



**HAL**  
open science

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

Davide Frey, Mathieu Gestin, Michel Raynal

► **To cite this version:**

Davide Frey, Mathieu Gestin, Michel Raynal. The Synchronization Power (Consensus Number) of Access-Control Objects: the Case of AllowList and DenyList. DISC 2023 - 37th International Symposium on Distributed Computing, Oct 2023, L'aquila, Italy. pp.1-32, 10.4230/LIPIcs.DISC.2023.21 . hal-04399298

**HAL Id: hal-04399298**

**<https://inria.hal.science/hal-04399298>**

Submitted on 17 Jan 2024

**HAL** is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.



Distributed under a Creative Commons Attribution 4.0 International License

# 1 The Synchronization Power (Consensus Number) 2 of Access-Control Objects: 3 The Case of AllowList and DenyList

4 Davide Frey ✉

5 Inria, IRISA, CNRS, Université de Rennes

6 Mathieu Gustin ✉

7 Inria, IRISA, CNRS, Université de Rennes

8 Michel Raynal ✉

9 IRISA, Inria, CNRS, Université de Rennes

## 10 — Abstract —

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

23 **2012 ACM Subject Classification** Theory of computation → Distributed computing models; Security  
24 and privacy → Access control; Security and privacy → Pseudonymity, anonymity and untraceability

25 **Keywords and phrases** Access control, AllowList/DenyList, Blockchain, Consensus number, Dis-  
26 tributed objects, Modularity, Privacy, Synchronization power.

27 **Digital Object Identifier** 10.4230/LIPIcs.DISC.2023.39

## 28 **1** Introduction

29 The advent of blockchain technologies increased the interest of the public and industry in  
30 distributed applications, giving birth to projects that have applied blockchains in a plethora  
31 of use cases. These include e-vote systems [1], naming services [2, 3], Identity Management  
32 Systems [4, 5], supply-chain management [6], or Vehicular Ad hoc Network [7]. However, this  
33 use of the blockchain as a swiss-army knife that can solve numerous distributed problems  
34 highlights a lack of understanding of the actual requirements of those problems. Because of  
35 these poor specifications, implementations of these applications are often sub-optimal.

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

38 Differently from the previous approaches, it aims to understand the amount of synchro-  
39 nization required between processes of a system to implement *specific* distributed objects. To  
40 achieve this goal it studies such objects under the lens of Herlihy’s consensus number [8]. This  
41 parameter is inherently associated to shared memory distributed objects, and has no direct  
42 correspondence in the message passing environment. However, in some specific cases, this  
43 information is enough to provide a better understanding of the objects analyzed, and thus,  
44 to gain efficiency in the message passing implementations. For example, recent papers [9, 10]



© Davide Frey, Mathieu Gustin, and Michel Raynal;  
licensed under Creative Commons License CC-BY 4.0  
37th International Symposium on Distributed Computing (DISC 2023).  
Editor: Rotem Oshman; Article No. 39; pp.39:1–39:32



Leibniz International Proceedings in Informatics  
LIPICs Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

45 have shown that cryptocurrencies can be implemented without consensus and therefore  
46 without a blockchain. In particular, Guerraoui et al. [9] show that  $k$ -asset transfer has a  
47 consensus number  $k$  where  $k$  is the number of processes that can withdraw currency from the  
48 same account [11]. Similarly, Alpos et al. [12] have studied the synchronization properties of  
49 ERC20 token smart contracts and shown that their consensus number varies over time as  
50 a result of changes in the set of processes that are approved to send tokens from the same  
51 account. These two results consider two forms of asset transfer: the classical one and the one  
52 implemented by the ERC20 token, which allows processes to dynamically authorize other  
53 processes. The consensus number of those objects depends on specific and well identified  
54 processes. From this study, it is possible to conclude that the consensus algorithms only need  
55 to be performed between those processes. Therefore, in these specific cases, the knowledge of  
56 the consensus number of an object can be directly used to implement more efficient message  
57 passing applications. Furthermore, even if this study uses a shared memory model, with  
58 crash prone processes, its results can be used to implement more efficient Byzantine resilient  
59 algorithm, in a message passing environment. This paper proposes to extend this knowledge  
60 to a broader class of applications.

61 Indeed, the transfer of assets, be them cryptocurrencies or non-fungible tokens, does not  
62 constitute the only application in the Blockchain ecosystem. In particular, as previously  
63 indicated, a number of applications like e-voting [1], naming [2, 3], or Identity Management [4,  
64 5] use Blockchain as a tool to implement some form of access control. This is often achieved  
65 by implementing two general-purpose objects: AllowLists and DenyLists. An AllowList  
66 provides an opt-in mechanism. A set of managers can maintain a list of authorized parties,  
67 namely the AllowList. To access a resource, a party (user) must prove the presence of an  
68 element associated with its identity in the AllowList. A DenyList provides instead an opt-out  
69 mechanism. In this case, the managers maintain a list of revoked elements, the DenyList. To  
70 access a resource, a party (user) must prove that no corresponding element has been added to  
71 the DenyList. In other words, AllowList and DenyList support, respectively, set-membership  
72 and set-non-membership proofs on a list of elements.

73 The proofs carried out by AllowList and DenyList objects often need to offer privacy  
74 guarantees. For example, the Sovrin privacy preserving Decentralized Identity-Management  
75 System (DIMS) [4] associates an AllowList<sup>1</sup> with each verifiable credential that contains  
76 the identifiers of the devices that can use this verifiable credential. When a device uses a  
77 credential with a verifier, it needs to prove that the identifier associated with it belongs to  
78 the AllowList. This proof must be done in zero knowledge, otherwise the verifier would learn  
79 the identity of the device, which in turn could serve as a pseudo-identifier for the user. For  
80 this reason, AllowList and DenyList objects support respectively a zero-knowledge proof of  
81 set membership or a zero-knowledge proof of set non-membership.

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

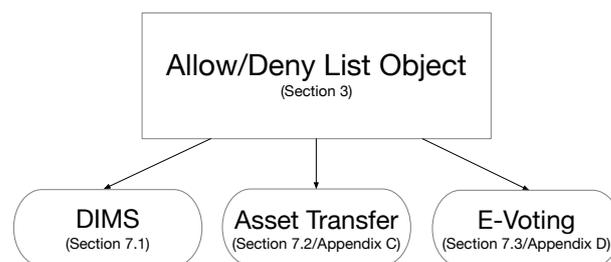
---

<sup>1</sup> In reality this is a variant that mixes AllowList and DenyList which we discuss in appendix A.

Existing systems [1, 2, 3, 4, 5] 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 [11]. 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.

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 (Section 5), while the DenyList requires the synchronization of all the verifiers of its set-non-membership proofs (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 protocol (Appendix B and C).



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  $\Pi$  be a set of  $N$  asynchronous sequential crash-prone processes  $p_1, \dots, p_N$ . 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.

125 **Communication**

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

132 **Histories and Linearizability**

133 A *history* [8] is a sequence of invocations and responses in the execution of an algorithm.  
 134 An invocation with no matching response in a history,  $H$ , is called a *pending* invocation. A  
 135 *sequential history* is one where the first event is an invocation, and each invocation—except  
 136 possibly the last one—is immediately followed by the associated response. A sub-history  
 137 is a sub-sequence of events in a history. A process sub-history  $H|p_i$  of a history  $H$  is a  
 138 sub-sequence of all the events in  $H$  whose associated process is  $p_i$ . Given an object  $x$ , we  
 139 can similarly define the object sub-history  $H|x$ . Two histories  $H$  and  $H'$  are equivalent if  
 140  $H|p_i = H'|p_i, \forall i \in \{1, \dots, N\}$ .

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

151 **Consensus number**

152 The consensus number of an object of type  $T$  (noted  $\text{cons}(T)$ ) is the largest  $n$  such that it  
 153 is possible to wait-free implement a consensus object from atomic read/write registers and  
 154 objects of type  $T$  in a system of  $n$  processes. If an object of type  $T$  makes it possible to  
 155 wait-free implement a consensus object in a system of any number of processes, we say the  
 156 consensus number of this object is  $\infty$ . Herlihy [11] proved the following well-known theorem.

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

160 We will determine the consensus number of the DenyList and the AllowList objects using  
 161 Atomic Snapshot objects and consensus objects in a set of  $k$  processes. A Single Writer Multi  
 162 Reader (SWMR) [13] Atomic Snapshot object is an array of fixed size, which supports two  
 163 operations: Snapshot and Update. The Snapshot() operation allows a process  $p_i$  to read the  
 164 whole array in one atomic operation. The Update( $v, i$ ) operation allows a process  $p_i$  to write  
 165 the value  $v$  in the  $i$ -th position of the array. Afek et al. showed that a SWMR Snapshot  
 166 object can be wait-free implemented from read/write registers [13], i.e., this object type has  
 167 consensus number 1. This paper assumes that all Atomic Snapshot objects used are SWMR.  
 168 A consensus object provides processes with a single one-shot operation *propose()*. When a

169 process  $p_i$  invokes  $propose(v)$  it proposes  $v$ . This invocation returns a *decided* value such  
 170 that the following three properties are satisfied.

171 ■ *Validity*: If a correct process decides value  $v$ , then  $v$  was proposed by some process;

172 ■ *Agreement*: No two correct processes decide differently; and

173 ■ *Termination*: Every correct process eventually decides.

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

## 175 2.2 Number theory preliminaries

### 176 Cryptographic Commitments

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

### 185 Zero Knowledge Proof of set operations

186 A Zero Knowledge Proof (ZKP) system is a cryptographic protocol that allows a prover to  
 187 prove some Boolean statement about a value  $x$  to a verifier without leaking any information  
 188 about  $x$ . A ZKP system is initialized for a specific language  $\mathcal{L}$  of the complexity class  $\mathcal{NP}$ .  
 189 The proving mechanism takes as input  $\mathcal{L}$  and outputs a proof  $\pi$ . Knowing  $\mathcal{L}$  and  $\pi$ , any  
 190 verifier can verify that the prover knows a value  $x \in \mathcal{L}^2$ . However, the verifier cannot learn  
 191 the value  $x$  used to produce the proof. In the following, it is assumed there exists efficient  
 192 non interactive ZKP systems of set-membership and set-non-membership (e.g., constructions  
 193 from [14] can be used).

## 194 3 The AllowList and DenyList objects: Definition

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

200 The AllowList object type is one of the two most common access control mechanisms.  
 201 To access a resource, a process  $p \in \Pi_V$  needs to prove it knows some element  $v$  previously  
 202 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  
 203 set of processes authorized to conduct proofs. We call verifiers the processes in  $\Pi_V$ . The sets  
 204  $\Pi_V$  and  $\Pi_M$  are predefined and static. They are parameters of the object. Depending on the  
 205 usage of the object, these subset can either be small, or they can contain all the processes in  
 206  $\Pi$ .

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

207 A process  $p \in \Pi_V$  proves that  $v$  was previously authorized by invoking a  $\text{PROVE}(v)$   
 208 operation. This operation is said to be valid if some manager in  $\Pi_M$  previously invoked an  
 209  $\text{APPEND}(v)$  operation. Intuitively, we can see the invocation of the  $\text{APPEND}(v)$  operation  
 210 as the action of authorizing some process to access the resource. On the other hand, the  
 211  $\text{PROVE}(v)$  operation, performed by a prover process,  $p \in \Pi_V$ , proves to the other processes  
 212 in  $\Pi_V$  that they are authorized. However, this proof is not enough in itself. The verifiers  
 213 of a proof must be able to verify that a valid  $\text{PROVE}$  operation has been invoked. To this  
 214 end, the  $\text{AllowList}$  object type is also equipped with a  $\text{READ}()$  operation. This operation  
 215 can be invoked by any process in  $\Pi$  and returns all the valid  $\text{PROVE}$  operations invoked,  
 216 along with the identity of the processes that invoked them. The list returned by the  $\text{READ}$   
 217 operation can be any arbitrary permutation of the list of  $\text{PROVE}$  operations. All processes  
 218 in  $\Pi$  can invoke the  $\text{READ}$  operation.<sup>3</sup>

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

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

225 ► **Definition 2.** *The  $\text{AllowList}$  object type supports three operations:  $\text{APPEND}$ ,  $\text{PROVE}$ ,  
 226 and  $\text{READ}$ . These operations appear as if executed in a sequence  $\text{Seq}$  such that:*

227 ■ *Termination.* A  $\text{PROVE}$ , an  $\text{APPEND}$ , or a  $\text{READ}$  operation invoked by a correct  
 228 process always returns.

