Asynchronous BFT Asset Transfer: Quasi-Anonymous, Light, and Consensus-Free

Timothé Albouy, Emmanuelle Anceaume, Davide Frey, Mathieu Gestin, Arthur Rauch, Michel Raynal, François Taïani

Asynchronous BFT Asset Transfer: Quasi-Anonymous, Light, and Consensus-Free

Timothé Albouy
Univ Rennes, Inria, CNRS, IRISA, 35042 Rennes, France

Emmanuelle Anceaume
Univ Rennes, Inria, CNRS, IRISA, 35042 Rennes, France

Davide Frey
Univ Rennes, Inria, CNRS, IRISA, 35042 Rennes, France

Mathieu Gestin
Univ Rennes, Inria, CNRS, IRISA, 35042 Rennes, France

Arthur Rauch
Univ Rennes, Inria, CNRS, IRISA, 35042 Rennes, France

Michel Raynal
Univ Rennes, Inria, CNRS, IRISA, 35042 Rennes, France

François Taïani
Univ Rennes, Inria, CNRS, IRISA, 35042 Rennes, France

Abstract

This article introduces a new asynchronous Byzantine-tolerant asset transfer system (cryptocurrency) with three noteworthy properties: quasi-anonymity, lightness, and consensus-freedom. Quasi-anonymity means no information is leaked regarding the receivers and amounts of the asset transfers. Lightness means that the underlying cryptographic schemes are *succinct*, and each process only stores data polylogarithmic in the number of its own transfers. Consensus-freedom means the system does not rely on a total order of asset transfers. The proposed algorithm is the first asset transfer system that simultaneously fulfills all these properties in the presence of asynchrony and Byzantine processes. To obtain them, the paper adopts a modular approach combining a new distributed object called agreement proofs and well-known techniques such as vector commitments, universal accumulators, and zero-knowledge proofs. The paper also presents a new non-trivial universal accumulator implementation that does not need knowledge of the underlying accumulated set to generate (non-)membership proofs, which could benefit other crypto-based applications.

2012 ACM Subject Classification Theory of computation → Distributed algorithms

Keywords and phrases Agreement proof, Anonymity, Asset transfer, Asynchrony, Byzantine process, Consensus-freedom, Cryptography, Deterministic algorithm, Distributed computing, Fault-tolerance, Light algorithm, Modularity, Message-passing communication, Non-interactive, Quorum, Succintness, Vector commitment, Universal accumulator, Zero-knowledge proof.

Funding This work was partially supported by the French ANR projects ByBloS (ANR-20-CE25-0002-01) and PriCLeSS (ANR-10-LABX-07-81), devoted to the modular design of building blocks for large-scale Byzantine-tolerant multi-users applications. This work was also partially funded by the SOTERIA project. SOTERIA has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 101018342. This content reflects only the author's view. The European Agency is not responsible for any use that may be made of the information it contains.

1 Introduction

Existing decentralized *asset transfer* systems[1] face a range of challenges pertaining to their privacy guarantees (*e.g.*, confidentiality), performance (*e.g.*, throughput, latency), costs (*e.g.*, in term of storage and computation), and scalability. Several solutions have been proposed to address those problems individually, but implementing a solution to one problem often negatively impacts other aspects of the system. In this paper, we propose a new asset transfer system that balances efficiency and user privacy, by guaranteeing the following desirable properties.

1. **Quasi-anonymity:** the amount and the receiver's identity of every asset transfer remain hidden from the adversary.
2. **Lightness:** all cryptographic schemes used by our solution are *succinct* (as defined in [6]), and the storage cost incurred by each process p_i is polylogarithmic in $|S_i|$ for a fixed security parameter λ (which we denote $O_\lambda(\text{polylog } |S_i|)$), where S_i is the set of all transfers of p_i.
3. **Consensus-freedom:** our algorithm does not rely on consensus or a total order of asset transfers.

As we show next, several endeavors have been conducted to create asset transfer systems that meet the quasi-anonymity, lightness, and consensus-freedom properties.

Anonymous and quasi-anonymous asset transfer systems. Anonymous Asset Transfer (AAT) systems have been studied since the introduction of Bitcoin [39] and can be categorized as either *token-based* or *account-based*. Token-based AAT uses two types of Zero Knowledge Proofs (ZKP): ZKP of token existence and ZKP that a token has not been spent yet. To transfer a token, the sender proves that it possesses a token that has not yet been added to a list of spent tokens. It then "destroys" this token (by adding it to the list of spent coins) and creates a new one belonging to the transfer's receiver. Zcash [29] is an example of this first category. Account-based AAT is mainly represented by Quisquis [19] and Zether [11]. They only store commitments to different accounts. For each transfer, the sender proves that its account has enough funds, and full anonymity is provided by randomly selecting all or a subset of the system accounts. The system introduced in this article is account-based (it does not use a list of spent coins).

All previously cited systems [29, 19, 11] provide full anonymity (*i.e.*, they ensure sender and receiver anonymity), but they also require consensus. It has been shown in [22] that fully anonymous asset transfer requires consensus to be implemented. To achieve consensus freedom despite this impossibility, it is possible to weaken the anonymity guarantees of asset transfer, as done in Zef [4] or the present paper. We call our weakened version of AAT *quasi-anonymous asset transfer* (*QAAT*).

Light asset transfer systems. In this paper's context, *lightness* means that the storage per process is polylogarithmic in the number of its own transfers, and all underlying cryptographic schemes are *succinct*.

A cryptographic proving scheme is succinct if and only if its proof size and verification time are at most polylogarithmic (and thus sublinear) in the "size of the problem" [6]. This "size of the problem" depends on the scheme. For instance, it may refer to the number of signatures

[1] The terms *money* or *currency* are sometimes discouraged when talking about decentralized payment systems, as they do not always satisfy the 3 properties of money, *i.e.*, unit of account, medium of exchange, and store of value [37].

to aggregate in an aggregated signatures scheme [8], or to the arithmetic circuit size used in a Succinct Non-interactive Argument scheme (SNARG). Intuitively, succinctness captures the fact that a practicable system has to use cryptographic implementations that are themselves practicable (*i.e.*, their computational and storage costs do not become prohibitively high with time).

Reducing the local storage cost of decentralized asset transfer systems has been a research challenge since the advent of Bitcoin [39], which introduced the notion of *light clients* (a.k.a. *Simplified Payment Verification*, or *SPV*), *i.e.*, clients storing only a subset of the whole system data (*e.g.*, block headers and Merkle proofs in SPV). But as efficient as they are for reducing the load of light clients, such solutions [12, 39] do not completely remove the need for *validators* (the processes verifying the transfers) to store the entire set of transfers. To overcome this limitation, alternative approaches seek to reduce the storage cost of validators to make it at most sublinear in the system's total number of asset transfers [32, 31]. For example, the Mina system [9] leverages SNARKs (Succinct Non-interactive Arguments of Knowledge) to obtain a succinct blockchain whose storage cost for validators is proportional to the number of accounts in the system.

The system presented in this paper goes one step further: to our knowledge, it is the first light asset transfer system, *i.e.*, it is succinct and has a storage cost for each correct process p_i that is polylogarithmic in the number of asset transfers of p_i.[2]

Consensus-free asset transfer systems. Contrary to a common belief, asset transfer in the presence of (Byzantine) faults does not require strong agreement such as consensus or, equivalently, total order [27, 28].[3] More formally, the only condition required to prevent double-spending in a payment system is for the processes to agree on the order in which transfers are issued from each individual account: if some account issues two conflicting transfers spending the same funds, the rest of the network, and especially the corresponding creditors, have to settle on which transfer (if any) is correct. This relaxed, per-account transfer ordering can be obtained by communication primitives weaker than consensus, such as *reliable broadcast* [10].[4]

Following the seminal result of [27], several papers have proposed payment systems based on reliable broadcast only, namely AFRT20 [1], Astro [14], and FastPay [3]. Astro has been later extended to permissionless environments with Pastro (permissionless Astro) [35], by combining weighted quorums coupled and proof-of-stake. Building on FastPay, Zef [4] provides confidential and untraceable transfers, but is not fully consensus-free as it relies on a classical blockchain, which typically requires Nakamoto-style consensus. (For simplicity, we have categorized Zef as consensus-free in Table 1.)

The benefits of leveraging communication primitives with weak ordering (such as reliable broadcast) instead of consensus (total order) when designing an asset transfer system are twofold.

- First, constructing a total order is costly and forces the system's processes to treat all asset transfers in the same sequential order (or block by block in the case of blockchains). This significantly impacts the system's throughput, *i.e.*, the number of asset transfers per

[2] Following the literature, we only consider the internal state and the cryptographic schemes used by our system when analyzing its storage cost. In particular, the cost induced by the network routing protocol is ignored.

[3] More specifically, consensus-freedom applies to asset transfer systems in which each account is owned by a single process.

[4] This is only the case for asset transfer systems where each account is owned by a single process. In systems where accounts can have multiple owners, consensus is needed.

System	Quasi-anon.	Succinct	Polylog storage	Consensus-free
Zcash [29]	✓(fully anon.)	✓	✗	✗
Mina [9]	✗	✓	✗	✗
Pastro [35]	✗	✓	✗	✓
Zef [4]	✓	✗	✗	✓
This paper	✓	✓	✓	✓

Table 1 Comparison of the properties of notable existing systems.

second (TPS) that can be handled. In contrast, the weak ordering provided by reliable broadcast makes it possible to process asset transfers concurrently, resulting in higher throughput [14].

- Second, building a total order in the presence of asynchrony and failures is bound by a fundamental impossibility (known as the FLP theorem [21]) stating that consensus cannot be implemented deterministically in an asynchronous system prone to process faults. To circumvent this impossibility, consensus-based cryptocurrency systems typically require stronger assumptions, *e.g.*, (partial) synchrony, randomization, or non-trivial cryptography. In comparison, reliable broadcast can be implemented in asynchronous systems even in the presence of Byzantine faults [2, 10, 30]. This makes reliable-broadcast-based asset-transfer systems more resilient because they deterministically guarantee safety and liveness in environments where their consensus-based counterparts cannot.

Combining quasi-anonymity, lightness, and consensus-freedom. Table 1 compares some notable existing asset transfer systems in terms of quasi-anonymity, lightness (made of succinctness and polylog storage), and consensus-freedom. To the best of our knowledge, the algorithm presented in this paper is the first to exhibit all properties while incurring polylog storage per process.

Roadmap. This paper is made up of 6 sections. Section 2 defines the computing model of our system. Section 3 provides the sequential specification of quasi-anonymous asset transfer (QAAT). Section 4 introduces the 4 schemes used in our system (including the novel Agreement Proof abstraction). Section 5 presents our QAAT algorithm and the intuition behind its correctness. Finally, Section 6 concludes the article. Due to space constraints, some developments, such as the proofs of our QAAT algorithm, and implementation examples of Section 4's schemes (among which a new non-trivial implementation universal accumulator implementation), appear in appendices.

2 System Model

Processes. The system comprises n sequential asynchronous processes denoted by $p_1, ..., p_n$. Each process p_i has a unique identity, which is known to other processes. To simplify the presentation without loss of generality, we assume that i is the identity of p_i. Up to $t < n/3$ processes can be Byzantine, where a Byzantine process is a process whose behavior does not follow the code specified by its algorithm [36, 41].[5] Byzantine processes may collude to fool non-Byzantine (a.k.a. correct) processes.

[5] The assumption $t < n/3$ is only needed for the *Agreement Proof* scheme of our system (see Section 4.1).

Network. Processes communicate by exchanging messages through a fully connected asynchronous point-to-point communication network, which is assumed to be reliable (*i.e.*, the network does not corrupt, drop, duplicate, or create messages). Let MSG be a message type and v the value contained in the message. The operation "send MSG(v) **to** p_j" is used for sending, and the callback "**when** MSG(v) **is** received" is used for receiving. For syntactic sugar, processes can also communicate using the macro-operation denoted broadcast MSG(v), that is a shorthand for "**for all** $j \in \{1, \cdots, n\}$ **do** send MSG(v) **to** p_j". When processes use this macro-operation to disseminate a message, we say that this message is *broadcast* and *received*. The macro-operation broadcast MSG(v) is unreliable. For example, if the invoking process crashes during its invocation, an arbitrary subset of processes receives the message MSG(v). Moreover, due to its very nature, a Byzantine process can send conflicting messages without using the macro-operation broadcast. Finally, the processes have access to ra_send/ra_receive operations (for "receiver-anonymous send/receive"), which function like the classical send/receive operations with the additional guarantees that *(i)* the message cannot be read by anyone other than the receiver, and that *(ii)* the receiver remains anonymous from an adversary eavesdropping the network. For instance, these two operations could be implemented by broadcasting an encrypted message to the whole network that only the intended receiver can decrypt, or by using onion routing [43].

Cryptographic schemes. In this paper, we use cryptographic schemes such as secure hash functions, asymmetric signatures, commitments, accumulators, and arguments of knowledge. In the following, we specify these schemes regarding abstract operations and guarantees. We provide concrete implementation examples in Appendix B, detailing how each choice constrains the adversary's power.

Security parameter λ. λ denotes the security parameter of the cryptographic schemes used in our algorithms. There is a trade-off between the security level of our cryptographic schemes (represented by λ) and their computational, communication, and storage costs. $\epsilon(\lambda)$ is a positive number between 0 and 1 that can be made arbitrarily small by increasing λ. Additionally, $O_\lambda()$ denotes the complexity for a fixed λ: for a given expression F, $O_\lambda(F)$ is equivalent to $O(\lambda \times F)$.

Different notions of adversaries. We consider an adversary with a *distributed* and a *cryptographic* aspect. The distributed aspect means that the adversary seeks to compromise the correctness of the system by controlling process faults (Byzantine faults in our case) and the scheduling of messages (asynchrony). The cryptographic aspect means that the adversary also strives to compromise the anonymity properties of the system. In this respect, we consider a *probabilistic polynomial-time* (PPT) adversary, *i.e.*, one that has bounded computational power. It has only access to the information publicly transiting on the network (*i.e.*, the messages communicated using the clear-text primitives send, broadcast, and received), not the messages privately exchanged between pairs of processes using the ra_send/ra_receive operations defined before. We denote our adversary as *Adv*.

Notations. The *invocation* by a process of an operation op with input parameter *in* and output result *out* is denoted op(*in*)/*out*. The "$\star$" symbol means that the corresponding value is unspecified, *e.g.*, $\mathsf{op}_i(\star)$ refers to an invocation of op by p_i with an arbitrary input. A pair made up of a and b is denoted $\langle a, b \rangle$. Table 2 summarizes key notations used throughout this paper.

Notation	Meaning
λ, $\epsilon(\lambda)$, Adv	Security parameter, Negligible real number in $[0; 1]$, Adversary
AT, QAAT	Asset transfer object, Quasi-anonymous asset transfer object
AP, σ, P_A	Agreement proof scheme, Agreement proof, AP predicate
VC, c, o	Vector commitment scheme, Commitment, Opening
UA, A, w, u	Universal accumulator scheme, Accumulator, Membership proof, Non-membership proof
ZKP, π, P_{ZK}	Zero-knowledge proof scheme, Zero-knowledge proof, ZKP predicate
$\mathbb{D}$ (for "data")	Input set (values) of the proving operations of Section 4's schemes
$\mathbb{F}$ (for "finite")	Output set (proofs) of the proving operations of Section 4's schemes
τ:$\langle snd, v, rcv, sn\rangle$	Asset transfer details: sender, amount, receiver, sequence number

Table 2 Key notations used in the paper.

3 Quasi-Anonymous Asset Transfer (QAAT): Sequential Specification

Asset Transfer (AT), and its extension, *Quasi-Anonymous Asset Transfer* ($QAAT$) are derived from the *concurrent* asset transfer specification of AFRT20 [1]. For simplicity, we assume that each account is owned by a single process.

AT operations. A distributed AT object provides processes with two operations:

- $\mathsf{balance}()/v$ returns the balance v of the calling process' account according to its current local vision.
- $\mathsf{transfer}(v, j)/r$ transfers the amount v from the account of the calling process to the account of p_j, returns $r =$ `commit` if the transfer succeeds, $r =$ `abort` otherwise. For simplicity, if $r =$ `commit`, we omit writing the result of the corresponding invocation: $\mathsf{transfer}_i(v, j)/$`commit` can be simply written $\mathsf{transfer}_i(v, j)$.

