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

Univ Rennes, Inria, CNRS, IRISA, 35042 Rennes-cedex, France

February 18, 2025

Abstract

This paper 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* (*i.e.*, they produce short-sized and quickly verifiable proofs) and each process only stores its own transfers while keeping communication cost as low as possible. 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 proof" and well-known techniques such as commitments, universal accumulators, and zero-knowledge proofs.

Keywords: Anonymity, Asset transfer, Asynchrony, Byzantine Fault Tolerance, Consensus-freedom, Cryptography, Distributed computing, Lightness.

Acknowledgments. The authors would like to thank Maxence Brugères, Petr Kuznetzov, and Victor Languille for their valuable comments which helped improve this paper significantly.

This work is partially supported by the French ANR projects ByBloS (ANR-20-CE25-0002-01) and PriCLeSS (ANR-10-LABX-07-81), and 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

Imagine a world without the ability to send money instantly across the globe, a world where financial borders seemed impassable. This was our world just a few decades ago. The advent of online money transfers, a real silent revolution, has turned our lives upside down and is reshaping the global economic landscape. Its potential for a positive impact on the world (*e.g.*, decentralized and secured financial transactions, democratized access to financial services, micro-payments) is far from exhausted. Existing decentralized money transfer, also called *decentralized or distributed asset transfer*, faces a range of challenges pertaining to privacy (*i.e.*, transfer confidentiality), performance (*i.e.*, throughput and latency), and communication/computational/storage costs. Addressing these challenges amounts to designing a system that enjoys:

- **Anonymity** to hide both the identity of the parties involved in the transfer (*i.e.*, the sender and/or the recipient)[1] and the amount of transferred assets,
- **Lightness** to maintain low operational costs in terms of computation (in particular, ensuring the *succinctness* of security proofs), storage (each participant only needs to store her own transfer history), and communication (*i.e.*, low bit complexity),
- **Consensus freedom** to operate deterministically in an asynchronous failure-prone setting.

Unfortunately, existing systems invariably satisfy only a subset of these three important properties.

First, until recently, all asset-transfer systems relied on a consensus primitive. Consensus was thought necessary to avoid double spending, but as shown in [1, 26, 27], a reliable broadcast primitive whose deliveries respect process sending order is sufficient when each account is owned by a single process. This has direct practical implications because consensus imposes a non-required sequential processing of all the asset transfers (or block by block in the case of blockchains). On the other hand, not only can consensus-free asset-transfer systems process transfers concurrently, resulting in higher throughput [2, 14], but they also make it possible to leverage deterministic algorithms that achieve progress in asynchronous settings. In practical systems, this means not having to wait for synchrony assumptions to be satisfied, a condition that would otherwise lead to long delays in large-scale systems. The interested reader may find possibility and impossibility results for payment channels in asynchronous asset transfer in [37].

Second, even if recent years have shown the emergence of a variety of consensus-free solutions for asset transfer, none of them, except for the recent Zef [3] feature any level of anonymity. Anonymous asset transfer systems typically rely on consensus to enable multiple users to hide behind a set of accounts or tokens. Indeed, recent work [22] has shown that consensus is necessary to guarantee full anonymity in a system where asset transfers from correct processes never fail. Zef [3] circumvents this impossibility by weakening the anonymity requirement, similar to what we do in this paper.

Finally, most systems (including Zef [3]) exhibit high storage and/or communication cost as summarized in Table 1. With respect to storage, most systems require nodes to store the entire history of all asset transfers (*e.g.*, the entire blockchain) or at least a number of transfers proportional to this entire history. With respect to communication, they feature a cost that is at least linear in the number of nodes, and quadratic in the case of Zef (Table 1).

Combining quasi-anonymity, lightness, and consensus freedom. The challenge addressed in this paper consists in designing the first asset transfer system that achieves anonymity, and consensus-freedom while incurring as-low-as-possible storage and communication costs. Specifically, the proposed solution achieves the following properties.

1. **Quasi-anonymity:** it hides the amount and the receiver's identity of each asset transfer.
2. **Lightness:** All the cryptographic schemes it uses are succinct—*i.e.*, with at most polylogarithmic proof size and verification time—the storage cost incurred by each process p_i is linear in the

[1] Note that when the anonymity of only the sender or the recipient is hidden, we talk about quasi-anonymity.

System	Anonymity	Comm.	Storage	Cons.-Free	Model
Zcash [28]	Full	$\Omega(\lambda n)$	$\Omega(\lambda \lvert T \rvert)$	No	Sync.
Mina [9, 41]	None	$\Omega(\lambda n)$	$\Omega(\lambda \lvert T \rvert / n + n \log n + n)$	No	Sync.
Pastro [33]	None	?	$\Omega(\lambda \lvert T \rvert + n)$	Yes	Async.
Zef [3]	Quasi	$\Omega(\lambda n^2)$	$\Omega(\lambda \lvert T \rvert + n)$ [2]	Yes	Async.
This paper	Quasi	$O(\lambda n)$	$O(\lambda + (\lvert T \rvert / n) \log n + n)$	Yes	Async.

Table 1: Comparison with notable existing systems. $|T|$ denotes the total number of transfers in an execution. The storage cost is given per process, and the communication cost is given for the whole system.

number of transfers of p_i for a fixed security parameter λ, and the communication cost of the algorithm remains as low as possible.