229 ■ **APPEND Validity.** The invocation of  $\text{APPEND}(x)$  by a process  $p$  is valid **if**:

- 230 -  $p \in \Pi_M \subseteq \Pi$ ; **and**
- 231 -  $x \in \mathcal{S}$ , where  $\mathcal{S}$  is a predefined set.

232 Otherwise, the operation is invalid.

233 ■ **PROVE Validity.** **If** the invocation of  $op = \text{PROVE}(x)$  by a process  $p$  is valid, **then**:

- 234 -  $p \in \Pi_V \subseteq \Pi$ ; **and**
- 235 - A valid  $\text{APPEND}(x)$  operation appears before  $op$  in  $\text{Seq}$ .

236 Otherwise, the invocation is invalid.

237 ■ **Progress.** **If** a valid  $\text{APPEND}(x)$  operation is invoked, **then** there exists a point in  $\text{Seq}$   
 238 such that any  $\text{PROVE}(x)$  operation invoked after this point by any process  $p \in \Pi_V$  will  
 239 be valid.

240 ■ **READ Validity.** The invocation of  $op = \text{READ}()$  by a process  $p \in \Pi_V$  returns the list of  
 241 valid invocations of  $\text{PROVE}$  that appears before  $op$  in  $\text{Seq}$  along with the names of the  
 242 processes that invoked each operation.

243 ■ **Optional - Anonymity.** Let us assume the process  $p$  invokes a  $\text{PROVE}(v)$  operation. If  
 244 the process  $p'$  invokes a  $\text{READ}()$  operation, then  $p'$  cannot learn the value  $v$  unless  $p$   
 245 leaks additional information.<sup>4</sup>

<sup>3</sup> Usually,  $\text{AllowList}$  objects are implemented in a message-passing setting. In these cases, the  $\text{READ}$  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  $\text{READ}$  operation explicit.

<sup>4</sup> 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 examples are discussed in Section 7. Thereby, the anonymity property does not contravene the  $\text{READ}$  validity property, which only discloses the process identity.

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

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

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

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

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

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

273 ■ *Termination.* A PROVE, an APPEND, or a READ operation invoked by a correct  
 274 process always returns.

275 ■ **APPEND Validity.** The invocation of APPEND( $x$ ) by a process  $p$  is valid **if**:

- 276 -  $p \in \Pi_M \subseteq \Pi$ ; **and**
- 277 -  $x \in \mathcal{S}$ , where  $\mathcal{S}$  is a predefined set.

278 Otherwise, the operation is invalid.

279 ■ **PROVE Validity.** **If** the invocation of a  $op = \text{PROVE}(x)$  by a correct process  $p$  is not  
 280 valid, **then**:

- 281 -  $p \notin \Pi_V \subseteq \Pi$ ; **or**
- 282 - A valid APPEND( $x$ ) appears before  $op_P$  in Seq.

283 Otherwise, the operation is valid.

284 ■ **PROVE Anti-Flickering.** **If** the invocation of a operation  $op = \text{PROVE}(x)$  by a correct  
 285 process  $p \in \Pi_V$  is invalid, **then** any PROVE( $x$ ) operation that appears after  $op$  in Seq is  
 286 invalid.<sup>5</sup>

<sup>5</sup> 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.

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

■ **Table 1** Transition function  $\Delta$  for the PROOF-LIST object.

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

#### 293 4 PROOF-LIST object specification

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

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

310 The sets  $\Pi_V$  and  $\Pi_M$  are static, predefined subsets of  $\Pi$ . There is no restriction on their  
 311 compositions. The choice of these sets only depends on the usage of the AllowList or the  
 312 DenyList. Depending on the usage, they can either contain a small subset of processes in  $\Pi$   
 313 or they can contain the whole set of processes of the system.

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

320  $\mathcal{NP}$ <sup>6</sup> 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  
 321 of valid proofs relative to the language  $\mathcal{L}_{\mathcal{A}}$ .  $\mathcal{P}_{\mathcal{L}_{\mathcal{A}}}$  can either represent Zero Knowledge Proofs  
 322 or explicit proofs.

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

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

329 ■ The set of valid state is  $Q = (\text{listed-values} = \{x \in \mathcal{S}\}, \text{proofs} = \{(p \in \Pi, \widehat{\mathcal{S}} \subseteq \mathcal{S}, P \in$   
 330  $\mathcal{P}_{\mathcal{L}_{\widehat{\mathcal{S}}}})\})$ , where *listed-values* is a subset of  $\mathcal{S}$  and *proofs* is a set of tuples. Each tuple in  
 331 *proofs* consists of a proof associated with the set it applies to and to the identifier of the  
 332 process that issued the proof;

333 ■ The set of valid initial states is  $Q_0 = (\emptyset, \emptyset)$ , the state where the *listed-values* and the  
 334 *proofs* arrays are empty;

335 ■ The set of possible operation is  $O = \{\text{APPEND}(x), \text{PROVE}(y), \text{READ}()\}$ , with  $x, y \in \mathcal{S}$ ;

336 ■ The set of possible responses is  $R = \left\{ \text{True}, \text{False}, (\widehat{\mathcal{S}} \subseteq \mathcal{S}, P \in \mathcal{P}_{\mathcal{L}_{\widehat{\mathcal{S}}}}), \{(p \in \Pi, \widehat{\mathcal{S}}' \subseteq$   
 337  $\mathcal{S}, P' \in \mathcal{P}_{\mathcal{L}_{\widehat{\mathcal{S}}}'})\} \right\}$ , where True is the response to a successful APPEND operation,  $(\widehat{\mathcal{S}}, P)$  is

338 the response to a successful PROVE operation,  $\{(p, \widehat{\mathcal{S}}', P')\}$  is the response to a READ  
 339 operation, and False is the response to a failed operation; and

340 ■ The transition function is  $\Delta$ . The PROOF-LIST object type supports 5 possible transi-  
 341 tions. We define the 5 possible transitions of  $\Delta$  in Table 1.

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

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

## 355 **5 The consensus number of the AllowList object**

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

<sup>6</sup> 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 belong to  $\mathcal{A}$  (DenyList).

## 39:10 The Synchronization Power of Access Control Objects

358 We provide a specification of the AllowList object defined as a PROOF-LIST object,  
 359 where  $C(y, \widehat{\mathcal{S}}) = 1$  and

$$360 \quad \forall y \in \mathcal{S}, y \in \mathcal{L}_{\mathcal{A}} \Leftrightarrow (\mathcal{A} \subseteq \mathcal{S} \wedge y \in \mathcal{A}). \quad (1)$$

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

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

<p><b>Shared variables</b></p> <p>AS-LV <math>\leftarrow</math> <math>N</math>-dimensions Atomic-Snapshot object, initially <math>\{\emptyset\}^N</math>;</p> <p>AS-PROOF <math>\leftarrow</math> <math>N</math>-dimensions Atomic-Snapshot object, initially <math>\{\emptyset\}^N</math>;</p> <p><b>Operation APPEND(<math>v</math>) is</b></p> <p>1: <b>IF</b> <math>(v \in \mathcal{S}) \wedge (p \in \Pi_M)</math> <b>THEN</b></p> <p>2:   local-values <math>\leftarrow</math> AS-LV.Snapshot()[<math>p</math>];</p> <p>3:   AS-LV.Update(local-values <math>\cup</math> <math>v</math>, <math>p</math>);</p> <p>4:   <b>RETURN</b> true;</p> <p>5: <b>ELSE</b> <b>return</b> false;</p> <p><b>Operation READ() is</b></p> <p>6: <b>RETURN</b> AS-PROOF.Snapshot();</p>	<p><b>Operation PROVE(<math>v</math>) is</b></p> <p>7: <b>IF</b> <math>p \notin \Pi_V</math> <b>THEN</b></p> <p>8:   <b>RETURN</b> false;</p> <p>9:   <math>\mathcal{A} \leftarrow</math> AS-LV.Snapshot();</p> <p>10: <b>IF</b> <math>v \in \mathcal{A}</math> <b>THEN</b></p> <p>11:   <math>\pi_{\text{set-memb}} \leftarrow</math> Proof(<math>v \in \mathcal{A}</math>);</p> <p>12:   proofs <math>\leftarrow</math> AS-PROOF.Snapshot()[<math>p</math>];</p> <p>13:   AS-PROOF.Update(proofs <math>\cup</math> <math>(p, \mathcal{A}, \pi_{\text{set-memb}})</math>, <math>p</math>);</p> <p>14:   <b>RETURN</b> <math>(\mathcal{A}, \pi_{\text{set-memb}})</math>;</p> <p>15: <b>ELSE</b> <b>return</b> false.</p>
---	---

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

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

373 **Proof.** Let us fix an execution  $E$  of the algorithm presented in algorithm 1. Each invocation  
 374 is a sequence of a finite number of local operations and Atomic-Snapshot accesses. Because  
 375 the Atomic Snapshot primitive can be wait-free implemented in the read-write shared memory  
 376 model, each correct process terminates each invocation in a finite number of its own steps.

377 Let  $H$  be the history of the execution  $E$ . We define  $\bar{H}$ , the completed history of  $H$ . Any  
 378 invocation in  $H$  can be completed in  $\bar{H}$ . We give the completed history  $\bar{H}$  of  $H$ :

- 379 ■ Any invocation of the APPEND operation that did not reach line 3 can be completed  
 380 with the line "Return false";
- 381 ■ Any invocation of the PROVE operation that did not reach line 13 can be completed  
 382 with the line "Return false";
- 383 ■ Any invocation of the APPEND operation that reached line 3 can be completed with line  
 384 4; and
- 385 ■ Any invocation of the PROVE operation that reached line 13 can be completed with line  
 386 14.

387 The linearization points of the APPEND, PROVE and READ operations are respectively  
 388 line 3, line 13 and line 6. For convenience, We call any operation in  $\bar{H}$  that returns "false"  
 389 an invalid operation. We verify that each operation in  $\bar{H}$  respects the specification:

- 390 ■ Any operation in  $\bar{H}$  run by a process  $p$  that is invalid is an operation that only modifies  
 391 the internal state of  $p$  and that was invoked by a faulty process or that was invoked by a

392 process without the write to invoke the operation. Therefore, these invalid operations do  
 393 not impact the validity and the progress properties of the AllowList object.

- 394 ■ If an APPEND operation invoked by a process  $p$  in  $\bar{H}$  returns "true", it implies that  $p$   
 395 reached line 3. Therefore  $p$  appended a value  $v$  to the array *listed-values* at the index  $p$ .  
 396 Process  $p$  is the only process able to write at this index. Because the Update operation is  
 397 atomic, and because  $p$  is the only process able to write in AS-LV[ $p$ ], the *listed-values* array  
 398 append-only property is preserved. Furthermore, the element added to *listed-value* belongs  
 399 to the set  $\mathcal{S}$ , and the process that appends the value belongs to the set of managers  $\Pi_M$ .  
 400 Therefore, any invocation of the APPEND operation in  $\bar{H}$  that returns "true" fulfills the  
 401 APPEND validity property. Hence, any APPEND invocation in  $\bar{H}$  follows the AllowList  
 402 specification.
- 403 ■ If an invocation of the PROVE operation by a process  $p$  in  $\bar{H}$  returns  $(\mathcal{A}, \pi)$ , then  $p \in \Pi_V$   
 404 reached line 13. Therefore,  $p$  appended a proof  $\pi$  to the *proofs* array at the index  $p$ , and  
 405 the proof is a valid proof that  $v \in \mathcal{A}$ . Process  $p$  is the only process allowed to modify the  
 406 *proofs* array at this index. There is no concurrency on the write operation. Furthermore,  
 407 the set  $\mathcal{A}$ , is a subset of the AS-LV array (line 9). Because the only way to add an element  
 408 to the AS-LV array is via an APPEND operation, because we consider the linearization  
 409 point of the PROVE operation to be at line 13, the PROVE validity property is ensured.  
 410 The progress property is ensured thanks to the atomicity of the Atomic Snapshot object.  
 411 If some process executes line 3 of the APPEND operation at time  $t_1$ , then any correct  
 412 process that reaches line 8 of the PROVE( $x$ ) operation at time  $t_2 > t_1$  will be valid.  
 413 Hence, any PROVE invocation in  $\bar{H}$  follows the AllowList specification.
- 414 ■ A READ operation always returns the values of the AS-PROOF array that were linearized  
 415 before the execution of line 6, thanks to the atomicity of the Atomic Snapshot object.  
 416 Furthermore, the returned value is always a set of successful PROVE operations (AS-  
 417 PROOF). This set is compounded of proofs associated to the name of the process that  
 418 invoked the operation. Therefore, the READ validity property is ensured. Hence, any  
 419 READ invocation in  $\bar{H}$  follows the AllowList specification.

