The Synchronization Power (Consensus Number) of Access-Control Objects: the Case of AllowList and DenyList

Davide Frey ✉
Inria, IRISA, CNRS, Université de Rennes, France

Mathieu Gestin ✉
Inria, IRISA, CNRS, Université de Rennes, France

Michel Raynal ✉
IRISA, Inria, CNRS, Université de Rennes, France

Abstract

This article studies the synchronization power of AllowList and DenyList objects under the lens provided by Herlihy's consensus hierarchy. It specifies AllowList and DenyList as distributed objects and shows that, while they can both be seen as specializations of a more general object type, they inherently have different synchronization power. While the AllowList object does not require synchronization between participating processes, a DenyList object requires processes to reach consensus on a specific set of processes. These results are then applied to a more global analysis of anonymity-preserving systems that use AllowList and DenyList objects. First, a blind-signature-based e-voting is presented. Second, DenyList and AllowList objects are used to determine the consensus number of a specific decentralized key management system. Third, an anonymous money transfer algorithm using the association of AllowList and DenyList objects is presented. Finally, this analysis is used to study the properties of these application, and to highlight efficiency gains that they can achieve in message passing environment.

2012 ACM Subject Classification Theory of computation → Distributed computing models; Security and privacy → Access control; Security and privacy → Pseudonymity, anonymity and untraceability

Keywords and phrases Access control, AllowList/DenyList, Blockchain, Consensus number, Distributed objects, Modularity, Privacy, Synchronization power

Digital Object Identifier 10.4230/LIPIcs.DISC.2023.21

Related Version *Full Version*: `https://doi.org/10.48550/arXiv.2302.06344v2` [19]

Funding This work was partially funded by the PriCLeSS project and by the SOTERIA H2020 project. PriCLeSS was granted by the Labex CominLabs excellence laboratory of the French ANR (ANR-10-LABX-07-01). SOTERIA received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No101018342. This content reflects only the author's view. The European Agency is not responsible for any use that may be made of the information it contains.

Acknowledgements We wish to thank the anonymous reviewers for their insightful comments and remarks that led to significant improvements to our paper.

1 Introduction

The advent of blockchain technologies increased the interest of the public and industry in distributed applications, giving birth to projects that have applied blockchains in a plethora of use cases. These include e-vote systems [16], naming services [1, 27], Identity Management Systems [18, 31], supply-chain management [30], or Vehicular Ad hoc Network [21]. However,

37th International Symposium on Distributed Computing (DISC 2023).
Editor: Rotem Oshman; Article No. 21; pp. 21:1–21:23
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

this use of the blockchain as a swiss-army knife that can solve numerous distributed problems highlights a lack of understanding of the actual requirements of those problems. Because of these poor specifications, implementations of these applications are often sub-optimal.

This paper thoroughly studies a class of problems widely used in distributed applications and provides a guideline to implement them with reasonable but sufficient tools.

Differently from the previous approaches, it aims to understand the amount of synchronization required between processes of a system to implement *specific* distributed objects. To achieve this goal it studies such objects under the lens of Herlihy's consensus number [24]. This parameter is inherently associated to shared memory distributed objects, and has no direct correspondence in the message passing environment. However, in some specific cases, this information is enough to provide a better understanding of the objects analyzed, and thus, to gain efficiency in the message passing implementations. For example, recent papers [22, 5] have shown that cryptocurrencies can be implemented without consensus and therefore without a blockchain. In particular, Guerraoui et al. [22] show that k-asset transfer has a consensus number k where k is the number of processes that can withdraw currency from the same account [23]. Similarly, Alpos et al. [3] have studied the synchronization properties of ERC20 token smart contracts and shown that their consensus number varies over time as a result of changes in the set of processes that are approved to send tokens from the same account. These two results consider two forms of asset transfer: the classical one and the one implemented by the ERC20 token, which allows processes to dynamically authorize other processes. The consensus number of those objects depends on specific and well identified processes. From this study, it is possible to conclude that the consensus algorithms only need to be performed between those processes. Therefore, in these specific cases, the knowledge of the consensus number of an object can be directly used to implement more efficient message passing applications. Furthermore, even if this study uses a shared memory model, with crash prone processes, its results can be used to implement more efficient Byzantine resilient algorithm, in a message passing environment. This paper proposes to extend this knowledge to a broader class of applications.

Indeed, the transfer of assets, be them cryptocurrencies or non-fungible tokens, does not constitute the only application in the Blockchain ecosystem. In particular, as previously indicated, a number of applications like e-voting [16], naming [1, 27], or Identity Management [18, 31] use Blockchain as a tool to implement some form of access control. This is often achieved by implementing two general-purpose objects: AllowLists and DenyLists. An AllowList provides an opt-in mechanism. A set of managers can maintain a list of authorized parties, namely the AllowList. To access a resource, a party (user) must prove the presence of an element associated with its identity in the AllowList. A DenyList provides instead an opt-out mechanism. In this case, the managers maintain a list of revoked elements, the DenyList. To access a resource, a party (user) must prove that no corresponding element has been added to the DenyList. In other words, AllowList and DenyList support, respectively, set-membership and set-non-membership proofs on a list of elements.

The proofs carried out by AllowList and DenyList objects often need to offer privacy guarantees. For example, the Sovrin privacy preserving Decentralized Identity-Management System (DIMS) [18] associates an AllowList[1] with each verifiable credential that contains the identifiers of the devices that can use this verifiable credential. When a device uses a credential with a verifier, it needs to prove that the identifier associated with it belongs to the AllowList. This proof must be done in zero knowledge, otherwise the verifier would learn

[1] In reality this is a variant that mixes AllowList and DenyList which we discuss in Appendix A.

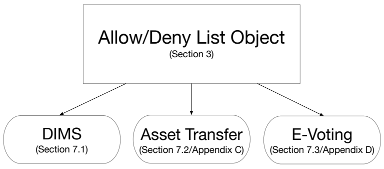
the identity of the device, which in turn could serve as a pseudo-identifier for the user. For this reason, AllowList and DenyList objects support respectively a zero-knowledge proof of set membership or a zero-knowledge proof of set non-membership.

Albeit similar, the AllowList and DenyList objects differ significantly in the way they handle the proving mechanism. In the case of an AllowList, no security risk appears if access to a resource is prohibited to a process, even if a manager did grant this right. As a result, a transient period in which a user is first allowed, then denied, and then allowed again to access a resource poses no problem. On the contrary, with a DenyList, being allowed access to a resource after being denied it poses serious security problems. Hence, the DenyList object is defined with an additional anti-flickering property prohibiting such transient periods. This property is the main difference between an AllowList and a DenyList object and is the reason for their distinct consensus numbers.

Existing systems [16, 1, 27, 18, 31] that employ AllowList and DenyList objects implement them on top of a heavy blockchain infrastructure, thereby requiring network-level consensus to modify their content. As already said, this paper studies this difference under the lens of the consensus number [23]. It shows that (i) the consensus number of an AllowList object is 1, which means that an AllowList can be implemented without consensus; and that (ii) the consensus number of a DenyList is instead equal to the number of processes that can conduct prove operations on the DenyList, and that only these processes need to synchronize. Both data structures can therefore be implemented without relying on the network-level consensus provided by a blockchain, which opens the door to more efficient implementations of applications based on these data structures.

To summarize, this paper presents the following three contributions. We note $CN(X)$ the consensus number of the object X.

1. It formally defines and studies AllowList and DenyList as distributed objects (Section 3).
2. It analyses the consensus number of these objects: it shows that the AllowList does not require synchronization between processes, i.e. $CN(\text{AllowList }) = 1$ (Section 5), while the DenyList requires the synchronization of all the k verifiers of its set-non-membership proofs, i.e $CN(\text{DenyList }) = k$ (Section 6).
3. It uses these theoretical results to give intuitions on their optimal implementations. Namely the implementation of a DIMS, as well as of an e-vote system and an Anonymous Asset-Transfer (AAT) algorithm (Appendix B and in the full version of this paper [19]). More precisely, the consensus number of an AAT algorithm depends on the required anonymity level, i.e. $CN(\text{AAT}) = CN(\text{DenyList}) = k = CN(k\text{-shared Asset Transfer object})$. The consensus number of an e-vote system depends on the number k of vote-casting servers, i.e $CN(\text{e-vote}) = k$. Finally, the consensus number of a the revocation mechanism in a DIMS is 2 in most cases.



To the best of our knowledge, this paper is the first to study the AllowList and DenyList from a distributed algorithms point of view. So we believe our results can provide a powerful tool to identify the consensus number of recent distributed objects that make use of them and to provide more efficient implementations of such objects.

2 Preliminaries

2.1 Computation Model

Model

Let Π be a set of N asynchronous sequential crash-prone processes $p_1, \cdots, p_N$. Sequential means that each process invokes one operation of its own algorithm at a time. We assume the local processing time to be instantaneous, but the system is asynchronous. This means that non-local operations can take a finite but arbitrarily long time and that the relative speeds between the clocks of the different processes are unknown. Finally, processes are crash-prone: any number of processes can prematurely and definitely halt their executions. A process that crashes is called *faulty*. Otherwise, it is called *correct*. The system is eponymous: a unique positive integer identifies each process, and this identifier is known to all other processes.

Communication

Processes communicate via shared objects of type T. Each operation on a shared object is associated with two *events*: an *invocation* and a *response*. An object type T is defined by a tuple (Q, Q_0, O, R, Δ), where Q is a set of states, $Q_0 \subseteq Q$ is the set of initial states, O is the set of operations a process can use to access this object, R is the set of responses to these operations, and $\Delta \subseteq \Pi \times Q \times O \times R \times Q$ is the transition function defining how a process can access and modify an object.