The above $\mathsf{balance}()$ operation is more restricted than that of AFRT20 [1] as it only returns the balance of the local process, rather than the balances of the entire system. This is needed to ensure confidentiality (since a given process should not be able to read the accounts of other processes). Interestingly enough, this small change also allows us to obtain a sequential specification (instead of a concurrent specification [1]), as shown next.

It is assumed that the account of each process p_i is initialized to a non-negative value denoted $\texttt{init}_i$ (in our QAAT implementation of Algorithm 3, $\texttt{init}_i$ is known only by p_i).

Histories and sequences.

- A correct local history of a correct process p_i, denoted L_i, is a sequence of invocations made by p_i. If an invocation o precedes another invocation o' in L_i, we say that "o precedes o' in the process order of p_i", which is written $o \rightarrow_i o'$.
- A Byzantine local history of a Byzantine process p_i is a sequence of $\mathsf{transfer}_i(v, j)$ invocations that could have been issued by process p_i from the point of view of correct processes. Let us remind that we do not know if Byzantine processes correctly implement the $\mathsf{balance}$ operation, so we omit the $\mathsf{balance}_i()$ invocations in Byzantine local histories. Multiple possible Byzantine local histories for a given Byzantine process may exist as long as the content and order of this sequence are consistent with the local histories of the correct processes. Just like correct local histories, each Byzantine local history L_i induces a total order relation $\rightarrow_i$ between the invocation of L_i.
- A global history H is an array of n local histories, one for each process: $H = [L_1, ..., L_n]$, where L_i is a correct local history if p_i is correct, or a Byzantine local history if p_i is

Byzantine.

- A sequence S of a global history H is a topological sort of H (*i.e.*, a total order respecting all the process orders) that contains all the invocations of every process in H and respects the process order $\rightarrow_i$ of each process p_i. In the following, $\rightarrow_S$ denotes the total order defined by S.

AT-sequence. Given a process ID i and a set of invocations O, the function $total(i, O)$ returns the account balance of p_i resulting from its operation invocations in O (by adding the initial balance of p_i to the funds received by p_i in O, and subtracting the funds sent by p_i in O):

$$total(i, O) = \texttt{init}_i + \left(\Sigma_{\mathsf{transfer}_\star(v,i)\in O}\, v\right) - \left(\Sigma_{\mathsf{transfer}_i(v,\star)\in O}\, v\right).$$

A sequence S of a global history H is an *asset-transfer-sequence* (*AT-sequence*) if and only if:

1. $\forall\ o = \mathsf{balance}_i()/v \in S : v = total(i, \{o' \in S \mid o' \rightarrow_S o\})$;
2. $\forall\ o = \mathsf{transfer}_i(v, j) \in S : v \leq total(i, \{o' \in S \mid o' \rightarrow_S o\})$.

Informally, these conditions mean that the balance operation must return the balance of the process' account when it is invoked in the sequence, and the transfer operation must succeed (*i.e.*, return `commit`) only if the balance of the debtor's account is sufficient. We say that a global history H can be *AT-sequenced* if we can produce an AT-sequence with its invocations.

AT properties. A distributed algorithm $\mathcal{A}$ implements the AT specification iff it provides balance and transfer operations as defined before, such that the following properties hold.

- **AT-Sequentiality.** Any global history H from an execution of $\mathcal{A}$ can be AT-sequenced.
- **AT-Termination.** All operations of $\mathcal{A}$ (balance and transfer) terminate for correct processes.

Quasi-Anonymous Asset Transfer extension. An algorithm $\mathcal{A}$ implements a Quasi-Anonymous AT object (QAAT for short) if it verifies the AT properties (stated above) and meets in addition the following privacy-preserving properties. Remind that *Adv* denotes an adversary seeking to guess private information from the asset transfers of the system.

- **QAAT-Receiver-Anonymity.** Given a global history H produced by any execution of $\mathcal{A}$, any invocation $\mathsf{transfer}_i(\star, j)$ contained in H, and $j' \in [1..n]$ a process identity chosen by *Adv* attempting to guess j, then $\Pr(j = j') \leq \min((1/n) + \epsilon(\lambda), 1)$.
- **QAAT-Confidentiality.** Given a global history H from an execution of $\mathcal{A}$, an arbitrary invocation $\mathsf{transfer}_i(v, \star)$ contained in H, and $v' \in \mathbb{R}^+$ an amount chosen by the adversary *Adv* attempting to guess v, then $\Pr(v = v') \leq \epsilon(\lambda)$.

Informally, if we consider an adversary *Adv* with absolutely no information on a transfer $\mathsf{transfer}_i(v, j)$, the best this adversary can do is to pick $v \in \mathbb{R}^+$ and $j \in [1..n]$ uniformly at random. Doing so, *Adv* will guess j correctly with a probability of $1/n$, and v with a probability of 0.[6] The above two properties capture that a Quasi-Anonymous Asset Transfer can be made arbitrarily close to this ideal situation by increasing the security parameter λ of the cryptographic schemes.[7]

[6] For simplicity, we assume that v is defined on $\mathbb{R}^+$, but in practice v is a bounded positive number.
[7] Recall that $\lim_{\lambda\rightarrow+\infty} \epsilon(\lambda) = 0$.

4 A Modular Approach: Underlying Building Blocks

The QAAT algorithm proposed in this paper follows a modular approach. It builds upon several distributed and cryptographic schemes we present in this section before moving to the actual QAAT algorithm in Section 5. These schemes all require an implicit *setup* operation that takes as input the security parameter λ and outputs the public parameters of the scheme, which are known to all processes, including the adversary. For instance, in a digital signature scheme, public parameters correspond to the set of all the system's public keys. All the cryptographic schemes that follow implicitly use some public parameters (although these parameters are not exposed explicitly in the specifications and algorithms). The domains of definition for the input values and output proofs of our schemes are respectively called $\mathbb{D}$ and $\mathbb{F}$. Each scheme features a proving algorithm that aims to convince a verifying algorithm of some claim through a proof. In the following, the proving algorithm is called the prover, while the verifying algorithm is called the verifier. Formally, both the prover and the verifier are poly-time probabilistic algorithms. For simplicity, we do not provide explicit termination guarantees for the purely cryptographic schemes presented in this section (VC, UA, and ZKP), as their operations consist only of mathematical computations and do not involve communication with other processes, unlike agreement proofs (AP).

4.1 A new distributed scheme: Agreement Proofs (AP)

An *Agreement proof* scheme (*AP*) is a new distributed scheme defined by two (explicit) operations ap_prove and ap_verify satisfying four properties. The AP scheme aims at producing transferable agreement proofs (APs) that the system has reached an agreement regarding some payload v originating from a given sender/prover p_i with sequence number sn_i. Sequence numbers uniquely identify each proof the prover generates, and a new sn is generated by the scheme at each new proof, making the scheme *multi-shot* (*i.e.*, a given prover can generate multiple proofs). In the asset transfer system we present in Section 5, the AP scheme precludes double-spending. In addition, an AP also shows that the system verified some arbitrary predicate P_A. The ap_prove operation is typically implemented by a distributed algorithm involving communication between a process and a system of verifiers. For instance, the AP scheme can be implemented using quorums, consensus, or reliable broadcast. An example of consensus-free AP implementation with a constant storage complexity and a communication complexity of $O(n)$ is presented in Appendix B.3.

AP predicate. Each object of an AP scheme is set up with a specific predicate P_A, called an *AP predicate*, taking in parameter some arbitrary *data*, and returning `true` or `false`. The prover p_i passes to the ap_prove operation the payload v and the *data* parameter of P_A, and the proof σ_i is correctly generated only if $P_A(data) = \texttt{true}$. Typically, the P_A predicate can be passed to the implicit setup operation of the AP scheme, although the P_A predicate could also change dynamically for a given AP object.

AP operations. We consider an AP object O_A set up with an AP predicate P_A.

- O_A.ap_prove$(v, data)/r$: Given the prover p_i, a value v and some *data*, returns $r = \langle \sigma_i, sn_i, i \rangle$ if the input $P_A(data)$ is true, where σ_i and sn_i are respectively an agreement proof and a sequence number for value v, or $r = \texttt{abort}$ otherwise.
- O_A.ap_verify$(\sigma_j, v, sn_j, j)/r$: Returns the validity $r \in \{\texttt{true}, \texttt{false}\}$ of an agreement proof σ_j for a value v of a prover p_j at a sequence number sn_j.

Validity of an AP σ. Given an AP object O_A, a value v, a sequence number sn_i and a prover p_i (correct or faulty), we say that some σ is a "*valid AP for v at sn_i from p_i*" if and

only if any invocation of $O_A.\mathsf{ap\_verify}(\sigma, v, sn_i, i)$ by any correct process would return true.

AP properties. We consider an AP object O_A set up with an AP predicate P_A.

- **AP-Validity.** If σ_i is a valid AP for a value v at sequence number sn_i from a correct prover p_i, then p_i has executed $O_A.\mathsf{ap\_prove}(v, \star)/\langle \sigma_i, sn_i \rangle$.
- **AP-Agreement.** There are no two different valid APs σ_i and σ_i' for two different values v and v' at the same sequence number sn_i and from the same prover p_i.
- **AP-Knowledge-Soundness.** If σ_i is a valid AP for value v at sequence number sn_i from a prover p_i (correct or faulty), then p_i knows some *data* such that $P_A(data)/\texttt{true}$.
- **AP-Termination.** Given a correct process p_i that executes $O_A.\mathsf{ap\_prove}(v, data)/r$ with value v and a *data*, if $P_A(data) = \texttt{true}$, then $r = \langle \sigma_i, sn_i \rangle$ and σ_i is a valid AP for v at sn_i from p_i, and if $P_A(data) = \texttt{false}$, then $r = \texttt{abort}$.

4.2 Vector Commitment (VC)

A *commitment* scheme (as defined and used in [24, 42]) allows one to create a commitment c to a chosen value $v \in \mathbb{D}$ while keeping it hidden from others, with the ability to reveal the committed value later through an opening o. In our system, we use a *vector commitment* scheme (*VC*), which guarantees that the commitments stay secure even if their associated input data is a vector of values $V \in \mathbb{D}^m$ of size $m \in \mathbb{N}^*$.

VC operations. We consider an arbitrary vector size $m \in \mathbb{N}^*$.

- $\mathsf{vc\_commit}(V)/\langle c, o \rangle$: Takes a vector $V \in \mathbb{D}^m$ and outputs the corresponding commitment c and opening o.
- $\mathsf{vc\_verify}(c, V, o)/r$: Outputs $r = \texttt{true}$ if o is a valid opening for the commitment c and vector $V \in \mathbb{D}^m$, and $r = \texttt{false}$ otherwise.

VC properties. We consider arbitrary vector sizes $m, \ell \in \mathbb{N}^*$.

- **VC-Correctness.** $\forall\ V \in \mathbb{D}^m$, if $\mathsf{vc\_commit}(V)/\langle c, o \rangle$ then $\mathsf{vc\_verify}(c, V, o)/\texttt{true}$.
  Informally, a commitment created from a vector opens to this vector (using an opening).
- **VC-Binding.** $\forall\ V \in \mathbb{D}^m, \mathsf{vc\_commit}(V)/\langle c, o \rangle : \Pr(\mathsf{vc\_verify}(c, V, o')/\texttt{true} \wedge o' \neq o) < \epsilon(\lambda)$.
  Informally, a commitment opens to only one vector (w.h.p.).
- **VC-Hiding.**
  $\forall\ c_1, c_2 \in \mathbb{F}, \forall\ V \in \mathbb{D}^m, \mathsf{vc\_commit}(V)/\langle c, \star \rangle : |\Pr(c = c_1) - \Pr(c = c_2)| < \epsilon(\lambda)$.
  Informally, a commitment does not leak any information on its committed vector (w.h.p).

4.3 Universal Accumulators (UA)

An *accumulator* scheme (notion introduced in [5]) produces a short commitment to a set of elements S. A *universal accumulator* (*UA*) is a special kind of accumulator that can prove both the inclusion or non-inclusion of an element in the set by using *membership* or *non-membership proofs*, respectively. One of the most widely known examples of accumulators is Merkle trees [38]; however, Merkle trees do not exhibit some desirable accumulator properties provided by more efficient implementations, such as RSA accumulators [7].

In the following, we consider a new dynamic and universal accumulator scheme that does not need the knowledge of the entire accumulated set S to add or delete elements or produce membership and non-membership proofs. Furthermore, processes are not required to update the (non-)membership of their (non-)accumulated elements when additions or deletions are performed on the accumulator. To the best of our knowledge, no previously proposed

accumulator possesses all these properties. Based on the Chinese remainder theorem [24], an implementation of this new accumulator is presented in Appendix B.1.

Accumulator manager. Each accumulator A has a manager, which is the only process allowed to call the ua_add(), ua_delete(), ua_prove_mem() and ua_prove_non_mem() operations (described below) for A. This restriction is allowed because the manager is the only process who knows some secret needed to execute these operations. For classical accumulators [38, 7], this secret is typically the plain accumulated set[8], however, this secret is polylog-sized (*i.e.*, $O_\lambda(\text{polylog } |S|)$) in our Chinese-remainder-based accumulator (Appendix B.1). Multi-manager accumulators are allowed by simply sharing the secret among multiple processes, but this inevitably compromises the indistinguishability of the accumulator for all co-managers. We assume that the empty accumulator of each manager is created during the system's setup phase (see Appendix C).

UA operations. We consider an accumulator A and its associated accumulated set S.

- ua_is_empty$(A)/b$: Takes A and outputs $b =$ `true` if $S = \varnothing$, and $b =$ `false` otherwise.
- ua_add$(A, v)/A'$: Takes A and a value v and outputs the updated accumulator A' containing v.
- ua_delete$(A, v)/r$: Takes A and a value v and outputs the updated accumulator $r = A'$ without v if $v \in S$, and $r =$ `abort` otherwise.
- ua_prove_mem$(A, v)/r$: Takes A and a value v and outputs a membership proof $r = w$ of value v in accumulator A if $v \in S$, and $r =$ `abort` otherwise.
- ua_prove_non_mem$(A, v)/r$: Takes A and a value v and outputs a non-membership proof $r = u$ of value v in A if $v \notin S$, and $r =$ `abort` otherwise.
- ua_verify_mem$(A, v, w)/r$: Outputs $r =$ `true` if w is a correct membership proof of value v for A, and $r =$ `false` otherwise.
- ua_verify_non_mem$(A, v, u)/r$: Outputs $r =$ `true` if u is a correct non-membership proof of value v for A, and $r =$ `false` otherwise.

UA properties. We consider an accumulator A and its associated set S.

- **UA-Soundness.** $\forall\ v \notin S, \text{ua\_prove\_mem}(A, S, v)/r : \Pr(r \neq \texttt{abort}) < \epsilon(\lambda)$ and $\forall\ v \in S, \text{ua\_prove\_mem}(A, S, v)/r : \Pr(r \neq \texttt{abort}) < \epsilon(\lambda)$.
  Informally, the probability of computing a membership proof for a non-accumulated element or a non-membership proof for an accumulated element is negligible.
- **UA-Completeness.** $\forall\ v \in S, \text{ua\_verify\_mem}(A, v, \text{ua\_prove\_mem}(A, S, v))/r : \Pr(r = \texttt{true}) > 1-\epsilon(\lambda)$ and $\forall\ v \notin S, \text{ua\_verify\_non\_mem}(A, v, \text{ua\_prove\_non\_mem}(A, S, v))/r : \Pr(r = \texttt{true}) > 1 - \epsilon(\lambda)$.
  Informally, all honestly accumulated values are verified as true with their corresponding membership proof with a negligible probability of error.
- **UA-Undeniability.** $\forall\ v \in \mathbb{D}, \forall\ w, u \in \mathbb{F} :$
  $\Pr(\text{ua\_verify\_mem}(A, v, w) \wedge \text{ua\_verify\_non\_mem}(A, v, u)) < \epsilon(\lambda)$.
  Informally, the probability of computing both a membership and non-membership proof for the same element is negligible.
- **UA-Indistinguishability.** No information on some accumulated set S is leaked from its associated accumulator A, membership proofs w or non-membership proofs u.[9]

[8] With the only exception that RSA accumulators do not need the accumulated set to add new elements.
[9] For a more formal definition of Indistinguishability, we refer the interested reader to [17].