420 All operations in  $\bar{H}$  follow the AllowList specification. Thus,  $\bar{H}$  is a legal history of the  
 421 AllowList object type, and  $H$  is linearizable. To conclude, the algorithm presented in  
 422 algorithm 1 is a wait-free implementation of the AllowList object type. ◀

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

## 424 6 The consensus number of the DenyList object

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

### 431 6.1 Lower bound

432 Algorithm 2 presents an implementation of a  $k$ -consensus object using a  $k$ -DenyList object  
 433 with  $\Pi_M = \Pi_V = \Pi$ , and  $|\Pi| = k$ . It uses an Atomic Snapshot object, AS-LIST, to allow  
 434 processes to propose values. AS-LIST serves as a helping mechanism [15]. In addition, the  
 435 algorithm uses the progress and the anti-flickering properties of the PROVE operation of

## 39:12 The Synchronization Power of Access Control Objects

436 the  $k$ -DenyList to enforce the  $k$ -consensus agreement property. The PROPOSE operation  
 437 operates as follows. First, a process  $p$  tries to prove that the element 0 is not revoked by  
 438 invoking PROVE(0). Then, if the previous operation succeeds,  $p$  revokes the element 0 by  
 439 invoking APPEND(0). Then,  $p$  waits for the APPEND to be effective. This verification is  
 440 done by invoking multiple PROVE operations until one is invalid. This behavior is ensured  
 441 by the progress property of the  $k$ -DenyList object. Once the progress has occurred,  $p$  is  
 442 sure that no other process will be able to invoke a valid PROVE(0) operation. Hence,  $p$  is  
 443 sure that the set returned by the READ operation can no longer grow. Indeed, the READ  
 444 operation returns the set of valid PROVE operation that occurred prior to its invocation. If  
 445 no valid PROVE(0) operation can be invoked, the set returned by the READ operation is  
 446 fixed (with regard to the element 0). Furthermore, all the processes in  $\Pi$  share the same  
 447 view of this set.

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

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

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

■ **Algorithm 2** Implementation of a  $k$ -consensus object using one  $k$ -DenyList object and one Atomic Snapshot

456

457 ► **Theorem 6.** *Algorithm 2 wait-free implements a  $k$ -consensus object.*

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

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

470 ■ The process  $p$  that invoked a failed PROPOSE operation in  $\bar{H}$  is faulty—by definition,  
 471 the process prematurely stopped before line 8. Therefore, the fact that  $p$  cannot decide  
 472 does not impact the termination nor the agreement properties of the  $k$ -consensus object.

473 ■ A successful PROPOSE operation returns  $\text{AS-LIST.Snapshot()}[f_{|processes|}(processes)]$ .  
 474 Furthermore, a process proposed this value in line 1. All the processes that invoke  
 475 PROPOSE conduct an APPEND(0) operation, and wait for this operation to be effective  
 476 using the while loop at line 4 to 6. Thanks to the anti-flickering property of the  $k$ -  
 477 DenyList object, when the APPEND operation is effective for one process—i.e. the  
 478 Progress happens, in other words, a PROVE(0) operation is invalid—, then it is effective  
 479 for any other process that would invoke the PROVE(0) operation. Hence, thanks to the  
 480 anti-flickering property, when a process obtains an invalid response from the PROPOSE(0)  
 481 operation at line 5, it knows that no other process can invoke a valid PROVE(0) operation.  
 482 This implies that the READ operation conducted at line 7 will return a fix set of processes,  
 483 and all the processes that reach this line will see the same set. Furthermore, because  
 484 each process invokes a PROPOSE(0) before the APPEND(0) at line 3, at least one valid  
 485 PROPOSE(0) operation was invoked. Therefore, the  $processes$  set is not empty. Because  
 486 each process ends up with the same set  $processes$ , and thanks to the determinism of the  
 487 function  $f_i$ , all correct processes output the same value  $v$  (Agreement property and non-  
 488 trivial value). The value  $v$  comes from the Atomic Snapshot object, composed of values  
 489 proposed by authorized processes (Validity property). Hence a successful PROPOSE  
 490 operation follows the  $k$ -consensus object specification.

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

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

## 494 6.2 Upper bound

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

498 We provide an instantiation of the DenyList object defined as a PROOF-LIST object,  
 499 where:

$$500 \quad \forall y \in \mathcal{S}, y \in \mathcal{L}_{\mathcal{A}} \Leftrightarrow (\mathcal{A} \subseteq \mathcal{S} \wedge y \notin \mathcal{A}).$$

501 And where :

$$502 \quad C(y, \hat{\mathcal{S}}) = \begin{cases} 1, & \text{if } \forall \mathcal{A}' \in \hat{\mathcal{S}}, y \notin \mathcal{A}' \\ 0, & \text{otherwise.} \end{cases}$$

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

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

## 39:14 The Synchronization Power of Access Control Objects

515 targeted application does not require the user’s anonymity, it is possible to use the plaintext  
516  $v$  as the helping value.

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

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

523 When a process invokes the PROVE( $v$ ) operation, it publishes  $C_v$ , the cryptographic  
524 commitment to  $v$ , using an atomic snapshot object. This commitment is published along  
525 with a timestamp [16] defined as follow. A local timestamp  $(p, c)$  is constituted of a process  
526 identifier  $p$  and a local counter value  $c$ . The counter  $c$  is always incremented before being  
527 reused. Therefore, each timestamp is unique. Furthermore, we build the strict total order  
528 relation  $\mathcal{R}$  such that  $(p, c)\mathcal{R}(p', c') \Leftrightarrow (c < c') \vee ((c = c') \wedge (p < p'))$ . The timestamp is used  
529 in coordination with the helping value  $C_v$  to ensure termination. A process  $p$  that invokes  
530 the PROVE( $v$ ) operation must parse all the values proposed by the other processes. If a  
531 PROVE( $v'$ ) operation was invoked by a process  $p'$  earlier than the one invoked by  $p$ —under  
532 the relation  $\mathcal{R}$ —, then  $p$  must affect a set "val" for the PROVE operation of  $p'$  via the  
533 consensus object. The set "val" is obtained by reading the AS-LV object. The AS-LV object  
534 is append-only—no operation removes elements from the object. Furthermore, the sets "val"  
535 are attributed via the consensus object. Therefore, this mechanism ensures that the sets on  
536 which the PROVE operations are applied always grow.

537 Furthermore, processes sequentially parse the CONS-ARR using the counter <sub>$p$</sub>  variable.  
538 This behavior, in collaboration with the properties of the consensus, ensures that all the  
539 process see the same tuples (winner, val) in the same order.

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

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

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

560 ► **Theorem 8.** *Algorithm 3 wait-free implements a  $k$ -DenyList object.*

561 **Proof.** Let us fix an execution  $E$  of the algorithm presented in Algorithm 3. The strict order  
562 relation  $\mathcal{R}$  used to prioritize accesses to the CONS-ARR array implies that each process that

**Shared variables**  
 AS-LV  $\leftarrow$   $N$ -dimensions Atomic-Snapshot object, initially  $\{\emptyset\}^N$ ;  
 AS-Queue  $\leftarrow$   $N$ -dimensions Atomic-Snapshot object, initially  $\{\emptyset\}^N$ ;  
 CONS-ARR <sub>$p$</sub>   $\leftarrow$  an array of  $k$ -consensus objects of size  $l > 0$ ;  
 AS-PROOF  $\leftarrow$   $l$ -dimensions Atomic-Snapshot object, initially  $\{\emptyset\}^l$ ;  
**Local variables**  
**For each**  $p \in \Pi_V$  :  
     evaluated <sub>$p$</sub>   $\leftarrow$  an array of size  $l > 0$ , initially  $\{\emptyset\}^l$ ;  
     counter <sub>$p$</sub>   $\leftarrow$  a positive integer, initially 0;  
**Operation APPEND**( $v$ ) **is**  
 1: **If** ( $v \in S$ )  $\wedge$  ( $p \in \Pi_M$ ) **then**  
 2:     local-values  $\leftarrow$  AS-LV.Snapshot()[ $p$ ];  
 3:     AS-LV.UPDATE(local-values  $\cup$   $v$ ,  $p$ );  
 4:     **Return** true;  
 5:     **Else return** false;  
**Operation PROVE**( $v$ ) **is**  
 6: **If**  $p \notin \Pi_V$  **then**  
 7:     **Return** false;  
 8:      $C_v \leftarrow$  Commitment( $v$ );  
 9: cnt  $\leftarrow$  counter <sub>$p$</sub> ;  
 10: AS-Queue.UPDATE(((cnt,  $p$ ),  $C_v$ ),  $p$ );  
 11: queue  $\leftarrow$  AS-Queue.Snapshot() \ evaluated <sub>$p$</sub> ;  
 12: **While** (cnt,  $p$ )  $\in$  queue **do**  
 13:     oldest  $\leftarrow$  the smallest clock value in queue under  $\mathcal{R}$ ;  
 14:     prop  $\leftarrow$  (oldest, AS-LV.snapshot());  
 15:     (winner, val)  $\leftarrow$  CONS-ARR[counter <sub>$p$</sub> ].propose(prop);  
 16:     ((counter <sub>$p_w$</sub> ,  $p_w$ ),  $C^*$ )  $\leftarrow$  winner;  
 17:     **If**  $C^* = C_v \wedge v \notin \text{val}$  **then**  
 18:          $\pi_{SNM} \leftarrow$  Proof( $v \notin \text{val}$ );  
 19:         AS-PROOF.Update(( $p_w$ , val,  $\pi_{SNM}$ , winner), counter <sub>$p_w$</sub> );  
 20:     evaluated <sub>$p$</sub>   $\leftarrow$  evaluated <sub>$p$</sub>   $\cup$  winner;  
 21:     queue  $\leftarrow$  queue \ winner;  
 22:     counter <sub>$p$</sub>   $\leftarrow$  counter <sub>$p$</sub>  + 1;  
 23:     **If**  $v \notin \text{val}$  **then**  
 24:         **Return** (val,  $\pi_{SNM}$ );  
 25:         **Else return** false;  
**Operation READ**() **is**  
 26: **Return** AS-PROOF.Snapshot();

■ **Algorithm 3**  $k$ -DenyList object type implementation using  $k$ -consensus objects and Atomic Snapshot objects.

563 enters the while loop in line 12 will only iterate a finite number of times. Furthermore, we  
 564 assume that  $k$ -consensus objects and atomic-snapshot objects are atomic. Therefore, each  
 565 process returns from a PROVE, an APPEND, or a READ operation in a finite number of its  
 566 own steps.