3. **Consensus freedom:** The proposed solution consists of a deterministic algorithm that can operate in an asynchronous setting prone to failures, thereby supporting responsive implementations.

Roadmap. This paper is made up of 7 main sections. Section 2 positions the contribution with respect to the state of the art. Section 3 defines the computing system model. Section 4 provides a concurrent specification of quasi-anonymous asset transfer (QAAT). Section 5 informally defines the cryptographic schemes used in our system. Section 6 presents a novel *Agreement Proof* abstraction, our QAAT system, and the intuition behind its correctness. Finally, Section 7 concludes the article.

Mote technical developments, such as the proofs of the QAAT algorithm, and the properties and implementations of Section 5's s chemes appear in appendices.

2 Background and State of the Art

As we briefly mentioned above, some notable systems have addressed each of the challenges we presented individually, but to the best of our knowledge, we are the first to address all of them in a single system. Table 1 compares some of the notable existing asset transfer systems evoked below.

Anonymous and quasi-anonymous asset transfer systems. Anonymous Asset Transfer (AAT) systems have been studied since the introduction of the online payment system Bitcoin in 2008 [36] and can be categorized as either *token-based* or *account-based*. The two types of systems differ by the persistency of the financial medium. A token can only be used once while an account has a recorded balance. To transfer a token, the sender must prove that it possesses it and that this token has never been used before. This is generally achieved by maintaining a list of unspent tokens and a list of spent tokens. Anonymity is typically obtained by relying on cryptographic primitives (Zero-knowledge proofs—ZKPs) that allow one party to convince another party that a given statement is true without divulging any information beyond the fact that the statement is true. An example of token-based systems is Zcash [28].

Asset transfer via accounts requires the sender to prove that its account has sufficient funds. Anonymity is guaranteed by having the sender select at random a subset of its accounts. The Quisquis [19] and the Zether [11] asset-transfer systems represent account-based AATs.

All the systems we mentioned [11, 19, 28] provide full anonymity (*i.e.*, they ensure both sender and receiver anonymity), but they also rely on consensus and therefore cannot be implemented deterministically in an asynchronous failure-prone system. Indeed, it was recently shown [22] that fully anonymous asset transfer requires consensus or equivalently total order to be implemented, if asset transfers from a

[2] The authors of [3] describe a public-key rotation method to further reduce Zef's storage cost, however this mechanism requires additional synchrony assumptions, as discussed in Section 6.2.5.

correct process must never fail.

We show in the this paper that guaranteeing quasi-anonymity in place of anonymity is sufficient to circumvent this impossibility result, similar to what is done by Zef [3].

Light asset transfer systems. The idea of lightness consists in guaranteeing that all underlying cryptographic schemes are *succinct*, that processes only store transactions they are involved in, and that communication cost remains as low as possible. A cryptographic proving scheme is succinct if and only if its proof size and verification time are at most polylogarithmic in the "size of the problem" [6], where the size of the problem depends on the scheme.[3] For instance, it may refer to the number of signatures in an aggregated signature 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 verification 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 the Bitcoin blockchain. The notion of light clients, implemented with the simplified payment verification (SPV) protocol [36] allows clients to only store block headers and Merkle proofs in place of full blocks. Still, their local storage grows linearly with the size of the blockchain. Other solutions, such as the one presented in [12], allow a succinct and secure construction of proofs, *i.e.*, Merkle mountain ranges, but still require the full blockchain to be maintained to update the proof with new blocks. In contrast, non-interactive proofs of proof-of-work succeed in constructing secure and succinct token-based asset transfer systems by sampling a polylogarithmic number of blocks of the blockchain [29, 30] but do not target full anonymity of the transfers. Finally, the Mina system [9] leverages SNARKs (Succinct Non-interactive Arguments of Knowledge) to obtain a succinct blockchain. However, their construction requires maintaining the full knowledge of all the accounts of the system to update the proof when a transfer occurs. In addition, Mina does not provide any level of anonymity for the transfers. Those solutions do not explicitly say that they rely on P2P for their communication. Assuming that this is the case, sending a piece of information costs $n\log(n)$ messages in average.

Consensus-free asset transfer systems. Fundamentally, an asset-transfer system must be double-spending-free; that is, spending a unit of value more than once must be impossible. This means that all parties 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 agree on which transfer (if any) is correct. This relaxed, per-account ordering of transfers can be obtained by Byzantine-tolerant communication primitives weaker than consensus, *e.g.*, *Byzantine-tolerant reliable broadcast* [10].[4] Following [26], several works have proposed payment systems based on reliable broadcast only, namely AFRT20 [1], Astro [14], and FastPay [2]. Astro was later extended to permissionless environments with Pastro [33], combining weighted quorums and proof-of-stake. Building on FastPay, Zef [3] provides quasi-anonymous transfers, but it incurs quadratic communication cost and storage cost linear in the total number of transfers.

3 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 may not follow the algorithm [34, 38].[5] Byzantine

[3]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 storage cost induced by the network routing protocol is ignored.

[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, total ordering is needed [26, 27], *i.e.*, consensus or atomic broadcast.

[5]The assumption $t < n/3$ is only needed for the *Agreement Proof* scheme of our system (see Section 6.1).

processes may collude to fool non-Byzantine (a.k.a. correct) processes.

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 [39].[6]

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 C, detailing how each choice constrains the adversary's power.

Security parameter λ. We denote by λ 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. We further assume that λ dominates $\log n$, *i.e.*, $\lambda = \Omega(\log n)$.

