma La Sapienza
Via Ariosto, 25
I-00185 Roma, Italy
{baldoni, bonomi, diluna}@dis.uniroma1.it

[‡] Dipartimento di Informatica
Universitá degli Studi di Roma La Sapienza
Via Salaria, 113
I-00198 Roma, Italy
ateniese@di.uniroma1.it

MIDLAB TECHNICAL REPORT 2/12 2012

Abstract

Preserving anonymity and privacy of customer actions within a complex software system, such as a cloud computing system, is one of the main issues to be solved in order to boost private computation outsourcing. In this paper, we propose a coordination paradigm, namely oblivious assignment with m slots of a resource $\mathcal{R}$ (with $m \geq 1$), allowing processes to compete in order to get a slot of $\mathcal{R}$, while ensuring at the same time both fairness in the resource slots assignment and that no process knows which slot of $\mathcal{R}$ is assigned to a process. We study oblivious assignment with m slots solvability issues related to the message pattern of the algorithm. We also present a distributed algorithm solving oblivious assignment with m slots within a distributed systems, assuming the existence of at least two *honest* processes and $m \leq n$ (where n is the number of processes). The algorithm is based on a rotating token paradigm and employs an adaptation of the ElGamal encryption scheme to work with multiple parties and ensuring obliviousness of the assignment. Finally, the correctness of the algorithm is formally proved.

Keywords: distributed coordination abstractions, secure computations, mutual exclusion, distributed systems.

1 Introduction

In this paper, we investigate the problem of oblivious assignment with m slots. Informally, we consider n non-anonymous processes competing for accessing one of the m slots of a resource $\mathcal{R}$. Each slot can be assigned to at most one process at a time. When the process does not need the resource anymore, it releases the slot it owns and the latter can be thus assigned to another requesting process. Note that, processes are utterly identifiable but we strive to protect the allocations of resource slots to processes. Thus, *processes are oblivious and in particular they are unaware of assignments between processes and resource slots*.

This problem is particularly interesting because it crystallizes the difficulty in coordinating processes that wish to interact with a resource without being noticed by anyone else. Resource sharing environments, channel assignments in telco systems are examples of domains where this problem can be relevant. As an example, an oblivious assignment scheme can help a group of clients of a cloud provider to hide and protect their allocation of resources within a virtualized environment or across distinct domains. Resources can thus be obliviously allocated to clients. Not even the cloud provider is aware of these various assignments.

We target organizations moving to the cloud, or outsourcing their services, that wish to access or allocate virtual resources anonymously. Cryptographic systems, such as fully homomorphic encryption [11], do not solve the oblivious assignment problem. Homomorphic encryption allows clients to perform computation over encrypted data ensuring that sensitive information remain inaccessible to the cloud provider. Nevertheless, the provider can derive which resources are allocated to which clients. This constitutes a side-channel leak we aspire to prevent. We stress that this type of side-channel has not been considered before in the context of cloud computing.

The paper firstly defines the oblivious assignment with m slots (O-mA) problem. More precisely, if an honest process p_i get a slot r_j, then no other process is aware of this assignment. We also provide a stronger form of this problem, namely strong oblivious assignment with m slots (SO-mA). In this case, given a process p_i, no other process can tell whether any slot was assigned to p_i or not. That is, it is not possible to infer whether a specific process is using a resource slot or not. We study solvability issues of O-mA and SO-mA problems related to the message pattern generated by distributed algorithms. This points out that token-based algorithms could implement SO-mA and O-mA. We also show that a trivial perpetual circulating algorithm solves our problem only in the presence of $n - 1$ honest processes (where n is the number of processes). Thus, we introduce a rotating token distributed algorithm solving O-mA and we assume the existence of at least two *honest* processes and $m \leq n$. The algorithm employs an adaptation of ElGamal encryption scheme to ensure obliviousness of the assignment. Finally, the correctness of the algorithm is formally proved. Let us remark that each non-honest process is able to detect bounds on transmission delays and when they holds. Non-honest process can use this knowledge to infer information about current slot assignments of correct processes.

The rest of the paper is organized as follows: related work is in Section 2 and the system model is defined in Section 3. Section 4 formalizes the oblivious assignment with m slots problem and provides some solvability conditions, while Section 5 presents a distributed algorithm solving the oblivious assignment problem. Finally, Section 6 concludes the paper. The proofs omitted in the text and can be found in Appendix A.

2 Related Work

Defining distributed algorithms for accessing resources in mutual exclusion has been a mainstream field of research in the eighties [18] and several efficient algorithms have been devised (e.g., [19], [21], [16] just

to cite a few). To facilitate fault tolerance without assuming failure detection, the general mutual exclusion problem has been extended to the k-mutual exclusion one [17], where at most k different processes can concurrently access the same resource; general strategies working in a failure-free environment have been adapted to solve this more general problem in an asynchronous message passing system (e.g. [14], [8]).
A different generalization of the mutual exclusion problem, namely k-assignment, has been presented in [9]. In k-assignment there are $k < n$ identical, named resources that may be requested by n processes and the authors shown that the problem can be solved in an asynchronous distributed system, as long as at most $k/2$ processes can fail.

Similarly, in the renaming problem [3], each participating process is initially associated to a unique identifier from a large name space and the final objective is to select unique identifiers from a smaller name space. Combining together renaming and k-exclusion, a more general specification, called k-assignment with m slots can be defined [4]. Informally, such a problem requires that at most k processes access concurrently one of the m distinct available slots. All these existing algorithms do not mask the assignment between slots and competing processes. On the contrary, they exploit their knowledge about assignments to minimize the number of exchanged messages.

Generally, the oblivious assignment problem can be solved using multiparty computation [22]. This is a paradigm that allows several parties to evaluate a function $f(x_1, \ldots, x_n)$, or multiple functions, without revealing the inputs $x_1, \ldots, x_n$. That is, every party p_i contributes x_i but at the end of the protocol it will only learn $f(x_1, \ldots x_n)$ and nothing else. Unfortunately, these generic techniques are notoriously very expensive and call for an exorbitant number of messages to be exchanged. However, there exist more efficient alternatives for many functionalities. The one that is more closely related to the oblivious assignment functionality is referred to as *mental poker*. Mental poker algorithms [20] allow people to play card games over networks without any trusted dealer. The basic idea is to assign cards to players such that cards stay private and can be safely shuffled. In addition, it is possible to detect cheaters. While the original scheme [20] represented each card with a large number of bits, more recent work [6] makes card sizes smaller and independent of the number of players.

The oblivious assignment problem does not fit completely within the mental poker framework. In our model, we must avoid starvation and ensure liveness and thus allow a process to pick a specific slot of a resource within a fixed amount of time (while this is not possible in mental poker). The release of a resource is also significantly simpler than discarding a card from hand. Indeed, we do not have to preserve the value of the slot (or card) and thus we can just set obliviously a boolean flag.

3 System Model

The distributed system is composed of a set of n processes $\Pi = \{p_1, p_2 \ldots, p_n\}$, each one having a unique identifier, that compete for m distinct slots $\{r_1, \ldots, r_m\}$ of a resource $\mathcal{R}$, where $m \leq n$. Each process p_i competes to get exclusive access to a slot of $\mathcal{R}$. At any time, each slot can be assigned to at most one process and allocated slots must be released within a finite period of time. Specifically, when process p_i needs to acquire one of the m slots of $\mathcal{R}$, it invokes a request() operation and waits until a grantResource() event occurs returning the id of the slot r_j assigned to p_i. To release the slot r_j, p_i invokes a release() operation. Processes do not crash.

We assume the existence of a coalition $\mathcal{C}$ (with $1 \leq |\mathcal{C}| \leq n-2$) of *honest-but-curious* processes [12]. Such processes act according to their algorithm but they can collaborate to acquire and share information about others processes. Processes not belonging to the coalition $\mathcal{C}$ are said to be *honest*, that is, they are correct and behave according to the algorithm and they do not attempt to infer other information, except the

ones obtained during the algorithm execution.