567 Let  $H$  be the history of  $E$ . We define  $\bar{H}$ , the completed history of  $H$ . We associate a  
 568 specific response with all pending invocations in  $H$ . The associated responses are:

- 569 ■ Any invocation of the APPEND operation that did not reach line 3 can be completed  
 570 with the line "**Return false**".
- 571 ■ Any invocation of the PROVE operation that did not reach line 10 can be completed  
 572 with the line "**Return false**".
- 573 ■ Any pending invocation of the PROVE operation by the process  $p$  that reached line 10  
 574 is completed with the line "**Return** (val,  $\pi_{SNM}$ );" if ( $p$ , value,  $\pi_{SNM}$ , winner) is in the  
 575 AS-PROOF array, and the value added by process  $p$  in line 10 is "winner". Otherwise,  
 576 the operation is completed with the line "**Return false**".
- 577 ■ Any pending invocation of the APPEND operation that reached line 3 can be completed  
 578 with line 4.

579 The linearization point of the APPEND and READ operations are respectively at line 3 and  
 580 26. Let us consider a valid PROVE operation invoked by a process  $p$  that is attributed a  
 581 tuple (winner, val) at the index counter <sub>$p_w$</sub>  of the CONS-ARR array. We say this operation is  
 582 linearized when the first AS-PROOF.Update labeled with counter <sub>$p_w$</sub>  in line 19 is executed  
 583 by any process.

584 For convenience, we call operations that return false invalid operations. The consensus  
 585 objects in CONS-ARR are accessed at most once by each process. There are only  $k = |\Pi_V|$   
 586 processes allowed to access these objects. Therefore, the  $k$ -consensus objects in the array  
 587 always return a value different from  $\emptyset$ . We now prove that all operations in  $\bar{H}$  follow the  
 588 DenyList specification:

- 589 ■ An invalid APPEND operation in  $\bar{H}$  only modifies the internal state of the process. This  
 590 operation does not modify the state of the shared object. It is either invoked by an  
 591 unauthorized process which fails in line 1, or by a faulty process. This operation follows  
 592 the specification;
- 593 ■ An invalid PROVE operation in  $\bar{H}$  is an operation that returns false in line 7 or 25. In

594 the first case, the process was not authorized to propose a proof. In the second case, the  
 595 value  $v$  used by the process is already inside the set "val" the process was attributed by  
 596 the consensus in line 15. This set is produced from the values added to the AS-LV object.  
 597 This object begins as an empty set, and values inside this set can only be added using  
 598 the APPEND operation. Therefore, the PROVE validity property is ensured.

599 ■ If an invocation of the APPEND operation in  $\bar{H}$  returns true, it implies that process  
 600  $p$  appended a value  $v$  to the *listed-values* array, at the index  $p$  at line 3. Because the  
 601 WRITE operation is atomic, and because  $p$  is the only process able to write in AS-ACC[ $p$ ],  
 602 the *listed-values* array append-only property is preserved. Hence a successful APPEND  
 603 operation follows the specification.

604 ■ If an invocation of the PROVE operation in  $\bar{H}$  returns True, it implies that: 1) process  
 605  $p$  was attributed a  $k$ -consensual set "val" on line 15, 2) from line 17,  $v \notin \text{val}$ , and 3) a  
 606 proof that  $v \notin \text{"val"}$  was added to the AS-PROOF object, either by  $p$  or by another  
 607 process performing the helping mechanism. First, to prove the progress property, we  
 608 assume a history where first, a process  $p'$  obtains a positive response from an APPEND( $v$ )  
 609 operation. Afterward, a process  $p$  invokes a PROVE( $v$ ) operation. Therefore, the value  $v$   
 610 will already be included in the AS-LV object at this time because  $p'$  received a positive  
 611 response from its invocation. Any process that executes the line 14 of the PROVE  
 612 operation after the invocation of  $p$  will propose a set where  $v$  is included. Therefore,  
 613 the set "val" that will be affected to  $p$  by the consensus on line 15 will include  $v$ . The  
 614 PROVE( $v$ ) operation invoked by  $p$  will be invalid. The progress property is ensured.

615 Second, the anti-flickering property is ensured by the helping mechanism and the  $k$ -  
 616 consensus objects used from line 10 to 22. The processes in  $\Pi_V$  that invoke the PROVE( $v$ )  
 617 operation will sequentially attribute a set "val" to each proving process, using the set  
 618 of  $k$ -consensus objects. Furthermore, this sequential attribution takes into account the  
 619 evolution of the AS-LV object. Therefore, the set associated with the object CONS-  
 620 ARR[ $i - 1$ ] is always included in the set associated with the object CONS-ARR[ $i$ ].

621 Furthermore, the CONS-ARR array is browsed sequentially by each process invoking the  
 622 PROVE operation. Therefore, if a process  $p$  that invokes a PROVE( $v$ ) operation with  
 623 a timestamp  $t$ , and this invocation is not valid in the end,  $p$  will nonetheless linearize  
 624 all the PROVE( $v$ ) operations that have a lower timestamp than  $t$  before returning from  
 625 the operation. Hence, all the valid PROVE( $v$ ) operations will be linearized before the  
 626 response of  $p$ 's invocation, and any invocation of a PROVE( $v$ ) operation that occurs after  
 627 the response of  $p$ 's invocation will fail.

628 Third, the PROVE validity property directly follows from point (2) and the anti-flickering  
 629 property. Hence a successful PROVE operation follows the specification.

630 ■ A READ operation always returns, and thanks to the atomicity of the Atomic Snapshot  
 631 object, it always returns the most up-to-date version of the AS-PROOF array.

632 All operations in  $\bar{H}$  follow the  $k$ -DenyList specification. Therefore the algorithm presented  
 633 in Algorithm 3 is a wait-free implementation of the  $k$ -DenyList object type. ◀

634 The following corollary follows from Theorem 6 and Theorem 8.

635 ► **Corollary 9.** *The  $k$ -DenyList object type has consensus number  $k$ .*

## 636 7 Discussion

637 This section presents several applications where the AllowList and the  $k$ -DenyList can be used  
 638 to determine consensus number of more elaborate objects. More importantly, the analysis of

639 the consensus of these use cases makes it possible to determine if actual implementations  
640 achieve optimal efficiency in terms of synchronization. If not, we use the knowledge of the  
641 consensus number of the AllowList and DenyList objects to give intuitions on how to build  
642 more practical implementations. More precisely, the fact that consensus numbers of AllowList  
643 and DenyList objects are (in most cases) smaller than  $n$  implies that most implementations  
644 can reduce the number of processes that need to synchronize in order to implement such  
645 distributed objects. The liveness of many consensus protocols is only ensured when the  
646 network reaches a synchronous period. Therefore, reducing the number of processes that need  
647 to synchronize can increase the system's probability of reaching such synchronous periods.  
648 Thus, it can increase the effectiveness of such protocols.

## 649 7.1 Revocation of a verifiable credential

650 We begin by analyzing Sovrin's Verifiable-Credential revocation method using the DenyList  
651 object [4]. Sovrin is a privacy-preserving Distributed Identity Management System (DIMS).  
652 In this system, users own credentials issued by entities called issuers. A user can employ one  
653 such credential to prove to a verifier they have certain characteristics. An issuer may want  
654 to revoke a user's credential prematurely. To do so, the issuer maintains an append-only list  
655 of revoked credentials. When a user wants to prove that their credential is valid, they must  
656 provide to the verifier a valid ZKP of set-non-membership proving that their credential is  
657 not revoked, i.e. not in the DenyList. In this application, the set of managers  $\Pi_M$  consists  
658 solely of the credential's issuer. Hence, the proof concerns solely the verifier and the user.  
659 The way Sovrin implements this verification interaction is by creating an ad-hoc peer-to-peer  
660 consensus instance between the user and the verifier for each interaction. Even if the resulting  
661 DenyList has consensus number 2, Sovrin implements the APPEND operation using an  
662 SWMR stored on a blockchain-backed ledger (which requires synchronizing the  $N$  processes  
663 of the system). Our results suggest instead that Sovrin's revocation mechanism could be  
664 implemented without a blockchain. The APPEND operation could be implemented using  
665 FIFO reliable broadcast, and the PROVE operation could be implemented using pairwise  
666 consensus between users and verifiers.

## 667 7.2 The Anonymous Asset Transfer object

668 The anonymous asset transfer object is another application of the DenyList and the AllowList  
669 objects. As described in Appendix B, it is possible to use these objects to implement the  
670 asset transfer object described in [9]. Our work generalizes the result by Guerroui et al. [9].  
671 Guerraoui et al. show that a joint account has consensus number  $k$  where  $k$  is the number of  
672 agents that can withdraw from the account. We can easily prove this result by observing  
673 that withdrawing from a joint account requires a denylist to record the already spent coins.  
674 Nevertheless, our ZKP capable construction makes it possible to show that an asset transfer  
675 object where the user is anonymous, and its transactions are unlinkable also has consensus  
676 number  $k$ , where  $k$  is the number of processes among which the user is anonymous. The two  
677 main implementations of Anonymous Asset Transfer, ZeroCash and Monero [17, 18], use a  
678 blockchain as their main double spending prevention mechanism. While the former provides  
679 anonymity on the whole network, the second only provides anonymity among a subset of  
680 the processes involved in the system. Hence, this second implementation could reduce its  
681 synchronization requirements accordingly.

682 Furthermore, our implementation uses AllowList as *ex-nihilo*-coin-creation prevention  
683 mechanism. Hence, both security properties of an anonymous asset transfer object can be

684 enforced separately. Using the fact that AllowLists have consensus number 1, this result  
 685 implies that this part of the protocol could be handled more efficiently. For example, the  
 686 most space and resource-intensive part of the ZeroCash protocol is the *ex-nihilo*-coin-creation  
 687 prevention mechanism. This part of the protocol is implemented in a synchronous way,  
 688 i.e., all processes synchronize to conduct it, which is, as we demonstrated, sub-optimal.  
 689 Therefore, this analysis may lead to a more efficient ZeroCash-like asset transfer protocol  
 690 implementation.

### 691 7.3 Distributed e-vote systems

692 Finally, another direct application of the DenyList object is the blind-signature-based e-vote  
 693 system with consensus number  $k$ ,  $k$  being the number of voting servers, which we present in  
 694 Appendix C. Most distributed implementations of such systems also use blockchains, whereas  
 695 only a subset of the processes involved actually require synchronization.

## 696 8 Related Works

### 697 Bitcoin and blockchain

698 Even though distributed consensus algorithms were already largely studied [19, 20, 21, 22, 23],  
 699 the rise of Ethereum—and the possibilities offered by its versatile smart contracts—led to  
 700 new ideas to decentralized already known applications. Among those, e-vote and DIMS [4]  
 701 are two examples.

702 Blockchains increased the interest in distributed versions of already existing algorithms.  
 703 However, these systems are usually developed with little concern for the underlying theoretical  
 704 basis they rely on. A great example is trustless money transfer protocols or crypto money. The  
 705 underlying distributed asset-transfer object was never studied until recently. A theoretical  
 706 study proved that a secure asset-transfer protocol does not need synchronicity between  
 707 network nodes [9]. Prior to this work, all proposed schemes used a consensus protocol, which  
 708 cannot be deterministically implemented in an asynchronous network [24]. The result is  
 709 that many existing protocols could be replaced by more efficient, Reliable Broadcast [23]  
 710 based algorithms. This work leads to more efficient implementation proposal for money  
 711 transfer protocol [10]. Alpos et al. then extended this study to the Ethereum ERC20 smart  
 712 contracts [12]. This last paper focuses on the asset-transfer capability of smart contracts.  
 713 Furthermore, the object described has a dynamic consensus number, which depends on the  
 714 processes authorized to transfer money from a given account. Furthermore, this work and  
 715 the one from Guerraoui et al. [9] both analyze a specific object that is not meant to be used  
 716 to find the consensus number of other applications. In contrast, our work aims to be used as  
 717 a generic tool to find the consensus number of numerous systems.