Histories and Linearizability

A *history* [24] is a sequence of invocations and responses in the execution of an algorithm. An invocation with no matching response in a history, H, is called a *pending* invocation. A *sequential history* is one where the first event is an invocation, and each invocation – except possibly the last one – is immediately followed by the associated response. A sub-history is a sub-sequence of events in a history. A process sub-history $H|p_i$ of a history H is a sub-sequence of all the events in H whose associated process is p_i. Given an object x, we can similarly define the object sub-history $H|x$. Two histories H and H' are equivalent if $H|p_i = H'|p_i$, $\forall i \in \{1, \cdots, N\}$.

In this paper, we define the specification of a shared object, x, as the set of all the allowed sub-histories, $H|x$. We talk about a sequential specification if all the histories in this set are sequential. A *legal history* is a history H in which, for all objects x_i of this history, $H|x_i$ belongs to the specification of x_i. The completion $\bar{H}$ of a history H is obtained by extending all the pending invocations in H with the associated matching responses. A history H induces an irreflexive partial order $<_H$ on operations, i.e. $op_0 <_H op_1$ if the response to the operation op_0 precedes the invocation of operation op_1. A history is sequential if $<_H$ is a total order. The algorithm executed by a correct process is *wait-free* if it always terminates after a finite number of steps. A history H is linearizable if a completion $\bar{H}$ of H is equivalent to some legal sequential history S and $<_H \subseteq <_S$.

Consensus number

The consensus number of an object of type T (noted $\text{cons}(T)$) is the largest n such that it is possible to wait-free implement a consensus object from atomic read/write registers and objects of type T in a system of n processes. If an object of type T makes it possible to wait-free implement a consensus object in a system of any number of processes, we say the consensus number of this object is ∞. Herlihy [23] proved the following well-known theorem.

▶ **Theorem 1.** *Let X and Y be two atomic objects type such that cons(X) $= m$ and cons(Y) $= n$, and $m < n$. There is no wait-free implementation of an object of type Y from objects of type X and read/write registers in a system of more than m processes.*

We will determine the consensus number of the DenyList and the AllowList objects using Atomic Snapshot objects and consensus objects in a set of k processes. A Single Writer Multi Reader (SWMR) [2] Atomic Snapshot object is an array of fixed size, which supports two operations: Snapshot and Update. The Snapshot() operation allows a process p_i to read the whole array in one atomic operation. The Update(v, i) operation allows a process p_i to write the value v in the i-th position of the array. Afek et al. showed that a SWMR Snapshot object can be wait-free implemented from read/write registers [2], i.e., this object type has consensus number 1. This paper assumes that all Atomic Snapshot objects used are SWMR. A consensus object provides processes with a single one-shot operation *propose()*. When a process p_i invokes *propose(v)* it proposes v. This invocation returns a *decided* value such that the following three properties are satisfied.

- *Validity*: If a correct process decides value v, then v was proposed by some process;
- *Agreement*: No two correct processes decide differently; and
- *Termination*: Every correct process eventually decides.

A k-consensus object is a consensus object accessed by at most k processes.

2.2 Number theory preliminaries

Cryptographic Commitments

A *cryptographic commitment* is a cryptographic scheme that allows a Prover to commit to a value v while hiding it. The commitment scheme is a two phases protocol. First, the prover computes a binding value known as commitment, C, using a function *Commit*. *Commit* takes as inputs the value v and a random number r. The prover sends this hiding and binding value C to a verifier. In the second phase, the prover reveals the committed value v and the randomness r to the verifier. The verifier can then verify that the commitment C previously received refers to the transmitted values v and r. This commitment protocol is the heart of Zero Knowledge Proof (ZKP) protocols.

Zero Knowledge Proof of set operations

A Zero Knowledge Proof (ZKP) system is a cryptographic algorithm that allows a prover to prove some Boolean statement about a value x to a verifier without leaking any information about x. A ZKP system is initialized for a specific language $\mathcal{L}$ of the complexity class $\mathcal{NP}$. The proving mechanism takes as input $\mathcal{L}$ and outputs a proof π. Knowing $\mathcal{L}$ and π, any verifier can verify that the prover knows a value $x \in \mathcal{L}$[2]. However, the verifier cannot learn the value x used to produce the proof. In the following, it is assumed there exists efficient non interactive ZKP systems of set-(non)-membership (e.g., constructions from [8]).

[2] The notation $x \in \mathcal{L}$ denotes the fact that x is a solution to the instance of the problem expressed by the language $\mathcal{L}$

3 The AllowList and DenyList objects: Definition

Distributed AllowList and DenyList object types are the type of objects that allow a set of managers to control access to a resource. The term ”resource” is used here to describe the goal a user wants to achieve and which is protected by an access control policy. A user is granted access to the resource if it succeeds in proving that it is authorized to access it. First, we describe the AllowList object type. Then we consider the DenyList object type.

The AllowList object type is one of the two most common access control mechanisms. To access a resource, a process $p \in \Pi_V$ needs to prove it knows some element v previously authorized by a process $p_M \in \Pi_M$, where $\Pi_M \subseteq \Pi$ is the set of managers, and $\Pi_V \subseteq \Pi$ is the set of processes authorized to conduct proofs. We call verifiers the processes in Π_V. The sets Π_V and Π_M are predefined and static. They are parameters of the object. Depending on the usage, these subset can either be small, or they can contain all the processes in Π.

A process $p \in \Pi_V$ proves that v was previously authorized by invoking a PROVE(v) operation. This operation is said to be valid if some manager in Π_M previously invoked an APPEND(v) operation. Intuitively, we can see the invocation of APPEND(v) as the action of authorizing some process to access the resource. On the other hand, the PROVE(v) operation, invoked by a prover process, $p \in \Pi_V$, proves to the other processes in Π_V that they are authorized. However, this proof is not enough in itself. The verifiers of a proof must be able to verify that a valid PROVE has been invoked. To this end, the AllowList object type is also equipped with a READ() operation. This operation can be invoked by any process in Π and returns a random permutation of all the valid PROVE invoked, along with the identity of the processes that invoked them. All processes in Π can invoke the READ operation.[3]

An optional anonymity property can be added to the AllowList object to enable privacy-preserving implementations. This property ensures that other processes cannot learn the value v proven by a PROVE(v) operation.

The AllowList object type is formally defined as a sequential object, where each invocation is immediately followed by a response. Hence, the sequence of operations defines a total order, and each operation can be identified by its place in the sequence.

▶ **Definition 2.** The *AllowList* object type supports three operations: APPEND, PROVE, and READ. These operations appear as if executed in a sequence Seq such that:

- *Termination.* A PROVE, an APPEND, or a READ operation invoked by a correct process always returns.
- APPEND *Validity.* The invocation of APPEND(x) by a process p is valid **if** $p \in \Pi_M \subseteq \Pi$ **and** $x \in \mathcal{S}$, where $\mathcal{S}$ is a predefined set. Otherwise, the operation is invalid.
- PROVE *Validity.* **If** the invocation of *op* =PROVE(x) by a process p is valid, **then** $p \in \Pi_V \subseteq \Pi$ **and** a valid APPEND(x) appears before *op* in Seq. Otherwise, the invocation is invalid.
- *Progress.* **If** a valid APPEND(x) is invoked, **then** there exists a point in Seq such that any PROVE(x) invoked after this point by any process $p \in \Pi_V$ will be valid.
- READ *Validity.* The invocation of *op* =READ() by a process $p \in \Pi_V$ returns the list of valid invocations of PROVE that appears before *op* in Seq along with the names of the processes that invoked each operation.

[3] Usually, AllowList objects are implemented in a message-passing setting. In these cases, the READ operation is implicit. Each process knows a local state of the distributed object, and can inspect it any time. In the shared-memory setting, we need to make this READ operation explicit.

- *Optional - Anonymity.* Let us assume the process p invokes a PROVE(v) operation. If the process p' invokes a READ() operation, then p' cannot learn the value v unless p leaks additional information.[4]

The AllowList object is defined in an append-only manner. This definition makes it possible to use it to build all use cases explored in this paper. However, some use cases could need an DenyList with an additional REMOVE operation. This variation is studied in Appendix A.

The DenyList object type can be informally presented as an access policy where, contrary to the AllowList object type, all users are authorized to access the resource in the first place. The managers are here to revoke this authorization. A manager revokes a user by invoking the APPEND(v) operation. A user uses the PROVE(v) operation to prove that it was not revoked. A PROVE(v) invocation is invalid only if a manager previously revoked the value v.

All the processes in Π can verify the validity of a PROVE operation by invoking a READ() operation. This operation is similar to the AllowList's READ operation. It returns the list of valid PROVE invocations along with the name of the processes that invoked it.

There is one significant difference between the DenyList and the AllowList object types. With an AllowList, if a user cannot access a resource immediately after its authorization, no malicious behavior can harm the system – the system's state is equivalent to its previous state. However, with a DenyList, a revocation not taken into account can let a malicious user access the resource and harm the system. In other words, access to the resource in the DenyList case must take into account the "most up to date" available revocation list.

To this end, the DenyList object type is defined with an additional property. The anti-flickering property ensures that if an APPEND operation is taken into account by one PROVE operation, it will be taken into account by every subsequent PROVE operation. Along with the progress property, the anti-flickering property ensures that the revocation mechanism is as immediate as possible. The DenyList object is formally defined as a sequential object, where each invocation is immediately followed by a response. Hence, the sequence of operations define a total order, and each operation can be identified by its place in the sequence.