For simplicity and compliance with the traditional definitions given in the cryptography literature [7], we do not give properties for the ua_add and ua_delete operations. We implicitly assume that they behave correctly, that is, we can only prove the membership of an element that has been added but never deleted, we can only prove the non-membership for an element that has never been added or that has been deleted, and the ua_delete operation returns `abort` if the element was never added.

4.4 Zero-Knowledge Succinct Non-Interactive Arguments (zk-SNARGs)

By abuse of notation, we refer to the last scheme used in this article as *zero-knowledge proofs* (*ZKP*), which correspond to proofs that some prover knows some secret data without divulging any other information on it to the verifier. Specifically, we use Zero-Knowledge Succinct Non-Interactive Arguments (zk-SNARGs), which include zk-SNARKs and zk-STARKs [40].

ZKP predicate. Each object of a ZKP scheme is set up with a specific predicate P_{ZK}, called a *ZKP predicate*, taking in parameters a public data *public_data* (known both by the prover and verifier) and a secret data *secret_data* (known only by the prover), and returning `true` or `false`. The prover p_i passes to the zkp_prove operation the *public_data* and *secret_data* parameters of P_{ZK}, and the proof σ_i is correctly generated only if $P_{ZK}(public\_data, secret\_data) = \texttt{true}$. Much like the P_A predicate, the P_{ZK} predicate is typically passed to the implicit setup operation of the ZKP scheme.

ZKP operations. We consider a ZKP object O_{ZK} set up with a ZKP predicate P_{ZK}.

- O_{ZK}.zkp_prove($public\_data, secret\_data$)/r: Returns a result r, which can be either a zero-knowledge proof $r = \pi$ if the input parameters *secret_data* and *public_data* satisfy P_{ZK}, or $r =$ `abort` otherwise.
- O_{ZK}.zkp_verify($\pi, public\_data$)/r: Returns the validity $r \in \{\texttt{true}, \texttt{false}\}$ of a zero-knowledge proof π with respect to the public data *public_data* and the ZKP predicate.

ZKP properties. We consider a ZKP object O_{ZK} set up with a ZKP predicate P_{ZK}.

- **ZKP-Knowledge-Soundness.** If O_{ZK}.zkp_verify($\pi, public\_data$) is true, then the prover knows some *secret_data* such that O_{ZK}.zkp_prove($public\_data, secret\_data$)/π and $P_{ZK}(public\_data, secret\_data)$ is true.
- **ZKP-Completeness.** For any pair $\langle public\_data, secret\_data \rangle$ such that $P_{ZK}(public\_data, secret\_data)$ is true, we must have O_{ZK}.zkp_verify(O_{ZK}.zkp_prove($public\_data, secret\_data$)) is true.
- **ZKP-Zero-Knowledge.** No information on some secret data *secret_data* is leaked by its associated public data *public_data* and zero-knowledge proof π.[10]

5 A Light Consensus-Free Quasi-anonymous AT Algorithm

The asset transfer system presented in this section leverages the building blocks presented in Section 4. Intuitively, each process p_i has an accumulator A_i containing all its transfers (debits and credits), and a sequence number sn_i for its last transfer. Agreement proofs guarantee that there can be at most one transfer per sequence number from p_i, in other words, they ensure that a process cannot spend twice the same funds. Non-membership proofs make sure that a given transfer is not already in the receiver's accumulator, in other words, they ensure that a process cannot redeem twice the same transfer.

[10] For a more formal definition of Zero-Knowledge, we refer the interested reader to [25].

Algorithm 1 Initialization of the variables of Algorithm 3 (code for p_i).

1 **init:** $bal_i \leftarrow \texttt{init}_i$; $sn_i \leftarrow 0$; $\langle bal\_c_i, bal\_o_i \rangle \leftarrow \mathsf{vc\_commit}(\texttt{init}_i)$;
$A_i \leftarrow$ empty universal accumulator of p_i;
$\sigma_i \leftarrow$ initial valid agreement proof for $\langle A_i, bal\_c_i \rangle$ at $sn_i = 0$ from p_i.

5.1 Setup and initialization of process variables

We present in Algorithm 1 the initialization of the variables of each process p_i in our system: bal_i (the balance of process p_i, only known by itself and initialized to $\texttt{init}_i$), sn_i (the current sequence number of p_i, initialized to 0), bal_c_i and bal_o_i (respectively the commitment and opening of $bal_i = \texttt{init}_i$), A_i (the universal accumulator for the transfers (debits and credits) of p_i, initialized to the empty accumulator where p_i is the only manager), and σ_i (the previous agreement proof of p_i, initialized to a valid AP for the initial accumulator and balance commitment of p_i). To keep our system *light*, we impose that the bal_i and sn_i variables are of constant size (*e.g.*, 64 bits).

At the start of the system, we must use a setup procedure to obtain all these variables, and in particular σ_i, which involves communication among the processes. To this end, we present in Appendix C a trustless setup procedure for bootstrapping our system.

5.2 AP and ZKP predicates

Our system relies on an AP object *AccountUpdate*, and a ZKP object *TransferValidity*, which are respectively set up with the AP predicate P_A and the ZKP predicate P_{ZK} given in Algorithm 2 (line 2). In our system, a process creates a new AP and ZKP each time it sends or receives an asset transfer. Hence, in these predicates, the prover can either be the sender or receiver of the asset transfer at hand. We also assume that these predicates have access to the identity of the prover, denoted *pvr*. In the following, *snd* and *rcv* refer to the sender and receiver (resp.).

Predicate P_A. The P_A predicate takes as input in its *data* parameter the previous AP of the prover σ_{pvr}, the current ZKP of the prover π_{pvr}, the prover's last sequence number sn_{pvr}, and the public data $public\_data_{pvr}$ of π_{pvr} (line 4).

P_A first checks that the π_{pvr} ZKP is valid (line 5). Then, if this is the first transfer of the prover (debit or credit), P_A checks that the prover's accumulator is empty (line 7). Finally, P_A verifies the prover's previous AP (line 8).

Predicate P_{ZK}. The P_{ZK} predicate takes as input in its *public_data* parameter the prover's accumulators before and after adding the new transfer A_{pvr} and A'_{pvr} (resp.), the commitments of the prover's balance before and after applying the new transfer bal_c_{pvr} and $bal\_c'_{pvr}$ (resp.), and the prover's commitment of the transfer trf_c_{pvr} (line 10).

The P_{ZK} predicate takes as input in its *secret_data* parameter the sender's accumulator A_{snd} (recall that the sender is not always the prover), the commitment of the sender's balance bal_c_{snd}, the sender's AP σ_{snd}, the sender's membership proof w_{snd} that the transfer is in A_{snd} (notice that all the previous parameters are equal to $\perp$ if the prover is not the receiver), the transfer τ, the prover's opening of the transfer bal_o_{pvr}, the prover's balance bal_{pvr}, the commitments of the prover's balance before and after applying the transfer bal_{pvr} and bal'_{pvr} (resp.), and the prover's non-membership proof u_{pvr} that the transfer was not already in A_{pvr} (line 11).

Recall that all checks of P_{ZK} are done in zero-knowledge, without divulging any data to

Algorithm 2 AP and ZKP predicates of Algorithm 3.

2 **init:** $AccountUpdate \leftarrow$ AP object setup with P_A;
$TransferValidity \leftarrow$ ZKP object setup with P_{ZK}.

3 **predicate** $P_A(data)$ **is**
4 $\langle \sigma_{pvr}, \pi_{pvr}, sn_{pvr}, public\_data_{pvr} \rangle \leftarrow data$;
5 **if** $\neg TransferValidity.\mathsf{zkp\_verify}(\pi_{pvr}, public\_data_{pvr})$ **then** return `false`;
6 $\langle A_{pvr}, \star, bal\_c_{pvr}, \star, \star \rangle \leftarrow public\_data_{pvr}$;
7 **if** $sn_{pvr} = 1 \wedge \neg \mathsf{ua\_is\_empty}(A_{pvr})$ **then** return `false`;
8 return $AccountUpdate.\mathsf{ap\_verify}(\sigma_{pvr}, \langle A_{pvr}, bal\_c_{pvr} \rangle, sn_{pvr}, pvr)$.

9 **predicate** $P_{ZK}(public\_data, secret\_data)$ **is**
10 $\langle A_{pvr}, A'_{pvr}, bal\_c_{pvr}, bal\_c'_{pvr}, trf\_c_{pvr} \rangle \leftarrow public\_data$;
11 $\langle A_{snd}, bal\_c_{snd}, \sigma_{snd}, w_{snd}, \tau, trf\_o_{pvr}, bal_{pvr}, bal\_o_{pvr}, bal\_o'_{pvr}, u_{pvr} \rangle \leftarrow secret\_data$;
12 **if** $\neg \mathsf{vc\_verify}(bal\_c_{pvr}, bal, bal\_o_{pvr})$ **then** return `false`;
13 **if** $\neg \mathsf{ua\_verify\_non\_mem}(A_{pvr}, \tau, u_{pvr})$ **then** return `false`;
14 **if** $\mathsf{ua\_add}(A_{pvr}, \tau) \neq A'_{pvr}$ **then** return `false`;
15 **if** $\mathsf{vc\_verify}(trf\_c_{pvr}, \langle \tau, u_{pvr}, A_{snd}, bal\_c_{snd}, \sigma_{snd}, w_{snd} \rangle, trf\_o_{pvr})$ **then**
16 return `false`;
17 $\langle snd, v, rcv, sn_{snd} \rangle \leftarrow \tau$;
18 **if** $pvr = snd = rcv$ **then** return $\mathsf{vc\_verify}(bal\_c'_{pvr}, bal_{pvr}, bal\_o'_{pvr})$;
19 **if** $pvr = snd$ **then** return $\mathsf{vc\_verify}(bal\_c'_{pvr}, bal_{pvr} - v, bal\_o'_{pvr}) \wedge bal_{pvr} \geq v$;
20 **if** $pvr = rcv$ **then** return $\mathsf{vc\_verify}(bal\_c'_{pvr}, bal_{pvr} + v, bal\_o'_{pvr})$
$\wedge\, \mathsf{ua\_verify\_mem}(A_{snd}, \tau, w_{snd})$
$\wedge\, AccountUpdate.\mathsf{ap\_verify}(\sigma_{snd}, \langle A_{snd}, bal\_c_{snd} \rangle, sn_{snd}, snd)$;
21 return `false`.

the verifier(s). P_{ZK} first checks that the commitment to the prover's balance indeed opens to its balance (line 12) and that the transfer does not already belong to the prover's old accumulator (line 13). P_{ZK} then checks if the prover's new accumulator can be obtained by adding the transfer to its old accumulator (line 14). Next, if the prover, sender and receiver are the same, P_{ZK} checks that the prover's balance is the same before and after the transfer (line 18)[11]. Otherwise, if the prover is only the sender, then P_{ZK} checks that the new prover's balance is obtained by subtracting the transfer amount from the old balance, and that the old balance was sufficient to do the transfer (line 19). Finally, if the prover is only the receiver, then P_{ZK} checks that the new prover's balance is obtained by adding the transfer amount to the old balance, that the sender's accumulator contains the transfer, and that the sender's AP is valid (line 20). If neither of the 3 last conditions passed, P_{ZK} returns `false` (line 21).

[11] This condition is needed to support empty transfers, which are used to enhance the sender anonymity of the system.

Algorithm 3 Light and consensus-free QAAT algorithm (code for p_i).

```
22 operation balance() is return bal_i.

23 operation transfer(v, j) is
24     if v > bal_i then return abort;
25     τ ← ⟨i, v, j, sn_i + 1⟩;
26     process_transfer(τ, ⊥, ⊥, ⊥, ⊥);
27     w_i ← ua_prove_mem(A_i, τ);
28     ra_send TRANSFER(τ, σ_i, A_i, w_i, bal_c_i) to p_j;   ▷ra_send is receiver-anonymous
29     return commit.

30 when TRANSFER(τ, σ_j, A_j, w_j, bal_c_j) is ra_received from p_j do
31     if τ.rcv = i ∧ ap_verify(σ_j, ⟨A_j, bal_c_j⟩, τ.sn, j) ∧ ua_verify_mem(A_j, τ, w_j) then
32         process_transfer(τ, A_j, bal_c_j, σ_j, w_j).

33 internal operation process_transfer(τ, A_j, bal_c_j, σ_j, w_j) is
34     u_i ← ua_prove_non_mem(A_i, τ);
35     A'_i ← ua_add(A_i, τ);
36     if i = τ.snd = τ.rcv then bal'_i ← bal_i;
37     else if i = τ.snd then bal'_i ← bal_i − τ.v else bal'_i ← bal_i + τ.v;
38     ⟨bal_c'_i, bal_o'_i⟩ ← vc_commit(bal'_i);
39     ⟨trf_c_i, trf_o_i⟩ ← vc_commit(⟨τ, u_i, A_j, bal_c_j, σ_j, w_j⟩);
40     public_data ← ⟨A_i, A'_i, bal_c_i, bal_c'_i, trf_c_i⟩;
41     secret_data ← ⟨A_j, bal_c_j, σ_j, w_j, τ, trf_o_i, bal_i, bal_o_i, bal_o'_i, u_i⟩;
42     π ← TransferValidity.zkp_prove(public_data, secret_data);
43     data ← ⟨σ_i, π, sn_i + 1, public_data⟩;
44     ⟨σ_i, ⋆⟩ ← AccountUpdate.ap_prove(⟨A'_i, bal_c'_i⟩, data);
45     sn_i ← sn_i + 1; A_i ← A'_i; bal_i ← bal'_i; bal_c_i ← bal_c'_i; bal_o_i ← bal_o'_i.
```

5.3 Algorithm

Algorithm 3 presents the code of our QAAT implementation for a process p_i. The balance operation simply returns the value of bal_i (line 22). In the transfer operation, p_i first checks it has enough funds (line 24). Then p_i creates the transfer details with its next sequence number (line 25), processes its own transfer using the process_transfer internal operation (line 26), creates a membership proof of the transfer in its accumulator (line 27), sends all the necessary information to the receiver in a TRANSFER message (line 28) and returns commit (line 29). When p_i receives a TRANSFER message, it processes the transfer using the process_transfer internal operation if it is the transfer receiver, and if the sender's AP and membership proof are valid (line 32).

In the process_transfer internal operation, p_i first proves the non-membership of the transfer in its old accumulator (line 34) and creates its new accumulator by adding the transfer (line 35). If p_i is both the sender and receiver, its balance does not change (line 36), otherwise p_i computes its new balance depending on whether it is the sender or receiver (line 37). Next, p_i commits its new balance (line 38) and the transfer information (line 39). Process p_i then creates the public data (line 40) and secret data (line 41) of the ZKP, and generates the ZKP (line 42). Next, p_i creates the data of the AP predicate (line 43). Finally, p_i generates the AP (line 44) and updates its local variables (line 45).

5.4 Intuition of Algorithm 3's proofs

Due to space constraints, the full developments on our system's correctness, quasi-anonymity, lightness and consensus-freedom are given in Appendix A. Nonetheless, we describe in this section the high-level intuition behind these proofs.

Correctness proof. The correctness of our system comes from the fact that it satisfies AT-Sequentiality and AT-Termination.

▶ **Lemma 1** (AT-Sequentiality). *Any global history H from an execution of* Algorithm 3 *can be AT-sequenced.*

Proof sketch. AT-Sequentiality is the property that poses the most important technical difficulty: we have to show that for any execution of our algorithm, there exists a way to create a sequence of invocations respecting all process orders (the AT-sequence) that contains all the transfer/balance invocations of all correct processes, and "fake" transfer invocations of Byzantine processes, such that the legality of the operations is respected. To do that, let us remark that it is sufficient to prove that we can create a strict (*i.e.*, irreflexive) partial order defining the precedence of all transfer/balance correct invocations and transfer Byzantine invocations (thus forming a directed acyclic graph of all these invocations), such that the legality of the operations is respected. Indeed, once we have this legal partial order of invocations, we can trivially extend it to a legal total order (*i.e.*, a topological sort), which would constitute the AT-sequence.

