Mutex-Based De-anonymization of an Anonymous Read/Write Memory

Emmanuel Godard, Damien Imbs, Michel Raynal, Gadi Taubenfeld

Mutex-based De-anonymization of an Anonymous Read/Write Memory

Emmanuel Godard[†], Damien Imbs[†], Michel Raynal[⋆,‡], Gadi Taubenfeld[◦]

[†]LIS, Université d'Aix-Marseille, France
[⋆]Univ Rennes IRISA, France
[‡]Department of Computing, Polytechnic University, Hong Kong
[◦]The Interdisciplinary Center, Herzliya 46150, Israel

Abstract. Anonymous shared memory is a memory in which processes use different names for the same shared read/write register. As an example, a shared register named A by a process p and a shared register named B by another process q can correspond to the very same register X, and similarly for the names B at p and A at q which can correspond to the same register $Y \neq X$. Hence, there is a permanent disagreement on the register names among the processes. This new notion of anonymity was recently introduced by G. Taubenfeld (PODC 2017), who presented several memory-anonymous algorithms and impossibility results.
This paper introduces a new problem, that consists in "de-anonymizing" an anonymous shared memory. To this end, it presents an algorithm that, starting with a shared memory made up of m anonymous read/write atomic registers (i.e., there is no a priori agreement on their names), allows each process to compute a local addressing mapping, such that all the processes agree on the names of each register. The proposed construction is based on an underlying deadlock-free mutex algorithm for $n \geq 2$ processes (recently proposed in a paper co-authored by some of the authors of this paper), and consequently inherits its necessary and sufficient condition on the size m of the anonymous memory, namely m must belong to the set $M(n) = \{m : \text{ such that } \forall\, \ell : 1 < \ell \leq n : \mathsf{gcd}(\ell, m) = 1\} \setminus \{1\}$. This algorithm, which is also symmetric in the sense process identities can only be compared by equality, requires the participation of all the processes; hence it can be part of the system initialization. Last but not least, the proposed algorithm has a noteworthy first-class property, namely, its simplicity.

Keywords: Anonymity, Anonymous shared memory, Asynchronous system, Atomic read/write register, Concurrent algorithm, Deadlock-freedom, Local memory, Mapping function, Mutual exclusion, Simplicity, Synchronization.

1 Introduction

Read/write registers. *Read/write registers* are the basic objects of sequential computing. From a theoretical point of view, they constitute the cells of a Turing machine tape, and from a programming point of view, they are the memory locations on top of which are built high-level objects such as stacks, queues, and trees (to cite a few of the most common).

In a concurrent programming context, a read/write register can be shared (accessed) by several processes to coordinate their actions or progress to a common goal. The most popular consistency condition for registers is *atomicity*, which states that all its read and write operations appear as if they have been executed sequentially, this sequence S being such that, if an operation op1 terminates before operation op2 starts, op1 appears before op2 in S, and a read operation returns the value written by the closest preceding write in S [13].

A register is said to be single-reader (SR) or multi-reader (MR) according to the number of processes that are allowed to read it. Similarly, a register can be single-writer (SW) or multi-writer (MW). A lot of algorithms have been proposed (e.g., see the textbooks [19,22]), which build MWMR registers from SWSR or SWMR registers in the presence of asynchrony and process crashes. In the other direction, an adaptive construction of SWMR registers from MWMR registers is described in [7].

Anonymous memory. While the notion of *process anonymity* has been studied for a long time from an algorithmic and computability point of view, both in message-passing systems (e.g., [2,5,24]) and shared memory systems (e.g., [4,6,11]), the notion of *memory anonymity* has been introduced only very recently in [23]. (See [21] for an introductory survey on process and memory anonymity).

Let us consider a shared memory SM made up of m atomic read/write registers. Such a memory can be seen as an array with m entries, namely $SM[1..m]$. In a non-anonymous memory system, for any index x, $1 \leq x \leq m$, if two or more processes invoke the address $SM[x]$ they access the very same register. As stated in [23], in the classical system model, there is an a priori agreement on the names of the shared registers. This a priori agreement facilitates the implementation of the coordination rules the processes have to follow to progress without violating the safety (consistency) properties associated with the application they solve [19,22].

This a priori agreement does no longer exist in a memory-anonymous system. In such a system the very same address identifier $SM[x]$ invoked by a process p_i and invoked by a different process p_j does not necessarily refer to the same atomic read/write register. More precisely, a memory-anonymous system is such that:

- for each process p_i an adversary defined, over the set $\{1, 2, \cdots, m\}$, a permutation $f_i()$ such that when p_i uses the address $SM[x]$, it actually accesses $SM[f_i(x)]$, and
- no process knows the permutations.

Let us notice that the read/write registers of a memory-anonymous system are necessarily MWMR.

Results on anonymous memory. In [23], mutual exclusion, consensus, and renaming, problems are addressed, and memory-anonymous algorithms and impossibility results are presented. Concerning deadlock-free mutual exclusion in failure-free asynchronous read/write systems, the following results are presented:

- A symmetric deadlock-free algorithm for two processes ("symmetric" means process identifiers are not ordered and can only be compared for equality, see Section 2.2).