Processes coordinate their access to slots of $\mathcal{R}$ by exchanging messages. We assume that for any pair of processes $p_i, p_j \in \Pi$, there exists a reliable FIFO point-to-point communication channel connecting them. Messages are delivered "*most of the time*" within δ time units, that is the underlying communication system is synchronous most of the time. However, there could be finite periods of time where the systems behaves as asynchronous. We assume that processes belonging to the coalition $\mathcal{C}$ are powerful enough to know both the communication bound δ and if the system is in a synchronus period or not. Such processes can use this knowledge to infer information about other honest processes.

4 Oblivious Assignment with m Slots

Given a generic resource $\mathcal{R}$, it can be used concurrently by different processes; however, any of its m slot can be used in an exclusive way. Let us remark that every process can always get at most one slot of $\mathcal{R}$, that is, *the assignment of multiple slots to a single process is not allowed*. At the same time, it must be guaranteed that competing processes will eventually obtain a slot of $\mathcal{R}$. In addition, resource assignment must be kept private.

4.1 Problem Definition

The *Oblivious assignment with m Slots* (O-mA) problem is specified by the following properties:

1. UniqueAssignment : If p_i and p_j access concurrently the resource $\mathcal{R}$, then the slot r_x assigned to p_i and the slot r_y assigned to p_j are distinct.
2. LockoutAvoidance : any process p_i that requests access to resource $\mathcal{R}$, eventually is assigned a slot r_j of $\mathcal{R}$.
3. ObliviousAssignment : if a slot r_j is assigned to an honest process p_i, then no other process is deterministically aware of this assignment.

As an example, consider a distributed system composed by two honest processes, p_1 and p_2, and $n-2$ honest-but-curious processes. Let r_1 and r_2 be two slots of a resource. Suppose that, after a run of the oblivious assignment scheme both processes obtain a slot and only two assignments are possible: (i) $\langle p_1, r_1\rangle$, $\langle p_2, r_2\rangle$ or (ii) $\langle p_1, r_2\rangle$, $\langle p_2, r_1\rangle$. The ObliviousAssignment property will ensure that the coalition of $n-2$ honest-but-curious processes won't be able to determine what was the actual assignment (i.e., whether (i) or (ii) above).

4.2 Strong Oblivious Assignment with m Slots (SO-mA)

We consider a stronger variant of the O-mA problem, which is referred to as SO-mA, where it is not possible to determine whether resources are allocated to a specific process. The SO-mA problem can be defined as O-mA by replacing the ObliviousAssignment property with the following one:

StrongObliviousAssignment : Fixed a process p_i, no other process can deterministically determine whether p_i owns or not any slot of a resource.

In the previous example, the $n-2$ honest-but-curious processes may not know what was the actual assignment but they can collectively determine that certain slots were assigned to p_1 and p_2. This violates the Strong Oblivious Assignment property.

4.3 Solvability Issues for O-mA and SO-mA problems

In the following, we will show a necessary condition for an algorithm to solve O-mA and SO-mA. In particular, we will show that there exists constraints on the message pattern that any algorithm must satisfy to solve our problem.

Lemma 1 *Let A be an assignment algorithm, ensuring properties 1 and 2, executed by processes in Π. If the message pattern of A expects a process $p_i \in \Pi$ to send a request message m to another process p_j to acquire a slot and $|\mathcal{C}| \geq 1$, then A cannot solve O-mA.*

Proof Let's consider the following run where p_c is a corrupted process: if a process p_i sends a message requesting a slot r_j to p_c, then p_c will learn that p_i is about to access the slot r_j. From this time on p_c declares the assignment $\langle p_i, r_j \rangle$. Due to the fact that A satisfies properties 1 and 2, eventually p_i will access the slot and this violates property 3. $\square_{Lemma\ 1}$

Thus, assignment algorithms that are based on explicit permissions for resource allocation cannot solve O-mA and SO-mA. Examples of such algorithms in the context of distributed mutual exclusion are ([14],[16],[17],[18]). A class of algorithms that satisfies the necessary condition of Lemma 1 is the one based on a rotating coordinator approach (also called perpetual circulating token [5],[15]) as shown in the next section.

5 A Rotating Token Algorithm for Solving O-mA

5.1 Ruling out trivial perpetual circulating token algorithms

Let us consider a trivial token circulating algorithm, namely *trivial-A*.The token owner can access a slot as soon as it receives the token, without sending out any notification. Once the token owner releases the slot, the token is passed to the next process in the logical ring. This algorithm satisfies property 1 and 2 and Lemma 1

The following Lemmas show that this simple algorithm implements O-mA and SO-mA only if there is at most one honest-but-curious process..

Lemma 2 *Consider an algorithm trivial-A running on the top of the distributed systems described in Section 3 and satisfying properties 1 and 2. If $|\mathcal{C}| \geq 2$, trivial-A cannot ensure SO-mA property.*

Proof Let's consider the following run where two honest-but-curious processes are respectively the predecessor and the successor of an honest process p_i in the ring and the communication delay is bounded by δ (see Section 3). When the token reaches p_i sent by p_{i-1} and p_i decides to access the slot r_j, if p_i keeps the slot for an interval of time greater than 2δ then p_{i-1} and p_{i+1} will infer deterministically that p_i has acquired a slot. This can be accomplished by looking at the timestamps of token messages sent from p_{i-1} to p_i and from p_i to p_{i+1}. This violates the SO-mA property. $\square_{Lemma\ 2}$

The next Lemma follows directly from the previous one:

Lemma 3 *Consider a distributed system with a bound δ on message transfer delay and an algorithm trivial-A running on top of it. If $m = 1$ and $|\mathcal{C}| \geq 2$, trivial-A cannot ensure O-mA property.*

5.2 A rotating token algorithm resilient to $|\mathcal{C}| \leq n-2$ honest-but-curious processes

Our algorithm is token-based and works in rounds. Each round is led by a coordinator p_c that takes care of the token creation, encoding, and dissemination for that specific round. The token circulates on the top of a logical ring formed by the processes (i.e. each process p_i passes the token to its neighbor $p_{i+1 mod n}$). Each round is characterized by two phases, *allocation phase* (corresponding to the management of request() operations), where resource slots are allocated to processes, and *release phase*, where each process frees its assigned slot once it has done with it. A round ends when all allocated slots are released. The next round is coordinated by the process that follows p_c in the logical ring. In the following, we will use the term *ticket* to indicate a numeric representation of a slot. The coordinator will create n tickets (that is, a ticket per process in the system) regardless of the number of actual slots.

Allocation Phase: The coordinator of the current round creates a token, $request\_token$, containing a set of tickets $\{tk_1, tk_2, \ldots, tk_n\}$, each one identifying a resource slot. Only m ouf of n tickets will univocally be associated to actual slots of the resource (i.e. *valid tickets*) while the remaining $n-m$ tickets (i.e. *invalid ticket*) represent dummy slots. Invalid tickets help prevent leakage of information on actual assignments.

At the beginning of each round, the coordinator picks one ticket, encrypts the $request\_token$ via ElGamal encryption [10], and forwards the token to the next process in the ring. Upon the receipt of the token, a process p_i picks a ticket, re-encrypts the token to make it indistinguishable, and forwards it to the next process in the ring. After getting the ticket, p_i will decrypt it by asking other processes for their ephemeral keys: if the ticket is valid and p_i requested a slot of $\mathcal{R}$, then it will trigger the grantResource event.

Release Phase: The release phase starts only when the $request\_token$ returns to the coordinator. The coordinator creates a $release\_token$, used to identify the released tickets, and starts to circulate it in the logical ring. A ticket is released by a process p_i in two cases: (i) p_i did not request a slot of $\mathcal{R}$ or, (ii) p_i finished with the slot (i.e., when invoking the release() operation). Every time the $release\_token$ is passed to the next process, it is re-encrypted to avoid information leakage.