### 718 E-vote

719 An excellent example of the usage of DenyList is to implement blind signatures-based e-vote  
 720 systems [25]. A blind signature is a digital signature where the issuer can sign a message  
 721 without knowing its content. Some issuer signs a cryptographic commitment—a cryptographic  
 722 scheme where Alice hides a value while being bound to it [26]—to a message produced by  
 723 a user. Hence, the issuer does not know the actual message signed. The user can then  
 724 un-commit the message and present the signature on the plain-text message to a verifier.  
 725 The verifier then adds this message to a DenyList. A signature present in the DenyList is no  
 726 longer valid. Such signatures are used in some e-vote systems [27, 28]. In this case, the blind

727 signature enables anonymity during the voting operation. This is the e-vote mechanism that  
728 we study in this article. They can be implemented using a DenyList to restrain a user from  
729 voting multiple times. This method is explored in Appendix C

730 There exists two other way to provide anonymity to the user of an e-vote system. The  
731 first one is to use a MixNet [29, 30, 31]. MixNet is used here to break the correlation between  
732 a voter and his vote. Finally, anonymity can be granted by using homomorphic encryption  
733 techniques [32, 33].

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

### 738 **Anonymous Money Transfer**

739 Blockchains were first implemented to enable trustless money transfer protocols. One of the  
740 significant drawbacks of this type of protocol is that it only provides pseudonymity to the  
741 user. As a result, transfer and account balances can be inspected by anyone, thus revealing  
742 sensitive information about the user. Later developments proposed hiding the user's identity  
743 while preventing fraud. The principal guarantees are double-spending prevention—i.e., a coin  
744 cannot be transferred twice by the same user—and *ex nihilo* creation prevention—i.e., a user  
745 cannot create money. Zcash [18] and Monero [17] are the best representative of anonymous  
746 money transfer protocols. The first one uses an AllowList to avoid asset creation and a  
747 DenyList to forbid double spending, while the second one uses ring signatures. We show in  
748 Appendix B that the DenyList and AllowList objects can implement an Anonymous Money  
749 Transfer object, and thus, define the synchronization requirements of the processes of the  
750 system.

## 751 **9 Conclusion**

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

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

## 767 **Acknowledgments**

768 We wish to thank the anonymous reviewers for their insightful comments and remarks that  
769 led to significant improvements to our paper. This work was funded by the SOTERIA

## 39:20 The Synchronization Power of Access Control Objects

770 project. SOTERIA has received funding from the European Union's Horizon 2020 research  
771 and innovation programme under grant agreement No101018342. This content reflects only  
772 the author's view. The European Agency is not responsible for any use that may be made of  
773 the information it contains.

## 774 — References —

- 775 1 Gaby G. Dagher, Praneeth Babu Marella, Matea Milojkovic, and Jordan Mohler. Broncovote:  
776 Secure voting system using ethereum’s blockchain. In *ICISSP*, 2018.
- 777 2 Ethereum name service documentation. online - <https://docs.ens.domains/> - accessed  
778 23/11/2022.
- 779 3 Harry A. Kalodner, Miles Carlsten, Paul Ellenbogen, Joseph Bonneau, and Arvind Narayanan.  
780 An empirical study of namecoin and lessons for decentralized namespace design. In *Workshop*  
781 *on the Economics of Information Security*, 2015.
- 782 4 Sovrin: A protocol and token for self-sovereign identity and decentralized trust. Technical  
783 report, Sovrin Foundation, 2018.
- 784 5 Lundkvist, Heck, Torstensson, Mitton, and Sena. Uport: A platform for self-sovereign identity.  
785 Technical report, Uport.
- 786 6 Ming K. Lim, Yan Li, Chao Wang, and Ming-Lang Tseng. A literature review of blockchain  
787 technology applications in supply chains: A comprehensive analysis of themes, methodologies  
788 and industries. *Computers and Industrial Engineering*, 154:107133, 2021.
- 789 7 Jyoti Grover. Security of vehicular ad hoc networks using blockchain: A comprehensive review.  
790 *Vehicular Communications*, 34:100458, 2022.
- 791 8 Maurice P Herlihy and Jeannette M Wing. Linearizability: A correctness condition for  
792 concurrent objects. *ACM Transactions on Programming Languages and Systems*, 12(3):463–  
793 492, 1990.
- 794 9 Rachid Guerraoui, Petr Kuznetsov, Matteo Monti, Matej Pavlovič, and Dragos-Adrian Seredin-  
795 schi. The consensus number of a cryptocurrency. *PODC ’19*, page 307–316, 2019.
- 796 10 Alex Auvolat, Davide Frey, Michel Raynal, and François Taïani. Money Transfer Made  
797 Simple: a Specification, a Generic Algorithm, and its Proof. *Bulletin European Association*  
798 *for Theoretical Computer Science*, 132, October 2020.
- 799 11 Maurice Herlihy. Wait-free synchronization. *ACM Trans. Program. Lang. Syst.*, 13(1):124–149,  
800 jan 1991.
- 801 12 Orestis Alpos, Christian Cachin, Giorgia Azzurra Marson, and Luca Zanolini. On the  
802 synchronization power of token smart contracts. In *41st IEEE ICDCS*, pages 640–651, 2021.
- 803 13 Yehuda Afek, Hagit Attiya, Danny Dolev, Eli Gafni, Michael Merritt, and Nir Shavit. Atomic  
804 snapshots of shared memory. *JACM*, 40(4):873–890, sep 1993.
- 805 14 Daniel Benarroch, Matteo Campanelli, Dario Fiore, Kobi Gurkan, and Dimitris Kolonelos. Zero-  
806 knowledge proofs for set membership: Efficient, succinct, modular. In *Financial Cryptography*  
807 *and Data Security*. Springer Berlin Heidelberg, 2021.
- 808 15 Keren Censor-Hillel, Erez Petrank, and Shahar Timnat. Help! *PODC ’15*, page 241–250, 2015.
- 809 16 Leslie Lamport. Time, clocks and the ordering of events in a distributed system. *Communica-*  
810 *tions of the ACM* 21, (7), 558-565, July 1978.
- 811 17 Nicolas van Saberhagen. Cryptonote v 2.0, october 2013.
- 812 18 Eli Ben Sasson, Alessandro Chiesa, Christina Garman, Matthew Green, Ian Miers, Eran  
813 Tromer, and Madars Virza. Zerocash: Decentralized anonymous payments from bitcoin. In  
814 *2014 IEEE Symposium on Security and Privacy*, pages 459–474, May 2014.
- 815 19 Miguel Castro and Barbara Liskov. Practical byzantine fault tolerance. *OSDI ’99*, page  
816 173–186, 1999.
- 817 20 Leslie Lamport. *The Part-Time Parliament*, volume 16. 1998.
- 818 21 Miguel Castro and Barbara Liskov. Proactive recovery in a Byzantine-Fault-Tolerant system.  
819 In *OSDI 2000*, October 2000.
- 820 22 Pierre-Louis Aublin, Sonia Ben Mokhtar, and Vivien Quéma. Rbft: Redundant byzantine  
821 fault tolerance. In *IEEE 33rd International Conference on Distributed Computing Systems*,  
822 pages 297–306, 2013.
- 823 23 Gabriel Bracha. Asynchronous byzantine agreement protocols. *Information and Computation*,  
824 75(2):130–143, 1987.

## 39:22 The Synchronization Power of Access Control Objects

- 825 24 Michael J. Fischer, Nancy A. Lynch, and Michael S. Paterson. Impossibility of distributed  
826 consensus with one faulty process. *J. ACM*, 32(2):374–382, apr 1985.
- 827 25 David Chaum. Blind signatures for untraceable payments. In *Advances in Cryptology*, pages  
828 199–203, 1983.
- 829 26 Torben Pryds Pedersen. Non-interactive and information-theoretic secure verifiable secret  
830 sharing. In *Advances in Cryptology — CRYPTO '91*, pages 129–140, 1992.
- 831 27 Atsushi Fujioka, Tatsuaki Okamoto, and Kazuo Ohta. A practical secret voting scheme for  
832 large scale elections. In *AUSCRYPT '92*, pages 244–251, 1993.
- 833 28 Miyako Ohkubo, Fumiaki Miura, Masayuki Abe, Atsushi Fujioka, and Tatsuaki Okamoto. An  
834 improvement on a practical secret voting scheme. In *Information Security*, pages 225–234,  
835 1999.
- 836 29 Ari Juels, Dario Catalano, and Markus Jakobsson. Coercion-resistant electronic elections.  
837 WPES '05, page 61–70, 2005.
- 838 30 Markus Jakobsson, Ari Juels, and Ronald L. Rivest. Making mix nets robust for electronic  
839 voting by randomized partial checking. In *11th USENIX Security Symposium*, August 2002.
- 840 31 Michael R. Clarkson, Stephen Chong, and Andrew C. Myers. Civitas: Toward a secure voting  
841 system. *IEEE SSP*, pages 354–368, 2008.
- 842 32 Olivier Baudron, Pierre-Alain Fouque, David Pointcheval, Jacques Stern, and Guillaume  
843 Poupard. Practical multi-candidate election system. PODC '01, page 274–283, 2001.
- 844 33 Ronald Cramer, Rosario Gennaro, and Berry Schoenmakers. A secure and optimally efficient  
845 multi-authority election scheme. In *EUROCRYPT '97*, pages 103–118, 1997.
- 846 34 Gang Wang, Zhijie Jerry Shi, Mark Nixon, and Song Han. Sok: Sharding on blockchain. In  
847 *Proceedings of the 1st ACM Conference on Advances in Financial Technologies*, page 41–61,  
848 2019.
- 849 35 Satoshi Nakamoto. Bitcoin: A peer-to-peer electronic cash system. Technical report, march  
850 2009.
- 851 36 Andreas Pfitzmann and Marit Hansen. Anonymity, unlinkability, undetectability, unobservabil-  
852 ity, pseudonymity, and identity management—a consolidated proposal for terminology. *Version*  
853 *v0*, 31, 01 2007.

### 854 **A** Variations on the *listed-values* array

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

#### 858 **One-process only**

859 We will first explore a limited scenario where the processes can only remove the values they  
860 wrote themselves. In this case, there are no conflicts on the append and remove operations.  
861 The *listed-values* array can be seen as an array of  $|\Pi_V|$  values. A process  $p_i$  can write the  
862  $i$ -th index of the *listed-values* array. It is the only process that modifies this array. Therefore,  
863 there are no conflicts upon writing. We would need to add a REMOVE operation to the  
864 AllowList and DenyList object. Because of this REMOVE operation, the AllowList could  
865 act as a DenyList. Indeed, let us assume the managers adds all elements of the universe  
866 of the possible identifiers to the AllowList in the first place. Then, this AllowList can  
867 implement a DenyList object, where the REMOVE operation of the AllowList is equivalent  
868 to the APPEND operation of the DenyList. Hence, the AllowList object would need an  
869 anti-flickering property to prevent concurrent PROVE operations from yielding conflicting  
870 results. This implies that an AllowList object implemented with a REMOVE operation

871 is equivalent to a DenyList object and has consensus number  $k$ , where  $k$  is the number of  
872 processes in  $\Pi_V$ .

### 873 Multi-process

874 The generalization of the previous single-write-remove *listed-values* array is a *listed-values*  
875 array where  $k_{AR}$  (AR for APPEND/REMOVE) processes can remove a value appended by  
876 process  $p_i$ . We assume each process  $p$  is authorized to conduct APPEND and REMOVE  
877 operations on its "own" register. Furthermore, each process  $p_i$  has a predefined authorization  
878 set  $\mathcal{A}_i \subseteq \Pi_M$ , defining which processes can APPEND or REMOVE on  $p_i$ 's register. We  
879 always have  $p_i \in \mathcal{A}_i$ . If  $p_j \in \mathcal{A}_i$ , then  $p_j$  is allowed to "overwrite" (remove) anything  $p_i$   
880 wrote. In this case, all authorized processes need to synchronize in order to write a value on  
881 the *listed-values* array. More precisely, we can highlight two cases.

882 The first case is the "totally shared array" case, where all processes share the same  
883  $\mathcal{A}_i = \Pi_M$ . Any modifications on the *listed-values* array by one process  $p_i$  can be in competition  
884 with any other process  $p_j \in \Pi_M$ . Therefore, there must be a total synchronization among all  
885 the processes of the managers' set to modify the *listed-values* array. When such behaviour  
886 is needed, both AllowList and DenyList require solving consensus among at least  $|\Pi_M|$   
887 processes to implement the APPEND and REMOVE operations.