- A theorem stating there is no deadlock-free algorithm if the number of processes n is not known.
- A condition on the size m of the anonymous memory which is necessary for any symmetric deadlock-free algorithm. More precisely, given an n-process system where $n \geq 2$, there is no deadlock-free mutual exclusion algorithm if the size m does not belong to the set $M(n) = \{ m \text{ such that } \forall\ \ell : 1 < \ell \leq n: \mathsf{gcd}(\ell, m) = 1\} \setminus \{1\}$.

Let us observe that the previous condition implies that it is not possible to design a symmetric deadlock-free mutex algorithm when the size of the anonymous memory m is an even integer greater than 2. As symmetric deadlock-free mutex algorithms suited to a non-anonymous memory do not require a parity-related property on the number of registers they use, it follows that, when the size of the memory m is an even integer greater than 2, non-anonymous read/write registers are computationally stronger than anonymous registers.

In the conclusion of [23], a few open problems are presented, one of them being "the existence of a symmetric starvation-free mutual exclusion algorithm for two processes", another one being "the existence of a symmetric deadlock-free mutual exclusion algorithm for more than two processes". This second problem was recently solved in [3] where an algorithm is presented, which assumes $m \in M(n)$. It follows that the very existence of this algorithm shows that the condition $m \in M(n)$ is also a sufficient condition for symmetric deadlock-free mutual exclusion in read/write anonymous memory systems.

Content of the paper. As shown in [3,23], the design of memory-anonymous algorithms is not a trivial task. We started this work with an attempt to design a starvation-free memory-anonymous mutual exclusion algorithm. This drove us to the observation that the fact "there is currently a competition among processes" must be memorized in one way or another to prevent a process from always defeating other processes, and thereby ensure starvation-freedom.

Finally, considering an n-process system, after many attempts, this work ended with a relatively simple symmetric *de-anonymization* algorithm, namely, an algorithm that transforms an anonymous read/write memory into a non-anonymous read/write memory. This algorithm requires the participation of all the processes, and assumes that processes do not fail. Once memory de-anonymization is obtained (e.g., at system initialization), it becomes possible to use algorithms based on a non-anonymous memory on top of anonymous memory.

The proposed construction relies on an underlying memory-anonymous symmetric deadlock-free mutual exclusion algorithm (the one introduced in [3]). Hence, it inherits its requirement on m, namely, $m \in M(n)$. It follows that, when m satisfies this condition, m anonymous registers and m non-anonymous registers have the same computability power from an anonymous/non-anonymous mutual exclusion point of view. Let us also notice that, if a non-anonymous memory algorithm executed on top of the proposed construction requires m' registers where m' does not belong to the set $M(n)$ defined above, it is sufficient to select the first integer greater than m' belonging to $M(n)$ as the value of m, and, at the non-anonymous memory upper layer, $(m - m')$

registers are ignored. Let us notice that the proposed construction is *universal* in the sense any concurrent non-anonymous memory algorithm can be executed on top of it.

On the difficulty of the problem. In a non-anonymous memory system, there is no ambiguity on the read/write registers used by the processes. As already said, its identifiers are unambiguously shared by all processes, and no other algorithm is concurrently using these registers. Differently, as, in an anonymous memory system, $SM[x]$ can denote different registers for distinct processes, a process must (in one way or another) write "enough" registers to transmit information to other processes. This is a direct consequence of the fact that there is no a priori agreement on the identities of the shared atomic read/write registers and the fact that – due to its very nature – no anonymous register can be a single-writer register.

Hence, the difficulty in the construction of a memory de-anonymization algorithm comes from the fact that, due to memory anonymity, it concurrently uses the same registers like the ones used by the underlying mutex algorithm it uses as a subroutine. As we will see, to circumvent this issue, the proposed memory de-anonymization algorithm will use (in a very simple way) the local memory of each process to store the value of an increasing counter, which simulates a shared non-anonymous register on which the processes agree and can consequently use to coordinate their local progress.

The de-anonymization problem addressed in this paper may seem of theoretical interest only (as many other problems appeared first). As long as its practical interest is concerned, we do not have to forget that, as nicely expressed by the physicist Niels Bohr "prediction is very difficult, especially when it about the future!". Nevertheless, the results presented in this paper shows that, from a computability point of view, there are cases where –in a failure-free context– anonymous read/write registers are as strong as non-anonymous registers.

Let us also notice that a similar problem (but much simpler, even trivial) appears in message-passing systems, where any two nodes (processes) are connected by a communication channel, locally known as internal ports by each process, $port_i[x]$ being the local name of the channel connecting process p_i to some process p_j. In this context, it is possible that for any two processes p_i and p_k, the local names $port_i[x]$ and $port_k[x]$ denote channels connecting them to two different processes, while $port_i[x]$ and $port_k[y]$, $x \neq y$, connect them to the same process. Differently, from process identities, values stored in ports are purely local and have no global meaning. Moreover, it is straightforward for a process to learn the name of the process it is connected to when it uses a given local port.

Simplicity is a first class property. The simplicity of the proposed algorithm does not mean it was simple to obtain. This was not a trivial task as simplicity is rarely obtained for free. As said by A.J. Perlis (the first Turing Award recipient) "Simplicity does not precede complexity, but follows it" [16]. Let us also remember the following sentence written by the mathematician/philosopher Blaise Pascal at the end of a letter to a friend: "I apologize for having written such a long letter, I had not enough time to write a shorter one". The implication "simple $\Rightarrow$ easy" is rarely true for non-trivial problems [1]. Simplicity requires effort, but is very rewarding. It is a first class scientific

property which participates in the beauty of science [9].