The token $release\_token$ circulates continuously till the coordinator verifies that the number of released tickets is equal to n. At this point, the round is completed and the next process in the ring becomes the new coordinator for a new round.

5.3 ElGamal Encryption with Multiple Parties

Notation and Assumptions. In the following, we use $y \leftarrow f(x)$ to indicate the assignment to y of the value obtained evaluating a function f over the input x, while we will use $y \xleftarrow{u} S$ to indicate that y is a random element uniformly selected from a set S. In the following, we will assume to have a cyclic subgroup G of prime order q and generator g where the Decisional Diffie-Hellman (DDH) assumption [7] holds. Informally, the DDH assumption states that given a triple (g^x, g^y, g^{xy}) with $x, y \xleftarrow{u} \mathbb{Z}_q$ it can be distinguished from a triple in the form (g^x, g^y, g^z), with $z \xleftarrow{u} \mathbb{Z}_q$, by using a probabilistic polynomial time algorithm, with negligible probability.
For a concrete instantiation, we consider G to be the set of quadratic residues of $\mathbb{Z}_p^*$ where p is a *safe* prime, i.e., $p = 2q+1$ with prime q. A generator g of the group G is simply found by selecting $\bar{g} \xleftarrow{u} \mathbb{Z}_p^*$ and setting $g = \bar{g}^2 \bmod p$ whenever $\bar{g} \neq 1$.

ElGamal Encryption. The idea behind ElGamal scheme is to use g^{xy} as a shared secret between sender and recipient. The private key is $y \xleftarrow{u} \mathbb{Z}_q$ while the public key is the value $g^y \in G$.
To encrypt an element $e \in G$, it is enough to randomly select an element $r \xleftarrow{u} \mathbb{Z}_q$ and compute the ciphertext

as a pair $(c_1, c_2) = (g^r, mg^{ry}) \in G \times G$. The recipient of the ciphertext (c_1, c_2) recovers m by computing $c_2/c_1^y \in G$.
Note that, under the DDH assumption, ElGamal encryption is semantically secure [7]. Intuitively, a semantically secure scheme does not leak any information about the encrypted message. In particular, given a ciphertext (c_1, c_2) of one of two messages m_0 and m_1, an adversary cannot tell which message was encrypted. This holds even if the adversary chooses both messages, as long as they are both in G.

Adaptation We adapt the ElGamal crypto-system to work with multiple parties. Each process p_i has a private key $Pr\_key_i \xleftarrow{u} \mathbb{Z}_q$, and the corresponding public key is calculated as $g^{Pr\_key_i}$. In addition, p_i also maintains the *group public key* as the value $g^Y = g^{\sum_{p_i \in \Pi} Pr\_key_i}$.
We use the ElGamal crypto-system to encrypt tickets whose values contain relevant information about slots of the resource $\mathcal{R}$ (e.g. such as network address, memory location, printer ID, etc...). Thus, generic numerical tickets must be mapped into elements of the subgroup G of quadratic residues in $\mathbb{Z}_p^*$.
The standard mapping-then-encrypt procedure works as follows: (i) Consider the ticket t as an element of $\mathbb{Z}_q$, (ii) set $\bar{t} = t + 1$, and (iii) encrypt the value $\bar{t}^2 \bmod p$. The decryption phase is more involved: (i) decrypt and recover the plaintext $\bar{m} = \bar{t}^2 \bmod p$, (ii) compute a square root of $\bar{m}$ as $m = \bar{m}^{(p+1)/4} \bmod p$, and return the ticket $m - 1$ if $m \leq q$, or $p - (m - 1)$ when $m > q$. In the rest of the paper we assume that tickets or any arbitrary messages are in G, either directly or through the mapping described above.

A ticket t is encrypted for the group of precesses as (g^r, tg^{rY}). Each process must contribute to the decryption phase in order to recover the ticket by computing the partial value $g^{rPr\_key_i}$. The product modulo p of these partial values from all processes is equal to g^{rY} which is used to recover t as in standard ElGamal. We define a function $removeLayer$ that receives as input a valid ciphertext and *removes* the component $g^{rPr\_key_i}$ from it, effectively allowing other processes to decrypt the message. This function is executed locally by the process p_i.

Notice that, ElGamal ciphertexts can easily be randomized, i.e., given a ciphertext (c_1, c_2) anyone can produce a new ciphertext (c_1', c_2') on the same message without knowing any secret key or learning the message itself. Indeed, given (g^r, tg^{rY}), it is enough to select $r^* \xleftarrow{u} \mathbb{Z}_q$ and compute a new and unlinkable ciphertext $(g^{r+r^*}, tg^{(r+r^*)Y})$. The security of this randomized ElGamal encryption still holds as shown in [13].

5.4 The Algorithm

In this section, we provide the details of the oblivious assignment scheme for our system model. In particular, we first describe the data structures maintained locally by each process p_i, then we provide the details about the coordinator selection and the round phases, i.e., the assignment phase and the release phase.
Data structures. Each process p_i maintains locally the following data structures:

- $round_i$: is an integer representing the round p_i is participating in;
- $coordinator_i$: is a boolean variable set to `true` when p_i is the coordinator for the current round, `false` otherwise;
- $state_i$: is a variable that can be set to $\{NCS, waiting, CS\}$ and it represents the state of p_i;
- $ticket_i$: is a pair $< rd, tk >$ where tk is an encrypted ticket associated to a slot (whether real or not) and rd essentially reveals the slot identifier;
- Pr_key_i/Pb_key_i: ElGamal private/public keys used to decrypt/encrypt tickets;
- $keys_i$: is a set variable, used in the assignment phase, to store all the temporary keys (i.e. *ephemeral keys*) needed to decrypt the selected ticket.

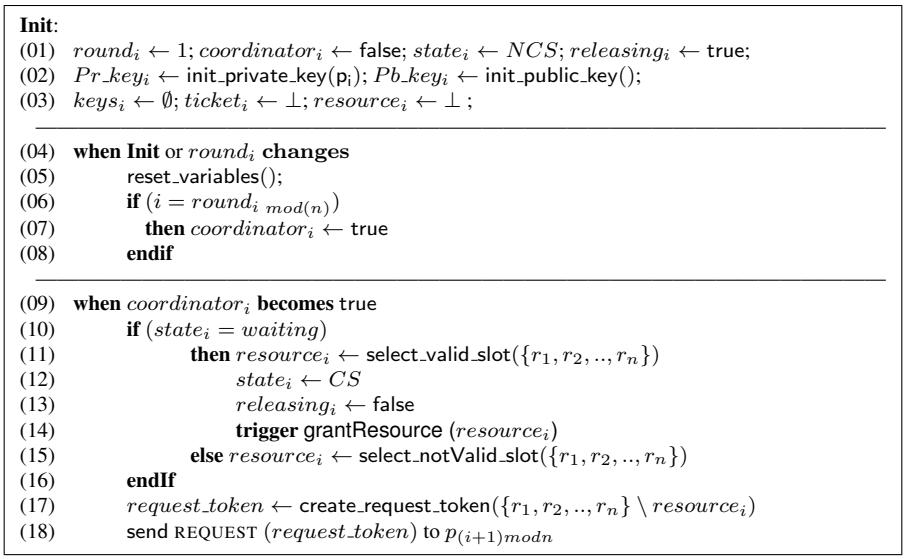

Figure 1: The rotating leader protocol (code for p_i)

• $resource_i$: is an integer representing the slot id obtained by p_i;

• $releasing_i$: is a boolean flag. It is set to true when p_i has no assigned slot of $\mathcal{R}$, false otherwise.