Different notions of adversaries. We consider an adversary denoted *Adv* with both *distributed* and *cryptographic* aspects. The distributed aspect 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, on the other hand, 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.

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 left unspecified, *e.g.*, 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.

4 Quasi-Anonymous Asset Transfer (QAAT): Concurrent Specification

Asset Transfer (*AT*), and its extension, *Quasi-Anonymous Asset Transfer* (*QAAT*) are derived from the *concurrent* asset transfer specification of AFRT20 [1]. We refer to this specification as concurrent because it does not consider asset transfer as a sequential object handling operation invocations in a total order. In the following, we assume that each account is owned by a single process.

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

[6] Note that both these solutions have a communication complexity of at most $O(\lambda n)$.

Notation	Meaning
λ, $\varepsilon(\lambda)$, 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
c, o	Commitment digest, 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
τ:$\langle snd, v, rcv, sn\rangle$	Asset transfer details: sender, amount, receiver, sequence number

Table 2: Key notations used in the paper.

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

The above $\mathsf{balance}()$ operation is weaker than that of AFRT20 [1] as it can only return the balance of the local process, and not of any process of the system. This is needed to ensure confidentiality (since a given process should not be able to read the accounts of other processes).

It is assumed that the account of each process p_i is initialized to a nonnegative value denoted $\texttt{init}_i$ (in our QAAT implementation of Algorithms 1–3, $\texttt{init}_i$ is known only by p_i).

Histories and sequences.

- Let $S = (o_k)_{k\in[1..|S|]}$ be a sequence of operation invocations. S might be finite (in which case $|S| \in \mathbb{N}$), or infinite (in which case $|S| = +\infty$, and $[1..|S|] = \mathbb{N}$). We note $\mathsf{set}(S)$ the set of invocations in S, $\rightarrow_S$ the total order defined by S, and $\mathsf{transferSet}(S)$ the set of $\mathsf{transfer}$ invocations contained in S (*i.e.*, $\mathsf{transferSet}(S) = \{k \in [1,|S|] \mid o_k = \mathsf{transfer}_\star(\star,\star)\}$).
- A local history of a correct process p_i, denoted L_i, is a sequence of operation 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 $o \rightarrow_i o'$.
- In a similar way, a local history of a Byzantine process p_i is a sequence L_i of operations invocations. However, because Byzantine processes might deviate arbitrarily from their prescribed behavior, these invocations do not necessarily correspond to how the behavior of Byzantine processes is *perceived* by correct processes.
- A global history H is an array of n local histories, one for each process: $H = [L_1,...,L_n]$.
- We use the notion of *mock history* to capture the perceived behavior of Byzantine processes [1]. Intuitively, a mock history $\widehat{H}$ of some global history H is a history that preserves the local histories L_i of correct processes p_i but might change the local histories of Byzantine processes. More formally, $\widehat{H} = [\widehat{L}_1,...,\widehat{L}_n]$ is a mock history of $H = [L_1,...,L_n]$ iff $\widehat{L}_i = L_i$ for all correct processes.

Active processes. Given a global history H, we say that a process is *active* in H if it is involved in a $\mathsf{transfer}_i(v,j)$ operation invocation appearing in a local history L_i of H, either as the calling process p_i or as the receiver p_j.

AT-sequence. Given a process ID i and a set of operation invocations O (performed by p_i and other processes), the function $total(i,O)$ returns the balance of the account of p_i resulting from the transfers it sends and receives according to 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 invocations is an *asset-transfer-sequence* (*AT-sequence*) if and only if:

1. $\forall\, i \quad \in [0..n], \forall\, o = \mathsf{balance}_i()/v \quad \in S : v = \mathit{total}\big(i, \{o' \in S \mid o' \rightarrow_S o\}\big)$;
2. $\forall\, i, j \in [0..n], \forall\, o = \mathsf{transfer}_i(v, j) \in S : v \leq \mathit{total}\big(i, \{o' \in S \mid o' \rightarrow_S o\}\big)$.

Informally, these conditions mean that the balance operation must return the balance of the process's 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 according to the sequence S.

We say that a global history H can be *AT-sequenced* iff for every correct process p_i there exists an *AT-sequence* S_i with the following properties:

- S_i contains exactly all of p_i's invocations (both $\mathsf{transfer}_i$ and $\mathsf{balance}_i$ invocations), and all transfers by other processes, *i.e.*, $\mathsf{set}(S_i) = \mathsf{set}(L_i) \cup \bigcup_{j \neq i} \mathsf{transferSet}(L_j)$.
- S_i respects p_i's process order, *i.e.*, $\rightarrow_i \subseteq \rightarrow_{S_i}$.

Intuitively, the above conditions ensure that each process p_i can explain its local execution (Condition 1 of the AT-sequence definition above) from the transfers contained in the system in a way that respects p_i's local order and the invariant that no account should ever be negative (Condition 2 of the AT-sequence definition).

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

- **AT-Termination.** All operation invocations of $\mathscr{A}$ (balance and transfer) by correct processes terminate.
- **AT-Sequentiality.** For any global history H capturing an execution of $\mathscr{A}$, there exists a mock history $\widehat{H}$ of H that can be AT-sequenced.