Roadmap. The paper is composed of 7 sections. Section 2 introduces the computing model, the notion of a symmetric algorithm, and mutual exclusion. Section 3 defines the de-anonymization problem. A first de-anonymization algorithm is presented in Section 4 and proved in Section 5. This algorithm requires each register of the de-anonymized memory to forever contain $1 + \log_2 m$ bits of control information. Then, the previous algorithm is enriched in Section 6 to obtain an algorithm which associates a single bit of permanent control information with each register of the de-anonymized memory. Section 7 concludes the paper.

Remark. On a practical side, it appears that the concept of an anonymous memory allows us to model epigenetic cell modifications [18]. Hence, it could be useful in biologically inspired distributed systems [14,15].

2 System Model, Symmetric Algorithm, and Mutex Algorithm

2.1 Process and Communication Model

Processes. The system is composed of a finite set of $n \geq 2$ asynchronous processes denoted p_1, .., p_n. The subscript i in p_i is only a notational convenience, which is not known by the processes. *Asynchronous* means that each process proceeds to its own speed, which can vary with time and remains always unknown to the other processes. Each process p_i knows its identity id_i and the total number of processes n. No two processes have the same identity.

Anonymous shared memory. The shared memory is made up of m atomic anonymous read/write registers denoted $SM[1...m]$. Hence, *all* registers are anonymous. As indicated in the Introduction, when p_i uses the address $SM[x]$, it actually uses $SM[f_i(x)]$, where $f_i()$ is a permutation defined by an external adversary. We will use the notation $SM_i[x]$ to denote $SM[f_i(x)]$, to stress the fact that no process knows the permutations.

It is assumed that all the registers are initialized to the same value. Otherwise, thanks to their different initial values, it would be possible to distinguish different registers, which consequently will no longer be fully anonymous.

To summarize: which adversaries? The adversaries considered in the paper are consequently asynchrony and memory anonymity. There are no process failures (this assumption is motivated by the fact that the proposed construction is based on a mutual exclusion algorithm, and mutual exclusion algorithms are impossible to build from read/write registers in the presence of process failures). Furthermore, unlike the mutual exclusion model where a process may never leave its remainder region, we assume that all the processes must participate in the algorithm.

2.2 Symmetric Algorithm

The notion of a *symmetric algorithm* dates back to the eighties [10,12]. Here, as in [23], a *symmetric algorithm* is an "algorithm in which the processes are executing exactly the

same code and the only way for distinguishing processes is by comparing identifiers. Identifiers can be written, read, and compared, but there is no way of looking inside an identifier. Thus it is not possible to know whether an identifier is odd or even".

Moreover, symmetry can be restricted by considering that the only comparison that can be applied to identifiers is equality. In this case, there is no order structuring the identifier name space. In the following, we consider the more restricting definition, namely, "symmetric" means " symmetric with comparison limited to equality".

Let us notice that, as all the processes have the same code and all the registers are initialized to the same value, process identities become a key element when one has to design an algorithm in such a constrained context.

2.3 One-Shot Mutual Exclusion

One-Shot Mutual Exclusion. Mutual exclusion is the oldest and certainly the most important of the synchronization problems. Formalized by E.W. Dijkstra in the mid-sixties [8], it consists in building what is called a lock (or mutex) object, defined by two operations, denoted $\mathsf{acquire}()$ and $\mathsf{release}()$. (Recent textbooks including mutual exclusion and variants of it are [19,22].)

The invocation of these operations by a process p_i always follows the following pattern: "$\mathsf{acquire}()$; *critical section*; $\mathsf{release}()$", where "critical section" is any sequence of code. Moreover, "one-shot" means that a process invokes at most once the operations $\mathsf{acquire}()$ and $\mathsf{release}()$. The mutex object satisfying the deadlock-freedom progress condition is defined by the following two properties.

- Mutual exclusion. No two processes are simultaneously in their critical section.
- Deadlock-freedom progress condition. If there is a process p_i that has a pending operation $\mathsf{acquire}()$, there is a process p_j (maybe $p_j \neq p_i$) that eventually executes its critical section.

As already mentioned, a memory-anonymous symmetric deadlock-free mutual exclusion algorithm is presented in [3]. This algorithm assumes that size m of the anonymous memory belongs to the set $M(n) = \{\, m \text{ such that } \forall \ell : 1 < \ell \leq n\text{: } \mathsf{gcd}(\ell, m) = 1\} \setminus \{1\}$. Hence, the mutex-based read/write memory de-anonymization algorithm presented in Section 4 is optimal with respect to the values of m for which deadlock-free mutual exclusion can be built despite memory anonymity.

3 The De-anonymization Problem

Definition. Given an n-process asynchronous system, in which the processes communicate via a set of m anonymous read/write registers $SM[1..m]$, the aim is for each process p_i to compute an addressing function $\mathsf{map}_i()$, which is a permutation over the set of the memory indexes $\{1, \cdots, m\}$, such that the two following properties are satisfied. It is assumed that all processes participate in the de-anonymization.

- Safety. For any $y \in \{1, \cdots, m\}$ and any process p_i, we have $SM_i[\mathsf{map}_i(y)] = SM[y]$.

– Liveness. There is a finite time after which all the processes have computed their addressing function $\mathsf{map}_i()$.