Hence, a method for constructing an AT-sequence must both describe (i) how the "fake" $\mathsf{transfer}_i(\star,\star)$ Byzantine invocations are added to the Byzantine local histories L_i of every Byzantine process p_i, and (ii) how the partial order of all invocations of the execution is constructed. We give in Appendix A.1 such a method, that we call M, for constructing an invocation sequence S. Using M, we "kill two birds with one stone" by using the set of valid agreement proofs that are produced by correct or faulty processes in the system, giving us both all Byzantine transfer invocations, and the partial order between all the transfer/balance invocations of the system. Finally, we show that any sequence S produced using M is an AT-sequence. (Full derivations in Appendix A.1.) ◀

▶ **Lemma 2** (AT-Termination). *All operations of* Algorithm 3 *(*balance *and* transfer*) terminate for correct processes.* (Proof in Appendix A.1.)

Quasi-anonymity proof. Intuitively, our system is quasi-anonymous because it uses cryptographic schemes that do not leak sensitive data (*i.e.*, vector commitments, universal accumulators, and zero-knowledge proofs). We prove the following lemmas in Appendix A.2.

▶ **Lemma 3** (QAAT-Receiver-Anonymity). *Given the global history H produced by any execution of* Algorithm 3*, any* $\mathsf{transfer}_i(\star, j)$ *invocation of H, and $j' \in [1..n]$ a process identity chosen by Adv trying to guess j, we must have:* $\Pr(j = j') \leq \min((1/n) + \epsilon(\lambda), 1)$. (Proof in Appendix A.2.)

▶ **Lemma 4** (QAAT-Confidentiality). *Given a global history H from an execution of* Algorithm 3*, an arbitrary* $\mathsf{transfer}_i(v, \star)$ *invocation of H, and $v' \in \mathbb{R}^+$ an amount chosen by the adversary Adv trying to guess v, we must have:* $\Pr(v = v') \leq \epsilon(\lambda)$. (Proof in Appendix A.2.)

Informal arguments for lightness and consensus-freedom. If we instantiate it with the implementations of Appendix B, our system is light (as it uses succinct cryptography and

our accumulator implementation requires storing polylog bits in the size of the accumulated set) and consensus-free (as we implement agreement proofs without consensus). More details are given in Appendix A.3.

5.5 Further enhancements

Transfer batching. One could argue that requiring users to commit an accumulator update to the system each time they send or receive a single transfer is inefficient. To address this problem, we propose aggregating an arbitrary number of transfers into a single accumulator update. As a result, a user could choose only to declare an accumulator update when making a payment, while cashing all receipts simultaneously when doing so. To implement transfer batching, we use a method known as folding in Appendix D.

Constant local storage (public key rotation). The QAAT system presented in this article is light, *i.e.*, the storage cost of a given process p_i is in $O_\lambda(\text{polylog } |S_i|)$, where S_i is the set of transfers (debits and credits) of p_i. This storage cost is justified by p_i's need to store some data whose size is proportional to its entire transfer history, to prove that it is not trying to redeem the same transfer several times. However, a public key rotation mechanism could be added to obtain a system with a constant storage cost. When a process p_i rotates its public key, it changes its old public key for a new one, while transferring all its funds to the account associated to the new public key. Once this is done, p_i can flush its old local data (and, in particular, its accumulator data) so that its storage cost stays bounded by a constant. Thus, the process only has to record information concerning this rotating transfer which can be seen as the genesis state of a newly created account. However, this mechanism would involve one major technical challenge: a sender has to retrieve the receiver's public key before it can send funds to it. To achieve this retrieval, the sender can initiate a handshake with the receiver, but due to asynchrony and the fact that the receiver may be faulty, the handshake may never complete, hampering the termination of the transfer.

Towards full anonymity. Our system is not fully anonymous, as we deterministically guarantee Receiver Anonymity and Confidentiality, but not Sender Anonymity. However, remark that our system allows "empty" transfers with amount 0 (or transfers to oneself), which do not change any balance. These empty transfers could be used to obfuscate the traffic of funds from the adversary's point of view, and if they are issued at the right moment, we conjecture that they could preclude (w.h.p.) timing attacks, *i.e.*, attacks where an adversary deanonymizes the sender (or receiver) of an asset transfer by observing the timing of messages transiting on the network. If so, our system would reach full anonymity asymptotically (by adding more empty transfers at "clutch moments"). The design of the heuristics determining the "clutch moments" for issuing empty transfers is left to future work.

6 Conclusion

This article considered the problem of asynchronous Byzantine-fault-tolerant asset transfer, with the additional constraint of satisfying the properties of quasi-anonymity (*i.e.*, no leak of information on the transfers' amounts and receivers), lightness (*i.e.*, succinct cryptography and polylogarithmic storage cost), and consensus-freedom (*i.e.*, no total order of transfers). These properties are important for achieving good performance, confidentiality, and user privacy in an asset transfer system. In this context, this article introduced Quasi-Anonymous Asset Transfer (QAAT), and presented a consensus-free and light QAAT implementation,

along with its formal proofs. To our knowledge, our asset transfer system is not only the first to fulfill the 3 properties, but also the first one to have a polylogarithmic storage cost.

In addition, the article presented a new distributed abstraction called Agreement Proofs, which captures the notion of distributed agreement in a transferable short-sized proof. It also presented a new implementation of universal accumulators based on the Chinese remainder theorem, the first one which does not need knowledge of the underlying accumulated set to add/delete elements or generate (non-)membership proofs.

Presently, our asset transfer solution still lacks some capabilities compared to more mature blockchain systems (*e.g.*, Sybil resistance or smart contracts) or mainstream payment networks (*e.g.*, overdrawn accounts or fraud detection), but we believe that it demonstrates how strong privacy features can still be guaranteed by systems with low computational, storage and network requirements. Furthermore, we conjecture that our system's scalability can be further enhanced, and in particular, that it can be made permissionless without sacrificing consensus-freedom, by leveraging techniques such as the ones introduced in [35].

References

1 Alex Auvolat, Davide Frey, Michel Raynal, and François Taïani. Money transfer made simple: a specification, a generic algorithm, and its proof. *Bulletin of EATCS*, 132:22–43, 2020.

2 Alex Auvolat, Davide Frey, Michel Raynal, and François Taïani. Byzantine-tolerant causal broadcast. *Theor. Comput. Sci.*, 885:55–68, 2021.

3 Mathieu Baudet, George Danezis, and Alberto Sonnino. FastPay: High-performance Byzantine fault tolerant settlement. In *Proc. 2nd ACM Conference on Advances in Financial Technologies (AFT'20)*, pages 163–177. ACM, 2020.

4 Mathieu Baudet, Alberto Sonnino, Mahimna Kelkar, and George Danezis. Zef: Low-latency, scalable, private payments. In *Proc. 22nd Workshop on Privacy in the Electronic Society (WPES@CCS'23)*, pages 1–16. ACM, 2023.

5 Josh Cohen Benaloh and Michael de Mare. One-way accumulators: A decentralized alternative to digital signatures (Extended abstract). In *Proc. 12th Workshop on the Theory and Application of of Cryptographic Techniques (EUROCRYPT'93)*, volume 765 of *Springer LNCS*, pages 274–285, 1993.

6 Nir Bitansky, Alessandro Chiesa, Yuval Ishai, Rafail Ostrovsky, and Omer Paneth. Succinct non-interactive arguments via linear interactive proofs. *J. Cryptol.*, 35(3):15, 2022.

7 Dan Boneh, Benedikt Bünz, and Ben Fisch. Batching techniques for accumulators with applications to IOPs and stateless blockchains. In *Proc. 39th Int'l Cryptology Conference*, Springer LNCS 11692, pages 561–586, 2019.

8 Dan Boneh, Ben Lynn, and Hovav Shacham. Short signatures from the weil pairing. *J. Cryptol.*, 17(4):297–319, 2004.

9 Joseph Bonneau, Izaak Meckler, Rao Vanishree, and Evan Shapiro. Mina: Decentralized cryptocurrency at scale (White paper), 2020.

10 Gabriel Bracha. Asynchronous Byzantine agreement protocols. *Inf. Comput.*, 75(2):130–143, 1987.

11 Benedikt Bünz, Shashank Agrawal, Mahdi Zamani, and Dan Boneh. Zether: Towards privacy in a smart contract world. In *Proc. 24th Int'l Conference on Financial Cryptography and Data Security (FC'20)*, volume 12059 of *Lecture Notes in Computer Science*, pages 423–443, 2020.

12 Benedikt Bünz, Lucianna Kiffer, Loi Luu, and Mahdi Zamani. Flyclient: Super-light clients for cryptocurrencies. In *Proc. IEEE Symposium on Security and Privacy*, pages 928–946, 2020.

13 Miranda Christ, Foteini Baldimtsi, Konstantinos Kryptos Chalkias, Deepak Maram, Arnab Roy, and Joy Wang. Sok: Zero-knowledge range proofs. *IACR Cryptol. ePrint Arch.*, page 430, 2024.

14 Daniel Collins, Rachid Guerraoui, Jovan Komatovic, Petr Kuznetsov, Matteo Monti, Matej Pavlovic, Yvonne-Anne Pignolet, Dragos-Adrian Seredinschi, Andrei Tonkikh, and Athanasios Xygkis. Online payments by merely broadcasting messages. In *Proc. 50th IEEE/IFIP Int'l Conference on Dependable Systems and Networks (DSN'20)*, pages 26–38. IEEE, 2020.

15 Ivan Damgård and Maciej Koprowski. Generic lower bounds for root extraction and signature schemes in general groups. In Lars R. Knudsen, editor, *Advances in Cryptology - EUROCRYPT 2002, International Conference on the Theory and Applications of Cryptographic Techniques, Amsterdam, The Netherlands, April 28 - May 2, 2002, Proceedings*, volume 2332 of *Lecture Notes in Computer Science*, pages 256–271. Springer, 2002.

16 Sourav Das, Zhuolun Xiang, Lefteris Kokoris-Kogias, and Ling Ren. Practical asynchronous high-threshold distributed key generation and distributed polynomial sampling. In *Proc. 32nd USENIX Security Symposium*, pages 5359–5376. USENIX Association, 2023.

17 David Derler, Christian Hanser, and Daniel Slamanig. Revisiting cryptographic accumulators, additional properties and relations to other primitives. In Kaisa Nyberg, editor, *Topics in Cryptology - CT-RSA 2015, The Cryptographer's Track at the RSA Conference 2015, San Francisco, CA, USA, April 20-24, 2015. Proceedings*, volume 9048 of *Lecture Notes in Computer Science*, pages 127–144. Springer, 2015.

18 Samuel Dobson, Steven D. Galbraith, and Benjamin Smith. Trustless unknown-order groups. *CoRR*, abs/2211.16128, 2022.

19 Prastudy Fauzi, Sarah Meiklejohn, Rebekah Mercer, and Claudio Orlandi. Quisquis: A new design for anonymous cryptocurrencies. In *Proc. 25th Int'l Conference on the Theory and Application of Cryptology and Information Security (ASIACRYPT'19)*, volume 11921 of *LNCS*, pages 649–678. Springer, 2019.

20 Amos Fiat and Adi Shamir. How to prove yourself: Practical solutions to identification and signature problems. In *Proc. Advances in Cryptology (CRYPTO'86)*, volume 263 of *LNCS*, pages 186–194. Springer, 1986.

21 Michael J. Fischer, Nancy A. Lynch, and Mike Paterson. Impossibility of distributed consensus with one faulty process. *J. ACM*, 32(2):374–382, 1985.

22 Davide Frey, Mathieu Gestin, and Michel Raynal. The synchronization power (consensus number) of access-control objects: the case of allowlist and denylist. In *Proc. 37th Int'l Symposium on Distributed Computing (DISC'2023)*, volume 281 of *LIPIcs*, pages 21:1–21:23. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2023.

23 Ariel Gabizon, Zachary J. Williamson, and Oana Ciobotaru. PLONK: permutations over lagrange-bases for oecumenical noninteractive arguments of knowledge. *IACR Cryptol. ePrint Arch.*, page 953, 2019.

24 Carl Friedrich Gauss. *Disquisitiones arithmeticae*. K. Gesellschaft der Wissenschaften zu Göttingen, 1801.

25 Oded Goldreich. *The Foundations of Cryptography - Volume 1: Basic Techniques*. Cambridge University Press, 2001.

26 Jens Groth. Short pairing-based non-interactive zero-knowledge arguments. In *Proc. 16th International Conference on the Theory and Application of Cryptology and Information Security*, volume 6477 of *Springer LNCS*, pages 321–340, 2010.

27 Rachid Guerraoui, Petr Kuznetsov, Matteo Monti, Matej Pavlovic, and Dragos-Adrian Seredinschi. The consensus number of a cryptocurrency. *Distributed Comput.*, 35(1):1–15, 2022.

28 Saurabh Gupta. A non-consensus based decentralized financial transaction processing model with support for efficient auditing (Master thesis). Arizona State University, 2016.

29 Daira Emma Hopwood, Sean Bowe, and Taylor Hornby Nathan Wilcox. Zcash protocol specification, 2016.

30 Damien Imbs and Michel Raynal. Trading off t-resilience for efficiency in asynchronous Byzantine reliable broadcast. *Parallel Process. Lett.*, 26(4):1650017:1–1650017:8, 2016.

31 Anurag Jain, Emmanuelle Anceaume, and Sujit Gujar. Extending the boundaries and exploring the limits of blockchain compression. In *Proc. 42nd Int'l Symposium on Reliable Distributed Systems (SRDS'23)*, pages 187–197. IEEE, 2023.

32 Aggelos Kiayias, Nikos Leonardos, and Dionysis Zindros. Mining in logarithmic space. In *Proc. ACM SIGSAC Conference on Computer and Communications Security (CCS'21)*, pages 3487–3501. ACM, 2021.

33 Chelsea Komlo and Ian Goldberg. FROST: flexible round-optimized schnorr threshold signatures. In *Proc. 27th Int'l Conference on Selected Areas in Cryptography (SAC'20)*, volume 12804 of *Lecture Notes in Computer Science*, pages 34–65. Springer, 2020.

34 Abhiram Kothapalli, Srinath T. V. Setty, and Ioanna Tzialla. Nova: Recursive zero-knowledge arguments from folding schemes. In *42nd Annual International Cryptology Conference (CRYPTO'22)*, volume 13510 of *LNCS*, pages 359–388. Springer, 2022.

35 Petr Kuznetsov, Yvonne-Anne Pignolet, Pavel Ponomarev, and Andrei Tonkikh. Permissionless and asynchronous asset transfer. *Distributed Computing*, 36(3):349–371, 2023.

36 Leslie Lamport, Robert E. Shostak, and Marshall C. Pease. The Byzantine generals problem. *ACM Trans. Program. Lang. Syst.*, 4(3):382–401, 1982.

37 Karl Marx. *Das Kapital. Kritik der politischen Ökonomie. Buch I: Der Produktionsprocess des Kapitals*. Otto Meissner, 1867.

38 Ralph C. Merkle. A digital signature based on a conventional encryption function. In *Proc. 7th Int'l Cryptology Conference (CRYPTO'87)*, volume 293 of *Lecture Notes in Computer Science*, pages 369–378. Springer, 1987.

39 Satoshi Nakamoto. Bitcoin: A peer-to-peer electronic cash system, 2008.

40 Anca Nitulescu. zk-snarks: a gentle introduction, 2020.

41 Marshall C. Pease, Robert E. Shostak, and Leslie Lamport. Reaching agreement in the presence of faults. *Journal of the ACM*, 27(2):228–234, 1980.

42 Torben P. Pedersen. Non-interactive and information-theoretic secure verifiable secret sharing. In *Proc. 11th Annual Int'l Cryptology Conference (CRYPTO'91)*, volume 576 of *Lecture Notes in Computer Science*, pages 129–140. Springer, 1991.

43 Michael G. Reed, Paul F. Syverson, and David M. Goldschlag. Anonymous connections and onion routing. *IEEE J. Sel. Areas Commun.*, 16(4):482–494, 1998.

44 Tim Ruffing, Viktoria Ronge, Elliott Jin, Jonas Schneider-Bensch, and Dominique Schröder. ROAST: robust asynchronous Schnorr threshold signatures. In *Proc. ACM SIGSAC Conference on Computer and Communications Security (CCS'22)*, pages 2551–2564. ACM, 2022.

45 Srinath T. V. Setty. Spartan: Efficient and general-purpose zksnarks without trusted setup. In *Proc. 40th Annual International Cryptology Conference (CRYPTO'20)*, volume 12172 of *Lecture Notes in Computer Science*, pages 704–737. Springer, 2020.