In addition, the algorithm also employes two tokens, namely $request\_token$ and $release\_token$. A token is essentially a set containing encrypted tickets and each ticket refers to real or dummy slots.

Round and Coordinator Change. The pseudo-code for the round and coordinator change is shown in Figure 1. We defined the following functions to simplify the code:

• init_private_key(p_i)/init_public_key(): initialize p_i's private and public keys.

• reset_variables(): reset all variables, except $round_i$, as declared into the Init statement.

• select_valid_slot($\{r_1, r_2, .., r_n\}$): given the set of (real and dummy) resource slots $\{r_1, r_2, .., r_n\}$, select a real slot.

• select_notValid_slot($\{r_1, r_2, .., r_n\}$): given the set of (real and dummy) resource slots $\{r_1, r_2, .., r_n\}$, select a dummy slot.

• create_request_token$(r_1, r_2, \ldots, r_{n-1})$: given the set of (real and dummy) resource slots $\{r_1, r_2, .., r_{n-1}\}$, creates a set of tickets and the corresponding request token.

A new round starts as soon as $round_i$ is updated (line 04 Figure 1, line 19 Figure 3) and this causes all local variable, except $round_i$, to be reset (line 05). Each process p_i checks whether it is a coordinator of the current round. If so, it sets the local $coordinator_i$ variable to true (lines 06 - 08). This triggers a new assignment phase lead by p_i (line 09). The new coordinator checks if it is in the waiting state (line 10) (that is, it is waiting for a slot) and, in that case, it selects a real slot of the resource (lines 11). Otherwise, the coordinator selects a dummy slot (line 12). After the selection, p_i creates and encrypts the $request\_token$ (line 14) and sends it to its "neighbor" $p_{i+1mod\ (n)}$ (line 15).

The request() operation and the assignment phase. The pseudo-code of the request() operation and the assignment phase is shown in Figure 2. The functions in the pseudo-code are defined as follows:

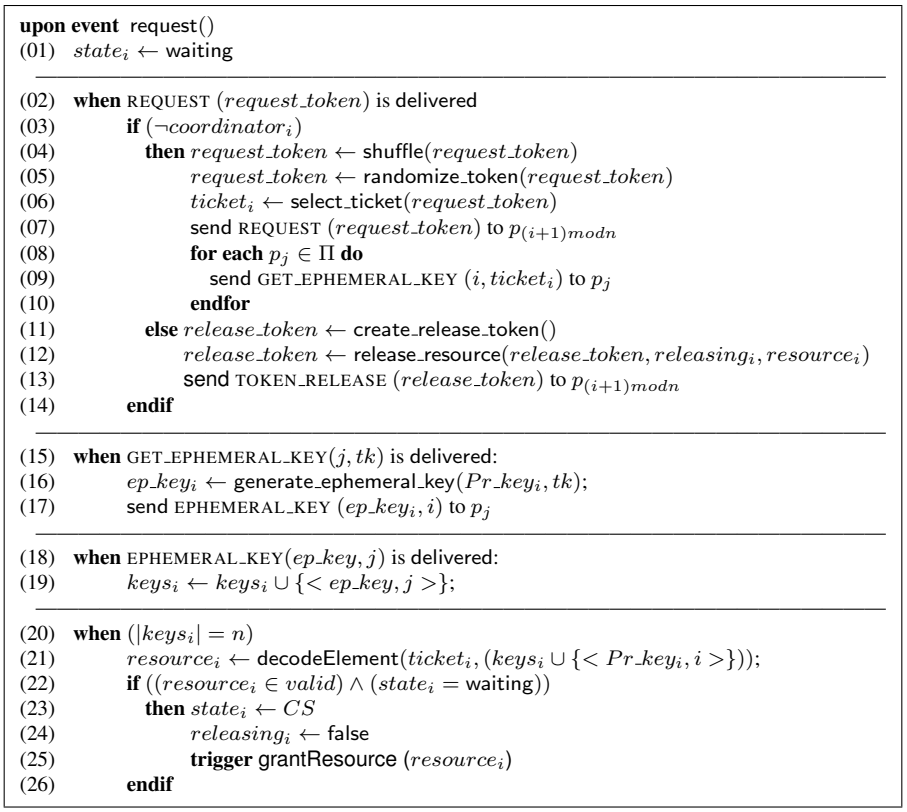

Figure 2: The request() protocol (code for p_i)

• shuffle(T): given a token T, randomly permute the sequence of tickets

• randomize_token(T): given token T, re-encrypt each ticket in T

• select_ticket(T): return a ticket tk randomly selected and removed from the token T.

• generate_ephemeral_key(tk): given a ticket $tk =< rd, \hat{r}_j >$, generate a temporary key (also called *ephemeral*) starting from the number rd included in tk, that can be used to decrypt the slot id $\hat{r}_j$.

• decodeElement($tk, \{k_1, k_2 \ldots k_j\}$): given a set of keys $\{k_1, k_2 \ldots k_j\}$ and a ticket tk, decrypt tk and return its cleartext value

• create_release_token(): create the $release\_token$ to collect released tickets.

• release_resource(T, b, r_j): given a token T, a boolean value b, and a slot r_j, process the token T according to the boolean value b. In particular, if b is true then the slot r_j is released otherwise the function does nothing.

When a process p_i needs a slot of $\mathcal{R}$, it invokes the request() operation. The variable $state_i$ is thus set to $waiting$ (line 01). When the $request\_token$ is delivered to p_i, it checks if it is the coordinator for this round (line 02). If p_i is not the coordinator, then it means that the assignment phase for this round is still running and a ticket can be chosen from the token. The selection consists of three steps: token shuffling (line 04) , token re-randomization (line 05) and finally ticket selection (line 06). Once a ticket has been selected, it has to be decrypted to recover the slot id. For this purpose, p_i sends a GET_EPHEMERAL_KEY message to other processes (lines 08 - 10).

If p_i is the coordinator, then all slots have been assigned for the current round and a release phase should

start (line 11 - 14). Hence, p_i creates the $release\_token$ (line 11), and embeds its encrypted $releasing_i$ flag into the token (line 12). Finally, the token is passed to $p_{i+1 mod\ n}$ (line 13).

When a process p_i receives a GET_EPHEMERAL_KEY(j, tk) message, it generates a temporary key that can only be used to decrypt the ticket tk (line 16) and returns it to p_jWhen all ephemeral keys are available (line 20), then p_i decrypts the ticket, recovers the slot id 21, and use the slot in case is a real one (lines 22 - 25).

The release() **operation and the release phase.** The pseudo-code of the request() operation and the assignment phase is shown in Figure 3.

```
upon event release():
(01)  state_i ← NCS
(02)  releasing_i ← true
-------------------------------------------------------------------------
(03)  when TOKEN_RELEASE (release_token) is delivered
(04)       if (¬coordinator_i)
(05)         then release_token ← release_resource(release_token, releasing_i, resource_i);
(06)              release_token ← randomize_token(release_token)
(07)              release_token ← remove_layer(release_token)
(08)              send TOKEN_RELEASE (release_token) to p_{(i+1)modn}
(09)         else if (isFree(release_token))
(10)                then coordinator_i ← false
(11)                     round_i ← round_i + 1;
(12)                     send NEW_ROUND (round_i) to p_{(i+1)modn}
(13)                else release_token ← create_release_token()
(14)                     release_token ← release_resource(release_token, releasing_i, resource_i);
(15)                     send TOKEN_RELEASE (release_token) to p_{(i+1)modn}
(16)              endif
(17)       endif
-------------------------------------------------------------------------
(18)  when NEW_ROUND(rd) is delivered
(19)       if (rd > round_i)
(20)         then round_i ← rd;
(21)       endif
(22)       send NEW_ROUND (rd) to p_{(i+1)modn}
```

Figure 3: The release() protocol (code for p_i)

In the pseudo-code, we used release_resource(T, b) and randomize_token(T) defined earlier and we use the following new functions:

- remove_layer(T): given an encrypted release token T, removes the encrypted layer of p_i
- isFree(T): given a token T, check if each slot in the release token T has been released.