▶ **Definition 3.** The *DenyList* object type supports three operations: APPEND, PROVE, and READ. These operations appear as if executed in a sequence Seq such that:

- *Termination.* A PROVE, an APPEND, or a READ operation invoked by a correct process always returns.
- APPEND *Validity.* The invocation of APPEND(x) by a process p is valid **if** $p \in \Pi_M \subseteq \Pi$ **and** $x \in \mathcal{S}$, where $\mathcal{S}$ is a predefined set. Otherwise, the operation is invalid.
- PROVE *Validity.* **If** the invocation of a op =PROVE(x) by a correct process p is not valid, **then** $p \notin \Pi_V \subseteq \Pi$ **or** a valid APPEND(x) appears before op_P in Seq. Otherwise, the operation is valid.
- PROVE *Anti-Flickering.* **If** the invocation of a operation op =PROVE(x) by a correct process $p \in \Pi_V$ is invalid, **then** any PROVE(x) that appears after op in Seq is invalid.[5]

[4] The Anonymity property only protects the value v. The system considered is eponymous. Hence, the identity of the processes is already known. However, the anonymity of v makes it possible to hide other information. For example, the identity of a client that issues a request to a process of the system. These example are discussed in Section 7.

[5] The only difference between the AllowList and the DenyList object types is this anti-flickering property. As it is shown in Section 5 and in Section 6, the AllowList object has consensus number 1, and the DenyList object has consensus number $k = |\Pi_V|$. Hence, this difference in term of consensus number is due solely to the anti-flickering property. It is an open question whether a variation of this property could transform any consensus number 1 object into a consensus number k object.

Table 1 Transition function Δ for the PROOF-LIST object.

Process	Operation	Initial state	Res-ponse	Final state	Conditions
$p_i \in \Pi_M$	APPEND(y)	$(\textit{listed-values} = \{x \in \mathcal{S}\},$ $\textit{proofs} = (\{(p_j \in \Pi, \widehat{\mathcal{S}} \subseteq \mathcal{S}, \mathsf{P} \in \mathcal{P}_{\mathcal{L}_{\widehat{\mathcal{S}}}})\}))$	True	$(\textit{listed-values} \cup \{y\},$ $\textit{proofs})$	$y \in \mathcal{S}$
p_i	APPEND(y)	$(\textit{listed-values} = \{x \in \mathcal{S}\},$ $\textit{proofs} = (\{(p_j \in \Pi, \widehat{\mathcal{S}} \subseteq \mathcal{S}, \mathsf{P} \in \mathcal{P}_{\mathcal{L}_{\widehat{\mathcal{S}}}})\}))$	False	$(\textit{listed-values}, \textit{proofs})$	$p_i \notin \Pi_M \vee y \notin \mathcal{S}$
$p_i \in \Pi_V$	PROVE(y)	$(\textit{listed-values} = \{x \in \mathcal{S}\},$ $\textit{proofs} = (\{(p_j \in \Pi, \widehat{\mathcal{S}} \subseteq \mathcal{S}, \mathsf{P} \in \mathcal{P}_{\mathcal{L}_{\widehat{\mathcal{S}}}})\}))$	$(\mathcal{A}, \mathsf{P})$	$(\textit{listed-values},$ $\textit{proofs} \cup \{(p_i, \mathcal{A}, \mathsf{P})\})$	$\forall y \in \mathcal{L}_{\mathcal{A}} \wedge \mathcal{A} \subseteq \textit{listed-values}$ $\wedge \forall \mathsf{P} \in \mathcal{P}_{\mathcal{L}_{\mathcal{A}}} \wedge \mathsf{C}(y, \widehat{\mathcal{S}}) = 1$
p_i	PROVE(y)	$(\textit{listed-values} = \{x \in \mathcal{S}\},$ $\textit{proofs} = (\{(p_j \in \Pi, \widehat{\mathcal{S}} \subseteq \mathcal{S}, \mathsf{P} \in \mathcal{P}_{\mathcal{L}_{\widehat{\mathcal{S}}}})\}))$	False	$(\textit{listed-values}, \textit{proofs})$	$\forall y \notin \mathcal{L}_{\mathcal{A}} \vee \mathcal{A} \not\subseteq \textit{listed-values}$ $\vee \forall \mathsf{P} \notin \mathcal{P}_{\mathcal{L}_{\mathcal{A}}} \vee \forall p_i \notin \Pi_V$ $\vee \mathsf{C}(y, \widehat{\mathcal{S}}) = 0$
$p_i \in \Pi$	READ()	$(\textit{listed-values} = \{x \in \mathcal{S}\},$ $\textit{proofs} = (\{(p_j \in \Pi, \widehat{\mathcal{S}} \subseteq \mathcal{S}, \mathsf{P} \in \mathcal{P}_{\mathcal{L}_{\widehat{\mathcal{S}}}})\}))$	*proofs*	$(\textit{listed-values}, \textit{proofs})$	

- READ *Validity.* The invocation of *op* =READ() by a process $p \in \Pi_V$ returns the list of valid invocations of PROVE that appears before *op* in Seq along with the names of the processes that invoked each operation.
- *Optional - Anonymity.* Let us assume the process p invokes a PROVE(v) operation. If the process p' invokes a READ() operation, then p' cannot learn the value v unless p leaks additional information.

4 PROOF-LIST object specification

Section 5 and Section 6 propose an analysis of the synchronization power of the AllowList and the DenyList object types using the notion of consensus number. Both objects share many similarities. Indeed, the only difference is the type of proof performed by the user and the non-flickering properties. Therefore, this section defines the formal specification of the PROOF-LIST object type, a new generic object that can be instantiated to describe the AllowList or the DenyList object type.

The PROOF-LIST object type is a distributed object type whose state is a pair of arrays (*listed-values*, *proofs*). The first array, *listed-values*, represents the list of authorized/revoked elements. It is an array of objects in a set $\mathcal{S}$, where $\mathcal{S}$ is the universe of potential elements. The second array, *proofs*, is a list of assertions about the *listed-values* array. Given a set of managers $\Pi_M \subseteq \Pi$ and a set of verifiers $\Pi_V \subseteq \Pi$, the PROOF-LIST object supports three operations. First, the APPEND(v) operation appends a value $v \in \mathcal{S}$ to the *listed-values* array. Any process in the manager's set can invoke this operation. Second, the PROVE(v) operation appends a valid proof about the element $v \in \mathcal{S}$ relative to the *listed-values* array to the *proofs* array. This operation can be invoked by any process $p \in \Pi_V$. Third, the READ() operation returns the *proofs* array.

The sets Π_V and Π_M are static, predefined subsets of Π. There is no restriction on their compositions. The choice of these sets only depends on the usage of the AllowList or the DenyList. Depending on the usage, they can either contain a small subset of processes in Π or they can contain the whole set of processes of the system.

To express the proofs produced by a process p, we use an abstract language $\mathcal{L}_{\mathcal{A}}$ of the complexity class $\mathcal{NP}$, which depends on a set $\mathcal{A}$. This language will be specified for the AllowList and the DenyList objects in Section 5 and Section 6. The idea is that p produces a proof π about a value $v \in \mathcal{S}$. A PROVE invocation by a process p is valid only if the proof π added to the *proofs* array is valid. The proof π is valid if $v \in \mathcal{L}_{\mathcal{A}}$ – i.e., v is a solution to the instance of the problem expressed by $\mathcal{L}_{\mathcal{A}}$, where $\mathcal{L}_{\mathcal{A}}$ is a language of the complexity class

$\mathcal{NP}$ [6] which depends on a subset $\mathcal{A}$ of the *listed-values* array ($\mathcal{A} \subseteq \mathcal{S}$). We note $\mathcal{P}_{\mathcal{L}_\mathcal{A}}$ the set of valid proofs relative to the language $\mathcal{L}_\mathcal{A}$. $\mathcal{P}_{\mathcal{L}_\mathcal{A}}$ can either represent Zero Knowledge Proofs or explicit proofs.

If a proof π is valid, then the PROVE operation returns $(\mathcal{A}, \mathsf{Acc}.\mathit{Prove}(v, \mathcal{A}))$, where $\mathsf{Acc}.\mathit{Prove}(v, \mathcal{A})$ is the proof generated by the operation, and where $\mathcal{A}$ is a subset of values in *listed-values* on which the proof was applied. Otherwise, the PROVE operation returns "False". Furthermore, the *proofs* array also stores the name of the processes that invoked PROVE operations.

Formally, the PROOF-LIST object type is defined by the tuple (Q, Q_0, O, R, Δ), where:

- The set of valid state is $Q = (\mathit{listed\text{-}values} = \{x \in \mathcal{S}\}, \mathit{proofs} = \{(p \in \Pi, \widehat{\mathcal{S}} \subseteq \mathcal{S}, \mathsf{P} \in \mathcal{P}_{\mathcal{L}_{\widehat{\mathcal{S}}}})\})$, where *listed-values* is a subset of $\mathcal{S}$ and *proofs* is a set of tuples. Each tuple in *proofs* consists of a proof associated with the set it applies to and to the identifier of the process that issued the proof;
- The set of valid initial states is $Q_0 = (\emptyset, \emptyset)$, the state where the *listed-values* and the *proofs* arrays are empty;
- The set of possible operation is $O = \{\text{APPEND}(x), \text{PROVE}(y), \text{READ}()\}$, with $x, y \in \mathcal{S}$;
- The set of possible responses is $R = \left\{\text{True, False, } (\widehat{\mathcal{S}} \subseteq \mathcal{S}, \mathsf{P} \in \mathcal{P}_{\mathcal{L}_{\widehat{\mathcal{S}}}}), \{(p \in \Pi, \widehat{\mathcal{S}}' \subseteq \mathcal{S}, \mathsf{P}' \in \mathcal{P}_{\mathcal{L}_{\widehat{\mathcal{S}}}})\}\right\}$, where True is the response to a successful APPEND operation, $(\widehat{\mathcal{S}}, \mathsf{P})$ is the response to a successful PROVE operation, $\{(p, \widehat{\mathcal{S}}', \mathsf{P}')\}$ is the response to a READ operation, and False is the response to a failed operation; and
- The transition function is Δ. The PROOF-LIST object type supports 5 possible transitions. We define the 5 possible transitions of Δ in Table 1.