46 Douglas R. Stinson and Reto Strobl. Provably secure distributed schnorr signatures and a (t, n) threshold scheme for implicit certificates. In Vijay Varadharajan and Yi Mu, editors, *Proc. 6th Australasian Conference on Information Security and Privacy (ACISP'01)*, volume 2119 of *Lecture Notes in Computer Science*, pages 417–434. Springer, 2001.

47 Harry W. H. Wong, Jack P. K. Ma, Hoover H. F. Yin, and Sherman S. M. Chow. How (not) to build threshold EdDSA. In *Proc. 26th Int'l Symposium on Research in Attacks, Intrusions and Defenses (RAID'23)*, pages 123–134. ACM, 2023.

A Proofs of Algorithm 3

In this appendix section, we provide full derivations on the proof of correctness (Appendix A.1) and quasi-anonymity (Appendix A.2) of Algorithm 3. We also provide informal arguments for the lightness and consensus-freedom of Algorithm 3 (Appendix A.3).

A.1 Proof of correctness

We start by defining notations used in the AT-Sequentiality proof, and then define a method M for constructing a sequence of invocations S from an execution of Algorithm 3.

AP notation. If some σ is a valid AP at a sequence number sn from a process p_i, we denote it by σ_i^{sn}.

Sequence of previous APs of an AP. Let us consider a valid AP σ_i^{sn} at sequence number $sn > 0$ from a process p_i (correct or faulty). By AP-Knowledge-Soundness, p_i knows some $data = \langle \sigma_i', \star, \star, sn-1, \star \rangle$ s.t. $P_A(data)$ is true, where σ_i' is an AP at sequence number $sn-1$ from p_i. From line 8 of P_A, σ_i' is also a valid AP at sequence number $sn-1$ from p_i. By induction and using AP-Agreement, we obtain that, for each sn' s.t. $0 \leq sn' \leq sn$ there is a single valid AP $\sigma_i^{sn'}$ at sn' from p_i. We say that the sequence of all $\sigma_i^{sn'}$ valid APs where $0 \leq sn' \leq sn$ is the *sequence of previous APs* of σ_i^{sn}, denoted $previous\_APs_i^{sn}$.

Transfer invocation corresponding to an AP. By AP-Knowledge-Soundness, for every valid AP σ_i^{sn} at a sequence number sn from a process p_i (correct or faulty) where $sn > 0$ (the sequence number $sn = 0$ is excluded as the initial AP σ_i^0 does not have a corresponding transfer invocation), the prover p_i must know some $data = \langle \star, \pi_i, \star, sn-1, public\_data \rangle$ satisfying $P_A(data)$. Moreover, given that $TransferValidity.\mathsf{zkp\_verify}(\pi_i, public\_data)$ must be true

(line 5), then by ZKP-Knowledge-Soundness, the prover p_i must know some *secret_data* containing a $\tau = \langle snd, v, rcv, sn' \rangle$ variable such that $P_{ZK}(public\_data, secret\_data)$ is true (if $i = snd$, then $sn = sn'$, otherwise $i = rcv$ and sn and sn' can be different). We denote the transfer invocation corresponding to τ by $\mathsf{transfer}_{snd}^{sn'}(v, rcv)$. In this case, we say that $\mathsf{transfer}_{snd}^{sn'}(v, rcv)$ is the transfer invocation *corresponding* to the AP σ_i^{sn}.

Sending APs, receiving APs, and null APs. We distinguish two sorts of valid APs that are produced in our system: *sending APs*, and *receiving APs*.

- Some valid σ_i^{sn} is a sending AP if it corresponds to a $\mathsf{transfer}_i^{sn}(\star, j)$ invocation (where i and j can be different). If the prover p_i is correct, the sending AP is produced during the process_transfer() call at line 26.
- Some valid σ_i^{sn} is a sending AP if it corresponds to a $\mathsf{transfer}_j^{sn'}(\star, i)$ invocation (where i and j can be different, and sn and sn' can be different). If the prover p_i is correct, the receiving AP is produced during the process_transfer() call at line 32.

Remind that a transfer invocation $\mathsf{transfer}_i^{sn}(v, i)$ from a process p_i to itself is allowed in our algorithm, which entails that the corresponding AP σ_i^{sn} is both a sending AP and a receiving AP. In this case, σ_i^{sn} is said to be a *null* AP, and $\mathsf{transfer}_i^{sn}(v, i)$ is said to be a *null* transfer invocation.

Matching non-null sending AP of a non-null receiving AP. Notice that, due to the verification of the sending AP in the P_{ZK} predicate at line 20 when the prover is the receiver, any non-null receiving AP σ_i^{sn} must have a *matching* sending AP $\sigma_j^{sn'}$.[12]

Transfer or balance invocation notations. If a correct process p_i performs a $\mathsf{balance}_i()$ invocation while its sn_i variable has value sn, we denote it by $\mathsf{balance}_i^{sn}()$.[13] For simplicity, we denote by $\mathsf{op}_i^{sn}()$ any invocation $\mathsf{balance}_i^{sn}()$ or $\mathsf{transfer}_i^{sn}(\star, \star)$.

Method M for constructing an invocation sequence S. We describe in the following a method, that we call M, for constructing a sequence S of all the operation invocations produced by an execution of Algorithm 3, captured as a global history $H = (L_1, ..., L_n)$. First, we incrementally construct a partially-ordered set (poset) $P = (I, \rightarrow_P)$, where I is the set of all operation invocations of the execution and $\rightarrow_P$ is a partial order on I defining the precedence of invocations. Initially, for every correct process p_i, I includes all the $\mathsf{transfer}_i(v, j)$ and $\mathsf{balance}_i()$ operation invocations in L_i, and the $\rightarrow_P$ order extends the $\rightarrow_i$ process order. We also define a set called *receiving_APs_to_process*, which contains non-null receiving APs that must be processed to construct P. Non-null receiving APs are particularly important, as they allow us to infer $\rightarrow_P$ relations between the transfer() invocations of different processes. The *receiving_APs_to_process* set is initialized with all non-null receiving APs produced during the execution by correct processes at line 44, in a process_transfer() call at line 32. We now describe the steps to construct P iteratively.

1. We initialize a new variable *new_receiving_APs* to the empty set $\varnothing$. This set will contain the new non-null receiving APs discovered during the current iteration, and it will replace the current *receiving_APs_to_process* at the end of the iteration. (Some non-null receiving APs can be "rediscovered" several times during the procedure, which does not pose a problem for the construction.)

[12] The reverse is not always true: if the receiver of a transfer is faulty, it may never produce a receiving AP for the corresponding sending AP.

[13] Recall that we do not consider balance() invocations from Byzantine processes.

2. For every non-null receiving AP $\sigma_{rcv}^{sn_{rcv}} \in$ *receiving_APs_to_process*, we get the matching non-null sending AP $\sigma_{snd}^{sn_{snd}}$ and its associated sequence of previous APs $previous\_APs_{snd}^{sn_{snd}}$, and execute the following sub-steps.
   a. For every sending AP $\sigma_{snd}^{sn'_{snd}} \in previous\_APs_{snd}^{sn_{snd}}$, we add its corresponding $\mathsf{transfer}_{snd}^{sn'_{snd}}(\star,\star)$ invocation in I if it was not already contained (typically because p_{snd} is correct).
   b. For every pair of invocations $o'_{snd} = \mathsf{transfer}_{snd}^{sn'_{snd}}(\star,\star), o''_{snd} = \mathsf{transfer}_{snd}^{sn''_{snd}}(\star,\star) \in I$ s.t. $sn'_{snd} < sn''_{snd}$, we have $o'_{snd} \rightarrow_P o''_{snd}$ (process order of p_{snd}).
   c. For every pair of invocations $o'_{rcv} = \mathsf{op}_{snd}^{sn'_{snd}}(), o'_{snd} = \mathsf{op}_{rcv}^{sn'_{rcv}}() \in I$ s.t. $sn'_{snd} \leq sn_{snd}$ and $sn_{rcv} \leq sn'_{rcv}$, we have $o'_{snd} \rightarrow_P o'_{rcv}$ (the sending precedes the reception).
   d. We add all receiving APs of $previous\_APs_{snd}^{sn}$ to *new_receiving_APs*.
3. We assign *new_receiving_APs* to *receiving_APs_to_process*.
4. Repeat from step 1 until *receiving_APs_to_process* is empty.

Once we have finished performing the above steps, we perform the transitive closure on the $\rightarrow_P$ relation to make sure that we end up with a well-formed partial order. Finally, we obtain a sequence S containing all the invocations of I by collapsing the $\rightarrow_P$ partial order into any possible total order $\rightarrow_S$.

▶ **Lemma 1** (AT-Sequentiality). *Any global history H from an execution of* Algorithm 3 *can be AT-sequenced.*

Proof. Let us consider any execution of Algorithm 3, captured as a global history $H = (L_1, ..., L_n)$. We subsequently show that any sequence of invocation S created from H using method M is an AT-sequence. Let us consider one of such sequences that we call S, with $\rightarrow_S$ its associated total order. We have the following for every operation invocation $o = \mathsf{op}_i^{sn}() \in S$ by process p_i at sequence number sn.

- Case where o is a $\mathsf{balance}_i^{sn}()/v$ invocation. In this case, p_i is necessarily correct, and we have to prove that $v = total(i, \{o' \in S \mid o' \rightarrow_S o\})$.
  By construction of S using method M, the following holds. For all the $\mathsf{transfer}_i^{sn'}(v', j)$ sent by p_i before o in S where $i \neq j$, p_i has subtracted v' to its balance bal_i at line 37 during the process_transfer() call at line 26. Likewise, for all the $\mathsf{transfer}_j^{sn'}(v', i)$ received by p_i before o in S where $i \neq j$, p_i has added v' to its balance bal_i at line 37 during the process_transfer() call at line 32. Therefore, the value v returned by $\mathsf{balance}_i^{sn}()/v$ is computed exactly like $total(i, \{o' \in S \mid o' \rightarrow_S o\})$.
- Case where o is a $\mathsf{transfer}_i^{sn}(v, j)$ invocation (remind that, implicitly, we only consider $\mathsf{transfer}()/r$ invocations that return $r =$ `commit`). In this case, we have to prove that $v \leq total(i, \{o' \in S \mid o' \rightarrow_S o\})$.
  If the sender p_i is correct, then the proof follows the same reasoning as the previous case, and from the fact that p_i checks that it has enough funds for the transfer at line 24.
  If the sender p_i is Byzantine, the proof relies on the P_{ZK} predicate. For each of the valid APs $\sigma_i^{sn'}$ at sn' from p_i (where $0 < sn' \leq sn$) that are used to construct S using method M, p_i must provide in the *secret_data* of the associated ZKP $\pi_i^{sn'}$ its old and new balances, respectively denoted bal_{pvr} and bal'_{pvr} at line 11. Be $\sigma_i^{sn'}$ a null AP, a non-null sending AP, or a non-null receiving AP, the transition from the old to the new balance of p_i is verified in zero-knowledge at line 18, 19, or 20, respectively.
  By construction of S using method M, we have the following for each of the valid APs $\sigma_i^{sn'}$ at sn' from p_i where $0 < sn' < sn$. If $\sigma_i^{sn'}$ is a sending AP corresponding to a $\mathsf{transfer}_i^{sn'}(v', j)$ invocation, then $\mathsf{transfer}_i^{sn'}(v', j)$ must appear before o in the $\rightarrow_P$ order.

Likewise, if $\sigma_i^{sn'}$ is a receiving AP corresponding to a $\mathsf{transfer}_j^{sn''}(v', i)$ invocation, then $\mathsf{transfer}_j^{sn''}(v', i)$ must appear before o in the $\rightarrow_S$ total order. Therefore, by induction, the balance of a Byzantine process is also computed exactly like $total(i, \{o' \in S \mid o' \rightarrow_S o\})$. Finally, due to the $bal_{pvr} \geq v$ check at line 19, the old balance of p_i right before o is guaranteed to be greater than transfer value, which concludes the proof. ◀

▶ **Lemma 2** (AT-Termination). *All operations of* Algorithm 3 *(*balance *and* transfer*) terminate for correct processes.*

Proof. The balance operation of Algorithm 3 terminates trivially. Furthermore, the only blocking instruction of the transfer operation of Algorithm 3 is the ap_prove operation at line 44, in the process_transfer internal operation (called at line 26). This ap_prove operation terminates by AP-Termination, so the transfer operation also terminates. ◀

A.2 Proof of quasi-anonymity

▶ **Lemma 3** (QAAT-Receiver-Anonymity). *Given the global history H produced by any execution of* Algorithm 3*, any* $\mathsf{transfer}_i(\star, j)$ *invocation of H, and $j' \in [1..n]$ a process identity chosen by Adv trying to guess j, we must have:* $\Pr(j = j') \leq \min((1/n) + \epsilon(\lambda), 1)$.

Proof. To determine if *Adv* can obtain any information on the receiver p_j of any $\mathsf{transfer}_i(\star, j)$ invocation given a global history H, we analyze the content of all message types produced by Algorithm 3. Our algorithm features 2 message types with the following content:

1. Accumulator update sent by a (sender or receiver) process p_i publicly to the network in the ap_prove operation at line 44: value $v = \langle A'_i, bal\_c'_i \rangle$ and data $data = \langle \sigma_i, \pi, sn_i + 1, \langle A_i, A'_i, bal\_c_i, bal\_c'_i, trf\_c_i \rangle \rangle$;
2. Message from a sender p_{snd} to a receiver p_{rcv}: $\langle \tau, \sigma_{snd}, A_{snd}, w_{snd}, bal\_c_{snd} \rangle$.

Note that sender-receiver messages (item 2) are exchanged outside using the ra_send operation, which ensures sender anonymity and confidentiality. Therefore, only accumulator updates (item 1) need to be considered. VC-Hiding guarantees that the vector commitments bal_c_i, $bal\_c'_i$, and trf_c_i do not leak any data on the corresponding committed values. Similarly, UA-Indistinguishability guarantees that the accumulators A_i and A'_i do not leak any data on their accumulated sets, and ZKP-Zero-Knowledge ensures that π does not leak any data on its secret input *secret_data*. Finally, the sequence number $sn_i + 1$ does not reveal whether p_i is a sender or receiver, only the number of transfers (debits or credits) in A_i. Thus, *Adv* does not learn any information from accumulator updates.

We conclude that without any information on the content of transfers already included or newly added to some history H, *Adv* cannot deduce the recipient of a transfer more accurately than random, *i.e.*, with a probability of less than $\min((1/n) + \epsilon(\lambda), 1)$, where ϵ is the statistical negligible function and λ is the security parameter of the system. ◀

▶ **Lemma 4** (QAAT-Confidentiality). *Given a global history H from an execution of* Algorithm 3*, an arbitrary* $\mathsf{transfer}_i(v, \star)$ *invocation of H, and $v' \in \mathbb{R}^+$ an amount chosen by the adversary Adv trying to guess v, we must have:* $\Pr(v = v') \leq \epsilon(\lambda)$.

Proof. The proof follows the same reasoning as that of Lemma 3. ◀

A.3 Informal arguments for lightness and consensus-freedom

In this section, we informally show that our asset transfer system (Algorithm 3) is consensus-free and light if it is instantiated with the AP, VC, UA and ZKP implementations of Appendix B. Our system's consensus-freedom follows the consensus-freedom of our AP implementation (Appendix B.3). The lightness of our system comes from its succinctness and polylogarithmic storage.

- The succinctness of our system comes from the succinctness of our Agreement Proof implementation (Appendix B.3) and CRASSe implementation of universal accumulators (Appendix B.1), as well as the succinctness of Pedersen vector commitments (Appendix B.2) and Spartan zk-SNARKs (Appendix B.4).
- Let us consider a correct process p_i. The polylogarithmic storage of our system comes from the fact that all its local variables, apart from the accumulator A_i, are in $O_\lambda(1)$ bits: by definition, bal_i and sn_i have $O_\lambda(1)$ bits (see Section 5.3), bal_c_i and bal_o_i have $O_\lambda(1)$ bits with Pedersen vector commitments, and σ_i has $O_\lambda(1)$ bits with our AP implementation (see Appendix B.3). Moreover, A_i has $O_\lambda(\text{polylog } |S_i|)$ bits with our CRASSe implementation (see Appendix B.1), where S_i is the accumulated set corresponding to A_i containing all the transfers (debits or credits) of p_i. Finally, the local variables of our AP implementation are also in $O_\lambda(1)$ bits (see Appendix B.3).