A slot is released by calling the release() operation. In this case, the variable $state_i$ is set to NCS and the flag $releasing_i$ is set to true (lines 01 - 02). When p_i receives $release\_token$, it checks whether it is the coordinator for the current round. If p_i is not the coordinator, according to its state, it releases or keeps the assigned slot (line 05), it re-randomizes the token (line 06), removes its encryption layer (line 07), and finally passes the token to its neighbor in the logical ring (line 08). If p_i is the coordinator for the current round, then it checks whether all other processes released their assigned slots (lines 09 - 16). If all slots were released, then p_i sends a NEW_ROUND message to its neighbor so that a new round can be started (line 12). Otherwise, the current token is discarded and a new turn of the release phase is started (lines 13 - 16). Finally, when a process p_i receives a NEW_ROUND message, it updates the local variable $round_i$ and

forwards the NEW_ROUND message to its neighbor (lines 19 - 22).

Correctness Proofs. Due to lack of space, we provide here only the statements of the main Lemmas. Proof of lemma 6 can be found in Appendix A while the other proofs can be found in [1].

Lemma 4 *Let $\Pi = \{p_1, p_2, \ldots, p_n\}$ be the set of processes of the distributed system and let $\{r_1, r_2, \ldots, r_m\}$ be the set of slots of the resource $\mathcal{R}$. Given the algorithm shown in Figures 1 - 3 and given two processes $p_i, p_j \in \Pi$, if p_i and p_j access concurrently the resource $\mathcal{R}$, then the slot r_x assigned to p_i and the slot r_y assigned to p_j are distinct.*

Lemma 5 *Let $\Pi = \{p_1, p_2, \ldots, p_n\}$ be the set of processes of the distributed system and let $\{r_1, r_2, \ldots, r_m\}$ be the slots of the resource $\mathcal{R}$. Given the algorithm shown in Figures 1 - 3, then any process p_i that invokes the* request() *operation will eventually obtain a slot r_j of $\mathcal{R}$.*

Lemma 6 *Let $\Pi = \{p_1, p_2 \ldots p_n\}$ be the set of processes of the distributed system and let $\{r_1, r_2, \ldots r_m\}$ be the slots of the resource $\mathcal{R}$. Given the algorithm shown in Figures 1 - 3, if $|\mathcal{C}| \leq n - 2$ then the O-mA property is satisfied.*

5.5 Discussion

Improving resource utilization: The algorithm might suffer in some runs from poor resource utilization, that is, a competing process (apart from the coordinator) may not obtain a valid slot even if there are several available. This impossibility leads to some utilization waste due to the non-deterministic behavior. Suppose the first competing process p_i is m hops away from the coordinator. It may happen that all m slots are assigned to the intermediate processes, i.e., those between the coordinator and p_i. However, intermediate processes may not necessarily compete for slots but they still get valid tickets. To avoid such a waste and improve resource utilization, we envision executing concurrent rounds with multiple coordinators. The idea is to let coordinators start new rounds as soon as slots become available. There could be up to m concurrent rounds in which each slot is managed by a distinct coordinator.

Comparison with an algorithm based on a Trusted Third Party: In Appendix A we considered an algorithm based on a fair Trusted Third Party (TTP) that regulates access to the slots. Each process sends a request to access a slot of R to the TTP. The TTP assigns one of the m slots, if available, and sends a reply to the process. We proved a bound on the maximum number of honest-but-curious processes (i.e., $|\mathcal{C}| \leq n-2$) that can be tolerated by the TTP algorithm to solve O-mA. Intuitively, the communication bound δ creates an information leakage that can be exploited by a coalition $\mathcal{C}$ with $|\mathcal{C}| = n - 1$ processes. Processes in $\mathcal{C}$ may collude and issue requests at exactly the same time to the TTP. If the resource is not allocated by the TTP to a honest-but-curious process within the time bound, then it's possible to infer that the honest process has received one slot. Such a slot can also be uniquely identified which implies that O-mA is violated.

This bound matches the one found in Lemma 6. Thus our algorithms has the same resiliency to honest-but-curious processes as the one based on a TTP.

Adapting the algorithm to satisfy SO-mA: Our basic scheme does not provide the SO-mA property. Indeed, if the round leader belongs to $\mathcal{C}$, it will figure out the number of processes that are using any slots of the resource. This is enough to violate, in some runs, the SO-mA property. It is possible to avoid this leakage by modifying the release phase implementing a $secure - or$ of the internal states of the processes. In particular, *secure-or* will return 1 if there is at least one process in CS (critical section) state, false

otherwise. But the number of processes in CS state is kept private. The *secure-or* can be implemented by simply exploiting the homomorphic property of ElGamal encryption. We will investigate further this idea in future work.

Towards Byzantine Adversaries: Our protocol assumes honest but curious adversaries, i.e., processes will faithfully follow the specifications of the protocol but are tempted to learn anything outside their domain. On the other hand, malicious or byzantine adversaries can actively and arbitrarily disrupt the protocol. For example, a byzantine adversary could inject, modify, or corrupt messages. In addition, the adversary can be adaptive in the sense that while disrupting the protocol it will adapt its attack strategy to render countermeasures ineffective. Designing an oblivious assignment scheme resilient against such a powerful attacker is still an open problem. However, we believe there exist technical tools that can be used to convert our basic scheme for honest-but-curious adversaries into a scheme resilient against byzantine adversaries. Inevitably, these tools will make our scheme substantially more expensive. In particular, it's possible to prevent injection of spurious messages via insubvertible encryption [2], that is, ciphertexts can still be randomized as in our scheme but no adversary can inject ciphertexts not produced by the round leader. At the same time, no existing ciphertext can be corrupted unless it is a legitimate re-randomization. The correctness of other operations such as partial decryption or release of tickets can be performed via standard zero-knowledge proofs. Such proofs are reasonably efficient in our setting. We leave the details of this approach to future work.

6 Conclusion

This paper introduced the oblivious assignment problem, i.e., a coordination problem, where n processes compete to get exclusive access to one of the m available slots of a resource $\mathcal{R}$, while still maintaining the obliviousness of the assignment. A rotating token algorithm solving the oblivious assignment problem has been introduced. This algorithm has been proven correct as long as at least two honest processes are in the distributed system. This bound matches the one proved considering a centralized TTP assigning slots. All these results consider that honest-but-curious processes know both when the communication delay is within a certain bound and the value of the bound. This knowledge can be exploited to break the algorithm (i.e. discover assignments of correct processes).

References

[1] G. Ateniese, R. Baldoni, S. Bonomi, and G. A. Di Luna. Oblivious Assignment with m Slots. Technical report, MIDLAB 2/12 - University of Rome "La Sapienza" - http://www.dis.uniroma1.it/ midlab/publications.php, 2012.

[2] G. Ateniese, J. Camenisch, and B. de Medeiros. Untraceable rfid tags via insubvertible encryption. In *Proceedings of the 12th ACM conference on Computer and communications security*, CCS '05, pages 92–101, New York, NY, USA, 2005. ACM.

[3] H. Attiya, A. Bar-Noy, D. Dolev, D. Peleg, and R. Reischuk. Renaming in an asynchronous environment. *Journal of the ACM*, 37:524–548, July 1990.

[4] H. Attiya and J. Welch. *Distributed Computing: Fundamentals, Simulations and Advanced Topics (2nd edition)*. John Wiley Interscience, March 2004.

[5] R. Baldoni, A. Virgillito, and R. Petrassi. A distributed mutual exclusion algorithm for mobile ad-hoc networks. *IEEE Symposium on Computers and Communications*, page 539, 2002.