The safety property states that, once a process p_i has computed $\mathsf{map}_i()$, its local anonymous memory address $SM_i[x]$, where $x = \mathsf{map}_i(y)$, denotes the shared register $SM[y]$.

Accessing the de-anonymized memory. Once de-anonymized, the way the memory is accessed by the processes is illustrated in Fig. 1. For any index y, $1 \leq y \leq m$, the processes access the same register as follow: $SM_i[\mathsf{map}_i[y]]$ used by p_i and $SM_j[\mathsf{map}_j[y]]$ used by p_j denote the same register.

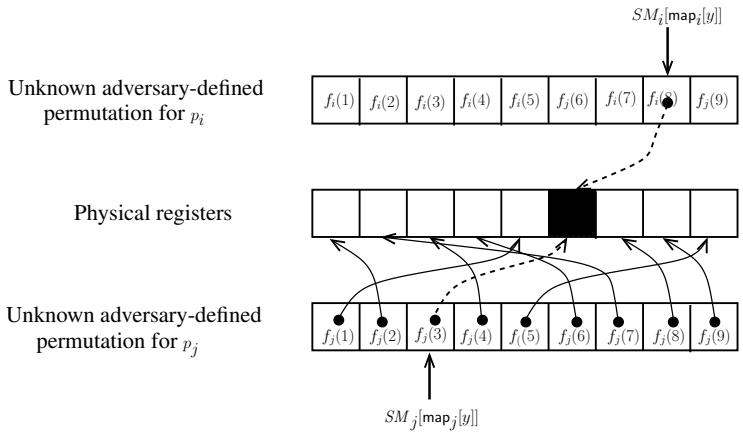

Fig. 1. Accessing the memory after de-anonymization

4 A Symmetric De-anonymization Algorithm

4.1 Memory De-anonymization in an n-Process Read/Write System

Underlying principle. The principle that underlies the design of the read/write memory de-anonymization algorithm (Algorithm 1) is based on an *competition/elimination* process, at the end of which a single winner process imposes its adversary-defined index permutation to all the processes, which becomes the shared names of the anonymous read/write registers, on which all processes agree.

The competition/elimination process uses an underlying mutual exclusion algorithm. Each process invokes $\mathsf{acquire}()$ and is eliminated when it leaves the critical section. The last process to enter the critical section is the winner.

Challenges. In order to detect which process is the last, the processes needs to collaborate to increase a counter whose value will reach n when the last process will enter the critical section. We stress that because the memory is anonymous there is no straightforward way to leverage a critical section. Since there is no agreement on the resources

(here the anonymous registers themselves), we underline that being in critical section does not grant any restricted access to the memory. In the following, properties of the underlying algorithm are described, which are used to build the required shared resource, namely a shared counter.

Properties of the underlying mutex algorithm that are used. In addition to the fact it solves mutual exclusion, the underlying mutex algorithm has behavioral properties that are implicitly used in the design of the de-anonymization algorithm and explicitly used in its proof.

- Property Mutex-1. A process writes only its identity or $\perp$ in an anonymous register.
- Property Mutex-2. When a process invokes $\mathsf{acquire}()$, it reads all anonymous registers.
- Property Mutex-3. When a process is allowed to enter the critical section, all registers contain its identity.
- Property Mutex-4. After a process is allowed to enter the critical section and before it invokes $\mathsf{release}()$, any other competing process can issue at most one write operation. It follows that, when a process p_i is inside the critical section, and x processes are inside their invocations of $\mathsf{acquire}()$, at least $(m - x)$ anonymous registers contain its identity id_i. Moreover, when a process releases the critical section (operation $\mathsf{release}()$), it writes $\perp$, in all the registers which contain its identity. Hence, at least $(m - x)$ such registers are reset to their initial value $\perp$.

Enriching the underlying mutex algorithm to share a counter. As can be seen from the previous properties, even when a process is alone in the critical section, it could happen that some of its writes are overwritten by another process. Property Mutex-4 states that a process, which is not in the critical section, may erase what was written by the process in critical section only once. That is no more than $(n - 1)$ registers can be erased. As $m - (n - 1) > 0$, by copying the value in *all* the anonymous registers, the process currently in the critical section ensures that at least one copy will not be overwritten. From property Mutex-2, the next process to enter the critical section will learn the correct value of the counter.

Sharing the counter in such a way is more easily done by integrating these operations within each read and write operation on the anonymous registers, issued by the underlying mutual exclusion algorithm. These basic operations are consequently enriched as described in Algorithm 2. These modifications are safe for the mutual exclusion algorithm since they do not interfere with operations and variables of this algorithm.

Let us remark that a similar technique, based on appropriate broadcast abstraction and quorums, is used in message-passing systems to update the local copies of a shared register [20]. Here the read and write operations issued by the underlying mutex algorithm are enriched to play the role of a broadcast abstraction.

Local variables. Each process s p_i manages three local variables.

- ct_i is a local counter initialized to 0, which will increase inside the integer interval $[0..n]$. The set of the n local variables ct_i implement a shared counter CT which increases by step 1 from its initial value 0 to n (line 2). (Actually, the set of the final values of the n local variables ct_i will be the set $\{1, 2, \ldots, n\}$.)

- $sm_i[1..m]$ is used to store a local copy of the anonymous memory $SM_i[1..m]$. A process p_i reads the anonymous memory by invoking $SM_i.\mathsf{scan}()$, which is an asynchronous (non-atomic) reading of all the anonymous registers.
- $last1_i$ is a Boolean, initialized to false, which will be set to true only by the last process that will access the critical section.