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

- **QAAT-Receiver-Anonymity.** For any global history H capturing an execution of $\mathscr{A}$, for any invocation $\mathsf{transfer}_i(\star, j)$ contained in H where p_i and p_j are correct processes, then, with high probability, p_j is indistinguishable among all the correct active processes of H.
- **QAAT-Confidentiality.** For any global history H capturing an execution of $\mathscr{A}$, for any invocation $\mathsf{transfer}_i(v, j)$ contained in H where p_i and p_j are correct processes, then, with high probability, v is indistinguishable among any value in $\mathbb{R}^+$.

Informally, if we consider an adversary *Adv* and a $\mathsf{transfer}_i(v, j)$ invocation appearing in a global history H, where p_i and p_j are correct processes, the best *Adv* can do (w.h.p.) 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 inversely proportional to the number of active processes in H, and v with a probability of 0.[7] The above two properties capture that a Quasi-Anonymous Asset Transfer can be made arbitrarily close to this ideal situation by increasing the number of correct active processes and the security parameter λ of the cryptographic schemes. Furthermore, in our QAAT implementation (Algorithms 1–3), since each correct process p_i is the only one knowing its initial account balance $\mathtt{init}_i$, QAAT-Confidentiality ensures that the account balance of p_i also remains private throughout the execution with high probability.

5 A Modular Approach: Underlying Building Blocks

The QAAT algorithm proposed in this paper follows a modular approach. Before moving to our actual contributions in Section 6, we present in this section the cryptographic schemes upon which our QAAT system builds, namely, *Commitments*, *Universal Accumulators* (*UA*), and *Zero-Knowledge Proofs* (*ZKP*). Each scheme features a proving algorithm that aims to convince a verifying algorithm of some

[7] For simplicity, we assume that v is defined on $\mathbb{R}^+$, but in practice, v is a bounded positive number.

claim through a proof. In the following, the proving algorithm is called the prover, while the verifying algorithm is called the verifier. Due to space constraints, the full definition of the properties of these cryptographic schemes is given in Appendix A.

Commitments. A *commitment* scheme allows one to create a commitment c to a chosen value v while keeping this value hidden from others, with the ability to reveal the committed value later through a proof o called opening.

A commitment scheme provides 2 operations: *(i)* $\mathsf{c\_commit}(v)/\langle c,o\rangle$ takes a value v and outputs the corresponding commitment c and opening o, and *(ii)* $\mathsf{c\_verify}(c,v,o)/r$ outputs $r = \texttt{true}$ if o is a valid opening for the commitment c and value v, and $r = \texttt{false}$ otherwise. Informally, a commitment scheme must be *binding* (it opens only to the committed value v) and *hiding* (it does not leak information on its committed value).

Universal Accumulators (UA). A *universal accumulator* (*UA*) scheme [7] produces a short commitment A (the accumulator) to a set of elements E, upon which the prover can produce both *membership proofs* w or *non-membership proofs* u of elements e in the set E, without divulging any other information. We assume each process creates an empty accumulator during the system's setup phase (see Appendix E).

Consider an accumulator A, its associated accumulated set E, and an arbitrary value v. A UA scheme provides 6 operations: *(i)* $\mathsf{ua\_is\_empty}(A)/b$ takes A and outputs $b = \texttt{true}$ if $E = \varnothing$, and $b = \texttt{false}$ otherwise; *(ii)* $\mathsf{ua\_add}(A,v)/A'$ takes A and v and outputs the updated accumulator A' containing v; *(iii)* $\mathsf{ua\_prove\_mem}(A,E,v)/r$ takes A, E, and v and outputs a membership proof $r = w$ of value v in accumulator A if $v \in E$, and $r = \texttt{abort}$ otherwise; *(iv)* $\mathsf{ua\_prove\_non\_mem}(A,E,v)/r$ takes A, E, and v and outputs a non-membership proof $r = u$ of value v in A if $v \notin E$, and $r = \texttt{abort}$ otherwise; *(v)* $\mathsf{ua\_verify\_mem}(A,v,w)/r$ outputs $r = \texttt{true}$ if w is a correct membership proof of v for A, and $r = \texttt{false}$ otherwise; and *(vi)* $\mathsf{ua\_verify\_non\_mem}(A,v,u)/r$ outputs $r = \texttt{true}$ if u is a correct non-membership proof of v for A, and $r = \texttt{false}$ otherwise.

Informally, a UA scheme must be *sound* (proofs are unforgeable), *complete* (the verification of (non-)membership works as intended), *undeniable* (an element cannot have both membership and non-membership proofs), and *indistinguishable* (the accumulator and proofs do not leak information on the set).

Zero-Knowledge Proofs (ZKP). A *zero-knowledge proof* (*ZKP*) scheme [42] produces proofs π that a prover knows some *secret_data* without divulging any other information on it to the verifier. Each object Obj_{ZK} of a ZKP scheme is set up with a specific predicate P_{ZK}, called a *ZKP predicate*, taking as parameters *public_data* (known both by the prover and verifier) and *secret_data* (known only by the prover), and returning `true` or `false`. The prover p_i passes to the $\mathsf{zkp\_prove}$ operation the *public_data* and *secret_data* parameters of P_{ZK}, and the proof π_i is correctly generated only if $P_{ZK}(\textit{public\_data},\textit{secret\_data}) = \texttt{true}$. The P_{ZK} predicate is typically passed to the implicit setup operation of the ZKP scheme.