The first transition of the Δ function models a valid APPEND invocation, a value $y \in \mathcal{S}$ is added to the *listed-values* array by a process in the managers' set Π_M. The second transition of the Δ function represents a failed APPEND invocation. Either the process p_i that invokes this function is not authorized to modify the *listed-values* array, i.e., $p_i \notin \Pi_M$, or the value it tries to append is invalid, i.e., $y \notin \mathcal{S}$. The third transition of the Δ function captures a valid PROVE operation, where a valid proof is added to the *proofs* array. The function C will be used to express the anti-flickering property of the DenyList implementation. It is a boolean function that outputs either 0 or 1. The fourth transition of the Δ function represents an invalid PROVE invocation. Either the proof is invalid, or the set on which the proof is issued is not a subset of the *listed-values* array. Finally, the fifth transition represents a READ operation. It returns the *proofs* array and does not modify the object's state.

The language $\mathcal{L}_\mathcal{A}$ does not directly depend on the *listed-values* array. Hence, the validity of a PROVE operation will depend on the choice of the set $\mathcal{A}$.

5 The consensus number of the AllowList object

This section provides an AllowList object specification based on the PROOF-LIST object. The specification is then used to analyze the consensus number of the object type.

We provide a specification of the AllowList object defined as a PROOF-LIST object, where $\mathsf{C}(y, \widehat{\mathcal{S}}) = 1$ and $\forall\ y \in \mathcal{S}, y \in \mathcal{L}_\mathcal{A} \Leftrightarrow (\mathcal{A} \subseteq \mathcal{S} \wedge y \in \mathcal{A})$.

[6] In this article, $\mathcal{L}_\mathcal{A}$ can be one of the following languages: a value v belongs to $\mathcal{A}$ (AllowList), or a value v does not belongs to $\mathcal{A}$ (DenyList).

In other words, y belongs to a set $\mathcal{A}$. Using the third transition of the Δ function, we can see that $\mathcal{A}$ should also be a subset of the *listed-values* array. Hence, this specification supports proofs of set-membership in *listed-values*. A PROOF-LIST object defined for such language follows the specification of the AllowList. To support this statement, we provide an implementation of the object.

To implement the AllowList object, Algorithm 1 uses two Atomic Snapshot objects. The first one represents the *listed-values* array, and the second represents the *proofs* array. These objects are arrays of N entries. Furthermore, we use a function "Proof" that on input of a set $\mathcal{S}$ and an element y outputs a proof that $y \in$ *listed-values*. This function is used as a black box, and can either output an explicit proof – an explicit proof can be the tuple $(y, \mathcal{A})$, where $\mathcal{A} \subseteq$ *listed-values* – or a Zero Knowledge Proof.

Algorithm 1 Implementation of an AllowList object using Atomic-Snapshot objects.

```
Shared variables
  AS-LV ← $N$-dimensions Atomic-Snapshot object, initially $\{\emptyset\}^N$;
  AS-PROOF ← $N$-dimensions Atomic-Snapshot object, initially $\{\emptyset\}^N$;
Operation APPEND($v$) is
  If $(v \in \mathcal{S}) \wedge (p \in \Pi_M)$ then
    local-values ← AS-LV.Snapshot()[$p$];
    AS-LV.Update(local-values $\cup$ $v$, $p$);
    Return true;
  Else return false;
Operation READ() is
  Return AS-PROOF.Snapshot();
Operation PROVE($v$) is
  If $p \notin \Pi_V$ then
    Return false;
  $\mathcal{A}$ ← AS-LV.Snapshot();
  If $v \in \mathcal{A}$ then
    $\pi_{\text{set-memb}}$ ← Proof($v \in \mathcal{A}$);
    proofs ← AS-PROOF.Snapshot()[$p$];
    AS-PROOF.Update(proofs $\cup$ $(p, \mathcal{A}, \pi_{\text{set-memb}})$, $p$);
    Return $(\mathcal{A}, \pi_{\text{set-memb}})$;
  Else return false.
```

▶ **Theorem 4.** *Algorithm 1 wait-free implements an AllowList object.*

Proof. The complete proof of this theorem is given in the full version of this paper [19]. ◀

▶ **Corollary 5.** *The consensus number of the AllowList object type is* 1.

6 The consensus number of the DenyList object

In the following, we propose two wait-free implementations establishing the consensus number of the DenyList object type. In this section and in the following, we refer to a DenyList with $|\Pi_V| = k$ as a k-DenyList object. This analysis of this parameter k is the core of the study conducted here. Because it is a statically defined parameter, the knowledge of this parameter can improve efficiency of DenyList implementation by reducing the number of processes that need to synchronize in order to conduct a proof.

6.1 Lower bound

Algorithm 2 presents an implementation of a k-consensus object using a k-DenyList object with $\Pi_M = \Pi_V = \Pi$, and $|\Pi| = k$. It uses an Atomic Snapshot object, AS-LIST, to allow processes to propose values. AS-LIST serves as a helping mechanism [12]. In addition, the algorithm uses the progress and the anti-flickering properties of the PROVE operation of the k-DenyList to enforce the k-consensus agreement property. The PROPOSE operation operates as follows. First, a process p tries to prove that the element 0 is not revoked by invoking PROVE(0). Then, if the previous operation succeeds, p revokes the element 0 by invoking APPEND(0). Then, p waits for the APPEND to be effective. This verification is done by invoking multiple PROVE operations until one is invalid. This behavior is ensured by the progress property of the k-DenyList object. Once the progress has occurred, p is sure that no other process will be able to invoke a valid PROVE(0) operation. Hence, p is sure that the set returned by the READ operation can no longer grow. Indeed, the READ operation returns the set of valid PROVE operation that occurred prior to its invocation. If no valid PROVE(0) operation can be invoked, the set returned by the READ operation is fixed (with regard to the element 0). Furthermore, all the processes in Π share the same view of this set.

Finally, p invokes READ() to obtain the set of processes that invoked a valid PROVE(0) operation. The response to the READ operation will include all the processes that invoked a valid PROVE operation, and this set will be the same for all the processes in Π that invoke the PROPOSE operation. Therefore, up to line 7, the algorithm solved the set-consensus problem. To solve consensus, we use an additional deterministic function $f_i : \Pi^i \rightarrow \Pi$, which takes as input any set of size i and outputs a single value from this set.

To simplify the representation of the algorithm, we also use the separator() function, which, on input of a set of proofs $(\{(p \in \Pi, \{\widehat{\mathcal{S}} \subseteq \mathcal{S}, \mathsf{P} \in \mathcal{P}_{\mathcal{L}_\mathcal{S}})\})$, outputs *processes*, the set of processes which conducted the proofs, i.e. the first component of each tuple.

Algorithm 2 k-consensus implementation using one k-DenyList object and one Atomic Snapshot.

```
Shared variables
  $k$-dlist ← $k$-DenyList object;
  AS-LIST ← Atomic Snapshot object, initially $\{\emptyset\}^k$
Operation PROPOSE($v$) is
1:  AS-LIST.update($v, p$);
2:  $k$-dlist.PROVE(0);
3:  $k$-dlist.APPEND(0);
4:  Do
5:    ret ← $k$-dlist.PROVE(0);
6:  Until (ret ≠ false);
7:  processes ← separator($k$-dlist.READ());
8:  Return AS-LIST.Snapshot()[$f_{|processes|}(processes)$].
```

▶ **Theorem 6.** *Algorithm 2 wait-free implements a k-consensus object.*

Proof. Let us fix an execution E of the algorithm presented in Algorithm 2. The progress property of the k-DenyList object ensures that the while loop in line 4 consists of a finite number of iterations – an APPEND(0) is invoked prior to the loop, hence, the PROVE(0) operation will eventually be invalid. Each invocation of the PROPOSE operation is a sequence of a finite number of local operations, Atomic Snapshot object accesses and k-DenyList object accesses which are assumed atomic. Therefore, each process terminates the PROPOSE operation in a finite number of its own steps. Let H be the history of E. We define $\bar{H}$ the completed history of H, where an invocation of PROPOSE which did not reach line 8 is completed with a line "return false". Line 8 is the linearization point of the algorithm. For convenience, any PROPOSE invocation that returns false is called an failed invocation. Otherwise, it is called a successful invocation.

We now prove that all operations in $\bar{H}$ follow the k-consensus specification:

- The process p that invoked a failed PROPOSE operation in $\bar{H}$ is faulty – by definition, the process prematurely stopped before line 8. Therefore, the fact that p cannot decide does not impact the termination nor the agreement properties of the k-consensus object.
- A successful PROPOSE operation returns AS-LIST.Snapshot()[$f_{|processes|}(processes)$]. Furthermore, a process proposed this value in line 1. All the processes that invoke PROPOSE conduct an APPEND(0) operation, and wait for this operation to be effective using the while loop at line 4 to 6. Thanks to the anti-flickering property of the k-DenyList object, when the APPEND operation is effective for one process – i.e. the Progress happens, in other words,a PROVE(0) operation is invalid – , then it is effective for any other process that would invoke the PROVE(0) operation. Hence, thanks to the

anti-flickering property, when a process obtains an invalid response from the PROPOSE(0) operation at line 5, it knows that no other process can invoke a valid PROVE(0) operation. This implies that the READ operation conducted at line 7 will return a fix set of processes, and all the processes that reach this line will see the same set. Furthermore, because each process invokes a PROPOSE(0) before the APPEND(0) at line 3, at least one valid PROPOSE(0) operation was invoked. Therefore, the *processes* set is not empty. Because each process ends up with the same set *processes*, and thanks to the determinism of the function f_i, all correct processes output the same value v (Agreement property and non-trivial value). The value v comes from the Atomic Snapshot object, composed of values proposed by authorized processes (Validity property). Hence a successful PROPOSE operation follows the k-consensus object specification.