Each register contains a tag and a value. In order not to confuse the values written in anonymous registers by processes executing statements of Algorithm 1 (not including the operations $\mathsf{acquire}()$ and $\mathsf{release}()$), and the values written by other processes executing the underlying mutex algorithm, all the values written in the anonymous memory are prefixed by a tag. More explicitly, the tag MUTEX is used by the mutex algorithm, while the tag DESA is used by the de-anonymization algorithm.

Each anonymous read/write register is initialized to $\text{MUTEX}\langle 0, \perp\rangle$. The first value (0) is the initial value of the global counter CT, while the second value ($\perp$) is the initial value used by the mutex algorithm.

operation $SM_i.\mathsf{scan}()$ **returns** $([SM_i[1], \cdots, SM_i[m]])$.

operation $\mathsf{de\text{-}anonymize}(id_i)$ **is** % code for process p_i
(1) $\mathsf{acquire}(id_i)$;
(2) $ct_i \leftarrow ct_i + 1$;
% ct_i is the local representation of the global counter CT. It is updated at each process
% by the read and write operations of the underlying mutex algorithm (see Algorithm 2)
(3) $last1_i \leftarrow (ct_i = n)$;
(4) $\mathsf{release}(id_i)$; % realizes an implicit broadcast of ct_i %
(5) **if** $(last1_i)$
(6) **then for each** $x \in \{1, \cdots, m\}$ **do** $SM_i[x] \leftarrow \text{DESA}(x)$ **end for**
% the permutation for p_i is: $\forall\, y \in \{1, \cdots, m\}$: $\mathsf{map}_i(y) = y$ %
(7) **else repeat** $sm_i \leftarrow SM_i.\mathsf{scan}()$ **until** ($\forall\, x:$ $sm_i[x]$ is tagged DESA) **end repeat**;
(8) **for each** $x \in \{1, \cdots, m\}$ **do** $\mathsf{map}_i(y) \leftarrow x$ where $sm_i[x]$=DESA(y) **end for**
% the perm. for p_i is: $\forall\, y \in \{1, \cdots, m\}$: $\mathsf{map}_i(y) = x$, where $sm_i[x] = \text{DESA}(y)$
(9) **end if**.

Algorithm 1: Memory de-anonymization in an n-process read/write system

Behavior of a process p_i: first invoke the mutex algorithm. All the processes invoke the operation $\mathsf{de-anonymize}(id_i)$. When a process p_i invokes it, it first acquires the critical section (line 1). The code inside the critical section is a simple increase of the shared counter CT globally implemented by the local variables ct_i (line 2). Hence, if p_i is the ℓ^{th} process to access the critical section, ct_i is updated from $\ell - 1$ to ℓ, and p_i will inform the other processes of this increase when it will invoke $\mathsf{release}()$ (line 4). Let us notice that, at line 3, p_i sets to true its local Boolean variable $last1_i$ only if it is the last process to execute the critical section. Then, the behavior of p_i depends on the fact it is or not the last process to enter the critical section (see below).

Behavior of a process p_i: the read and write operations used by the mutex algorithm. As already indicated, to ensure correct dissemination of the last increase of CT (update of the local variable ct_j at a process p_j), the read and write operations that allow the mutex algorithm to access the anonymous registers are modified as described in Algorithm 2.

```
operation read of SM_i[x] executed by the mutex algorithm is
(1)  ⟨ct, val⟩ ← SM_i[x];
(2)  ct_i ← max(ct_i, ct);
(3)  return(val).

operation write of v in SM_i[x] executed by the mutex algorithm is
(4)  SM_i[x] ← MUTEX⟨ct_i, v⟩;
(5)  return(ok).
```

Algorithm 2: Modified read and write operations (code for p_i)

As the operation release() of the mutex algorithm writes $\perp$ (i.e., the MUTEX$\langle CT, \perp\rangle$) in at least $(m-(n-1))$ anonymous registers (Property Mutex-4), it follows that if a process p_i accesses later the critical section, it updated its local counter ct_i when it executed acquire(), which reads all anonymous registers (Property Mutex-1).

Behavior of a process p_i: the winner imposes its addressing permutation to all. The de-anonymization is done at lines 5-9. The $(n-1)$ processes that won the first $(n-1)$ critical sections execute line 7, in which they loop until they see all the registers tagged DESA.

Let p_ℓ be the last process that entered the critical section (hence, $ct_\ell = n$ and $last1_\ell$ is the only Boolean equal to true). This process imposes its adversary-defined addressing permutation as the common addressing, which realizes a non-anonymous memory. To this end, for any $x \in \{1, \cdots, m\}$, p_ℓ writes DESA(x) in $SM_\ell[x]$ (line 6). Hence, for any x we have $\mathsf{map}_\ell(x) = x$.

Let p_i be any other process that is looping at line 7 until it sees all the registers tagged DESA. When this occurs, it computes $\mathsf{map}_i()$, which is such that for any $x \in \{1, \cdots, m\}$, if $sm_i[x] =$DESA(y) then $\mathsf{map}_i(x) = y$ (line 7).

4.2 Using the De-anonymized Memory

It follows from the de-anonymization algorithm that when a process has written the tag DESA in all registers, thanks to their local mapping function $\mathsf{map}_i()$, all the processes share the same indexes for the same registers.