888 The second case is the "cluster" case: a subset of processes share a sub-array, which they  
889 can write. In this case, each process in a given cluster must synchronize before writing (or  
890 removing) a value. The synchronization required is only between this cluster's  $k_{AR}$  authorized  
891 process. This corresponds to some extent to a sharded network [34].

## 892 **B** Anonymous Asset-Transfer object type

893 Decentralized money transfer protocols were popularized by Bitcoin [35]. Guerraoui et al.  
894 proposed a theoretical analysis [9] that proved that the underlying object, the asset transfer  
895 object, has consensus number 1 if each account is owned by a single process. This result  
896 implies that the expensive Proof Of Work (POW) leveraged by the Bitcoin implementation  
897 is an over-engineered solution in a message-passing setting. A less expensive solution based  
898 on the Reliable-Broadcast primitive works as well [10]. The paper by Guerraoui et al. also  
899 studies the case where multiple processes share accounts. In this case, the consensus number  
900 of the resulting object is  $k$ , the maximum number of processes sharing a given account.

901 These works give a good insight into the problem of asset transfer, but they only study  
902 pseudonymous systems, where all transactions can be linked to a single pseudonym. With  
903 the growing interest in privacy-enhancing technologies, cryptocurrency communities try  
904 to develop anonymous and unlinkable money transfer protocols [18, 17]. The subsequent  
905 question is to know the consensus number associated with the underlying distributed object.  
906 The formalization of the AllowList and the DenyList objects presented in this article makes  
907 it possible to answer this question. This section is dedicated to this proof.

### 908 B.1 Problem formalization

#### 909 Asset-Transfer object type definition

910 The Asset-Transfer object type allows a set of processes to exchange assets via a distributed  
911 network. We reformulate the definition proposed by Guerraoui et al. [9] to describe this  
912 object:

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

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

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

937 The paper by Guerraoui et al. [9] informs us that the consensus number of such an object  
 938 depends on the map  $\mu$ . If  $\sum_{i \in \{0, \dots, |\Pi|\}} \mu(w)[i] \leq 1, \forall w \in \mathcal{W}$ , then the consensus number of  
 939 the object type is 1. Otherwise, the consensus number is  $\max_{w \in \mathcal{W}} (\sum_{i \in \{0, \dots, |\Pi|\}} \mu(w)[i])$ . In  
 940 other words, the consensus number of such object type is the maximum number of different  
 941 processes that can invoke a TRANSFER operation on behalf of a given wallet.

#### 942 From continuous balances to token-based Asset-Transfer

943 The definition proposed by Guerraoui et al. uses a continuous representation of the balance  
 944 of each account. Implementing anonymous money transfer with such a representation would  
 945 require a mechanism to hide the transaction amounts [18]. As such a mechanism would not  
 946 affect the synchronization properties of the anonymous-money-transfer object, we simplify  
 947 the problem by considering a token-based representation. This means that the algorithm  
 948 can transfer only tokens of a predefined weight. To move from one representation to the  
 949 other, we operate a bijection between the finite history of a continuous account-based money  
 950 representation and a token-based representation.

951 Let  $\mathcal{W}$  be a set of accounts. Let  $\mu : \mathcal{W} \rightarrow \{0, 1\}^{|\Pi|}$  be the owner map, and let  $D_{(I, V)}$  be a  
 952 discretization function such that  $D_{(I, V)} : \mathcal{S} \subseteq \mathbb{R}^+ \rightarrow \mathbb{N}^I$ , for some  $I \in \mathbb{N}^*$ .  $D_{(I, V)}$  is defined  
 953 as follows:

$$954 \quad \forall i \in \{0, \dots, I\}, \forall x \in \mathcal{S} : \sum_{i=1}^I D_{(I, V)}(x)[i] = x$$

$$955 \quad \text{where } D_{(I, V)}(x) = ((D_{(I, V)}(x)[i] = V) \vee (D_{(I, V)}(x)[i] = 0)).$$

958 We need to define  $\mathcal{S}$  in order to make  $D_{(I,V)}$  bijective. Let the array  $\{InitBal_1, \dots, InitBal_{|\mathcal{W}|}\}$   
 959 be the initial balances of the  $|\mathcal{W}|$  accounts, with  $InitBal_i \in \mathbb{R}^+, \forall i \in \{1, \dots, p\}$ . Let us fix  
 960 an execution  $E$  of an Asset Transfer object. Let  $H$  be the history of  $E$ , and let  $\bar{H}$  be the  
 961 linearization of this history. Let us assume the amounts transferred are in  $\mathbb{N}^{*+}$ . Let  $T$  be  
 962 the array of transferred amounts. Let  $V$  be the greatest common divisor of all the elements  
 963 in  $T$  and the initial balance array. Let  $TotalTrans_i$  be the total number of transactions the  
 964 account receives  $i$  in  $\bar{H}$ . Let  $balance(i, j)$  be the balance of the account  $i$  after its  $j$ 'th asset  
 965 reception. Let  $I$  be the greatest amount of money possessed by an account in  $\bar{H}$  divided  
 966 by  $V$ , i.e.,  $I = \frac{\max_{i \in \{1, \dots, |\mathcal{W}|\}} (\max_{j \in \{1, \dots, TotalTrans_i\}} (balance(i, j)))}{V}$ . Hence, we can apply  $D_{(I,V)}$  to  
 967 each element of the initial balance array. We can define  $\mathcal{S}$  as the infinite set  $\{0, V, 2V, 3V, \dots\}$ .  
 968 Using such an  $\mathcal{S}$ , the map  $D_{(I,V)}$  is bijective when applied to  $\bar{H}$ .

969 Thanks to this bijection, all transactions can be seen as a given amount of "coins"  
 970 transferred. A coin corresponds to a given amount of asset  $V$ . The amount of coins of each  
 971 account is represented by an array of size  $I$ , with values  $V$  or  $0$ . Hence, we have a discretized  
 972 version of the asset transfer object, and there exists a bijection between the continuous setup  
 973 and the discretized setup.

974 We use the discrete version of the Asset Transfer object in the following to reason about  
 975 Anonymous Asset transfers. Specifically, a transfer in the tokenized version for a value of  $kV$   
 976 consists of  $k$  TRANSFER operations, each transferring a token of value  $V$ .

### 977 Anonymity set

978 Let  $S$  be a set of actors. We define "anonymity" as the fact that, from the point of view of  
 979 an observer,  $o \notin S$ , the action,  $v$ , of an actor,  $a \in S$ , cannot be distinguished from the action  
 980 of any other actor,  $a' \in S$ . We call  $S$  the anonymity set of  $a$  for the action  $v$  [36].

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

### 998 The Anonymous Asset-Transfer object type

999 We give a new definition of the Asset-Transfer object type that takes anonymity into account.  
 1000 The first difference between a Pseudonymous Asset Transfer object type and an anonymous  
 1001 one is the absence of a BALANCE operation. The wallet owner can compute the balance  
 1002 of its own wallet using a LOCALBALANCE function that is not part of the distributed

1003 object. The TRANSFER operation is also slightly modified. Let us consider a sender that  
 1004 wants to transfer a token  $T_O$  to a recipient. The recipient creates a new token  $T_R$  with the  
 1005 associated cryptographic offline proofs (in practice,  $T_R$  can be created by the sender using  
 1006 the public key of the recipient). Specifically, it associates it with a private key. This private  
 1007 key is known only to the recipient: its knowledge represents, in fact, the possession of the  
 1008 token. Prior to the transfer operation, the recipient sends token  $T_R$  to the sender. The  
 1009 sender destroys token  $T_O$  and activates token  $T_R$ . The destruction prevents double spending,  
 1010 and the creation makes it possible to transfer the token to a new owner while hiding the  
 1011 recipient's identity. Furthermore, this process of destruction and creation makes it possible  
 1012 to unlink the usages of what is ultimately a unique token.

1013 Each agent maintains a local wallet that contains the tokens (with the associated offline  
 1014 proofs) owned by the agent. The owner of a wallet  $w$  can invoke TRANSFER operations  
 1015 using any of the processes in  $\mu(w)$ . A transfer carried out from a process  $p$  for wallet  $w$  is  
 1016 associated with an anonymity set  $\mathcal{AS}_p^w$  of size equal to the number of wallets associated  
 1017 with process  $p$ :  $|\mathcal{AS}_p^w| = \sum_{i \in \mathcal{W}} \mu(i)[p]$ . The setup with the maximal anonymity set for  
 1018 each transaction is an Anonymous Asset Transfer object where each wallet can perform  
 1019 a TRANSFER operation from any process: i.e.,  $\mu(i) = \{1\}^{|\Pi|}, \forall i \in \mathcal{W}$ . The token-based  
 1020 Anonymous Asset Transfer object type is defined as follows:

1021 ► **Definition 11.** *The Anonymous Asset Transfer object type supports only one operation:*  
 1022 *the TRANSFER operation. It is defined for a set  $\Pi$  of processes and a set  $\mathcal{W}$  of wallets. An*  
 1023 *account is defined by the amount of tokens it controls at time  $t$ . Each account is initially*  
 1024 *attributed an amount  $v_0$  of tokens. We define a map  $\mu : \mathcal{W} \rightarrow \{0, 1\}^{|\Pi|}$  which associates*  
 1025 *each wallet to the processes that can invoke TRANSFER on behalf of these wallets. When*  
 1026 *considering a TRANSFER( $T_O, T_R$ ) operation,  $T_O$  is the cryptographic material of the initiator*  
 1027 *that proves the existence of a token  $T$ , and  $T_R$  is the cryptographic material produced by the*  
 1028 *recipient used to create a new token. The TRANSFER operation respects three properties:*

- 1029 ■ (Termination) *The TRANSFER operation always returns if it is invoked by a correct*  
 1030 *process.*
- 1031 ■ (TRANSFER Validity) *A TRANSFER( $T_O, T_R$ ) operation invoked at time  $t$  is valid if:*
  - 1032 ■ (Existence) *The token  $T_O$  already existed before the transaction, i.e., either it is one of*  
 1033 *the tokens initially created, or it has been created during a valid TRANSFER( $T'_O, T_O$ )*  
 1034 *operation invoked at time  $t' < t$ .*
  - 1035 ■ (Double spending prevention) *No TRANSFER( $T_O, T'_R$ ) has been invoked at time*  
 1036  *$t'' < t$ .*
- 1037 ■ (Anonymity) *A TRANSFER( $T_O, T_R$ ) invoked by process  $p$  does not reveal information*  
 1038 *about the owner  $w$  and  $w'$  of  $T_O$  and  $T_R$ , except from the fact that  $w$  belongs to the*  
 1039 *anonymity set  $\mathcal{AS}_p^w$ .*

1040 Let us extract knowledge from this definition. The TRANSFER validity property implies  
 1041 that the wallet owner can provide existence and non-double-spending proofs to the network.  
 1042 It implies that any other owner in the same anonymity set and with the same cryptographic  
 1043 material (randomness and associated element) can require the transfer of the same token.

1044 We know the material required to produce a TRANSFER proof is stored in the wallet.  
 1045 Furthermore, we can assume that all the randomness used by a given wallet owner is produced  
 1046 by a randomness Oracle that derives a seed to obtain random numbers. Each seed is unique  
 1047 to each wallet. We assume the numbers output by an oracle seems random to an external  
 1048 observer, but two processes that share the same seed will obtain the same set of random  
 1049 numbers in the same order.

1050 Finally, a transaction must be advertised to other processes and wallet owners via the  
 1051 TRANSFER operation. Therefore, proofs of transfer are public. We know these proofs are  
 1052 deterministically computed thanks to our deterministic random oracle model. Furthermore,  
 1053 only one sender and recipient are associated with each transfer operation. Therefore, the  
 1054 public proof cryptographically binds (without revealing them) the sender to the transaction.  
 1055 Hence, the public proof is a cryptographic commitment, which can be opened by the sender  
 1056 or any other actor who knows the same information as the sender.