Consider a ZKP object Obj_{ZK} set up with a ZKP predicate P_{ZK}. A ZKP scheme provides 2 operations: *(i)* $Obj_{ZK}.\mathsf{zkp\_prove}(\textit{public\_data},\textit{secret\_data})/r$ outputs 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 $r = \texttt{abort}$ otherwise, and *(ii)* $Obj_{ZK}.\mathsf{zkp\_verify}(\pi,\textit{public\_data})/r$ outputs the validity $r \in \{\texttt{true},\texttt{false}\}$ of a zero-knowledge proof π w.r.t. the *public_data* and the ZKP predicate P_{ZK}. Informally, a ZKP scheme must be *knowledge-sound* (if a proof π is valid, then the prover knows the associated *secret_data*), *complete* (a valid proof π can be generated from some *public_data* and *secret_data* that satisfy P_{ZK}), and *zero-knowledge* (a proof π does not leak information on *secret_data*).

6 A Light Consensus-Free Quasi-Anonymous Asset Transfer Algorithm

Our novel asset-transfer system leverages the building blocks presented in Section 5 and comprises two main contributions. The first, Agreement Proofs, presented in Section 6.1 lies in a novel construct that produces transferable proofs that the processes have reached an agreement on some payload. The second, presented in Section 6.2 consists of our novel asset-transfer algorithm, which leverages the building blocks presented in section 5, and our novel Agreement Proofs to guarantee *quasi-anonymity*, *lightness*, and *consensus freedom*.

6.1 A new distributed scheme: Agreement Proofs (AP)

An *Agreement proof* scheme (*AP*) is a novel distributed scheme defined by two (explicit) operations ap_prove and ap_verify. It aims at producing transferable agreement proofs (APs) that the system has reached an agreement regarding some payload value v originating from a given sender/prover p_i with sequence number sn_i. Sequence numbers uniquely identify each proof the prover generates using ap_prove, making the scheme *multi-shot* (*i.e.*, a given prover can generate multiple proofs). Our asset-transfer algorithm, presented in Section 6.2, leverages APs to prevent double-spending. Hence, the AP scheme, specified in the following, provides easily interpretable properties that formalize the notion of cryptographic certificates, which are often used in distributed systems [3]. In Appendix D, we provide a consensus-free AP implementation with a storage complexity of $O(\lambda + n)$ and a communication complexity of $O(n\lambda)$.

AP predicate. Each object of an AP scheme is set up with a specific predicate P_A, called an *AP predicate*, taking in the AP's sequence number *sn* and some arbitrary *data* as parameters, and returning `true` or `false`. The prover p_i passes the payload v and the *data* parameter of P_A to the ap_prove operation, which generates a correct proof σ_i only if $P_A(v, data, sn_i) = \texttt{true}$, where sn_i is the sequence number of the current ap_prove invocation by p_i, *i.e.*, the number of times the prover p_i has invoked ap_prove up to the current invocation. We further assume that during system initialization, a valid *genesis AP* can be generated by the set-up procedure for a pair $(v, data)$ at sequence number $sn_i = 0$. (We return to the set-up procedure in Section 6.2.1 and discuss implementation details in Appendix E.)

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

- $Obj_A.\texttt{ap\_prove}(v, data)/r$: Given the prover p_i, a payload value v and some *data*, the operation returns $r = \sigma_i$ if the predicate $P_A(v, data, sn_i)$ is `true`, where sn_i is the sequence number of the current ap_prove invocation by p_i, and σ_i is an agreement proof for value v at sequence number sn_i, or $r = \texttt{abort}$ otherwise.
- $Obj_A.\texttt{ap\_verify}(\sigma_j, v, sn_j, j)/r$: The operation 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 Obj_A, a payload 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 $Obj_A.\texttt{ap\_verify}(\sigma, v, sn_i, i)$ by any correct process would return true.

AP properties. An AP object Obj_A set up with an AP predicate P_A satisfies four properties.

- **AP-Validity.** If σ_i is a valid AP for a value v at sequence number $sn_i > 0$ from a correct prover p_i, then p_i has executed $Obj_A.\texttt{ap\_prove}(v, \star)/\sigma_i$ as its sn_i^{th} invocation of $Obj_A.\texttt{ap\_prove}(\star, \star)$.
- **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. More formally, $Obj_A.\texttt{ap\_verify}(\sigma_i, v, sn_i, i) = Obj_A.\texttt{ap\_verify}(\sigma_i', v', sn_i, i) = \texttt{true}$ implies $v = v'$.
- **AP-Knowledge-Soundness.** If σ_i is a valid AP for value v at sequence number $sn_i > 0$ from a prover p_i (correct or faulty), then p_i knows some *data* such that $P_A(v, data, sn_i)/\texttt{true}$.
- **AP-Termination.** Given a correct process p_i that executes $Obj_A.\texttt{ap\_prove}(v, data)/r$ with value v and a *data*, if $P_A(v, data, sn_i) = \texttt{true}$ where sn_i is the sequence number of the current ap_prove

1 **init:** $bal_i \leftarrow \texttt{init}_i$; $sn_i \leftarrow 0$; $\langle bal\_c_i, bal\_o_i \rangle \leftarrow \texttt{c\_commit}(\texttt{init}_i)$; $T_i \leftarrow \varnothing$;
$A_i \leftarrow$ 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.

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

invocation by p_i, then $r = \sigma_i$ where σ_i is a valid AP for v at sn_i from p_i. If $P_A(v, data, sn_i) = \texttt{false}$ then $r = \texttt{abort}$.