[6] A Barnett and N. P. Smart. Mental Poker Revisited. In Kenneth G. Paterson, editor, *Cryptography and Coding, Proceedings of the 9th IMA International Conference*, volume 2898 of *Lecture Notes in Computer Science*, pages 370–383. Springer Verlag, 2003.

[7] D. Boneh. The decision diffie-hellman problem. 1423:48–63, 1998. 10.1007/BFb0054851.

[8] S. Bulgannawar and N. H. Vaidya. A distributed k-mutual exclusion algorithm. In *International Conference on Distributed Computer Systems*, pages 153–160, 1995.

[9] J. E. Burns and G. L. Peterson. The ambiguity of choosing. In *Proceedings of the eighth annual ACM Symposium on Principles of distributed computing*, Priciple of Distributed Computing '89, pages 145–157, New York, NY, USA, 1989. ACM.

[10] T. El Gamal. A public key cryptosystem and a signature scheme based on discrete logarithms. In *Proceedings of CRYPTO 84 on Advances in cryptology*, pages 10–18, New York, NY, USA, 1985. Springer-Verlag New York, Inc.

[11] C. Gentry. Fully homomorphic encryption using ideal lattices. In *Proceedings of the 41st annual ACM symposium on Theory of computing*, pages 169–178. ACM, 2009.

[12] O. Goldreich. *Foundations of Cryptography: Volume 2, Basic Applications*. Cambridge University Press, New York, NY, USA, 2004.

[13] P. Golle, M. Jakobsson, A. Juels, and P. Syverson. Universal re-encryption for mixnets. In *In Proceedings of the 2004 RSA conference*, pages 163–178. Springer-Verlag, 2002.

[14] H. Kakugawa, S. Fujita, M. Yamashita, and T. Ae. A distributed k-mutual exclusion algorithm using k-coterie. *Information Processing Letters*, 49(4):213 – 218, 1994.

[15] G. Le Lann. Distributed systems - towards a formal approach. In *Congress of International Federation for Information Processing*, pages 155–160, 1977.

[16] M. Maekawa. A square root n algorithm for mutual exclusion in decentralized systems. *ACM Transaction on Computer System*, 3(2):145–159, 1985.

[17] K. Raymond. A distributed algorithm for multiple entries to a critical section. *Information Processing Letters*, 30(4):189–193, 1989.

[18] M. Raynal. *Algorithms for mutual exclusion*. MIT Press, Cambridge, MA, USA, 1986.

[19] G. Ricart and A. K. Agrawala. An optimal algorithm for mutual exclusion in computer networks. *Communincations of the ACM*, 24(1):9–17, 1981.

[20] A. Shamir, R. L. Rivest, and L. M. Adleman. Mental Poker. Technical Report MIT-LCS-TM-125, Massachusetts Institute of Technology, 1979.

[21] I. Suzuki and T. Kasami. A distributed mutual exclusion algorithm. *ACM Transaction on Computer System*, 3(4):344–349, 1985.

[22] A. C. Yao. Protocols for secure computations. In *23st Annual IEEE Symposium on Foundations of Computer Science*, pages 160–164. IEEE Computer Society, 1982.

Appendix A - Third Party Impossibilities and Proof Omitted in the text

6.1 Third Party Impossibilities

To state and prove solvability results, we introduce a Trusted Third Party (TTP) that manages the assignment. Each process sends a request to access a slot of $\mathcal{R}$ to the TTP. The TTP assigns one of the m resource, if available, and sends the reply to the process. To ensure SO-mA, the number of resources m has to be greater than the number of processes in $\mathcal{C}$. To see this, consider the case where the TTP assigns all slots to processes in $\mathcal{C}$. Clearly, in this case, processes in $\mathcal{C}$ can collectively determine that honest processes do not own any slot. For this reason, whenever we refer to the SO-mA problem, we implicitly assume that $m > |\mathcal{C}|$.

Lemma 7 *Consider an asynchronous system, with a trusted third party TTP, composed of n processes and m slot of a resource $\mathcal{R}$, where $m < n$. Assume TTP is running an assignment algorithm that ensures properties 1 and 2.*
If $|\mathcal{C}| < n$, then TTP solves both the O-mA and SO-mA problems.

Proof To violate the StrongObliviousAssignment property, processes in $\mathcal{C}$ must deterministically determine whether honest processes own slots or not. The only information leakage, however, is the acknowledgment from the TTP when a slot is requested. Due to the communication asynchrony, a process in $\mathcal{C}$ waiting for an acknowledgement message from the TTP cannot establish whether all slots have been assigned or there is an unpredictable message delay. $\square_{Lemma\ 7}$

Lemma 8 *Consider a distributed system, with a trusted third party TTP, composed of n processes with a bound δ on message transfer delay and m slots of a resource $\mathcal{R}$, where $m < n$. Assume TTP is running an assignment algorithm that ensures properties 1 and 2 and whose response time for each TTP assignment operation is bounded by δ'.*
If $|\mathcal{C}| \geq n - 1$, then TTP cannot solve the O-mA problem.

Proof In the synchronous system model, a process is able to compute a bound on the time elapsed between a slot request to the TTP and the time the reply is received. This bound is $2\delta + \delta'$. Consider an adversary $\mathcal{A}$ that coordinates the processes in $\mathcal{C}$. $\mathcal{A}$ instructs the processes in $\mathcal{C}$ to issue m requests to the TTP. If any of them does not obtain a slot r_j within $2\delta + \delta'$, it means that r_j has been assigned to an honest process, say p_i. Thus, $\mathcal{A}$ infers the assignment $\langle p_i, r_j \rangle$, violating the ObliviousAssignment property. $\square_{Lemma\ 8}$

Lemma 9 *Consider a distributed system, with a trusted third party TTP, composed of n processes with a bound δ on message transfer delay and m slots of a resource $\mathcal{R}$, where $m < n$. Assume TTP is running an assignment algorithm that ensures properties 1 and 2 and whose response time for each TTP assignment operation is bounded by δ'.*
Finally, let w be an integer (with $1 \leq w \leq m$) representing the maximum number of slots of $\mathcal{R}$ concurrently assigned by TTP.
If $w \geq n - |\mathcal{C}|$ and w is fixed, then TTP cannot solve the SO-mA problem.

Proof Assume that $w \geq |\Pi \setminus C|$. We show that there exists an execution of the protocol that violates the StrongObliviousAssignment property. Let us define $d = w - (n - |\mathcal{C}|)$. This is the number of available slots when all honest processes have already obtained a slot. Since $w \leq m$ and $m \leq n - 1$, we have that d is at

most $|\mathcal{C}| - 1$. Consider an adversary $\mathcal{A}$ that coordinates the processes in $\mathcal{C}$. $\mathcal{A}$ instructs $d + 1$ processes in $\mathcal{C}$ to issue requests to the TTP at time t_0 (note that $d + 1 \leq |\mathcal{C}|$). In the synchronous system model, a process is able to compute a bound on the time elapsed between a slot request to the TTP and the time the reply is received. This bound is $2\delta + \delta'$. Thus, at time $t_0 + 2\delta + \delta'$, the adversary learns how many slots have been assigned to $d + 1$ processes of $\mathcal{C}$. If any of the $d + 1$ processes did not obtained a slot, then the adversary infers that all honest processes have obtained a slot. $\square_{Lemma\ 9}$

6.2 Proofs omitted in the text

Lemma 4 *Let $\Pi = \{p_1, p_2 \dots p_n\}$ be the set of processes of the distributed system and let $\{r_1, r_2, \dots r_m\}$ be the set of slots of the resource $\mathcal{R}$. Given the algorithm shown in Figures 1 - 3 and given two processes $p_i, p_j \in \Pi$, if p_i and p_j access concurrently the resource $\mathcal{R}$, then the slot r_x assigned to p_i and the slot r_y assigned to p_j are distinct.*