When this occurs, process p_k could start executing its local algorithm defined by the upper layer application, but if it writes an application-related value in some of these registers, this value can overwrite a value DESA() stored in a register not yet read by other processes. To prevent this problem from occurring, all the values written by a process

at the application level are prefixed by the tag APPL, and include a field containing the common index y associated with this register. In this way, any process p_i will be able to compute its local mapping function $\mathsf{map}_i()$, and can start its upper layer application part, as soon as it has computed $\mathsf{map}_i()$.

Let us notice that one bit is needed to distinguish the tag DESA and the tag APPL. Hence, each of a value DESA(x) and a value APPL$(x, -)$ requires $(1 + \log_2 m)$ control bits.

5 Proof of the Algorithm

Lemma 1. *Each process exits* acquire() *and, denoting i_k the index of the k^{th} process that enters the critical section, when p_{i_k} invokes* release()*, it writes the value* MUTEX$\langle k, \perp\rangle$ *in at least $(m - (n - 1))$ anonymous registers.*

Proof Let us first observe that, as (i) the underlying mutex algorithm is independent of the values of the local variables ct_i, (ii) is deadlock-free, and (iii) each process invokes acquire() only once, it is actually starvation-free.

Let p_{i_1} be the first process that enters the critical section. As $ct_{i_1} = 0$, it follows that after line 2 we have $ct_{i_1} = 1$. Then, when p_{i_1} invokes release(), it writes MUTEX$\langle 1, \perp\rangle$ in at least $(m - (n - 1))$ anonymous registers (Property Mutex-4 and line 4 of Algorithm 2). It follows then (i) from Property Mutex-2 and lines 1-2 of Algorithm 2), and (ii) Property Mutex-1, Property Mutex-3, and line 4 of Algorithm 2, that when another process p_{i_2} enters the critical section, p_{i_2} has previously read and written all registers, from which we conclude from lines 1-5 of Algorithm 2 that $ct_{i_2} = 1$. It follows that p_{i_2} increases ct_{i_2} from 1 to 2 at line 2 of Algorithm 1.

The previous reasoning being repeated n times, we eventually have: $ct_{i(x)} = x$ at each process $p_{i(x)}$, $1 \leq x \leq n - 1$, and $ct_{i_n} = n$ at process p_{i_n}. It follows that no process blocks forever when it executes the lines 1-4 of Algorithm 1. $\square_{Lemma\ 1}$

Lemma 2. *The local mapping function $\mathsf{map}_i()$ computed by each process p_i is a permutation over the set of register indexes $\{1, \cdots, m\}$. Moreover, for any index $y \in \{1, \cdots, m\}$ and any pair of processes p_i and p_j, $SM_i[\mathsf{map}_i(y)]$ and $SM_j[\mathsf{map}_j(y)]$ address the very same register.*

Proof Let us assume that a process p_i executes line 6. From Lemma 1 there is a single such process p_i. Let p_j be any other process that executes lines 7-8. Due to the "repeat" loop of line 7, p_j executes line 8 only after all registers contain the tag DESA. Only p_i writes the registers with this tag, and (at line 6) wrote DESA(y) inside $SM_i[y]$, for each $y \in \{1, ..., m\}$. Hence, when p_j reads DESA(y) from $SM_j[x]$, it learns that this register is known by p_i as $SM_i[y]$. At line 8, p_j consequently considers x as the value of $\mathsf{map}_j(y)$. It follows that $SM_j[\mathsf{map}_j(y)]$ (i.e., $SM_j[x]$) and $SM_i[\mathsf{map}_i(y)]$ (which is $SM_i[y]$) denote the very same read/write register. As this is true for any process $p_j \neq p_i$, the lemma follows. $\square_{Lemma\ 2}$

Lemma 3. *Any process p_i terminates the operation* de-anonymize().

Proof The proof follows from Lemma 1, which states that all processes enter and leave the critical section. Moreover, as p_{i_n} executes line 6 of Algorithm 1, it follows that no other process can block forever at line 7 of this algorithm, which concludes the proof of the lemma. $\square_{Lemma\ 3}$

Theorem 1. *Algorithm* 1 *is a symmetric algorithm that solves the* de-anonymization *problem in a system made up of* n *asynchronous processes communicating by reading and writing* m *anonymous read/write atomic registers, where* m *belongs to the set* $M(n) = \{m$ *such that* $\forall \ell :\ 1 < \ell \leq n$*:* $\mathsf{gcd}(\ell, m) = 1\} \setminus \{1\}$.

Proof A simple examination of the code shows that process identities are compared only by equality, from which follows the "symmetry" property. The rest of the proof follows from Lemma 2 and Lemma 3. $\square_{Theorem\ 1}$

6 Reducing the Size of Control Information

Algorithm 1 requires that, once de-anonymized, each register must contain forever $1 + \mathsf{log}_2 m$ bits of control information. This section shows that this information can be reduced to a single bit.

Revisiting the shared memory. Each read/write register $SM[x]$ is now assumed to be composed of two parts $SM[x].BIT$ and $SM[x].RM$, more precisely, we have $SM[x] = \langle SM[x].BIT,\ SM[x].RM\rangle$. $SM[x].BIT$ is for example the leftmost bit of $SM[x]$, and $SM[x].RM$ the other bits. The meaning and the use of $SM[x].RM$ are exactly the same as $SM[x]$ in Algorithm 1 and Algorithm 2. For each x, $SM[x].BIT$ is initialized to 0, while (as in Algorithm 1) $SM[x].RM$ is initialized to $\text{MUTEX}\langle 0, \perp\rangle$.