6.2 QAAT Algorithm

Our quasi-anonymous asset-transfer (QAAT) algorithm leverages agreement proofs to guarantee that there can be at most one transfer per sequence number from any process p_i ensuring that a process cannot spend the same funds twice. In addition, it leverages non-membership proofs to guarantee that a given transfer is not already in the receiver's accumulator, ensuring that a process cannot redeem the same transfer twice.

6.2.1 Setup and initialization of process variables

We present in Algorithm 1 the initialization of each of the variables maintained by each process p_i in our system: bal_i (the balance of process p_i, only known to p_i and initialized to $\texttt{init}_i$), sn_i (the current sequence number of p_i, *i.e.*, the one for p_i's latest transfer, initialized to 0), bal_c_i and bal_o_i (respectively the commitment and opening of $bal_i = \texttt{init}_i$), T_i (the set of transfers details of p_i, including debits and credits, initially empty), A_i (a universal accumulator of T_i, recording all p_i's debits and credits and initialized to the empty accumulator), 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). We assume that the bal_i and sn_i variables are of constant size (*e.g.*, 64 bits).

To initialize all these variables, and in particular σ_i, which involves communication among the processes, we employ the trustless setup procedure presented in Appendix E.

6.2.2 AP and ZKP predicates

Our system relies on an AP object *AccountUpdate*, and a ZKP object *TransferValidity*, both shown in Algorithm 2. They are respectively set up with the AP predicate P_A and the ZKP predicate P_{ZK} on line 2. A process creates new AP and ZKP objects 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.). The statement **assert** B, where B evaluates to a Boolean, is a shortcut for "**if** $\neg B$ **then** `return false`". Predicates return `true` by default.

Predicate P_A. The P_A predicate takes 3 parameters: the payload v (used both for the AP generation and verification), the *data* (used only for the AP generation), and the current sequence number of the prover sn_{pvr}. The v parameter contains the new accumulator of the prover, A'_{pvr}, as well as the commitment to its new balance, $bal\_c'_{pvr}$. The *data* parameter contains the previous AP of the prover, σ_{pvr}, the current ZKP of the prover, π_{pvr}, and some data $old\_state\_data_{pvr}$ about the state of the prover before the transfer. This $old\_state\_data_{pvr}$ contains the prover's preceding accumulator A_{pvr} and a commitment to its previous balance bal_c_{pvr}.

P_A starts with some unpacking (line 4), and tuple concatenation (operation $\oplus$) to obtain the public data $public\_data_{pvr}$ describing the current transfer (line 5). $public\_data_{pvr}$ encompasses the prover's accumulator before applying the transfer A_{pvr}, a commitment bal_c_{pvr} to the prover's old balance, the

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

3 **predicate** $P_A(\langle A'_{pvr}, bal\_c'_{pvr}\rangle, \langle \sigma_{pvr}, \pi_{pvr}, old\_state\_data_{pvr}\rangle, sn_{pvr})$ **is**

▷*Unpacking data about the prover's preceding state*

4 $\langle A_{pvr}, bal\_c_{pvr}\rangle \leftarrow old\_state\_data_{pvr}$;

5 $public\_data_{pvr} \leftarrow \langle A_{pvr}, bal\_c_{pvr}\rangle \oplus \langle A'_{pvr}, bal\_c'_{pvr}, sn_{pvr}\rangle$;

▷π_{pvr} *proves the knowledge of a valid transfer fulfilling the ZKP predicate* P_{ZK} *(line 9) from* $\langle A_{pvr}, bal\_c_{pvr}\rangle$ *to* $\langle A'_{pvr}, bal\_c'_{pvr}\rangle$

6 **assert** $TransferValidity$.zkp_verify$(\pi_{pvr}, public_data_{pvr})$;

▷*If this is the prover's first transfer, its preceding accumulator* A_{pvr} *is empty*

7 **if** $sn_{pvr} = 1$ **then assert** ua_is_empty(A_{pvr});

▷*The prover's preceding AP* σ_{pvr} *is valid at sequence number* $sn_{pvr} - 1$

8 **assert** $AccountUpdate$.ap_verify$(\sigma_{pvr}, \langle A_{pvr}, bal_c_{pvr}\rangle, sn_{pvr} - 1, pvr)$.

9 **predicate** $P_{ZK}(public\_data, secret\_data)$ **is**

▷*Unpacking public and private data*

10 $\langle A_{pvr}, bal\_c_{pvr}, A'_{pvr}, bal\_c'_{pvr}, sn_{pvr}\rangle \leftarrow public\_data$;

11 $\langle A'_{snd}, bal\_c_{snd}, \sigma_{snd}, w_{snd}, \tau, bal_{pvr}, bal\_o_{pvr}, bal\_o'_{pvr}, u_{pvr}\rangle \leftarrow secret\_data$;

▷*The public commitment for balance should match the private data*

12 **assert** c_verify$(bal_c_{pvr}, bal_{pvr}, bal_o_{pvr})$;

▷*The prover's accumulator has received the transfer* τ, $A'_{pvr} = A_{pvr} \uplus \{\tau\}$

13 **assert** ua_verify_non_mem$(A_{pvr}, \tau, u_{pvr}) \wedge$ ua_add$(A_{pvr}, \tau) = A'_{pvr}$;

▷*The prover's balance has been properly updated*

14 $\langle snd, v, rcv, sn_{snd}\rangle \leftarrow \tau$; **assert** $v \geq 0$;