Proof We prove this lemma by contradiction. Assume that $r_x = r_y$. Thus, p_i and p_j executed line 26, Figure 2 and $resource_i$ is equal to $resource_j$. The latter implies that there exist two tickets $ticket_i$ and $ticket_j$ that store the same encrypted value. But this is a contradiction.
Note that, processes obtain tickets by selecting them from the $request\_token$ (line 10, Figure 2) and the request token is created by the coordinator of the current round at the beginning of the assignment phase (line 14, Figure1). But the request token does not contain duplicates and resource slots are distinct, thus there can be only one ticket associated to r_x (or r_y). $\square_{Lemma\ 4}$

Lemma 10 *Give the algorithm shown in Figures 1 - 3, then every round eventually terminates.*

Proof The proof is done by induction. Let us first prove that from round 1, eventually all the processes will move to round 2 and then we will show that the same happens also in a generic round i.
Basic step: round 1. Round 1 starts with the Init statement where all local variables are initialized to their default values (lines 01 - 02, Figure 1) and the coordinator for the current round is selected (line 07). Note that, since process identifiers are unique, in each round, a unique coordinator is selected by using the deterministic function modn (where n is the number of processes of the distributed system). In particular, when executing this procedure at round 1, p_1 will set its $coordinator_1$ to true ((line 07)), and the assignment phase starts with the send of the REQUEST message (line 18), containing the token to p_2.
Delivering such message, p_2 will forward it to p_3 and so on, until the REQUEST message come back to p_1. Note that, since processes communicate through reliable FIFO point-to-point channels and a failure free scenario is assumed, then eventually the request token come back to p_1 and the release phase starts.
In particular, p_1 will create a release token and will send it to p_2 trough a TOKEN_RELEASE message (line 13).
Delivering the TOKEN_RELEASE message, p_2 will execute line 08 forwarding the release token to p_3 and so on, until also the release token come back to p_1. Delivering the TOKEN_RELEASE message, p_1 will check if all the slots have been released (line 09, Figure 3).
If it is so, then p_1 will set its local variable $coordinator_1$ to false (line 10, Figure 3), it will increments its round counter to 2, i.e. it will goes to round 2 (line 11, Figure 3) and it will send a new_round message to p_2 that delivering it will execute lines 19 - 22, Figure 3 going to the second round as well. In addition, p_2 will also execute lines 04 - 08, Figure 3 and will become the new coordinator.

On the contrary, if there exists some process that still own a slot of the resource $\mathcal{R}$, then p_1 will circulate again a new release token in the ring (lines 13 - 16, Figure 3). Note that, since all the processes follows the protocol, eventually all of them will release the assigned slot and then eventually the condition in line 09 will become true and a new round will be triggered.

Induction step: round i. Note that, moving from a generic round $i-1$ to the current round i, the coordinator of the round $i-1$ send a new_round message that will circulate in the logical ring. As a consequence, every process p_j delivering such a message will execute lines 18 - 22, Figure 3 updating their $round_j$ local variable. As a consequence, every p_j will execute lines 04 - 08, Figure 1 re-initializing its local variables, coming back to the scenario described in the basic step and the claim follows.

$\square_{Lemma\ 10}$

Corollary 1 *Give the algorithm shown in Figures 1 - 3, then every process $p_i \in \Pi$ will become coordinator infinitely often.*

Proof Due to Lemma 10, every round eventually terminates and the round counter is incremented. Considering that (i) the number of processes n is finite and (ii) the coordinator is the process p_j such that $j = round_{mod\ n}$, it means that every process p_i will become coordinator once every n rounds. $\square_{Corollary\ 1}$

Lemma 5 *Let $\Pi = \{p_1, p_2 \dots p_n\}$ be the set of processes of the distributed system and let $\{r_1, r_2, \dots r_m\}$ be the slots of the resource $\mathcal{R}$. Given the algorithm shown in Figures 1 - 3, then any process p_i that invokes the* request() *operation, eventually obtains a slot r_j of $\mathcal{R}$.*

Proof Suppose by contradiction that there exists a process $p_i \in \Pi$ that invokes the request() operation at some time t and it never obtains a slot of $\mathcal{R}$. In this case, p_i never executes line 14, Figure 1 or line 26, Figure 2.

When p_i invokes the request() operation, it executes line 01, Figure 2 and sets the variable $state_i$ to $waiting$. Due to Corollary 1, p_i will become coordinator at some time t' after t and its $coordinator_i$ variable will be set to true. Thus, the procedure in lines 09 - 18, Figure 1 is executed and, in particular, the condition in line 10 p_i will be satisfied. But this implies that a valid slot will be assigned to p_i by executing line 11.

$\square_{Lemma\ 5}$

Lemma 6 *Let $\Pi = \{p_1, p_2 \dots p_n\}$ be the set of processes of the distributed system and let $\{r_1, r_2, \dots r_m\}$ be the slots of the resource $\mathcal{R}$. Given the algorithm shown in Figures 1 - 3, if $|\mathcal{C}| \leq n - 2$ then the O-mA property is satisfied.*

We divide the proof of Lemma 6 in two subproofs.

Lemma 7 *If $|\mathcal{C}| \leq n - 2$, then the O-mA property is satisfied during the assignment phase.*

We consider the worst case scenario in which a non-adaptive adversary $\mathcal{A}$ compromises $n-2$ processes. Let p_i and p_j be the two honest processes. We will show that if $\mathcal{A}$ is able to guess the correct assignment of p_i and p_j with probability better than $\frac{1}{2} + \mathsf{negl}(n)$, where $\mathsf{negl}(n)$ is a negligible function [1] in the security

[1] A function $f : \mathbb{N} \to \mathbb{R}$ is negligible if $\forall p$ polynomial function $p : \mathbb{N} \to \mathbb{R}\ \exists n_0\ s.t\ \forall n > n_0\ |f(n)| \leq \frac{1}{|p(n)|}$

parameter $k = |p|$, than there exists a probabilistic polynomial-time algorithm that can solve DDH in G (breaking the semantic security of ElGamal encryption).

To formalize our proof, we define the Correct Assignment Indistinguishability Experiment (CAIE):

- The adversary $\mathcal{A}$ outputs the set of processes $\mathcal{C}$ to be compromised.
- An assignment phase is run. At the end of it, the adversary $\mathcal{A}$ is provided with a transcript L of all messages exchanged and, in addition, the memory content of all processes in C.
- The adversary $\mathcal{A}$ outputs a guess $a =< p_i, r_k >, < p_j, r_t >$.
- The output of the experiment is defined to be 1 if a is the correct assignment of slots for p_i, p_j, 0 otherwise.

Recall that $\mathcal{A}$ embodies honest-but-curious processes, and thus cannot inject or manipulate messages but must faithfully follow the algorithm specifications. We want to show that the probability that $\mathcal{A}$ outputs the correct guess is $1/2$ plus a negligible function in the security parameter.

Lemma 8 *Under the DDH assumption in G, the probability that $\mathcal{A}$ wins in* CAIE *is* $\leq 1/2 + \mathsf{negl}(k)$.

Proof We fix an adversary $\mathcal{A}$ and set $|Pr[\mathbf{CAIE}_{\mathcal{A}}(k) = 1] - 1/2| = \epsilon(k)$. We build an adversary $\mathcal{A}'$ that uses $\mathcal{A}$ to violate the semantic security of ElGamal encryption. The adversary $\mathcal{A}'$ obtains a public key g^{y_i} and plays the CPA-security game [12]. In this game, $\mathcal{A}'$ generates two challenge messages of the same length and receives the encryption of one of them. $\mathcal{A}'$ must guess which message was encrypted.