1057 In order to study the consensus number of this object, we consider that wallet owners can  
 1058 share their cryptographic material with the entire network, thereby giving up their anonymity.  
 1059 This would not make any sense in an anonymous system, but it represents a valuable tool to  
 1060 reason about the consensus number of the object. This sharing process can be implemented  
 1061 by an atomic register (and therefore has no impact on the consensus number, as we discuss  
 1062 later).

1063 Processes can derive the sender's identity from the shared information using a local  
 1064 "uncommit" function. The "uncommit" function takes as input an oracle, a random seed,  
 1065 token elements, and an "on-ledger" proof of transfer of a token and outputs a wallet owner  
 1066 ID if the elements are valid. Otherwise, it outputs  $\emptyset$ .

## 1067 B.2 Consensus number of the Anonymous Asset-Transfer object type

### 1068 Lower bound

1069 Algorithm 4 presents an algorithm that implements a  $k$ -consensus object, using only  $k$ -  
 1070 Anonymous Asset Transfer objects and SWMR registers. The  $k$  in  $k$ -Anonymous Asset  
 1071 Transfer object refers here to the size of the largest  $\mu(w), \forall w \in \mathcal{W}$ . The signature of the  
 1072 TRANSFER operation is slightly modified from the one presented in the previous section.  
 1073 The sent token is a randomized token produced using the original token  $TokenMat$  and a  
 1074 random number. Thus, the  $T_O$  used is the result of the function  $randomize(TokenMat, O,$   
 1075  $seed)$  where  $O$  is a randomness oracle that uses the seed to provide random numbers. The  
 1076 oracle and the seed are shared by all the processes in the system. In Algorithm 4, the  
 1077 receiving token  $T_R^p$  is a dummy token that is never used.

<p><b>Shared variables:</b>          AT <math>\leftarrow</math> <math>k</math>-Anonymous-AT object, initialized with <math>k</math> wallets,              each one of the <math>k</math> wallets possesses the elements              necessary to transfer one shared token.          RM-LEDGER <math>\leftarrow</math> Atomic Snapshot object, initially <math>\{\emptyset\}^k</math>;          V-LED <math>\leftarrow</math> Atomic Snapshot object, initially <math>\{\emptyset\}^k</math>;          O <math>\leftarrow</math> A random oracle;          TokenMat <math>\leftarrow</math> secret associated with a unique token;</p> <p><b>Local variables:</b>          seed <math>\leftarrow</math> random number;  <math>T_R^p \leftarrow</math> receiving token;</p>	<p><b>Operation PROPOSE(<math>v</math>) is:</b>          1: RM-LEDGER[p].update(seed, <math>p</math>);          2: V-LED[p].update(<math>v, p</math>);          3: <math>T_O \leftarrow randomize(TokenMat, O, seed)</math>;          4: res <math>\leftarrow</math> AT.transfer(<math>T_O, T_R^p</math>);          5: RML <math>\leftarrow</math> RM-LEDGER.snapshot();          6: VL <math>\leftarrow</math> V-LED.snapshot();          7: <b>For</b> <math>i</math> in <math>\{1, \dots, k\}</math> <b>do</b>:          8:     <b>If</b> uncommit(O, RML[<math>i</math>], TokenMat, res) <math>\neq \emptyset</math> <b>then</b>:          9:         <b>Return</b> VL[<math>i</math>];          10: <b>Return</b> False;</p>
---	---

■ **Algorithm 4** Implementation of a  $k$ -consensus object using  $k$ -Anon-AT objects

1078 ► **Theorem 12.** *Algorithm 4 wait-free implements  $k$ -consensus.*

1079 **Proof.** Let us fix an execution  $E$  of the algorithm. Each PROPOSE operation only requires  
 1080 a finite number of AT-transfer operations and a finite number of accesses to Atomic Snapshot  
 1081 objects. Both objects are assumed to be atomic. The number of wallet owners is finite.  
 1082 Therefore, each process finishes the invocation of PROPOSE in a finite number of its own  
 1083 steps. Let  $H$  be the history of  $E$ . We define  $\bar{H}$  the completion of  $H$ , where:

## 39:28 The Synchronization Power of Access Control Objects

1084 ■ Any invocation of PROPOSE in  $H$  which does not reach line 4 is completed with the  
1085 line "return False";

1086 ■ Any invocation of PROPOSE in  $H$  that reaches line 4 is completed with the lines 5 to 9.  
1087 We call operations that return false failed operations. The other operations are called  
1088 successful operations. We verify that the completed history  $\bar{H}$  follow the specification of a  
1089  $k$ -consensus object:

1090 ■ Any process that invokes a failed PROPOSE in  $\bar{H}$  is a faulty process. The fact that this  
1091 process cannot decide on a value does not impact the validity, the agreement, or the  
1092 termination properties.

1093 ■ Any invocation of a successful PROPOSE operation in  $\bar{H}$  returns the value proposed by  
1094 the unique process that successfully transferred the token associated with TokenMat. If  
1095 a process reaches line 5 at time  $t$ , then a unique TRANSFER succeeded at time  $t' < t$ .  
1096 Hence, the uncommit operation returns 1 at least and at most once, ensuring the Validity  
1097 property. The agreement is ensured because no two processes can spend the same coin.  
1098 Furthermore, the coin associated with TokenMat is transferred to a dummy account which  
1099 cannot invoke TRANSFER operations. Therefore, the agreement property is verified.

1100 All invocations of the PROPOSE operation in  $\bar{H}$  follow the specification, and the algorithm  
1101 presented in Algorithm 4 is wait-free. In conclusion, the proposed implementation of a  
1102  $k$ -consensus object is linearizable. ◀

1103 ▶ **Corollary 13.** *The consensus number of the Anon-AT object is at least  $k$ .*

### 1104 Upper Bound

1105 We give an implementation of the Anon-AT object using only Atomic Snapshot objects,  
1106 DenyList objects, and AllowList objects. Each wallet owner can request a TRANSFER  
1107 operation to  $k$  different processes. The proposed implementation uses disposable tokens that  
1108 are either created at the initialization of the system or during the transfer of a token. When  
1109 a token is destroyed, a new token can be created, and the new owner of the token is the only  
1110 one to know the cryptographic material associated with this new token. In the following,  
1111 we use the zero-knowledge version of the DenyList and AllowList object types, where all  
1112 set-(non-)membership proofs use a zero-knowledge setup. In addition, we use an AllowList  
1113 object to ensure that a token exists (no ex-nihilo creation), and we use a DenyList object to  
1114 ensure that the token is not already spent (double-spending protection).

1115 The underlying cryptographic objects used are out of the scope of this paper. However,  
1116 we assume our implementation uses the ZeroCash [18] cryptographic implementation, which  
1117 is a sound anonymous asset transfer protocol. More precisely, we will use a high-level  
1118 definition of their off-chain functions. It is important to point out that using the ZeroCash  
1119 implementation, it is possible to transfer value from a pseudonymous asset transfer object to  
1120 an anonymous one using a special transaction called "Mint". To simplify our construction,  
1121 we assume that each wallet is created with an initial amount of tokens  $v_0$  and that our object  
1122 does not allow cross-chain transfers. We, therefore, have no "Mint" operation.

1123 ZeroCash uses a TRANSFER operation called *pour* that performs a transfer operation  
1124 destroying and creating the associated cryptographic material. Here, we use a modified  
1125 version of *pour* which does not perform the transfer or any non-local operation. It is a black-  
1126 box local function that creates the cryptographic material required prove the destruction  
1127 of the source token ( $T_O$ ) and the creation of the destination one ( $T_R$ ). Our modified *pour*  
1128 function takes as input the source token, the private key of the sender ( $sk_s$ ), and the public

1129 key of the recipient ( $\text{pk}_r$ ):  $\text{pour}(T_O, \text{pk}_r, \text{sk}_s) \rightarrow tx$ ,  $tx$  being the cryptographic material that  
 1130 makes it possible to destroy  $T_O$  and create  $T_R$ .

1131 There might be multiple processes transferring tokens concurrently. Therefore, we define  
 1132 a deterministic local function  $\text{ChooseLeader}(\mathcal{A}, tx)$ , which takes as input any set  $\mathcal{A}$  and a  
 1133 transaction  $tx$ , and outputs a single participant  $p$  which invoked  $\text{BL.PROVE}(tx)$ .<sup>7</sup>

<p><b>Shared variables:</b></p> <p>DL <math>\leftarrow</math> <math>k</math>-DenyList object, initially <math>(\emptyset, \emptyset)</math>;</p> <p>AL <math>\leftarrow</math> AllowList object, initially <math>(\{\{token_{(i,j)}\}_{i=1}^t\}_{j=1}^k}, \emptyset)</math></p> <p><b>Operation</b> <math>\text{TRANSFER}(T_O, \text{pk}_r, \text{sk}_s)</math> <b>is:</b></p> <p>1: <math>tx \leftarrow \text{Pour}(T_O, \text{pk}_r, \text{sk}_s)</math></p> <p>2: <b>If</b> <math>\text{verify}(tx)</math> and <math>tx \in \text{AL}</math> and <math>tx \notin \text{DL}</math> <b>then:</b></p> <p>3:   AL.PROVE(<math>tx</math>);</p> <p>4:   DL.PROVE(<math>tx</math>);</p>	<p>5:   DL.APPEND(<math>tx</math>);</p> <p>6:   <b>Do:</b></p> <p>7:     <math>\text{ret} \leftarrow \text{DL.PROVE}(tx)</math>;</p> <p>8:     <b>While</b> <math>\text{ret} \neq \text{false}</math>;</p> <p>9:     <b>If</b> <math>\text{ChooseLeader}(\text{DL.READ}(), tx.T_R)=p</math> <b>then:</b></p> <p>10:       AL.append(<math>tx.T_R</math>);</p> <p>11:       <b>Return</b> <math>tx.T_R</math>;</p> <p>12:     <b>Return</b> <math>\text{False}</math>;</p>
---	---

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

1134 ► **Theorem 14.** *Algorithm 5 wait-free implements an Anon-AT object.*

1135 **Proof.** Let us fix an execution  $E$  of the algorithm.  $E$  only uses DenyList objects and SWMR  
 1136 registers (AllowList objects have consensus number 1 and can be implemented using SWMR  
 1137 registers). Each TRANSFER operation only requires a finite number of DenyList.PROVE  
 1138 and DenyList.APPEND operations, which are assumed atomic. Each process finishes the  
 1139 invocation of TRANSFER in a finite number of its own steps. Let  $H$  be the history of  $E$ .  
 1140 We define  $\bar{H}$  the completion of  $H$ , where:

- 1141 ■ Any pending invocation of TRANSFER in  $H$  that did not reached line 11 is completed  
 1142 with the line "return False";
- 1143 ■ Any pending invocation of PROPOSE in  $H$  that reached line 11 is completed with line  
 1144 12.

1145 We call operations that return false failed operations. The other operations are called  
 1146 successful operations. In the following, we analyze if the completed history  $\bar{H}$  follows the  
 1147 Anonymous Asset-Transfer object specification:

- 1148 ■ Any process that fails in an invocation of a TRANSFER operation in  $\bar{H}$  is a faulty  
 1149 process. It cannot transfer money or create or double-spend a token. Therefore, it does  
 1150 not contradict the properties of the Anonymous Asset Transfer object. This faulty process  
 1151 can lose tokens (by destroying it and not transferring it), but because we assume a crash  
 1152 is definitive, all its tokens are lost anyway.
- 1153 ■ A  $\text{TRANSFER}(T_O, T_R)$  operation invoked by a process  $p$  in  $\bar{H}$  that succeeds proves to  
 1154 the network that the token  $T_O$  exists and was not spent. After creating the cryptographic  
 1155 material for the new token and for destroying the old one, the operation verifies the  
 1156 correctness of the material and the validity of the old and new tokens (line 2). The prove  
 1157 operation in line 3 necessarily succeeds as  $tx \in \text{AL}$ . The prove in line 4 may instead fail  
 1158 if another process is performing a concurrent transfer operation for the same  $T_O$ . This

<sup>7</sup> In reality, the signature of chooseLeader would be more complicated as the function needs  $T_O, \text{pk}_r, \text{sk}_s$  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  $\text{sk}_s$ . We omit these details in the following to simplify the presentation.