▷*Trfs of sending APs are tagged with the AP's s.n. / a sending process has sufficient funds*

15 **if** $pvr = snd$ **then assert** $sn_{snd} = sn_{pvr} \wedge bal_{pvr} \geq v$;

16 **if** $pvr = snd = rcv$ **then assert** c_verify$(bal_c'_{pvr}, bal_{pvr}, bal_o'_{pvr})$;

17 **else if** $pvr = snd$ **then assert** c_verify$(bal_c'_{pvr}, bal_{pvr} - v, bal_o'_{pvr})$;

18 **else if** $pvr = rcv$ **then**

19 **assert** c_verify$(bal_c'_{pvr}, bal_{pvr} + v, bal_o'_{pvr})$;

▷*A receiving prover implies a valid corresp. sending AP* σ_{snd} *from the sender*

20 **assert** ua_verify_mem$(A'_{snd}, \tau, w_{snd})$;

21 **assert** $AccountUpdate$.ap_verify$(\sigma_{snd}, \langle A'_{snd}, bal_c_{snd}\rangle, sn_{snd}, snd)$;

22 **else** return `false`.

Algorithm 2: AP and ZKP predicates of Algorithm 3.

prover's accumulator after the transfer A'_{pvr}, and a commitment $bal\_c'_{pvr}$ to the prover's new balance. P_A then checks that the ZKP π_{pvr} is valid (line 6) using the public data describing the transfer. Then, if this is the first transfer of the prover (debit or credit), P_A checks that the prover's preceding accumulator is empty (line 7). Finally, P_A verifies the prover's preceding AP (line 8).

Predicate P_{ZK}. The P_{ZK} predicate takes two parameters as input: *public_data* and *secret_data*. The former, *public_data*, consists of the tuple constructed at line 5. The latter, *secret_data*, consists of the sender's accumulator A_{snd} (recall that the sender is not always the prover), a commitment bal_c_{snd} of the sender's balance, 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 as in this

case there are not used in the body of P_{ZK}), the transfer details τ, the prover's opening of the transfer bal_o_{pvr}, the prover's balance bal_{pvr}, 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).

All the checks of P_{ZK} are done in zero-knowledge, without divulging any data to the verifier(s). P_{ZK} first checks that the commitments to the prover's balance and to the prover's transfer indeed open respectively to its balance (line 12). P_{ZK} then checks that the prover's accumulator has been correctly updated, *i.e.*, that the transfer did not already belong to the prover's old accumulator, and that the prover's new accumulator can be obtained by adding the transfer to its old accumulator (line 13). Finally, the P_{ZK} predicate verifies two properties of the transfer: *(i)* that the prover's balance has been updated according to the transfer's information in τ (lines 14–19), and *(ii)*, in case the prover is the receiver, that the passed AP σ_{snd} is valid and matches τ on the sender's side (lines 20–21). In more detail, P_{ZK} first verifies that if the prover is a sender, the transfer τ is stamped with the same sequence number sn_{pvr} as that of the prover's current AP (line 15, sn_{pvr} is passed as a parameter to P_{ZK} from P_A at line 6). Then, if the prover, sender and receiver are the same, P_{ZK} checks that the prover's balance remains unchanged (line 16)[8]. Otherwise, if the prover is only the sender, P_{ZK} ensures the transfer amount has been subtracted from the prover's balance, and that the prover had enough funds to perform the transfer (line 17). Finally, if the prover is only the receiver, then P_{ZK} verifies that the new prover's balance was obtained by adding the transfer's amount to its old balance (line 19), that the sender's accumulator contains the transfer (line 20), and that the sender's AP is valid (line 21). If the prover is neither the sender nor the receiver of τ, P_{ZK} returns `false` (line 22).

6.2.3 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 23). In the transfer operation, p_i first checks it has enough funds (line 25). Then p_i creates the transfer details τ with its next sequence number (line 26), processes its own transfer using the process_transfer internal operation (line 27), creates a membership proof w_i of the transfer in its accumulator (line 28), sends all the necessary information to the receiver in a TRANSFER message (line 29) and returns `commit` (line 30). 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 33).

In the process_transfer internal operation, p_i first computes its new balance bal'_i depending on whether it is the sender or receiver (line 35), and commits its new value (line 36). Next, p_i proves the non-membership of the transfer τ in its old accumulator A_i (line 37) and creates its new accumulator by adding the transfer (line 38). Process p_i then constructs the ZK proof π that the transfer τ is valid (lines 39–42). It 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 Agreement Proof (AP) predicate (line 43). Finally, p_i generates the AP σ_i (line 44) before updating its local state (line 45).

6.2.4 Intuition of Algorithm 3's proofs

In this section, we sketch the high-level intuition behind the correctness of our solution. The full correctness developments are given in Appendix B.

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

Lemma 6.1 (AT-Sequentiality)**.** *For any global history H capturing an execution of* Algorithm 3*, there exists a mock history $\widehat{H}$ of H that can be AT-sequenced.*

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

23 **operation** balance() **is** return bal_i.

24 **operation** transfer(v, j) **is**
25 **if** $v < 0 \vee v > bal_i$ **then** return `abort`;
26 $\tau \leftarrow \langle i, v, j, sn_i + 1 \rangle$;
27 process_transfer$(\tau, \bot, \bot, \bot, \bot)$; ▷*Producing ZKP and Agreement Proof for* τ
28 $w_i \leftarrow$ ua_prove_mem(A_i, T_i, τ); ▷*Membership proof that* τ *is now in* A_i
29 ra_send TRANSFER$(\tau, \sigma_i, A_i, w_i, bal_c_i)$ to p_j; ▷ra_send *is receiver-anonymous*
30 return `commit`.