To simplify both the writing and the reading of the improved algorithm, we write

- " $SM_i[x] \leftarrow \text{DESA}(x)$" when the first bit of $SM_i[x]$ is not modified by the write (line 6),
- "$SM_i.\mathsf{scan}()$ when we are interested in the $SM_i.RM$" part of the registers only (line 7),
- "$BIT_i[x] \leftarrow 1$" when the remaining part of $SM_i[x]$ is not modified by the write (line 15),
- "$BIT_i.\mathsf{scan}()$" when we are interested in the bits $SM_i.BIT$ only (line 16).

Behavior of a process p_i. Algorithm 3 is the improved algorithm. It is Algorithm 1 (lines 1-9), followed by a second global synchronization phase (lines 10-17), which is similar to the one at lines 1-9.

After the processes have executed line 9 (end of the first global synchronization phase), each of them knows its mapping function $\mathsf{map}_i()$, but no process knows that all the other processes know their own mapping function. This motivates the second use of the mutual exclusion algorithm, which, as the left bit of any register $SM[x].BIT$ still contains its initial value 0, ensures that when the last process (say p_k) that entered the second critical section exits it, it knows that all the processes have computed their mapping function, and no process that executes the "repeat" loop of line 16 can exit it.

```
operation SM_i.scan() returns ([SM_i[1], ··· , SM_i[m]]).

operation de-anonymize2(id_i) is    % code for p_i
      % the lines 1-9 are the same as in Algorithm 1; the lines 10-17 are new
(1)   acquire(id_i);
(2)     ct_i ← ct_i + 1;
(3)     last1_i ← (ct_i = n);
(4)   release(id_i); % realizes an implicit broadcast of ct_i %
(5)   if (last1_i)
(6)     then for each x ∈ {1, ··· , m} do SM_i[x] ← DESA(x) end for
              % the permutation for p_i is: ∀ y ∈ {1, ··· , m}: map_i(y) = y %
(7)     else repeat sm_i ← SM_i.scan() until (∀ x : sm_i[x] is tagged DESA) end repeat;
(8)          for each x ∈ {1, ··· , m} do map_i(y) ← x where sm_i[x]=DESA(y) end for
             % perm. for p_i is ∀ y ∈ {1, ··· , m}: map_i(y) = x, where sm_i[x] = DESA(y)
(9)   end if;
(10)  acquire(id_i);
(11)    ct_i ← ct_i + 1;
(12)    last2_i ← (ct_i = 2n);
(13)  release(id_i); % realizes an implicit broadcast of ct_i %
(14)  if (last2_i)
(15)    then for each x ∈ {1, ··· , m} do BIT_i[x] ← 1 end for
(16)    else  repeat bit_i ← BIT_i.scan() until (∃ x : bit_i[x] = 1) end repeat
(17)  end if.
```

Algorithm 3: Algorithm with a single bit of control information

To identify the last process that entered the (second) critical section, when a process p_i is inside the critical section it increases the abstract register CT (line 11), and sets $last2_i$ to `true` only if it discovers it is the last process that accessed the critical section (line 12), More precisely, we have the following.

- If p_i is not the last process to increase CT (locally represented by ct_i), $last2_i$ is equal to `false`, and consequently p_i waits until it sees at least one register whose bit $SM_i[x].BIT$ is equal to 1 (line 16). When this occurs p_i learns that the second phase is terminated (hence it knows that all the processes have computed their mapping function), and it can proceed to execute an upper layer non-anonymous register algorithm.
- Differently, if p_i is the last process to increase CT, it changes to 1 the left bit of all the registers (line 15), which unblocks all the other processes. As the bits $SM_i[x].BIT$ are never reset to 0, eventually all the processes know that each of them knows its mapping function.

As they follow the same synchronization pattern, the proof of the second part of Algorithm 3 (lines 10-17) is the same as the one of its first global synchronization phase (lines 1-9), which is the same as the one of Algorithm 1.

7 Conclusion

In addition to introducing the memory de-anonymization problem, this paper has shown that, in an n-process system where $n \geq 2$ and process identities can only be compared with equality, a shared memory made up of m anonymous read/write registers and a shared memory made up of m non-anonymous read/write registers have the same computability power for the values of m satisfying the necessary condition for deadlock-free anonymous mutex algorithms from [23], namely m must belong to the set $M(n) = \{\ m \mid$ such that $\forall \ell :\ 1 < \ell \leq n$: $\mathsf{gcd}(\ell, m) = 1\} \setminus \{1\}$. Let us observe that, as it includes an infinite sequence of prime numbers, $M(n)$ is infinite. It follows that, once de-anonymization (in which all processes participate) is obtained, it becomes possible to use a symmetric starvation-free mutex algorithm, thereby obtaining a symmetric starvation-free mutex algorithm working on top of an anonymous memory[1].

We emphasize that the above construction (of running a starvation-free mutex algorithm on top of a de-anonymization layer), does not solve the original open problem from [20], regarding the existence of a memory-anonymous two-process starvation-free mutex algorithm. In the definition of the mutex problem participation is not required (a process may never leave its remainder code), while our implementation of the de-anonymization layer, assumes that participation is required, or, equivalently, that the number of participants is known by all processes.