## 39:30 The Synchronization Power of Access Control Objects

1159 potential conflict is resolved by the ChooseLeader function, which takes as input the list  
1160 of successful prove operations on the DenyList. The determinism of the ChooseLeader  
1161 function and the agreement provided by the READ operation ensure that all processes  
1162 add the same  $tx.T_R$  at line 10 even if multiple processes issue concurrent conflicting  
1163 TRANSFER operations for the same  $T_O$ . The previous statement enforces double-  
1164 spending prevention. The non-creation property is ensured by the PROVE operation  
1165 conducted on the AllowList on line 3. Finally, anonymity is enforced using the ZKP  
1166 version of the AllowList and the DenyList objects.  
1167 All invocations of the TRANSFER operation in  $\bar{H}$  follow the specification of the Anonymous  
1168 Asset Transfer object. In conclusion, the proposed algorithm is a wait-free implementation  
1169 of the Anonymous Asset Transfer object. ◀

1170 ▶ **Corollary 15.** *The consensus number upper bound of a  $k$ -anon-AT object is  $k$ . Using*  
1171 *this corollary and the previous one, we further deduct that  $k$ -anon-AT object has consensus*  
1172 *number  $k$ .*

### 1173 B.3 From the size of the anonymity set to the consensus number

1174 A relation exists between the size of the anonymity set of an AAT object and the consensus  
1175 number of this object. The goal of this section is to explore this relationship.

1176 Let the relation between wallets and processes be represented by a graph. In the graph,  
1177 vertices are either wallets or processes. Edges only link wallets to processes, and represent  
1178 the fact that a process can order a TRANSFER operation on behalf on the associated wallet.  
1179 Then, on the one hand, the consensus number  $k$  of a  $k$ -anon-AT is the maximum multiplicity  
1180 of any wallet in the graph—i.e.  $k = \max_{w \in W} |\mu(w)|$ . On the other hand, the size of the  
1181 anonymity set for a wallet  $w$  and a given TRANSFER operation invoked by a process  $p$  is  
1182 the multiplicity of this process in the graph. Furthermore, the size of the anonymity set for a  
1183 wallet  $w$  and any TRANSFER operation is the minimum multiplicity of a process associated  
1184 to  $w$ .

## 1185 C E-vote system implementation using a DenyList object

1186 In this section, we show that DenyList objects can provide upper bounds on the consensus  
1187 number of a complex objects. As an example, we study an e-vote system. An e-vote system  
1188 must comply with the same properties as an "in-person" voting. An "in-person" voting  
1189 system must ensure four security properties, two for the organizers and two for the voters.  
1190 First, the organizers of the vote must ensure that each person who votes has the right to do  
1191 so. Second, each voter must vote only once. Third, a voter must verify that their vote is  
1192 considered in the final count. Fourth, an optional property is voter anonymity. Depending  
1193 on the type of vote, the voter may want to hide their identity.

1194 We want to design a distributed e-vote system, where a voter can submit their vote to  
1195 multiple different voting servers—while ensuring the unicity of their vote. We assume each  
1196 server is a process of the distributed infrastructure. The voters act as clients, submitting  
1197 vote requests to the servers. We assume the "right to vote" property is ensured using tokens.  
1198 The Token is a one-time-used pseudonym that links a vote to a voter. Users obtains their  
1199 Tokens from an issuer. All the voting servers trust this issuer. Neither the voters nor the  
1200 issuer has access to the e-vote object—except if one of the actors assumes multiple roles.  
1201 Using these specifications, we define the e-vote object type as follows:

1202 ► **Definition 16.** *The e-vote object type supports two operations: VOTE(Token, v) and*  
 1203 *VOTE-COUNT(). Token is the voter's identifier, and v is the ballot. Moreover, these*  
 1204 *operations support three mandatory properties and one optional property:*

- 1205 1. *(Vote Validity) A VOTE(Token, v) invoked at time t is valid if:*
  - 1206 ■ *Token is a valid token issued by an issuer trusted by the voting servers; and*
  - 1207 ■ *No valid VOTE(Token, v') operation was invoked at time t' < t, where v' ≠ v.*
- 1208 2. *(VOTE-COUNT validity) A VOTE-COUNT() operation returns the set of valid VOTE*  
 1209 *operations invoked.*
- 1210 3. *(optional - Anonymity) A Token does not link a vote to its voter identity, even if the*  
 1211 *voting servers and the issuers can collude.*

1212 In the following, we analyze an e-vote system based on signatures. The issuer issues a  
 1213 signature to the voter. The message of the signature is a nonce. The tuple (signature, nonce)  
 1214 is used as a token. When a voter issues a vote request, the server verifies the signature's  
 1215 validity and proceeds to vote if the signature is valid.

1216 Adding anonymity to a signature-based e-vote system can be easily achieved using blind  
 1217 signatures [25]. A blind signature algorithm is a digital signature scheme where the issuer  
 1218 does not learn the value it signs. Because the signed value is usually a nonce, the issuer  
 1219 does not need to verify the value—a value chosen maliciously will not grant the voter more  
 1220 privileges than expected.

1221 Formally, a Blind signature algorithm is defined by the tuple (Setup, Commit, Sign,  
 1222 Uncommit, Verify), where Setup creates the common values of the scheme (generators, shared  
 1223 randomness, etc...) and secret/public key pairs for all issuers. The public keys are shared  
 1224 with all the participants in the system. Commit is a commitment scheme that is hiding and  
 1225 binding; it outputs a commitment to a value—the nonce—randomly chosen by the user. The  
 1226 Sign algorithm takes as input a commitment to a nonce and the secret key of an issuer. It  
 1227 outputs a signature on the commitment. The Uncommit algorithm takes a signature on a  
 1228 commitment and the issuer's public key as input and outputs the same signature on the  
 1229 uncommitted message (the original message). Finally, the Verify algorithm outputs 1 if the  
 1230 uncommitted signature is a valid signature by an issuer on the message  $m$ .

1231 Algorithm 6 provides a wait-free implementation of an e-vote system using any signature  
 1232 scheme, one  $k$ -DenyList object, and one SWMR atomic snapshot object. Here, the value of  $k$   
 1233 corresponds to the number of voting servers, and  $k=|\Pi_V| = |\Pi_M|$ . The idea of the algorithm  
 1234 is to use the APPEND operation to state that a token has already been used to cast a  
 1235 vote. In order to obtain a wait-free implementation, we use a helper value [15] stored in the  
 1236 Atomic Snapshot object AS-prevote. The vote operation is conducted as follows: the voting  
 1237 server  $V$  communicates the vote it will cast in the AS-prevote object. Then,  $V$  conducts a  
 1238 PROVE(Token) operation to prove that the Token has not yet been used. Then,  $V$  invokes  
 1239 an APPEND(Token) operation and waits until the APPEND is effective—the do-while loop  
 1240 in line 8 to 10. Finally,  $V$  uses the READ operation to verify that it is the only process  
 1241 that proposed a vote for this specific Token. If it is the case, the vote is added to the vote  
 1242 array—the AS-vote array. Otherwise, the vote is added to the vote array only if the other  
 1243 servers that voted using the same Token proposed the same ballot in line 4.

1244 Other implementations can be proposed in the case of two concurrent transactions  
 1245 requested by the same voter—to different servers—with different values. For example, it is  
 1246 possible to modify the algorithm presented in Algorithm 6 using a deterministic function  
 1247 to choose one value among all the potential votes. This modification does not impact the  
 1248 properties of this implementation.

## 39:32 The Synchronization Power of Access Control Objects

1249 We now provide an informal proof of the linearizability of this implementation. The  
 1250 anti-flickering property of the  $k$ -DenyList object ensures the termination of the while loop.  
 1251 Therefore, the implementation is wait-free. The same property ensures that the *vote-values*  
 1252 variables are the same for all voters with the same Token, thus ensuring the unicity of the  
 1253 vote. The proof of authorization of the vote is ensured by the signature verification in line  
 1254 1. The anonymity property is also fulfilled if a blind signature scheme is used. Therefore,  
 1255 the use of anonymous DenyList objects is not required. Hence, we can conclude that the  
 1256 consensus number of the  $k$ -DenyList object type is an upper bound on the consensus number  
 1257 of an e-vote system.

```

Shared variables:
  k-dlist  $\leftarrow$   $k$ -DenyList object;
  AS-prevote  $\leftarrow$  Atomic Snapshot object, initially  $\{\emptyset\}^k$ 
  AS-vote  $\leftarrow$  Atomic Snapshot object, initially  $\{\emptyset\}^k$ 
Operation VOTE( $(signature, pk, token, v)$ ) is:
1: If Verify( $signature, token, pk$ )  $\neq 1$  then:
2:   Return false;
3: AS-prevote.update( $(token, v), p$ );
4: ret  $\leftarrow$  k-dlist.PROVE( $token$ )
5: If ret = false then:
6:   Return false;
7: k-dlist.APPEND( $token$ );
8: Do:
9:   ret  $\leftarrow$  k-dlist.PROVE( $token$ );
10: While ret  $\neq$  false;

11: votes  $\leftarrow$  k-dlist.READ();
12: client-votes  $\leftarrow$  all values in votes where token is  $token$ ;
13: voters  $\leftarrow$  all processes in client-votes;
14: vote-values  $\leftarrow$  all values in AS-prevote
    where token is  $token$  and processes that added the value are in voters;
15: If vote-values =  $\{v\}^l, \forall l \geq 1$  then:
16:   previous-votes  $\leftarrow$  AS-vote.SNAPSHOT()[ $p$ ];
17:   AS-vote.UPDATE(previous-votes  $\cup$  ( $token, v, voters$ ),  $p$ );
18:   Return true;
19: Else return false;
Operation VOTE-COUNT() is:
20: votes  $\leftarrow$  all values in AS-vote.Snapshot().
    Only one occurrence of each tuple ( $token, v, voters$ ) is kept;
21: Return votes;
  
```

■ **Algorithm 6** Implementation of an e-vote object using one  $k$ -DenyList object and Atomic Snapshots

1258 It is also possible to build a wait-free implementation of a  $k$ -consensus object using one  
 1259 e-vote object. This implementation is presented in Algorithm 7. The idea of this algorithm  
 1260 is that each process will try to vote with the same Token. Because only one vote can be  
 1261 accepted, the e-vote object will only consider the first voter. Ultimately, every process will  
 1262 see the same value in the vote object. This value is the result of the consensus.

1263 Each invocation is a sequence of a finite number of local operations and e-vote object  
 1264 accesses, which are assumed atomic. Therefore, each process terminates the PROPOSE  
 1265 operation in a finite number of its own steps. A vote can only be taken into account if it was  
 1266 proposed by some process, which enforces the validity property. The agreement property  
 1267 comes from the unicity of the vote. Finally, the non-trivial property of the consensus object  
 is ensured because the decided value is any value  $v$  chosen by the winning process.

```

Shared variables:
  vote-obj  $\leftarrow$  e-vote object where the only authorized token is 0;
   $\sigma \leftarrow$  signature on 0 by a trusted issuer;
Operation PROPOSE( $v$ ) is:
1: vote-obj.VOTE( $(\sigma, 0), v$ );
2:  $\{(winner\text{-}token, value)\} \leftarrow$  vote-obj.VOTE-COUNT();
3: Return value;
  
```

■ **Algorithm 7** Implementation of a  $k$ -consensus object using one e-vote object

1268 The consensus number of a blind-signature-based e-vote system is bounded on one side  
 1269 by the consensus number of a  $k$ -consensus object and the other side by the consensus number  
 1270 of a  $k$ -DenyList object. Hence, the blind-signature-based e-vote object type has consensus  
 1271 number  $k$ .  
 1272