B Implementations of the Schemes of Section 4

B.1 A new accumator implementation: Chinese-remainder-based accumulator with sublinear storage (CRASSe)

In this section we introduce a new non-trivial implementation of a cryptographic accumulator based on the Chinese Remainder Theorem (CRT) [24]. We call this new accumulator implementation CRASSe, for "Chinese-remainder-based accumulator with sublinear storage". To our knowledge, this accumulator is the first one that does not require the manager to know the plain accumulated set S to add/delete elements or prove their (non-)membership. It is also the first accumulator that incurs a storage cost for the manager that is only polylogarithmic in the size of the accumulated set S, which we denote $O_\lambda(\text{polylog } |S|)$. Algorithm 7 implements all operations presented in the universal accumulator specification of Section 4.3.

B.1.1 Preliminaries

Notations. We use the following notations in our accumulator implementation.

- $a||b$ is the concatenation of two bit-strings a and b.
- $\boldsymbol{a}$ (note the bold font) is a vector of an arbitrary size n with elements $\{a_1, .., a_n\}$.
- $[n]$ denotes the set of integers $\{1, .., n-1\}$.
- $x \xleftarrow{\$} S$ denotes sampling a random element $x \in S$ uniformly.
- $\mathsf{Primes}(a, b)$ is the set of primes greater than 2^a and smaller than 2^b.
- H_{Primes} is a mapping of each index $i \in [\pi(2^b) - \pi(2^a)]$ to a distinct prime in $\mathsf{Primes}(a, b)$, where π is the prime counting function.
- $\mathsf{GGen}(\lambda)$ is a randomized algorithm that generates a group of unknown order in a range $[A, B]$ such that A, B and $A - B$ are all integers exponential in λ.
- $a \mid b$ means "a divides b" and $a \nmid b$ means "a does not divide b".

Most of the above notations are standard cryptographic notations. We reuse the notations introduced by [7].

Generic groups of Unknown Order. We adopt the generic group model for groups of unknown order as defined in [15] and used in [7].

▶ **Definition 5** (Groups of Unknown Order). *A group of unknown order $\mathbb{G}$ has order $|\mathbb{G}| \xleftarrow{\$} [A, B]$ where A and B are two public integer parameters. The elements of the group $\mathbb{G}$ are defined by a random injective function $\sigma : \mathbb{Z}_{|\mathbb{G}|} \to \{0,1\}^\ell$ for some ℓ such that $2^\ell \gg |G|$. The group elements are denoted $\sigma(0), ..., \sigma(\mathbb{G}-1)$. Let $\mathcal{A}$ be a probabilistic generic group algorithm initialized with a list $\mathcal{L}$ of encodings. $\mathcal{A}$ can query oracles $\mathcal{O}_1$ and $\mathcal{O}_2$.*

- *$\mathcal{O}_1$ selects $r \in \mathbb{Z}_{|\mathbb{G}|}$, appends $\sigma(r)$ to $\mathcal{L}$ and returns $\sigma(r)$.*
- *When $\mathcal{L}$ has size q, $\mathcal{O}_2(i, j, \pm)$ selects two indices $i, j \in \{1, ..., q\}$, ouputs $\sigma(i \pm j)$ and appends it to $\mathcal{L}$.*

Concrete examples of such groups are RSA groups, ideal class groups of imaginary quadratic fields and hyperelliptic Jacobians. Note that RSA groups require a trusted setup to compute the RSA modulus, and are thus not viable for our trustless implementation. Dobson *et al.* [18] provide an algorithm (Algorithm 4 of [18]) to compute efficient genus-3 hyperelliptic Jacobians of unknown order trustlessly.

Assumptions. We define in the following cryptographic assumptions used in our construction.

▶ **Assumption 1** (Random Oracle Model (ROM)). *Informally, the ROM assumes a public oracle that implements a completely random function. Such a function cannot exist in the real world, as it would take infinite space to represent. However, cryptographic hash functions are designed to behave similarly and are assumed to be ROs in practice.*

▶ **Assumption 2** (Discrete Logarithm Assumption (DL)). *Let $\mathcal{G} \leftarrow \mathsf{GGen}(\lambda)$ be a generic group of unknown order and let $\langle g_1, ..., g_n \rangle$ and let Adv be a generic polynomial time algorithm. The DL assumption holds for $\mathcal{G}$ iff Adv succeeds with at most a negligible probability in outputting $a_1, ..., a_n$ and $b_1, ..., b_n$ s.t. $\prod_{i=1}^n g_i^{a_i} = \prod_{i=1}^n g_i^{b_i}$.*

▶ **Theorem 6** (Chinese Remainder Theorem (CRT)). *Let $\ell_1, ..., \ell_k$ be pairwise coprime integers greater than 1 and $a_1, ..., a_k$ integers s.t. $0 \leq a_i < \ell_i$. Then $\exists! x \in \mathbb{Z}$ s.t. $0 \leq x < \prod_{i=1}^k n_i$ and $\forall\ i \in \{1, ..., k\}$, $x \equiv a_i \mod n_i$.*

B.1.2 Construction

Sublinear representation of a set of bounded integers. Our aim is to store a compressed version of a set of positive b-bits integers $\boldsymbol{a} = \{a_1, ..., a_k\}$. To achieve this goal, we make use of the CRT. Let b_1 and b_2 be polylogarithmic bit sizes in b, with $b_i = \log_2(b)^{s_i}$ and $s_1 > s_2$. Note that the choice of b_1 and b_2 determines the maximum size of the set (number of indexes). Let $\boldsymbol{\ell} = \{\ell_1, ..., \ell_k\}$ be a set of k distinct b_1-bit prime numbers in $\mathsf{Primes}(b_2, b_1)$. We assume that the index/prime mapping of $\boldsymbol{\ell}$ is provided in the form of the public function H_{Primes}, such that $\ell_i = H_{\mathsf{Primes}}(i)$. Thus, we do not consider the storage of $\boldsymbol{\ell}$. We can write $a_i = q_i \ell_i + r_i$, where $q_i = \lfloor a_i / \ell_i \rfloor$ and $r_i = a_i \mod \ell_i$ are the quotient and the remainder of the integer division of a_i by ℓ_i. Recall that as the integers in $\boldsymbol{\ell}$ are primes and the set $\boldsymbol{r} = \{r_1, ..., r_k\}$ contains integers $r_i < \ell_i$, there exists a unique integer $x < \prod_{i=1}^k \ell_i$ that satisfies the equation of the CRT (Theorem 6). Since x is bounded by $\prod_{i=1}^k \ell_i$, x can be stored as a (kb_1)-bit integer, which is polylogarithmically more efficient than storing k

b-bits integers. We can now delete $\boldsymbol{q}$ and $\boldsymbol{r}$. We finally have obtained the space-efficient construction $\langle x, \ell \rangle$ composed of $2kb_1 = 2\log_2(b)^{s_1}$ bits. While we cannot extract elements of $\boldsymbol{a}$ from this representation anymore (this would require storing the quotients), we can still use it to test if an element e_i was used to construct $\langle x, \ell \rangle$ by verifying if $\ell_i \mid \ell$ and $x \equiv e_i \mod \ell_i$. However, we remark that if an element $a_i \equiv e_i \mod \ell_i$ was stored inside $\boldsymbol{a}$, then one would obtain a false positive for e_i. Thus, to achieve statistical soundness (collision resistance), we require the use of a secure hash function to attribute to each element its index and, thus, its prime divisor. This renders negligible the probability of attributing the same index i to two elements congruent modulo ℓ_i. In the next section, we show how to compute proofs of an element's membership and non-membership in zero knowledge.

Accumulator (non)-membership proofs. Let us consider a generic group of unknown order $\mathbb{G}$ with generators g_1, g_2. We define our accumulator as $A = \langle g_1^x, g_2^\ell \rangle \in \mathbb{G}^2$ with secret parameters $sp = \langle x, \ell \rangle$ stored by the accumulator manager (AM). Notice that the public representation of A (the digest) is of constant size. Furthermore, it is not possible to retrieve sp from A under the DL assumption (Assumption 2). We base our (non-)membership proofs on the proof of knowledge of representation (Algorithm 4) of Boneh *et al.* [7], which we slightly modify to prove the knowledge of a Chinese remainder. We then render it non-interactive using the Fiat-Shamir heuristic [20]. The resulting non-interactive proof of knowledge of Chinese remainder (NIPoKCR) can then be used by the AM to prove membership of an element e, *i.e.*, if x was computed using e. Algorithm 5 details briefly the operations performed by the prover and verifier of a NIPoKCR execution. To prove non-membership, we additionally provide a proof of knowledge of "non-Chinese remainder" (Algorithm 6) that proves that for an element e and prime ℓ_e, $x \not\equiv e \mod \ell_e \vee \ell_e \nmid \ell$. This second proof uses a range proof to convince that the remainder of the integer division of x by ℓ_e is in interval $[0, \ell_e - 1]$. While we do not provide an algorithm for the range proof, many compatible solutions are in the literature [13].

Empty accumulator. During setup, the AM samples random initial secret parameters $sp_0 = \langle x_0, \ell_0 \rangle \xleftarrow{\$} \mathbb{Z}^+ \times \mathsf{Primes}(b_1 + 1, b)$. The AM then computes the value of its empty accumulator $A_0 = \langle g_1^{x_0}, g_2^{\ell_0} \rangle$ and appends it to its public parameters. Note that ℓ_0 is sampled outside of $\mathsf{Primes}(b_2, b_1)$ to avoid any collision with further accumulated elements. This randomness is necessary to prevent any PPT adversary from being able to compute the accumulator even if it collects the set of accumulated elements $S \backslash \{x_0\}$.

Algorithm 4 Proof of Knowledge of Representation (PoKRep) [7].

params: $\mathbb{G} \xleftarrow{\$} \mathsf{GGen}(\lambda)$; $(g_1, ..., g_n) \in \mathbb{G}^n$;
inputs: $w \in \mathbb{G}$;
witness: $\boldsymbol{x} \in \mathbb{Z}^n$;
claim: $\mathsf{Rep}(x) = \prod_{i=1}^n g_i^{x_i} = w$;
Verifier sends $\ell \xleftarrow{\$} \mathsf{Primes}(2, \lambda)$;
For each x_i, Prover finds q_i, r_i s.t. $x_i = q_i \ell + r_i$, sets $\boldsymbol{q} \leftarrow \langle q_1, ..., q_n \rangle \in \mathbb{Z}^n$ and $\boldsymbol{r} \leftarrow \langle r_1, ..., r_n \rangle \in [\ell]^n$; Prover sends $Q \leftarrow \prod_{i=1}^n g_i^{q_i}$ and $\boldsymbol{r}$ to Verifier;
Verifier accepts if $\boldsymbol{r} \in [\ell]^n$ and $Q^\ell \mathsf{Rep}(r) = w$.

Algorithm 5 Non-Interactive Proof of Knowledge of Chinese Remainder (NIPoKCR).

params: $\mathbb{G} \xleftarrow{\$} \mathsf{GGen}(\lambda)$; $\langle g_1, g_2 \rangle \in \mathbb{G}$;
inputs: $e \in \mathbb{Z}$; $X \leftarrow g_1^x \in \mathbb{G}$; $L \leftarrow g_2^\ell \in \mathbb{G}$; $\ell_e \leftarrow H_{\mathsf{Primes}}(H(e)) \in \mathsf{Primes}(s_1, s_2)$;
witness: $x \in \mathbb{Z}$; $\ell \in \mathbb{G}$;
claim: $x \equiv e \mod \ell_e \wedge \ell_e \mid \ell$.
Prover computes q_e, r_e and q_x, r_x s.t. $e = q_e\ell_e + r_e$ and $x = q_x\ell_e + r_x$; Prover computes q_ℓ s.t. $\ell = \ell_e q_\ell$; Prover sends $Q_1 \leftarrow g_1^{q_x} g_2^{q_e}$ and $Q_2 \leftarrow g_2^{q_\ell}$ to Verifier;
Verifier computes $r_e = e \mod \ell_e$ and accepts if $Q_1^{\ell_e}(g_1 g_2)^{r_e} = X g_2^e \wedge Q_2^{\ell_e} = L$.

Algorithm 6 Non-Interactive Proof of Knowledge of Non-Chinese Remainder (NIPoKNCR).

params: $\mathbb{G} \xleftarrow{\$} \mathsf{GGen}(\lambda)$; $\langle g_1, g_2 \rangle \in \mathbb{G}$;
inputs: $e \in \mathbb{Z}$; $X \leftarrow g_1^x \in \mathbb{G}$; $L \leftarrow g_2^\ell \in \mathbb{G}$;$\ell_e \leftarrow H_{\mathsf{Primes}}(H(e)) \in \mathsf{Primes}(s_1, s_2)$;
witness: $x \in \mathbb{Z}$; $\ell \in \mathbb{G}$;
claim: $x \not\equiv e \mod \ell_e \vee \ell_e \nmid \ell$.
If $\ell_e \nmid \ell$, then Prover computes the Bezout coefficient a, b s.t. $a\ell + b\ell_e = 1$ and sends a and $B = g_2^b$;
Else Prover determines q_e, r_e and q_x, r_x s.t. $e = q_e\ell_e + r_e$ and $x = q_x\ell_e + r_x$, then computes $Q_x = g_1^{q_x}$, $R_x = g_1^{r_x}$, $Q_e = g_2^{q_e}$ and a range proof π for R_x proving that $r_x \in [\ell_e]$ and sends Q_x, R_x, Q_e and π to Verifier;
Verifier accepts if $L^a B^{\ell_e} = g_2$ or computes $r_e = e \mod \ell_e$ and accepts if π is a valid range proof for R_x and $X g_2^e = Q_x^{\ell_e} R_x Q_e^{\ell_e} + g_2^{r_e} \wedge R_x \neq g_1^{r_e}$.

B.1.3 Security proofs

In the following, we consider an accumulator $A = \langle g_1^x, g_2^\ell \rangle$ with public parameters $pp = \langle \mathcal{G}, g_1, g_2, H_{\mathsf{Primes}}, A_0 \rangle$ and secret parameters $sp = \langle x, \ell \rangle$.