All operations in $\bar{H}$ follow the k-consensus specification. To conclude, the algorithm presented in Algorithm 2 is a wait-free implementation of the k-consensus object type. ◀

▶ **Corollary 7.** *The consensus number of the k-DenyList object type is at least k.*

6.2 Upper bound

This section provides a DenyList object specification based on the PROOF-LIST object. The specification is then used to analyze the upper bound on the consensus number of the object type.

We provide an instantiation of the DenyList object defined as a PROOF-LIST object, where $\forall y \in \mathcal{S}, y \in \mathcal{L}_{\mathcal{A}} \Leftrightarrow (\mathcal{A} \subseteq \mathcal{S} \wedge y \notin \mathcal{A})$ and where:

$$\mathsf{C}(y, \widehat{\mathcal{S}}) = \begin{cases} 1, & \text{if } \forall \mathcal{A}' \in \widehat{\mathcal{S}}, y \notin \mathcal{A}' \\ 0, & \text{otherwise.} \end{cases}$$

In other words, the first equation ensures that y does not belong to a set $\mathcal{A}$, while the second equation ensures that the object fulfills the anti-flickering property. Hence, this instantiation supports proofs of set-non-membership in *listed-values*. A PROOF-LIST object defined for such language follows the specification of the DenyList. To support this statement, we provide an implementation of the object.

To build a k-DenyList object which can fulfill the anonymity property, it is required to build an efficient helping mechanism that preserves anonymity. It is impossible to disclose directly the value proven without disclosing the user's identity. Therefore, we assume that a process p that invokes the PROVE(v) operation can deterministically build a cryptographic commitment to the value v. Let C_v be the commitment to the value v. Then, any process $p' \neq p$ that invokes PROVE(v) can infer that C_v was built using the value v. However, a process that does not invoke PROVE(v) cannot discover to which value C_v is linked. If the targeted application does not require the user's anonymity, it is possible to use the plaintext v as the helping value.

Algorithm 3 presents an implementation of a k-DenyList object using k-consensus objects and Atomic Snapshots. The APPEND and the READ operations are analogous to those of Algorithm 1.

On the other hand, the PROVE operation must implement the anti-flickering property. To this end, a set of k-consensus objects and a helping mechanism based on commitments are used.

When a process invokes the PROVE(v) operation, it publishes C_v, the cryptographic commitment to v, using an atomic snapshot object. This commitment is published along with a timestamp [28] defined as follow. A local timestamp (p, c) is constituted of a process identifier p and a local counter value c. The counter c is always incremented before being reused. Therefore, each timestamp is unique. Furthermore, we build the strict total order relation $\mathcal{R}$ such that $(p, c)\mathcal{R}(p', c') \Leftrightarrow (c < c') \vee ((c = c') \wedge (p < p'))$. The timestamp is used in coordination with the helping value C_v to ensure termination. A process p that invokes the PROVE(v) operation must parse all the values proposed by the other processes. If a PROVE(v') operation was invoked by a process p' earlier than the one invoked by p – under the relation $\mathcal{R}$ – , then p must affect a set "val" for the PROVE operation of p' via the consensus object. The set "val" is obtained by reading the AS-LV object. The AS-LV object is append-only – no operation removes elements from the object. Furthermore, the sets "val" are attributed via the consensus object. Therefore, this mechanism ensures that the sets on which the PROVE operations are applied always grow.

Furthermore, processes sequentially parse the CONS-ARR using the counter$_p$ variable. This behavior, in collaboration with the properties of the consensus, ensures that all the process see the same tuples (winner, val) in the same order.

Finally, if a process p observes that a PROVE operation conducted by a process $p' \neq p$ is associated to a commitment C_v equivalent to the one proposed by p, then p produces the proof of set-non-membership relative to v and the set "val" affected to p' in its name. We consider that a valid PROVE operation is linearized when this proof of set-non-membership is added to AS-PROOF in line 19. Hence, when p produces its own proof – or if another process produces the proof in its name – it is sure that all the PROVE operations that are relative to v and that have a lower index in CONS-ARR compared to its own are already published in the AS-PROOF Atomic Snapshot object. Therefore, the anti-flickering property is ensured. Indeed, because the affected sets "val" are always growing and because of the total order induced by the CONS-ARR array, if p reaches line 25, it previously added a proof to AS-PROOF in the name of each process $p' \neq p$ that invoked a PROVE(v) operation and that was attributed a set at a lower index than p in CONS-ARR. Hence, the operation of p' was linearized prior to the operation of p.

A PROVE operation can always be identified by its published timestamp. Furthermore, when a proof is added to the AS-PROOF object, it is always added to the index counter$_{p_w}$. Therefore, if multiple processes execute line 19 for the PROVE operation labeled counter$_{p_w}$, the AS-PROOF object will only register a unique value.

Furthermore, we use a function "Proof" that on input of a set $\mathcal{S}$ and an element x outputs a proof that $x \notin \mathcal{S}$. This function is used as a black box, and can either output an explicit proof – an explicit proof can be the tuple $(x, \mathcal{S})$ – , or a Zero Knowledge Proof.

▶ **Theorem 8.** *Algorithm 3 wait-free implements a k-DenyList object.*

Proof. The proof of this theorem is given in the full version of this paper [19]. ◀

The following corollary follows from Theorem 6 and Theorem 8.

▶ **Corollary 9.** *The k-DenyList object type has consensus number k.*

7 Discussion

This section presents several applications where the AllowList and the k-DenyList can be used to determine the consensus numbers of more elaborate objects. More importantly, the analysis of the consensus number of these use cases makes it possible to determine if actual implementations achieve optimal efficiency in terms of synchronization. If not, we use the knowledge of the consensus number of the AllowList and DenyList objects to give intuitions

Algorithm 3 k-DenyList implementation using k-consensus objects and Atomic Snapshot objects.

```
Shared variables
  AS-LV ← $N$-dimensions Atomic-Snapshot object, initially $\{\emptyset\}^N$;
  AS-Queue ← $N$-dimensions Atomic-Snapshot object, initially $\{\emptyset\}^N$;
  CONS-ARR$_p$ ← an array of $k$-consensus objects of size $l > 0$;
  AS-PROOF ← $l$-dimensions Atomic-Snapshot object, initially $\{\emptyset\}^l$;
Local variables
  For each $p \in \Pi_V$ :
    evaluated$_p$ ← an array of size $l > 0$, initially $\{\emptyset\}^l$;
    counter$_p$ ← a positive integer, initially 0;
Operation APPEND($v$) is
  If $(v \in \mathcal{S}) \wedge (p \in \Pi_M)$ then
    local-values ← AS-LV.Snapshot()[$p$];
    AS-LV.UPDATE(local-values $\cup$ $v$, $p$);
    Return true;
  Else return false;
Operation PROVE($v$) is
  If $p \notin \Pi_V$ then
    Return false;
  $C_v$ ← Commitment($v$);
  cnt ← counter$_p$;
  AS-Queue.UPDATE(((cnt, $p$), $C_v$), $p$);
  queue ← AS-Queue.Snapshot() \ evaluated$_p$;
  While (cnt, $p$) $\in$ queue do
    oldest ← the smallest clock value in queue under $\mathcal{R}$;
    prop ← (oldest, AS-LV.snapshot());
    (winner, val) ← CONS-ARR[counter$_p$].propose(prop);
    ((counter$_{p_w}$, $p_w$), $C^*$) ← winner;
    If $C^* = C_v \wedge v \notin$ val then
      $\pi_{SNM}$ ← Proof($v \notin val$);
      AS-PROOF.Update(($p_w$, val, $\pi_{SNM}$, winner), counter$_{p_w}$);
    evaluated$_p$ ← evaluated$_p$ $\cup$ winner;
    queue ← queue \ winner;
    counter$_p$ ← counter$_p$ + 1;
  If $v \notin$ val then
    Return (val, $\pi_{\mathrm{SNM}}$);
  Else return false;
Operation READ() is
  Return AS-PROOF.Snapshot();
```

on how to build more practical implementations. More precisely, the fact that the consensus numbers of AllowList and DenyList objects are (in most cases) smaller than n implies that most implementations can reduce the number of processes that need to synchronize in order to implement such distributed objects. The liveness of many consensus algorithms is only ensured when the network reaches a synchronous period. Therefore, reducing the number of processes that need to synchronize can increase the system's probability of reaching such synchronous periods. Thus, it can increase the effectiveness of such algorithms.

7.1 Revocation of a verifiable credential

We begin by analyzing Sovrin's Verifiable-Credential revocation method using the DenyList object [18]. Sovrin is a privacy-preserving Distributed Identity Management System (DIMS). In this system, users own credentials issued by entities called issuers. A user can employ one such credential to prove to a verifier they have certain characteristics. An issuer may want to revoke a user's credential prematurely. To do so, the issuer maintains an append-only list of revoked credentials. When a user wants to prove that their credential is valid, they must provide to the verifier a valid ZKP of set-non-membership proving that their credential is not revoked, i.e. not in the DenyList. In this application, the set of managers Π_M consists solely of the credential's issuer. Hence, the proof concerns solely the verifier and the user. The way Sovrin implements this verification interaction is by creating an ad-hoc peer-to-peer consensus instance between the user and the verifier for each interaction. Even if the resulting DenyList has consensus number 2, Sovrin implements the APPEND operation using an SWMR stored on a blockchain-backed ledger (which requires synchronizing the N processes of the system). Our results suggest instead that Sovrin's revocation mechanism could be implemented without a blockchain by only using pairwise consensus.

7.2 The Anonymous Asset Transfer object

The anonymous asset transfer object is another application of the DenyList and the AllowList objects. As described in Appendix B, it is possible to use these objects to implement the asset transfer object described in [22]. Our work generalizes the result by Guerroui et al. [22]. Guerraoui et al. show that a joint account has consensus number k where k is the number of agents that can withdraw from the account. We can easily prove this result by observing that withdrawing from a joint account requires a denylist to record the already spent coins.