As stated in [23], the memory-anonymous communication model "enables us to understand better the intrinsic limits for coordinating the actions of asynchronous processes". It consequently enriches our knowledge of what can be (or cannot be) done when an adversary replaced a common addressing function, by individual and independent addressing functions, one per process.

Among problems that remain open, there are the design of de-anonymization algorithms (symmetric with equality only, or symmetric with equality, greater than, and lower than) not based on an underlying memory anonymous mutex algorithm, and the statement of a necessary and sufficient condition on the value of m (size of the anonymous memory) for which de-anonymization is possible (for each type of symmetry).

Acknowledgments

This work was partially supported by the French ANR project DESCARTES (16-CE40-0023-03) devoted to layered and modular structures in distributed computing. The authors want to thank the referees for their constructive comments.

References

1. Aigner M. and Ziegler G., Proofs from THE BOOK (4th edition). Springer, 274 pages, ISBN 978-3-642-00856-6 (2010)

[1] Peterson's mutual exclusion algorithm is such a symmetric algorithm [17]. As it requires $2n-1$ non-anonymous atomic registers, we need to have both $m \in M(n)$ and $m \geq 2n-1$.

2. Angluin D., Local and global properties in networks of processes. *Proc. 12th Symposium on Theory of Computing (STOC'80)*, ACM Press, pp. 82-93, (1980)
3. Aghazadeh Z., Imbs D., Raynal M., Taubenfeld G., and Woelfel Ph., Optimal memory-anonymous symmetric deadlock-free mutual exclusion. *Proc. 38th ACM Symposium on Principles of Distributed Computing (PODC'19)*, ACM Press, 10 pages (2019)
4. Attiya H., Gorbach A., and Moran S., Computing in totally anonymous asynchronous shared-memory systems. *Information and Computation*, 173(2):162-183 (2002)
5. Bonnet F. and Raynal M., Anonymous asynchronous systems: the case of failure detectors. *Distributed Computing*, 26(3):141-158 (2013)
6. Bouzid Z., Raynal M., and Sutra P., Anonymous obstruction-free (n, k)-set agreement with $(n - k + 1)$ atomic read/write registers. *Distributed Computing*, 31(2):99-117 (2018)
7. Delporte C., Fauconnier H., Gafni E., and Lamport L., Adaptive register allocation with a linear number of registers. *Proc. 27th Int'l Symposium on Distributed Computing (DISC'13)*, Springer LNCS 8205, pp. 269-283 (2013)
8. Dijkstra E.W., Solution of a problem in concurrent programming control. *Communications of the ACM*, 8(9):569 (1965)
9. Dijkstra E.W., Some beautiful arguments using mathematical induction. *Algorithmica*, 13(1):1-8 (1980)
10. Garg V.K. and Ghosh J., Symmetry in spite of hierarchy. *Proc. 10th Int'l Conference on Distributed Computing Systems (ICDCS'90)*, IEEE Computer Press, pp. 4-11 (1990)
11. Guerraoui R. and Ruppert E., Anonymous and fault-tolerant shared-memory computations. *Distributed Computing*, 20:165-177 (2007)
12. Johnson R.E., and Schneider F.B., Symmetry and similarity in distributed systems. *Proc.4th ACM Symposium on Principles of Distributed Computing (PODC'85)*, pp. 13-22, ACM Press (1985)
13. Lamport L., On interprocess communication, Part I: basic formalism. *Distributed Computing*, 1(2):77-85 (1986)
14. Navlakha S. and Bar-Joseph Z., Algorithms in nature: the convergence of systems biology and computational thinking. *Molecular systems biology*, 7(546):1-11 (2011)
15. Navlakha S. and Bar-Joseph Z., Distributed information processing in biological and computational systems. *Communications of the ACM*, 58(1):94-102 (2015)
16. Perlis A.J., Epigrams on programming, *ACM SIGPLAN Notices*, 17(1):7-13 (1982)
17. Peterson G.L., Myths about the mutual exclusion problem, *Information Processing Letters*, 12(3):115-116 (1981)
18. Rashid S., Taubenfeld G., and Bar-Joseph Z., Genome wide epigenetic modifications as a shared memory consensus. *6th Workshop on Biological Distributed Algorithms (BDA'18)*, London (2018)
19. Raynal M., *Concurrent programming: algorithms, principles and foundations*. Springer, 515 pages, ISBN 978-3-642-32026-2 (2013)
20. Raynal M., *Fault-tolerant message-passing distributed systems: an algorithmic approach*. Springer, 492 pages, ISBN 978-3-319-94140-0 (2018)
21. Raynal M. and Cao J., Anonymity in distributed read/write systems: an introductory survey. *Proc. 6th Int'l Conference on Networked Systems (NETYS'18)*, Springer LNCS 11028, pp. 122-140 (2018)
22. Taubenfeld G., *Synchronization algorithms and concurrent programming*. Pearson Education/Prentice Hall, 423 pages, ISBN 0-131-97259-6 (2006)
23. Taubenfeld G., Coordination without prior agreement. *Proc. 36th ACM Symposium on Principles of Distributed Computing (PODC'17)*, ACM Press, pp. 325-334 (2017)
24. Yamashita M. and Kameda T., Computing on anonymous networks: Part I -characterizing the solvable cases. *IEEE Transactions on Parallel Distributed Systems*, 7(1):69-89 (1996)