▶ **Lemma 7** (UA-cfw-Indistinguishability [17]). *Let $S_1 \leftarrow S \backslash \{v_0\}$ be the set of accumulated elements in A without the random initial value v_0. A is said to be collision-freeness-weakening (cfw) indistinguishable if for any PPT adversary Adv with knowledge of S_1 and alternative element set S_2, Adv cannot decide with probability better than $\frac{1}{2} + \epsilon(\lambda)$ whether S_1 or S_2 is accumulated in A. Adv has access to oracles ϑ, which it can query with element x_i to execute addition for $x_i \notin S_1 \cup S_2$, deletion for $x_i \in S_1 \cap S_2$, obtain membership proofs for $x_i \in S_1 \cap S_2$ and non-membership proofs for $x_i \notin S_1 \cup S_2$.*

Proof. Let $sp = \langle x, \ell \rangle$ be the secret parameters of $A = \langle g_1^x, g_2^\ell \rangle$. First, we recall that neither x nor ℓ can directly be computed from A under the DL assumption (Assumption 2). However, Adv may attempt to reconstruct A from S_1 and S_2. Let A_1 and A_2 be the accumulators constructed from S_1 and S_2 by Adv using the same public parameters. Recall that A was seeded with an additional random non-zero integer v_0 and random large prime ℓ_0 at setup, which the adversary cannot retrieve from the public parameters. Thus, $A \neq A_1$ and the probability that Adv guesses v_0 and ℓ_0 at random is negligible. Finally, Adv may try to obtain more information about sp through queries to ϑ. Suppose Adv queries ϑ to perform the addition of $x_i \notin S_1 \cup S_2$. Then Adv would only obtain a new version of A, which is still secure under the DL assumption. The same case applies to element deletion. Suppose Adv tasks ϑ with producing a membership proof for some element $x_i \in S_1 \cap S_2$. We see in the NIPoKCR protocol (Algorithm 5) that Adv obtains $Q_1 \leftarrow g_1^{q_x} g_2^{q_e}$ and $Q_2 \leftarrow g_2^{q_\ell}$, which do not leak any data on sp. Similarly, if Adv tasks ϑ with producing a non-membership proof for

Algorithm 7 Chinese-remainder-based accumulator with sublinear storage (CRASSe).

operation acc_setup(λ) **is**
  $\mathcal{G} = \langle \mathbb{G}, A, B \rangle \leftarrow \mathsf{GGen}(\lambda)$;
  $\langle g_1, g_2 \rangle \leftarrow \mathcal{O}_1$;
  Setup H_{Primes} with prime domain $\mathsf{Primes}(b_2, b_1)$;
  $v_0, \ell_0 \xleftarrow{\$} \mathbb{Z}^+ \times \mathsf{Primes}(b_1 + 1, b)$; ▷*Sample a random prime initial value*
  $sp \leftarrow \langle v_0, \ell_0 \rangle$;
  $A_0 \leftarrow \langle g_1^{v_0}, g_2^{\ell_0} \rangle$;
  $pp \leftarrow \langle \mathcal{G}, g_1, g_2, H_{\mathsf{Primes}}, A_0 \rangle$;
  return $\langle pp, sp, A_0 \rangle$.

operation ua_add(pp, sp, A, v) **is**
  if ua_prove_mem(pp, sp, A, v) $\neq$ `abort` **then** return A;
  $\langle \mathcal{G}, g_1, g_2, H_{\mathsf{Primes}} \rangle \leftarrow pp$;
  $\langle x, \ell \rangle \leftarrow sp$;
  $\ell_v \leftarrow H_{\mathsf{Primes}}(H(v))$;
  Solve the CRT for $x' \equiv x \mod \ell$ and $x' \equiv v \mod \ell_v$;
  $\ell' \leftarrow \ell \times \ell_v$;
  $sp \leftarrow \langle x', \ell' \rangle$;
  $A \leftarrow \langle g_1^{x'}, g_2^{\ell'} \rangle$;
  return $\langle sp, A \rangle$.

operation ua_delete(pp, sp, A, v) **is**
  if ua_prove_non_mem(pp, sp, A, v) $\neq$ `abort` **then** return A;
  $\langle \mathcal{G}, g_1, g_2, H_{\mathsf{Primes}} \rangle \leftarrow pp$;
  $\langle x, \ell \rangle \leftarrow sp$;
  $\ell_v \leftarrow H_{\mathsf{Primes}}(H(v))$;
  $\ell' \leftarrow \ell / \ell_v$;
  $x' \leftarrow x \mod \ell'$;
  $sp \leftarrow \langle x', \ell' \rangle$;
  $A \leftarrow \langle g_1^{x'}, g_2^{\ell'} \rangle$;
  return $\langle sp, A \rangle$.

operation ua_prove_mem(pp, sp, A, v) **is**
  return $\mathsf{NIPoKCR}_{\mathsf{prove}}(pp, sp, A, v)$.

operation ua_prove_non_mem(pp, sp, A, v) **is**
  return $\mathsf{NIPoKNCR}_{\mathsf{prove}}(pp, sp, A, v)$.

operation ua_verify_mem(pp, A, v, w) **is**
  return $\mathsf{NIPoKCR}_{\mathsf{verify}}(pp, A, v, w)$.

operation ua_verify_non_mem(pp, A, v, u) **is**
  return $\mathsf{NIPoKCR}_{\mathsf{verify}}(pp, A, v, u)$.

some element $x_i \notin S_1 \cup S_2$, and follows the NIPoKNCR protocol. (Algorithm 6) shows that all data sent to *Adv* is secure under the DL assumption except a single Bezout coefficient, which does not reveal any data on x or ℓ on its own. We conclude that the probability of *Adv* selecting the right set more accurately than at random is negligible. ◀

▶ **Lemma 8** (UA-Undeniability). *Let S be the set of elements accumulated in A. The probability that any PPT adversary Adv outputs w, u for an element e s.t.* ua_verify_mem(A, e, w)/`true` *and* ua_verify_non_mem(A, e, u)/`true` *is negligible.*

Proof. Suppose that there exists an element e which can be proven both membership and non-membership of for an accumulator $A = \langle x, \ell \rangle$. Then the statements $x \equiv e \mod \ell_e$ and $x \not\equiv e \mod \ell_e$ would both be true, which is impossible. ◀

▶ **Lemma 9** (UA-Completeness). *For element $a \in S$ and $b \notin S$,* $\Pr(\text{ua\_prove\_mem}(pp, sp, A, a) = \texttt{abort}) < \epsilon(\lambda)$ *and* $\Pr(\text{ua\_prove\_non\_mem}(pp, sp, A, b) = \texttt{abort}) < \epsilon(\lambda)$.

Proof. The completeness property implies that NIPoKCR and NIPoKNCR are valid proofs of knowledge (PoK) for the relation of congruence. The PoK property of NIPoKCR and NIPoKNCR follow directly from the proofs of PoKRep from Boneh *et al.* [7], on which we base our protocols in the particular case $n = 2$. The modification to prove congruence (and therefore the CRT) is straightforward. We consider accumulator $A = \langle X = g_1^x, L = g_2^\ell \rangle$ with secret parameters $\langle x, \ell \rangle$. To prove membership of an element e with corresponding prime modulus ℓ_e using NIPoKCR, the AM first has to prove that $\ell_e \mid \ell$ by computing q_ℓ s.t. $\ell = \ell_e q_\ell$. Then it commits to q_ℓ using $Q_\ell = g_2^{q_\ell}$. The statement can then be verified against the accumulator by computing $L = Q_\ell^{l_e}$, where $L = g_2^\ell \in A$. Now that the existence of an element paired with ℓ_e is ensured, the prover computes $q_x = \lfloor x/\ell_e \rfloor$ using its secret parameters $\langle x, \ell \rangle$. It also computes $q_e = \lfloor e/\ell_e \rfloor$ to reduce the verification time of the verifier, and commits to all values using $Q = g_1^{q_x} g_2^{q_e}$. The verifier can then compute $r_e = e \mod \ell_e$ and accepts if $X g_2^e = Q^{\ell_e}(g_1 g_2)^{r_e}$. The same principle is used in NIPoKNCR to prove non-membership of an element, where the prover shows that $\ell_e \nmid \ell \vee x \not\equiv e \mod \ell_e$. The statement $\ell_e \nmid \ell$ can simply be proven by finding the Bezout coefficients a, b s.t. $a\ell + b\ell_e = 1$ and obfuscating b using $B = g_2^b$ to prevent the adversary from computing ℓ, as done for non-membership of RSA accumulators [7]. Since $x \not\equiv e \mod \ell_e$, we do not disclose $r_x = x \mod \ell_e$ as it would leak information about the element accumulated with ℓ_e. Instead, the prover sends $R_x = g_1^{r_x}$ and proves that R_x commits to $r_x \in [\ell_e]$ using a range proof. Then, the verifier accepts if $Q_x^{\ell_e} R_x Q_e^{\ell_e} g_2^{r_e} = X g_2^e$ and $R_x \neq g_1^{r_e}$. ◀

▶ **Lemma 10** (UA-Soundness). *Let S be the set of elements accumulated in A. The probability that any PPT adversary Adv outputs w_a, u_b for elements a, b s.t.* ua_verify_mem$(pp, A, a, w_a) =$ `true` $\wedge\, a \notin S$ *or* ua_verify_non_mem$(pp, A, b, u_b) =$ `true` $\wedge$ $a \in S$ *is negligible.*

Proof. We saw in Lemma 9 that membership and non-membership proofs are handled by our proofs of knowledge NIPoKCR and NIPoKNCR. However, what those protocols prove is not strict equality with set elements but congruence, *i.e.*, for any element $e \in S$ and corresponding prime ℓ_e, $x \equiv e \mod \ell_e$. Therefore, if an element $a \notin S$ is associated with prime ℓ_e and $a \equiv e \mod \ell_e$, then $x \equiv a \mod \ell_e$ and membership of a could be proven. To ensure that the probability of such a collision remains negligible no matter the number of accumulated elements in S, we use a random oracle H (Assumption 1) to attribute each element e_i to a random index $i = H(e_i)$, and therefore to its prime ℓ_i. Let us write $a_i \equiv q\ell_i + r_{a_i} \mod \ell_i$.

Without loss of generality, we assume that $r_{a_i} = 0$ to maximize the range of possible values of q. Since $\ell_i \in \mathsf{Primes}(b_2, b_1)$, we know that $2^{b_2} < \ell_i < 2^{b_1}$. Furthermore, since all accumulated elements are b-bit integers, we have $0 \leq a_i < 2^b$. Thus, we have:

$$0 \leq a_i < 2^b,$$
$$0 \leq q\ell_i < 2^b,$$
$$0 \leq q < \frac{2^b}{2^{b_2}}.$$

Because q is incremented with a step of ℓ_i, there can be at most $1 + \frac{2^b}{2^{(2b_2)}}$ distinct values of q. Thus, the probability that an element $a_i \notin S$ is congruent to a random pair $\langle e_i, \ell_i \rangle$ s.t. $e_i \in S$ is $\frac{1+\frac{2^b}{2^{(2b_2)}}}{2^b} = \frac{1}{2^b} + \frac{1}{2^{(2b_2)}}$, which is negligible. The probability of Adv succeeding in colliding with a given index is even lower, that is $\Pr(\mathsf{BinomialDistribution}(1 + \frac{2^b}{2^{(2b_2)}}, \frac{1}{\pi(b_1)-\pi(b_2)}))$. Conversely, it is not possible to prove the non-membership of an element $e \in S$, as one cannot prove that $x \not\equiv e \mod \ell_e$ if $x \equiv e \mod \ell_e$. ◀

B.2 Extended Pedersen commitments for vectors

For our commitment scheme (specified in Section 4.2), we opt for the generalization of Pedersen commitments to vectors as defined in [26], which are both perfectly hiding and additively homomorphic under the discrete log assumption. The commitment scheme has public parameters $(\mathbb{G}, p, g, g_1, ..., g_n)$, where $\mathbb{G}_p$ is a finite cyclic group of prime order and $|p| = \lambda$, and $g, g_1, ..., g_n$ are group elements. A commitment to a vector $\langle m_1, ..., m_n \rangle$ with blinding factor r is of the form $c = g^r \prod_{i=1}^{n} g_i^{m_i}$. By additively homomorphic, we mean that for any commitments C_1, C_2 with openings $\langle m_1, ..., m_n, r \rangle$ and $\langle m'_1, ..., m'_n, r' \rangle$, $C = C_1 + C_2$ opens to $\langle m_1 + m'_1, ..., m_n + m'_n, r + r' \rangle$. Note that all commitments are constant-size (1 group element).

Vector commitments with partial openings. For some applications, a given process may want to only store a partial opening to one of the values of a vector commitment. For instance, a global vector commitment may be used in a system of processes, where each process has a "personal" value in the vector. In this context, each process only stores the (constant-size) global commitment and the (constant-size) partial opening of its personal value, and can prove to other processes that its personal value "belongs" to (*i.e.*, can be opened from) the global commitment (using the partial opening). It is also possible to update one of the values of a vector commitment if we provide (along with the new value) the partial opening of the old value. These vector commitment updates can be applied commutatively, *i.e.*, no matter whether we apply two different updates of two values in a vector commitment in one order or the other, it always yields the same updated vector commitment as a result.

Vector commitments with partial openings can also be implemented using Pedersen commitments [26]. In our agreement proof (AP) implementation (see Appendix B.3), we use a global vector commitment with partial openings that keeps track of the sequence numbers of every process in the system: to increment its sequence number, a process has to provide the partial opening to its old sequence number. As every process only has to store this global commitment and its personal partial opening (which both have $O_\lambda()$ bits), it allows us to obtain an AP implementation with a constant storage cost of $O_\lambda()$.

To support vector commitments with partial openings, we enrich the specification of Section 4.2 with two new operations. Let us consider a vector of values V and its corresponding vector commitment $c = \mathsf{vc\_commit}(V)$.

- $\mathsf{vc\_val\_update}(c, i, o, v')/r$: Takes a vector commitment c, an index i, an opening of the old value at index i, and a new value v', and returns a pair $r = \langle c', o' \rangle$ where c' is an updated vector commitment (committing the same vector as c, except that the i^{th} value is now v') and o' is the opening of v' at index i in c' if o is a valid opening for the i^{th} value of c, or $r = \texttt{abort}$ otherwise.
- $\mathsf{vc\_val\_verify}(c, i, v, o)/r$: Returns $r = \texttt{true}$ if o is a valid opening for the commitment c and value v at index i, or $r = \texttt{false}$ otherwise.

The enriched specification of vector commitments with these new operations stays *correct*, *binding* and *hiding*: informally, if we consider some vector commitment $c_{\texttt{init}}$ for a vector V and some value $v_{\texttt{init}} \in V$ at index i with partial opening $o_{\texttt{init}}$, the VC-Correctness, VC-Binding and VC-Hiding properties of Section 4.2 still hold even if we replace the $\mathsf{vc\_commit}(V)/\langle c, o \rangle$ and $\mathsf{vc\_verify}(c, V, o)/\texttt{true}$ mentions by $\mathsf{vc\_val\_update}(c_{\texttt{init}}, i, o_{\texttt{init}}, v)/\langle c, o \rangle$ and $\mathsf{vc\_val\_verify}(c, i, v, o)/\texttt{true}$, respectively.

B.3 A consensus-free quorum-based Agreement Proof implementation

In this section, we provide an implementation of the agreement proof scheme of Section 4.1 that does not rely on consensus, is succinct, and has a constant storage requirement of $O_\lambda(1)$. Our AP implementation leverages 2 schemes:

1. threshold digital signatures (see Appendix B.3.1), for creating constant-size agreement proofs σ (*i.e.*, quorums of signatures) and verifying them without the set of all public keys;
2. vector commitments with partial openings (see Appendix B.2) for committing the vector of all ⟨sequence number, process identity⟩ pairs of the system.

B.3.1 Threshold signatures

Threshold signatures are a family of aggregated digital signatures that are produced in a system of n processes, by having at least k out of n processes (the threshold) produce individual signatures for the same message, which we call *intermediary signatures*. Once these intermediary signatures have been produced, they are aggregated (typically by a coordinating process) into a fixed-size aggregated signature, called the threshold signature σ. We call a threshold signature with a threshold of k signers out of n processes a *k-threshold signature*. Unlike multi-signature schemes, which keep track of the identities of their signers, threshold signatures use a single system-wide public key, which is the same for all threshold signatures produced by the scheme. The global public key is typically generated during the setup phase of the system. Naturally, a k-threshold signature for a message m is valid iff it aggregates k valid distinct intermediary signatures for m.

Various asymmetric signature schemes can be transformed into threshold signature schemes [46, 44, 16, 47, 33]. Furthermore, the verification algorithm of most threshold versions of a classical signature scheme remains unchanged and can thus be efficiently verified independently of the number of co-signers. In particular, Roast [44] implements an asynchronous threshold Schnorr signature scheme that supports arbitrary choices of t, guarantees unforgeability against a dishonest majority ($n > 2t$), and guarantees that as long as $t + 1$ honest signers participate, a valid signature is outputted the remaining $n - t - 1$ signers are malicious and try to prevent the creation of the signature. It is also optimal-round efficient, with one preprocessing round before knowing the messages to be signed and one signing round per message.

Succinctness of threshold signatures. The most efficient threshold signature implementations (*e.g.*, [44]) all guarantee that, for a fixed λ and any number of signers k, the size and verification time of a threshold signature is equivalent to that of a single intermediary signature. Therefore, these implementations are succinct.

Constant size of the verification data in threshold signatures. Unlike classical signature schemes, or even multi-signature schemes such as BLS [8], which require knowing all the identities of the signers of a quorum of signatures, threshold signature schemes only require knowing one global public key for the system for verifying a quorum of signatures. In other words, given the maximum number of signers n in the system, instead of having quorums of signatures of size $O_\lambda(n)$ bits with classic or BLS-style schemes (as these quorums must also include the whole set of signers), threshold signatures can produce quorums of signatures of size $O_\lambda(1)$ bits (once a quorum has been generated, the individual identities of the signers are not needed for its verification).