Nevertheless, our ZKP capable construction makes it possible to show that an asset transfer object where the user is anonymous, and its transactions are unlinkable also has consensus number k, where k is the number of processes among which the user is anonymous. The two main implementations of Anonymous Asset Transfer, ZeroCash and Monero [35, 7], use a blockchain as their main double spending prevention mechanism. While the former provides anonymity on the whole network, the second only provides anonymity among a subset of the processes involved in the system. Hence, this second implementation could reduce its synchronization requirements accordingly.

7.3 Distributed e-vote systems

Finally, another direct application of the DenyList object is the blind-signature-based e-vote system with consensus number k, k being the number of voting servers, which we present in the full version of this paper [19]. Most distributed implementations of such systems also use blockchains, whereas only a subset of the processes involved actually require synchronization.

8 Related Works

Bitcoin and blockchain. Even though distributed consensus algorithms were already largely studied [10, 29, 11, 4, 9], the rise of Ethereum – and the possibilities offered by its versatile smart contracts – led to new ideas to decentralized already known applications. Among those, e-vote and DIMS [18] are two examples.

Blockchains increased the interest in distributed versions of already existing algorithms. However, these systems are usually developed with little concern for the underlying theoretical basis they rely on. A great example lies in trustless money transfer algorithms or crypto money. The underlying distributed asset-transfer object was never studied until recently. A theoretical study proved that a secure asset-transfer algorithm does not need synchrony between network nodes [22]. Prior to this work, all proposed schemes used a consensus algorithm, which cannot be deterministically implemented in an asynchronous network [17]. The result is that many existing algorithms could be replaced by more efficient, Reliable Broadcast [9] based algorithms. This work leads to more efficient implementation proposal for money transfer algorithm [5]. Alpos et al. then extended this study to the Ethereum ERC20 smart contracts [3]. This last paper focuses on the asset-transfer capability of smart contracts. Furthermore, the object described has a dynamic consensus number, which depends on the processes authorized to transfer money from a given account. Furthermore, this work and the one from Guerraoui et al. [22] both analyze a specific object that is not meant to be used to find the consensus number of other applications. In contrast, our work aims to be used as a generic tool to find the consensus number of numerous systems.

E-vote. An excellent example of the usage of DenyList is to implement blind signatures-based e-vote systems [13]. A blind signature is a digital signature where the issuer can sign a message without knowing its content. Some issuer signs a cryptographic commitment – a cryptographic scheme where Alice hides a value while being bound to it [33] – to a message produced by a user. Hence, the issuer does not know the actual message signed. The user can then un-commit the message and present the signature on the plain-text message to a verifier. The verifier then adds this message to a DenyList. A signature present in the DenyList is no longer valid. Such signatures are used in some e-vote systems [20, 32]. In this case, the blind signature enables anonymity during the voting operation. This is the e-vote mechanism that we study in this article. They can be implemented using a DenyList to restrain a user from voting multiple times. This method is explored in the full version of this paper [19].

There exists two other way to provide anonymity to the user of an e-vote system. The first one is to use a MixNet [26, 25, 14]. MixNet is used here to break the correlation between a voter and his vote. Finally, anonymity can be granted by using homomorphic encryption techniques [6, 15].

Each technique has its own advantages and disadvantages, depending on the properties of the specific the e-vote system. We choose to analyze the blind signature-based e-vote system because it is a direct application of the distributed DenyList object we formalize in this paper.

Anonymous Money Transfer. Blockchains were first implemented to enable trustless money transfer algorithms. One of the significant drawbacks of this type of algorithm is that it only provides pseudonymity to the user. As a result, transfer and account balances can be inspected by anyone, thus revealing sensitive information about the user. Later developments proposed hiding the user's identity while preventing fraud. The principal guarantees are double-spending prevention – i.e., a coin cannot be transferred twice by the same user – and *ex nihilo* creation prevention – i.e., a user cannot create money. Zcash [7] and Monero [35] are the best representative of anonymous money transfer algorithms. The first one uses an AllowList to avoid asset creation and a DenyList to forbid double spending, while the second one uses ring signatures. We show in Appendix B that the DenyList and AllowList objects can implement an Anonymous Money Transfer object, and thus, define the synchronization requirements of the processes of the system.

9 Conclusion

This paper presented the first formal definition of distributed AllowList and DenyList object types. These definitions made it possible to analyze their consensus number. This analysis concludes that no consensus is required to implement an AllowList object. On the other hand, with a DenyList object, all the processes that can propose a set-non-membership proof must synchronize, which makes the implementation of a DenyList more resource intensive.

The definition of AllowList and DenyList as distributed objects made it possible to thoroughly study other distributed objects that can use AllowList and DenyList as building blocks. For example, we discussed authorization lists and revocation lists in the context of the Sovrin DIMS. We also provided several additional examples in the Appendix. In particular, we show in Appendix B that an association of DenyList and AllowList objects can implement an anonymous asset transfer algorithm and that this implementation is optimal in terms of synchronization power. This result can also be generalized to any asset transfer algorithm, where the processes act as proxies for the wallet owners. In this case, synchronization is only required between the processes that can potentially transfer money on behalf of a given wallet owner.

References

1 Ethereum name service documentation. online - https://docs.ens.domains/ - accessed 23/11/2022.

2 Yehuda Afek, Hagit Attiya, Danny Dolev, Eli Gafni, Michael Merritt, and Nir Shavit. Atomic snapshots of shared memory. *JACM*, 40(4):873–890, September 1993. doi:10.1145/153724.153741.

3 Orestis Alpos, Christian Cachin, Giorgia Azzurra Marson, and Luca Zanolini. On the synchronization power of token smart contracts. In *41st IEEE ICDCS*, pages 640–651, 2021. doi:10.1109/ICDCS51616.2021.00067.

4 Pierre-Louis Aublin, Sonia Ben Mokhtar, and Vivien Quéma. Rbft: Redundant byzantine fault tolerance. In *IEEE 33rd International Conference on Distributed Computing Systems*, pages 297–306, 2013. doi:10.1109/ICDCS.2013.53.

5 Alex Auvolat, Davide Frey, Michel Raynal, and François Taïani. Money Transfer Made Simple: a Specification, a Generic Algorithm, and its Proof. *Bulletin European Association for Theoretical Computer Science*, 132, October 2020.

6 Olivier Baudron, Pierre-Alain Fouque, David Pointcheval, Jacques Stern, and Guillaume Poupard. Practical multi-candidate election system. In *PODC*, pages 274–283, 2001. doi:10.1145/383962.384044.

7 Eli Ben Sasson, Alessandro Chiesa, Christina Garman, Matthew Green, Ian Miers, Eran Tromer, and Madars Virza. Zerocash: Decentralized anonymous payments from bitcoin. In *2014 IEEE Symposium on Security and Privacy*, pages 459–474, May 2014. doi:10.1109/SP.2014.36.

8 Daniel Benarroch, Matteo Campanelli, Dario Fiore, Kobi Gurkan, and Dimitris Kolonelos. Zero-knowledge proofs for set membership: Efficient, succinct, modular. In *Financial Cryptography and Data Security*. Springer Berlin Heidelberg, 2021.

9 Gabriel Bracha. Asynchronous byzantine agreement protocols. *Information and Computation*, 75(2):130–143, 1987. doi:10.1016/0890-5401(87)90054-X.

10 Miguel Castro and Barbara Liskov. Practical byzantine fault tolerance. In *OSDI '99*, pages 173–186, 1999.

11 Miguel Castro and Barbara Liskov. Proactive recovery in a Byzantine-Fault-Tolerant system. In *OSDI 2000*, October 2000. URL: https://www.usenix.org/conference/osdi-2000/proactive-recovery-byzantine-fault-tolerant-system.

12 Keren Censor-Hillel, Erez Petrank, and Shahar Timnat. Help! In *PODC '15*, pages 241–250, 2015. doi:10.1145/2767386.2767415.

13 David Chaum. Blind signatures for untraceable payments. In *Advances in Cryptology*, pages 199–203, 1983.

14 Michael R. Clarkson, Stephen Chong, and Andrew C. Myers. Civitas: Toward a secure voting system. *IEEE SSP*, pages 354–368, 2008.

15 Ronald Cramer, Rosario Gennaro, and Berry Schoenmakers. A secure and optimally efficient multi-authority election scheme. In *EUROCRYPT '97*, pages 103–118, 1997.

16 Gaby G. Dagher, Praneeth Babu Marella, Matea Milojkovic, and Jordan Mohler. Broncovote: Secure voting system using ethereum's blockchain. In *ICISSP*, 2018.

17 Michael J. Fischer, Nancy A. Lynch, and Michael S. Paterson. Impossibility of distributed consensus with one faulty process. *J. ACM*, 32(2):374–382, April 1985. doi:10.1145/3149.214121.

18 Sovrin Foundation. Sovrin: A protocol and token for self-sovereign identity and decentralized trust. Technical report, Sovrin Foundation, 2018.

19 Davide Frey, Mathieu Gestin, and Michel Raynal. The synchronization power (consensus number) of access-control objects: The case of allowlist and denylist, 2023. doi:10.48550/arXiv.2302.06344.

20 Atsushi Fujioka, Tatsuaki Okamoto, and Kazuo Ohta. A practical secret voting scheme for large scale elections. In *AUSCRYPT '92*, pages 244–251, 1993.

21 Jyoti Grover. Security of vehicular ad hoc networks using blockchain: A comprehensive review. *Vehicular Communications*, 34:100458, 2022. doi:10.1016/j.vehcom.2022.100458.

22 Rachid Guerraoui, Petr Kuznetsov, Matteo Monti, Matej Pavlovič, and Dragos-Adrian Seredinschi. The consensus number of a cryptocurrency. In *PODC '19*, pages 307–316, 2019. doi:10.1145/3293611.3331589.

23 Maurice Herlihy. Wait-free synchronization. *ACM Trans. Program. Lang. Syst.*, 13(1):124–149, January 1991. doi:10.1145/114005.102808.