We build $\mathcal{A}'$ as follows:

- $\mathcal{A}'$ starts A and obtains $\mathcal{C}$.
- $\mathcal{A}'$ sets g^{y_i} as public key of p_i. We assume w.l.o.g. that $1 < i < j$ and that the leader is p_1. $\mathcal{A}'$ simulates an assignment phase for $\mathcal{A}$ and generates a transcript L. It is easy to see that, even though $\mathcal{A}'$ does not know the secret key of p_i, the simulation is perfect and the transcript L is correct. Indeed, $\mathcal{A}'$ knows the actual content of encrypted tickets and can respond to decryption queries (via ephemeral keys).

  $\mathcal{A}'$ picks two encrypted tickets $t_0 = (g^r, r_0 g^{rY}), t_1 = (g^{r'}, r_1 g^{r'Y})$ uniformly at random from the token entering p_i. It then sets (r_0, r_1) as the challenge messages in the CPA-game. $\mathcal{A}'$ receives the response to the challenge $(g^r, r_b g^{ry_i})$, and embeds the ticket $t' = (g^r, r_b g^{ry_i} g^{r \prod_{\forall p_k \in \Pi \setminus \{p_i\}} y_k})$ into the token exiting from p_i. $\mathcal{A}'$ cannot respond to decryption query of t'. If in the transcript L a process in $\mathcal{C}$ selects t', $\mathcal{A}'$ stops the execution of $\mathcal{A}$, it aborts, and restarts the simulation.
- $\mathcal{A}$ outputs its guess a. If p_i is paired with t_0, $\mathcal{A}'$ outputs 0, otherwise $\mathcal{A}'$ outputs 1.

$\mathcal{A}'$ runs in polynomial time and wins whenever $\mathcal{A}$ wins in the CAIE experiment. Since ElGamal is semantically secure, it must be that $\epsilon(k) = \mathsf{negl}(k)$ and this proves the Lemma.

$\square_{Lemma\ 8}$

The lemma 7 follows directly from lemma 8, the experiment CAIE is well defined for more than two honest parties. The adversary $\mathcal{A}'$ can use an adversary $\mathcal{A}$ that breaks CAIE with more than two honest processes to break the ElGamal encryption with multiple messages. $\mathcal{A}$ creates two possible assignments of tickets to the k honest processes $a_0 = (r_0, r_1, .., r_{k-1}),\ a_1 = (r'_0, r'_1, .., r'_{k-1})$ and use $\mathcal{A}$ to distinguish the encryption of one of these two assignments under an unknown private key.

Lemma 9 *If $|\mathcal{C}| \leq n-2$, then the O-mA property is satisfied during the release phase.*

The security proof for the release phase is a simple adaptation of the proof for the assignment phase and it is omitted.

Appendix B - Cryptographic Details

This appendix provide the details on the implementation of the cryptographic functions used in the algorithm shown in Figures 1 - 3.

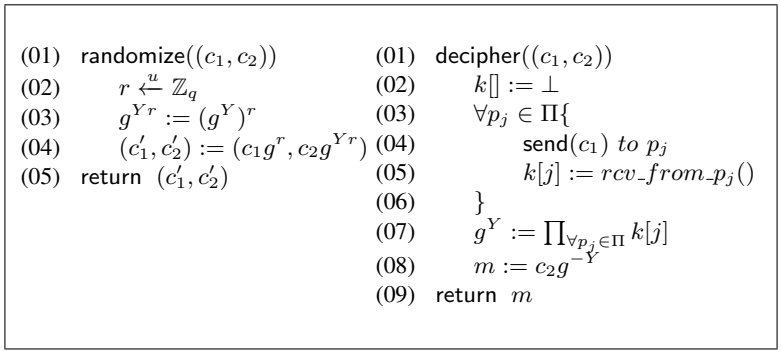

Figure 4: Crypto Operations A

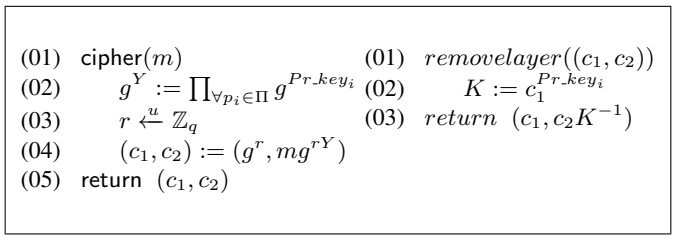

Figure 5: Crypto Operations B

Ticket Assignment The round leader p_1 encodes the set of tickets, encrypts them, and sends them to all processes positioned logically as a ring structure. In particular, the set of tickets are first encoded as elements of G, $T : \{t_1, t_2, \ldots, t_x\}$, then encrypted, obtaining $T' : \{(g^{r_1}, t_1 g^{r_1 Y}), (g^{r_2}, t_2 g^{r_2 Y}), \ldots, (g^{r_x}, t_x g^{r_x Y})\}$, and sent through the ring structure. Each process in the ring can perform several actions on the incoming set of encrypted tickets. It can permute the set, re-randomize each element, or remove its encryption layer. Thus, the basic idea is to let each process pick uniformly at random an encrypted ticket from T' which will have to be opened in cooperation with all other processes in the ring. The remaining tickets are then re-randomized and the new set is shuffled and sent to the next process. The assignment ends when the set of encrypted tickets reaches again the leader.

Ticket Release The release of tickets is handled by the round leader p_1. The basic idea is to collect *used* tickets by querying each process. To make the release oblivious, processes will still use ElGamal encryption to encode a boolean flag True/False. If a process wants to release a ticket t_j, it will alter the j-th ciphertext to

indicate a True value and hence that the j-th ticket is now released. Before altering any encryption though, the process p_i must remove its key component $g^{rPr\_key_i}$ from all ciphertexts and re-randomize the results.

This is accomplished as follows: The round leader p_1 prepares a vector $V := [(g^{r_1}, g^{r_1 Y}), (g^{r_2}, g^{r_2 Y}), \ldots, (g^{r_n}, g^{r_n Y})]$ and sends it through the ring of processes. $V[j]$ is essentially an ElGamal encryption of '1' that corresponds to the j-th ticket. To release a ticket t_j, it is enough to alter the ciphertext $(g^{r_j}, g^{r_j Y})$ into $(g^{r_j}, Dg^{r_j Y})$, for some dummy value $D \neq 1$. Before forwarding V to the next process, each process p_i will have to re-randomize every ciphertexts in V and, at the same time, remove its key component $g^{rPr\_key_i}$.

Functions We define a series of functions to capture the cryptographic operations described above. The functions are: decode_element, randomize_token, release_resource and remove_layer. The function decode_element takes as input a ticket from the assignment token, and returns the element decrypted, this function needs to communicate with all processes . The function randomize_token takes as input a token, randomizes all the elements. The function release_resource takes as input a release token and a boolean value and adds the value to the token. The function remove_layer takes as input a release token and removes the layer of encryption of the local process.

```
(01)randomize_token(T)                (01)remove_layer(T)
(02)    T' := ⊥                       (02)    T' := ⊥
(03)    ∀t ∈ T{                       (03)    ∀t ∈ T{
(04)        t' := randomize(t)        (04)        t' := removelayer(t)
(05)        T' := T' ∪ {t'}           (05)        T' := T' ∪ {t'}
(06)    }                             (06)    }
(07)return T'                         (07)return T'
```

Figure 6: Functions A

```
(01)decode_element((c1, c2))          (01)release_resource(T, v)
(02)    k[] := ⊥                      (02)    (c1, c2) := (g^r, v g^{rY})
(03)    ∀(p_j ∈ Π{                    (03)    T := T ∪ {(c1, c2)}
(04)        send(c1) to p_j           (04)    T := shuffle(T)
(05)        k[j] := rcv_from_p_j()    (05)    T := randomizeToken(T)
(06)    }                             (06)    T := removeLayer(T)
(07)    D := ∏_{∀p_j ∈ Π} k[j]        (07)return T
(08)    m := c2 D^{-1}
(09)return m
```

Figure 7: Functions B