Verifying intermediary signatures without storing all individual public keys. As said previously, only a single global public key is needed to verify a threshold signature, instead of all the signers' public keys. However, the coordinator must verify all intermediary signatures' validity before generating the threshold signature. To do that without having to store all the system's public keys (to keep our $O_\lambda(1)$ storage cost), each individual signer can send with their intermediary signature their public key, and a proof that their public key indeed belongs to the global public key. To make sure that it does not save two signatures by the same process, the coordinator saves the signer's identity along with each intermediary signature. Once the threshold signature has been produced, all the temporary data (intermediary signatures and sender identities) can be deleted.

B.3.2 Algorithm

In this section, we present our AP implementation, which eschews consensus by using threshold signatures for quorums. It achieves lightness by using succinct cryptographic primitives and ensuring a constant storage cost of $O_\lambda(1)$.

Algorithm 8 describes the code of this implementation for a correct process p_i and a AP predicate P_A. It uses 5 local variables: $sigs_i$ (a set of intermediary signatures used for generating threshold signatures), v_i (the value of the current ap_prove execution), sn_i (the current sequence number of p_i), sn_c_i (a vector commitment with partial openings, as defined in Appendix B.2, whose vector contains at each index $j \in [1..n]$ the $\langle sn_j, j\rangle$ pair of process p_j), and o_i (the partial opening of the initial $\langle 0, i\rangle$ pair at index i in sn_c_i).

The ap_verify operation simply verifies that the provided AP σ_j is a valid $(\lfloor\frac{n+t}{2}\rfloor+1)$-threshold signature for the provided value v, sequence number sn_j and process p_j (line 1).

The ap_prove operation first verifies that the provided *data* satisfies the AP predicate (line 5). If it does, p_i then saves the provided value v in the local variable v_i (line 6) and increments sn_i to obtain its next sequence number (line 7). Next, p_i generates the first signature of its payload, called the initialization signature (line 8), and broadcasts it in a QUORUMINIT message which requests other processes of the system to sign its payload (line 9). Process p_i then waits for a quorum of intermediary signatures from other processes (line 10), and when it is received, p_i aggregates all intermediary signatures into a $(\lfloor\frac{n+t}{2}\rfloor+1)$-threshold signature (line 11), flushes the temporary data (line 12) and returns the AP and next sequence number (line 13).

When p_i receives a QUORUMINIT message from a process p_j, it first checks that the provided initialization signature is valid (line 15) and that the provided *data* satisfies the AP

predicate (line 16). Then, p_i waits for the provided sn_j to be the next one (in FIFO order) to be processed (line 17). After that, p_i produces its intermediary signature (line 18) and sends it to p_j in a QUORUMSIG message (line 19). Finally, p_i updates the pair of p_j in sn_c_i with its new sn_j (line 20) and saves the new partial opening of this update in o_i if $i = j$ (line 21).

When p_i receives a QUORUMSIG message, it saves the provided intermediary signature if it is valid and if p_i has not already produced an AP for the payload (line 23).

Algorithm 8 Light and consensus-free quorum-based Agreement Proof algorithm for an AP predicate P_A, and assuming $n > 3t$ (code for p_i).

```
1  init: sigs_i ← ∅; v_i ← ⊥; sn_i ← 0; sn_c_i ← global VC for the vector containing a
     ⟨0, j⟩ pair for every j ∈ [1..n]; o_i ← partial opening of the ⟨0, i⟩ pair at index i in
     sn_c_i.
2  operation ap_verify(σ_j, v, sn_j, j) is
3      return { true   if σ_j is a valid (⌊(n+t)/2⌋+1)-threshold signature for ⟨v, sn_j, j⟩,
              { false  otherwise.
4  operation ap_prove(v, data) is
5      if P_A(data) = false then return abort;
6      v_i ← v;                                          ▷save v for condition at line 23
7      sn_i ← sn_i + 1;                                  ▷increment sn_i
8      sig_i ← initialization signature of ⟨v, data, sn_i, i⟩ by p_i;
9      broadcast QUORUMINIT(v, data, sn_i, i, sig_i, o_i);
10     wait (sigs_i has strictly more than (n+t)/2 intermediary signatures for ⟨v, sn_i, i⟩);
11     σ_i ← aggregation of sigs_i into a (⌊(n+t)/2⌋+1)-threshold signature for ⟨v, sn_i, i⟩;
12     sigs_i ← ∅; v_i ← ⊥;                              ▷flush temporary data
13     return ⟨σ_i, sn_i⟩.
14 when QUORUMINIT(v, data, sn_j, j, sig_j, o_j) is received do
15     if sig_j is not a valid initialization signature for ⟨v, data, sn_j, j⟩ by p_j then return;
16     if P_A(data) = false then return;
17     wait (vc_val_verify(sn_c_i, j, ⟨sn_j − 1, j⟩, o_j));
18     sig_i ← intermediary signature for ⟨v, sn_j, j⟩ by p_i;
19     send QUORUMSIG(i, sig_i) to p_j;
20     ⟨sn_c_i, o_j⟩ ← vc_val_update(sn_c_i, j, o_j, ⟨sn_j, j⟩);   ▷update global VC sn_c_i
21     if i = j then o_i ← o_j.                          ▷save new partial opening if p_i is the prover
22 when QUORUMSIG(j, sig_j) is received do
23     if sig_j is a valid intermediary signature for ⟨v_i, sn_i, i⟩ by p_j and AP σ_i for v_i not
         already produced by p_i then sigs_i ← sigs_i ∪ {sig_j}.
```

Let us remark that, like in our asset transfer algorithm of Algorithm 3, we impose the fact that the size of the sn_i variable must be in $O_\lambda(1)$. Let us also note that this implementation of the Agreement Proof scheme has a message complexity of only $O(n)$.

B.3.3 Proof of Algorithm 8

In the following correctness proofs of the AP scheme, we consider an AP object O_A set up with an AP predicate P_A.

▶ **Lemma 11** (AP-Validity). *If σ_i is a valid AP for a value v at sequence number sn_i from a correct process p_i, then p_i has executed O_A.ap_prove$(v, \star)/\langle \sigma_i, sn_i \rangle$.*

Proof. Let us assume that σ_i is a valid AP for value v at sequence number sn_i from a correct process p_i. By definition of the ap_verify operation at line 3, σ_i is a valid $(\lfloor \frac{n+t}{2} \rfloor + 1)$-threshold signature for v at sn_i from p_i. This threshold signature must have been aggregated from at least $\lfloor \frac{n+t}{2} \rfloor + 1$ intermediary signatures. Given the system assumption $n > 3t$, we have $\lfloor \frac{n+t}{2} \rfloor + 1 > \lfloor \frac{4t}{2} \rfloor + 1 = 2t + 1 \geq t + 1$. Therefore, at least one correct process must have produced an intermediary signature for $\langle v, sn_i, i \rangle$ at line 18. However, to execute this line, this correct process must have verified the initialization signature of p_i at line 15. By the unforgeability of signatures, the only way to produce this initialization signature is for p_i to execute line 8, during an O_A.ap_prove$(v, \star)/\langle \sigma_i, sn_i \rangle$ execution. ◀

▶ **Lemma 12** (AP-Agreement). *There are no two different valid APs σ_i and σ'_i for two different values v and v' at the same sequence number sn_i and from the same process p_i.*

Proof. Let us assume, on the contrary, that there exists two different valid APs σ_i and σ'_i for two different values v and v' at the same sequence number sn_i and from the same process p_i (correct or faulty). By definition of the ap_verify operation at line 3, σ_i and σ'_i are valid $(\lfloor \frac{n+t}{2} \rfloor + 1)$-threshold signatures for v and v' (resp.) at sn_i from p_i. These threshold signatures must have been aggregated from the intermediary signatures of two sets of processes, A and B, which have respectively signed $\langle v, sn_i, i \rangle$ and $\langle v', sn_i, i \rangle$. We have $|A| \geq \lfloor \frac{n+t}{2} \rfloor + 1 \leq |B|$, or equivalently, $|A| > \frac{n+t}{2} < |B|$.[14] We thus have $|A \cap B| = |A| + |B| - |A \cup B| > 2\frac{n+t}{2} - |A \cup B| \geq 2\frac{n+t}{2} - n = t$. Therefore, at least one correct process p_j must belong both to A and B, and must have signed both $\langle v, sn_i, i \rangle$ and $\langle v', sn_i, i \rangle$ for $v \neq v'$. But by the fact that correct processes produce intermediary signatures for some sn_i at line 18 and right after update the $\langle sn_i, i \rangle$ pair in their *sn_c* commitment at line 20, it follows that correct processes produce at most one intermediary signature for a given sn_i and sender identity i, which contradicts the fact that p_i must belong both to A and B. ◀

▶ **Lemma 13** (AP-Knowledge-Soundness). *If σ_i is a valid AP for value v at sequence number sn_i from a prover p_i (correct or faulty), then p_i knows some data such that $P_A(data)/$`true`.*

Proof. Let us assume that σ_i is a valid AP for value v at sequence number sn_i from a process p_i (correct or faulty). By definition of the ap_verify operation at line 3, σ_i is a valid $(\lfloor \frac{n+t}{2} \rfloor + 1)$-threshold signature for v at sn_i from p_i. This threshold signature must have been aggregated from at least $\lfloor \frac{n+t}{2} \rfloor + 1$ intermediary signatures. Given the system assumption $n > 3t$, we have $\lfloor \frac{n+t}{2} \rfloor + 1 > \lfloor \frac{4t}{2} \rfloor + 1 = 2t + 1 \geq t + 1$. Therefore, at least one correct process must have produced an intermediary signature for $\langle v, sn_i, i \rangle$ at line 18. However, to execute this line, this correct process must have verified the satisfaction of P_A by the received *data* at line 16, which entails that the prover p_i must have known the *data* such that $P_A(data)/$`true`. ◀

▶ **Lemma 14** (AP-Termination). *Given a correct process p_i that executes O_A.ap_prove$(v, data)/r$ with value v and a data, if $P_A(data) =$ `true`, then $r = \langle \sigma_i, sn_i \rangle$ and σ_i is a valid AP for v at sn_i from p_i, and if $P_A(data) =$ `false`, then $r =$ `abort`.*

Proof. Let us assume that a correct process p_i executes O_A.ap_prove$(v, data)/r$ with value v and a *data*. If $P_A(data) =$ `false`, then p_i passes the condition at line 5 and return `abort`.

[14] Recall that $\forall\, i \in \mathbb{Z}, r \in \mathbb{R} : (i \geq \lfloor r \rfloor + 1) \iff (i > r)$.

If $P_A(data) = \texttt{true}$, p_i continues the execution, produces an initialization signature at line 8, broadcasts a QUORUMINIT message at line 9 and wait for a quorum of signatures from the system at line 10.

Upon receiving this QUORUMINIT message, each correct process p_j passes the conditions at line 15 (as p_i has correctly generated its initialization signature sig_i) and at line 16 (as p_i has verified the satisfaction of P_A by $data$ at line 5), and then wait that the received sn_i is the next in FIFO order to be processed at line 17. As p_i uses its sequence numbers in FIFO order, then p_i has necessarily sent a QUORUMINIT message for each $sn'_i \in [1..sn_i]$, and p_j will eventually receive all these QUORUMINIT messages. By induction, p_j will pass the wait statement at line 17 for each $sn'_i \in [1..sn_i]$, because if $sn'_i = 1$, the condition line 17 is satisfied as the sn_c_j vector commitment is initialized at line 1 with a $\langle 0, i\rangle$ pair at index i, and if $sn'_i > 1$, then p_j will eventually replace the $sn'_i - 2$ by $sn'_i - 1$ at line 20. Therefore, all correct processes, which are at least $n - t$, will eventually pass the wait statement at line 17. Given the system assumption $n > 3t$, we have $n - t = \frac{2n-2t}{2} > \frac{n+3t-2t}{2} = \frac{n+t}{2}$. Therefore, strictly more than $\frac{n+t}{2}$ (or, equivalently, at least $\lfloor \frac{n+t}{2} \rfloor + 1$)[15] correct processes will produce an intermediary signature at line 18 and send it back to p_i at line 19.

Finally, p_i will receive this quorum of intermediary signatures at line 23, which will unlock the wait instruction at line 10, and p_i will aggregate all intermediary signatures into a valid AP σ_i, and will return $\langle \sigma_i, sn_i\rangle$ at line 13. ◀

B.3.4 Consensus-freedom and lightness of Algorithm 8

The consensus-freedom of Algorithm 8 follows trivially from the fact that the only communication primitives that it uses are classic send/receive operations and a best-effort broadcast operation, and because it always terminates in the presence of failures and asynchrony. The lightness of Algorithm 8 comes from its succinctness and constant storage.

- The succinctness of Algorithm 8 comes from the succinctness of threshold signatures: as shown in Appendix B.3.1, the best implementations of threshold signatures guarantee that the size and verification time of threshold signatures (and therefore also the agreement proofs σ_i produced by Algorithm 8) are equivalent to that of a single intermediary signature. Therefore, Algorithm 8 is succinct.
- The constant storage of Algorithm 8 comes from its local variables: the $sigs_i$ set and v_i values are emptied at the end of each ap_prove execution (line 12), and by definition, sn_i and sn_c_i and o_i have $O_\lambda(1)$ bits. Therefore, Algorithm 8 has a constant storage.

B.4 Transparent zk-SNARK with time-optimal prover

For our zk-SNARK scheme, we choose Spartan [45], which is both transparent (trustless setup) and prover time-optimal.

C Trustless system setup

Like most distributed or cryptographic systems, our system requires a setup phase to compute the initial public and private parameters enabling processes to participate in the system. We assume the initial knowledge of a genesis data structure to specify the process identifiers and the initial amounts of their respective accounts. Our system is set up in a trustless manner,

[15] See footnote 14.

i.e., no trusted party is involved in the distributed computation of the system parameters nor holds a trapdoor or secret that could otherwise be used to compromise the system's safety. We refer to the distributed setup operation of the system as system_setup(). Note that the generation of the genesis data is not part of the system_setup(). This trustless setup is possible because the cryptographic schemes implementing our schemes also provide a trustless setup (see Appendix B).

We provide the following implementation draft to demonstrate the feasibility of system_setup().

1. Assume public knowledge of genesis_data $\leftarrow \bigcup_{i=1}^{n} \langle pk_i, A_i, bal\_c_i \rangle$, where A_i is the empty accumulator of process p_i and bal_c_i is a commitment to the initial balance of p_i ($\texttt{init}_i$).
2. Assume private knowledge of the opening of bal_c_i, secret parameters of A_i and secret key of signature key pair $\langle sk_i, pk_i \rangle$ for each process p_i.
3. Execute the setup operation of each cryptographic scheme.
4. Setup the global vector commitment of ⟨sequence number, process identity⟩ pairs (sn_c_i in Algorithm 8) with public nonce H(genesis_data) (with $H()$ a secure hash function), and initial vector $\bigcup_{i=1}^{n} \{0||i\}$ (we concatenate the initial $sn = 0$ with the identity of each process p_i). The sn_c_i commitment is global, and hence is the same for all processes at the setup.
5. Generate a threshold signature σ_i (compatible with Algorithm 8) that will serve as the initial agreement proof of p_i for each $v_i = \langle A_i, bal\_c_i \rangle$.
6. Each process p_i can now safely delete genesis_data and output its initial parameters $\langle sn\_c_i, A_i, bal\_c_i, \sigma_i, bal\_o_i \rangle$.

D Transfer batching

In this section, we briefly propose a method to commit to batches of transfers (instead of single transfers) and to verify those batches succinctly, *i.e.*, faster than if the batched transfers were verified individually. Such a method is advantageous both in terms of anonymity and efficiency, which is explained in more detail in Section 5.5.

Incrementally Verifiable Computation (IVC). Informally, an IVC can be represented as a function F that takes as input a previous execution of F and some additional input. For example, F could be the function implementing the operation process_transfer() of Algorithm 3. Then each execution of F would take as input an accumulator, an agreement proof, a balance commitment and a transfer commitment of a process p_i and update the accumulator and balance commitment of p_i accordingly. Note that by "chaining" several iterations of F, we obtain the processing of a batch of transfers.

Folding scheme for IVC proofs. Folding schemes such as Nova [34] or Sangria [23] allow creating efficient SNARKS that a given number of iterations of an IVC were correctly executed by generating a proof for each step, then combining them successively on-the-fly in constant time (factor of 2 for Nova). Once the prover wishes to demonstrate the correct processing of its IVC iterations, the IVC proof is compressed into a single, small and succinct SNARK proof.