24 Maurice P Herlihy and Jeannette M Wing. Linearizability: A correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems*, 12(3):463–492, 1990.

25 Markus Jakobsson, Ari Juels, and Ronald L. Rivest. Making mix nets robust for electronic voting by randomized partial checking. In *11th USENIX Security Symposium*, August 2002. URL: https://www.usenix.org/conference/11th-usenix-security-symposium/making-mix-nets-robust-electronic-voting-randomized.

26 Ari Juels, Dario Catalano, and Markus Jakobsson. Coercion-resistant electronic elections. In *WPES*, pages 61–70, 2005. doi:10.1145/1102199.1102213.

27 Harry A. Kalodner, Miles Carlsten, Paul Ellenbogen, Joseph Bonneau, and Arvind Narayanan. An empirical study of namecoin and lessons for decentralized namespace design. In *Workshop on the Economics of Information Security*, 2015.

28 Leslie Lamport. Time, clocks and the ordering of events in a distributed system. *Communications of the ACM 21, (7), 558-565*, July 1978. URL: https://www.microsoft.com/en-us/research/publication/time-clocks-ordering-events-distributed-system/.

29 Leslie Lamport. The part-time parliament. In *ACM TOCS*, pages 133–169, 1998. doi:10.1145/279227.279229.

30 Ming K. Lim, Yan Li, Chao Wang, and Ming-Lang Tseng. A literature review of blockchain technology applications in supply chains: A comprehensive analysis of themes, methodologies and industries. *Computers and Industrial Engineering*, 154:107133, 2021. doi:10.1016/j.cie.2021.107133.

31 Nitin Naik and Paul Jenkins. uport open-source identity management system: An assessment of self-sovereign identity and user-centric data platform built on blockchain. In *IEEE International Symposium on Systems Engineering (ISSE)*, pages 1–7, 2020. doi:10.1109/ISSE49799.2020.9272223.

32 Miyako Ohkubo, Fumiaki Miura, Masayuki Abe, Atsushi Fujioka, and Tatsuaki Okamoto. An improvement on a practical secret voting scheme. In *Information Security*, pages 225–234, 1999.

33 Torben Pryds Pedersen. Non-interactive and information-theoretic secure verifiable secret sharing. In *Advances in Cryptology — CRYPTO '91*, pages 129–140, 1992.

34 Andreas Pfitzmann and Marit Hansen. Anonymity, unlinkability, undetectability, unobservability, pseudonymity, and identity management–a consolidated proposal for terminology. *Version v0*, 31, January 2007.

35 Nicolas van Saberhagen. Cryptonote v 2.0, October 2013.

A Variations on the *listed-values* array

In the previous sections, we assumed the *listed-values* array was append-only. Some use cases might need to use a different configuration for this array. In this section, we explore use cases where the *listed-values* array is no longer append-only.

Let us start by considering the simplest case in which processes can only remove the values they wrote themselves. This results in no conflicts between APPEND and REMOVE operations. The *listed-values* array can be seen as an array of $|\Pi_V|$ values. A process p_i can write the i-th index of the *listed-values* array. As only p_i can modify this value, there are no conflicts upon writing (append/remove). This allows us to easily add a REMOVE operation to an AllowList or DenyList object. In the case of the AllowList, this is particularly interesting because it effectively allows the AllowList to act as a DenyList. Let us assume the managers add all the elements of the universe of the possible identifiers to the AllowList in the first place. Then, this AllowList can implement a DenyList object, where the REMOVE operation of the AllowList is equivalent to the APPEND operation of the DenyList. The resulting AllowList with REMOVE needs an anti-flickering property to prevent concurrent PROVE operations from yielding conflicting results. This makes the AllowList with REMOVE equivalent to a DenyList object: its consensus number is k, where k is the number of processes in Π_V.

A more complex case arises when multiple processes can remove a written value. We associate each process p_i with a predefined authorization set $\mathcal{A}_i \subseteq \Pi_M$, defining which processes can APPEND or REMOVE on p_i's register. We always have $p_i \in \mathcal{A}_i$. If $p_j \in \mathcal{A}_i$, then p_j is allowed to "overwrite" (remove) anything p_i wrote. In this case, APPEND and REMOVE operation can conflict with each other and authorized processes need to synchronize when modifying the *listed-values* array. Specifically, let $k_{\text{AR}_i} = |\mathcal{A}_\rangle|$ be the number of processes that can modify the ith array position and let $k_{\text{AR}} = \mathsf{max}_i(k_{\text{AR}_i})$ be the largest value of k_{AR_i} over all the array positions. Then the consensus number of the APPEND and REMOVE operation is k_{AR_i}.

B Anonymous Asset-Transfer object type

Existing work by Guerraoui et al [22] and Auvolat [5] provides good insight into the problem of asset transfer, but it only studies pseudonymous systems, where all transactions can be linked to a single pseudonym. We now show how our formalization of AllowList and DenyList allows us to reason about anonymous and unlinkable asset transfer solutions [7, 35].

B.1 Problem formalization

The Asset-Transfer object type allows a set of processes to exchange assets via a distributed network. We reformulate the definition proposed by Guerraoui et al. [22]:

▶ **Definition 10.** The (pseudonymous) Asset-Transfer object type proposes two operations, TRANSFER and BALANCE. The object type is defined for a set Π of processes and a set $\mathcal{W}$ of accounts. An account is defined by the amount of assets it contains at time t. Each account is initially attributed an amount of assets equal to $v_0 \in \mathbb{Z}^{+*}$. We define a map $\mu : \mathcal{W} \to \{0,1\}^{|\Pi|}$ which associates each account to the processes that can invoke TRANSFER operations for these wallets. The Asset Transfer object type supports two operations, TRANSFER and BALANCE. When considering a TRANSFER(i,j,v) operation, $i \in \mathcal{W}$ is called the initiator, $j \in \mathcal{W}$ is called the recipient, and $v \in \mathbb{N}$ is called the amount transferred. Let $T(i,j)_t$ be the sum of all valid TRANSFER operations initiated by process i and received by process j before time t. These operations respect three properties:

- (Termination) TRANSFER and BALANCE operations always return if they are invoked by a correct process.
- (TRANSFER Validity) The validity of an operation TRANSFER(x,y,v) invoked at time t by a process p is defined in a recursive way. If no TRANSFER(x,i,v), $\forall i \in \mathcal{W}$ was invoked before time t, then the operation is valid if $v \le v_0$ and if $p \in \mu(x)$. Otherwise, the operation is valid if $v \le v_0 + \sum_{i\in\mathcal{W}} T(i,x)_t - \sum_{j\in\mathcal{W}} T(x,j)_t$ and if $p \in \mu(x)$.
- (BALANCE Validity) A BALANCE operation invoked at time t is valid if it returns $v_0 + \sum_{i\in\mathcal{W}} T(i,x)_t - \sum_{j\in\mathcal{W}} T(x,j)_t$ for each account x.

The Asset transfer object is believed to necessitate a double-spending-prevention property. This property is captured by the TRANSFER Validity property of Definition 10. Indeed, the double-spending-prevention property is defined to avoid ex-nihilo money creation. In a wait-free implementation, a valid transfer operation is atomic. Therefore, double spending is already prevented. A TRANSFER operation takes into account all previous transfers from the same account.

The paper by Guerraoui et al. [22] informs us that the consensus number of such an object depends on the map μ. If $\sum_{i\in\{0,\cdots,|\Pi|\}} \mu(w)[i] \leq 1, \forall\ w \in \mathcal{W}$, then the consensus number of the object type is 1. Otherwise, the consensus number is $\max_{w\in\mathcal{W}}(\sum_{i\in\{0,\cdots,|\Pi|\}} \mu(w)[i])$. In other words, the consensus number of such object type is the maximum number of different processes that can invoke a TRANSFER operation on behalf of a given wallet.

From continuous balances to token-based Asset-Transfer

The definition proposed by Guerraoui et al. uses a continuous representation of the balance of each account. Implementing anonymous money transfer with such a representation would require a mechanism to hide the transaction amounts [7]. As such a mechanism would not affect the synchronization properties of the AAT object, we simplify the problem by considering a token-based representation. A transfer in the tokenized version for a value of kV consists of k TRANSFER operations, each transferring a token of value V. The full version of the paper [19] provised a bijection that makes it possible to move to and from the continous and token-based representations.

Anonymity set

Let S be a set of actors. We define "anonymity" as the fact that, from the point of view of an observer, $o \notin S$, the action, v, of an actor, $a \in S$, cannot be distinguished from the action of any other actor, $a' \in S$. We call S the anonymity set of a for the action v [34].

Implementing Anonymous Asset Transfer requires hiding the association between a token and the account or process that owns it. If a "token owner" transfers tokens from the same account twice, these two transactions can be linked together and are no longer anonymous. Therefore, we assume that the "token owner" possesses offline proofs of ownership of tokens. These proofs are associated with shared online elements, allowing other processes to verify the validity of transactions. We call *wallet* the set of offline proofs owned by a specific user. We call the individual who owns this wallet the *wallet owner*. A wallet owner can own multiple wallets, while a wallet is owned by only one owner. Furthermore, we assume each process can invoke TRANSFER operations on behalf of multiple wallet owners. Otherwise, a single process, which is in most cases identified by its ip-address or its public key, would be associated with a single wallet and the system could not be anonymous. With the same reasoning, we can assume that a wallet owner can request many processes to invoke a TRANSFER operation on his or her behalf. Otherwise, the setup would not provide "network anonymity", but only "federated anonymity", where the wallet is anonymous among all other wallets connected to this same process. In our model, processes act as proxies.

The Anonymous Asset-Transfer object type

The first difference between a Pseudonymous Asset Transfer object type and an anonymous one is the absence of a BALANCE operation. The wallet owner can compute the balance of its own wallet using a LOCALBALANCE function that is not part of the distributed object. The TRANSFER operation is also slightly modified. Let us consider a sender that wants to transfer a token T_O to a recipient. The recipient creates a new token T_R with the associated cryptographic offline proofs (in practice, T_R can be created by the sender using the public key of the recipient). Specifically, it associates it with a private key. This private key is known only to the recipient: its knowledge represents, in fact, the possession of the token. Prior to the transfer operation, the recipient sends token T_R to the sender. The sender destroys token T_O and activates token T_R. The destruction prevents double spending, and the creation makes it possible to transfer the token to a new owner while hiding the recipient's identity. Furthermore, this process of destruction and creation makes it possible to unlink the usages of what is ultimately a unique token.

Each agent maintains a local wallet that contains the tokens (with the associated offline proofs) owned by the agent. The owner of a wallet w can invoke TRANSFER operations using any of the processes in $\mu(w)$. A transfer carried out from a process p for wallet w is associated with an anonymity set $\mathcal{AS}_p^w$ of size equal to the number of wallets associated with process p: $|\mathcal{AS}_p^w| = \sum_{i\in\mathcal{W}} \mu(i)[p]$. The setup with the maximal anonymity set for each transaction is an Anonymous Asset Transfer object where each wallet can perform a TRANSFER operation from any process: i.e., $\mu(i) = \{1\}^{|\Pi|}, \forall i \in \mathcal{W}$. The token-based Anonymous Asset Transfer object type is defined as follows:

▶ **Definition 11.** The Anonymous Asset Transfer object type supports only one operation: the TRANSFER operation. It is defined for a set Π of processes and a set $\mathcal{W}$ of wallets. An account is defined by the amount of tokens it controls at time t. Each account is initially attributed an amount v_0 of tokens. We define a map $\mu : \mathcal{W} \to \{0,1\}^{|\Pi|}$ which associates each wallet to the processes that can invoke TRANSFER on behalf of these wallets. When considering a TRANSFER(T_O, T_R) operation, T_0 is the cryptographic material of the initiator that proves the existence of a token T, and T_R is the cryptographic material produced by the recipient used to create a new token. The TRANSFER operation respects three properties:

- (Termination) The TRANSFER operation always returns if it is invoked by a correct process.
- (TRANSFER Validity) A TRANSFER(T_O, T_R) operation invoked at time t is valid if:
  - (Existence) The token T_O already existed before the transaction, i.e., either it is one of the tokens initially created, or it has been created during a valid TRANSFER(T'_O, T_O) operation invoked at time $t' < t$.
  - (Double spending prevention) No TRANSFER(T_O, T'_R) has been invoked at time $t'' < t$.
- (Anonymity) A TRANSFER(T_O, T_R) invoked by process p does not reveal information about the owner w and w' of T_O and T_R, except from the fact that w belongs to the anonymity set $\mathcal{AS}_p^w$.

The TRANSFER validity property implies that the wallet owner can provide existence and non-double-spending proofs to the network. It implies that any other owner in the same anonymity set and with the same cryptographic material (randomness and associated element) can require the transfer of the same token. We know the material required to produce a TRANSFER proof is stored in the wallet. Furthermore, we can assume that all the randomness used by a given wallet owner is produced by a randomness Oracle that derives a seed to obtain random numbers. Each seed is unique to each wallet. We assume the numbers output by an oracle seem random to an external observer, but two processes that share the same seed will obtain the same set of random numbers in the same order.

A transaction must be advertised to other processes and wallet owners via the TRANSFER operation. Therefore, proofs of transfer are public. We know these proofs are deterministically computed thanks to our deterministic random oracle model. Furthermore, only one sender and recipient are associated with each transfer operation. Therefore, the public proof cryptographically binds (without revealing them) the sender to the transaction. Hence, the public proof is a cryptographic commitment, which can be opened by the sender or any other actor who knows the same information as the sender.

In order to study the consensus number of this object, we consider that wallet owners can share their cryptographic material with the entire network, thereby giving up their anonymity. This would not make any sense in an anonymous system, but it represents a valuable tool to reason about the consensus number of the object. This sharing process can be implemented by an atomic register (and therefore has no impact on the consensus number).

Processes can derive the sender's identity from the shared information using a local "uncommit" function. The "uncommit" function takes as input an oracle, a random seed, token elements, and an "on-ledger" proof of transfer of a token and outputs a wallet owner ID if the elements are valid. Otherwise, it outputs $\emptyset$.

B.2 Consensus number of the Anonymous Asset-Transfer object type

Lower bound

Algorithm 4 presents an algorithm that implements a k-consensus object, using only k-Anonymous Asset Transfer objects and SWMR registers. The k in k-Anonymous Asset Transfer object refers here to the size of the biggest $\mu(w), \forall\ w \in \mathcal{W}$.

Algorithm 4 Implementation of a k-consensus object using k-Anon-AT objects.

```
Shared variables:
  AT ← $k$-Anonymous-AT object, initialized with $k+1$ wallets,
        each one of the $k$ first wallets possesses the elements
        necessary to transfer one shared token, the $k+1$-th
        wallet is the recipient of the transfers, it is not controlled
        by any process;
  RM-LEDGER ← Atomic Snapshot object, initially $\{\emptyset\}^k$;
  V-LED ← Atomic Snapshot object, initially $\{\emptyset\}^k$;
  O ← A random oracle;
  TokenMat ← secret associated with a unique token;
Local variables:
  seed ← random number;

Operation PROPOSE($v$) is:
  RM-LEDGER[p].update(seed, $p$);
  V-LED[p].update($v, p$);
  res ← AT.transfer(TokenMat, O, seed, $k+1$);
  RML ← RM-LEDGER.snapshot();
  VL ← V-LED.snapshot();
  For $i$ in $\{1, \cdots, k\}$ do:
    If uncommit(O, RML[$i$], TokenMat, res) $\neq \emptyset$ then:
      Return VL[$i$];
  Return False;
```

▶ **Theorem 12.** *Algorithm 4 wait-free implements k-consensus.*

Proof. The proof of Theorem 12 is given in the full version of this paper [19]. ◀

Upper Bound

We give an implementation of the Anon-AT object using only Atomic Snapshot objects, DenyList objects, and AllowList objects. Each wallet owner can request a TRANSFER operation to k different processes. The proposed implementation uses disposable tokens that are either created at the initialization of the system or during the transfer of a token. When a token is destroyed, a new token can be created, and the new owner of the token is the only one to know the cryptographic material associated with this new token. In the following, we use the zero-knowledge version of the DenyList and AllowList object types, where all set-(non-)membership proofs use a zero-knowledge setup. In addition, we use an AllowList object to ensure that a token exists (no ex-nihilo creation), and we use a DenyList object to ensure that the token is not already spent (double-spending protection).

The underlying cryptographic objects used are out of the scope of this paper. However, we assume our implementation uses the ZeroCash [7] cryptographic implementation, which is a sound anonymous asset transfer algorithm. More precisely, we will use a high-level definition of their off-chain functions. It is important to point out that using the ZeroCash implementation, it is possible to transfer value from a pseudonymous asset transfer object to an anonymous one using a special transaction called "Mint". To simplify our construction, we assume that each wallet is created with an initial amount of tokens v_0 and that our object does not allow cross-chain transfers. We, therefore, have no "Mint" operation.

ZeroCash uses a TRANSFER operation called *pour* that performs a transfer operation destroying and creating the associated cryptographic material. Here, we use a modified version of *pour* which does not perform the transfer or any non-local operation. It is a black-box local function that creates the cryptographic material required prove the destruction of the source token (T_O) and the creation of the destination one (T_R). Our modified *pour* function takes as input the source token, the private key of the sender (sk_s), and the public key of the recipient (pk_r): $pour(T_O, \mathsf{pk}_r, \mathsf{sk}_s) \rightarrow tx$, tx being the cryptographic material that makes it possible to destroy T_O and create T_R.

There might be multiple processes transferring tokens concurrently. Therefore, we define a deterministic local function ChooseLeader$(\mathcal{A}, tx)$, which takes as input any set $\mathcal{A}$ and a transaction tx, and outputs a single participant p which invoked BL.PROVE(tx).[7]

Algorithm 5 Anon-AT object implementation using SWMR registers, AllowList objects, and DenyList objects.

```
Shared variables:
  DL ← $k$-DenyList object, initially $(\emptyset, \emptyset)$;
  AL ← AllowList object, initially $(\{(token_{(i,j)})_{i=1}^{t}\}_{j=1}^{k}, \emptyset)$
Operation TRANSFER($T_O, \mathsf{pk}_r, \mathsf{sk}_s$) is:
  $tx$ ← Pour($T_O, \mathsf{pk}_r, \mathsf{sk}_s$)
  If verify($tx$) and $tx \in$ AL and $tx \notin$ DL then:
      AL.PROVE($tx$);
      DL.PROVE($tx$);
      DL.APPEND($tx$);
      Do:
        ret ← DL.PROVE(tx);
      While ret $\neq$ false;
      If ChooseLeader(DL.READ(), $tx.T_R$)=$p$ then:
        AL.append($tx.T_R$);
        Return $tx.T_R$;
  Return False;
```

▶ **Theorem 13.** *Algorithm 5 wait-free implements an Anon-AT object.*

Proof. The proof of Theorem 13 is given in the full version of this paper [19]. ◀

▶ **Corollary 14.** *The consensus number upper bound of a k-anon-AT object is k. Using this corollary and Theorem 12, we further deduct that k-anon-AT object has consensus number k.*

[7] In reality, the signature of chooseLeader would be more complicated as the function needs T_O, pk_r, sk_s in addition to tx. These additional elements make it possible to uncommit tx, thereby matching the values of the PROVE operation with $tx.T_R$. Note that this does not pose an anonymity threat as this is a local function invoked by the owner of sk_s. We omit these details to simplify the presentation.