31 **when** TRANSFER$(\tau, \sigma_j, A_j, w_j, bal_c_j)$ **is** ra_received **from** p_j **do**
32 **if** $\left\{ \begin{array}{l} \tau.rcv = i \wedge \text{ua\_verify\_mem}(A_j, \tau, w_j) \wedge \\ AccountUpdate.\text{ap\_verify}(\sigma_j, \langle A_j, bal\_c_j \rangle, \tau.sn, j) \end{array} \right\}$ **then**
33 process_transfer$(\tau, A_j, bal_c_j, \sigma_j, w_j)$.

34 **internal operation** process_transfer$(\tau, A_j, bal_c_j, \sigma_j, w_j)$ **is**
▷*Computing new balance with corresponding commitment and opening*
35 $bal'_i \leftarrow bal_i$; **if** $i = \tau.snd$ **then** $bal'_i \leftarrow bal'_i - \tau.v$; **if** $i = \tau.rcv$ **then** $bal'_i \leftarrow bal'_i + \tau.v$
36 $\langle bal\_c'_i, bal\_o'_i \rangle \leftarrow$ c_commit(bal'_i); ▷*New commitment and opening for* bal'_i
37 $u_i \leftarrow$ ua_prove_non_mem(A_i, T_i, τ); ▷*Non-membership proof,* τ *not in* A_i
38 $A'_i \leftarrow$ ua_add(A_i, τ); ▷*Adding new transfer to accumulator*
▷*Constructing ZK proof that transfer is valid*
39 $old\_state\_data \leftarrow \langle A_i, bal\_c_i \rangle$;
40 $public\_data \leftarrow old\_state\_data \oplus \langle A'_i, bal\_c'_i, sn_i + 1 \rangle$;
41 $secret\_data \leftarrow \langle A_j, bal\_c_j, \sigma_j, w_j, \tau, bal_i, bal\_o_i, bal\_o'_i, u_i \rangle$;
42 $\pi \leftarrow TransferValidity$.zkp_prove$(public_data, secret_data)$;
▷*Obtaining Agreement Proof (AP) on transfer's validity*
43 $data \leftarrow \langle \sigma_i, \pi, old\_state\_data \rangle$;
44 $\sigma_i \leftarrow AccountUpdate$.ap_prove$(\langle A'_i, bal_c'_i \rangle, data)$;
▷*Updating local state*
45 $sn_i \leftarrow sn_i + 1$; $A_i \leftarrow A'_i$; $bal_i \leftarrow bal'_i$; $bal\_c_i \leftarrow bal\_c'_i$; $bal\_o_i \leftarrow bal\_o'_i$.

Algorithm 3: QAAT algorithm (code for p_i).

Proof sketch. The proof first constructs the mock history $\widehat{H}$ by starting from the UAs linked to the APs generated by correct processes, and then recursively traversing these UAs and APs using the predicates P_A and P_{ZK} to uncover UAs issued by Byzantine processes that are causally linked to the operations of correct processes. Each UA yields information on a transfer invocation at a given sequence number, which we then use to construct the mock local execution $\widehat{L}_j$ of each Byzantine process p_j.

Given a correct process p_i, we then construct a partial $\rightsquigarrow_i$ order on $\mathscr{S}_i = \mathsf{set}(\widehat{L}_i) \cup \bigcup_{j \neq i} \mathsf{transferSet}(\widehat{L}_j)$. This construction is somewhat technical for two reasons: (1) Transfers might not be received in the order they were issued (in particular by p_i), which implies $\rightsquigarrow_i$ should not enforce any local process order $\rightarrow_j$ other than that of p_i. (2) Simultaneously, $\rightsquigarrow_i$ should be constraining enough to guarantee that balance invocations by p_i exactly reflect previous transfers (Property n.1 of an AT-sequence in Section 4) and that each transfer is backed by sufficient funds (Property n.2). We construct $\rightsquigarrow_i$ incrementally using two binary relations: $\overset{\mathbf{1}}{\rightsquigarrow}_i$ that captures p_i's local order and ensures that all transfers are sufficiently funded, and $\overset{\mathbf{2}}{\rightsquigarrow}_i$ to guarantee balance invocations by p_i can be properly explained. Part of the proof focuses on the acyclicity of $\overset{\mathbf{1}}{\rightsquigarrow}_i \cup \overset{\mathbf{2}}{\rightsquigarrow}_i$, so that $\rightsquigarrow_i$ can be defined as the transitive closure

$(\overset{\mathbf{1}}{\rightsquigarrow}_i \cup \overset{\mathbf{2}}{\rightsquigarrow}_i)^+$. Finally, we chose S_i as a topological sort of $(\mathscr{S}_i, \rightsquigarrow_i)$ and prove it fulfills the two properties of AT-Sequences. (Full derivations in Appendix B.1.) □

Lemma 6.2 (AT-Termination)**.** *All operation invocations of* Algorithm 3 *(*balance *and* transfer*) terminate for correct processes.* (Proof in Appendix B.1.)

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

Lemma 6.3 (QAAT-Receiver-Anonymity)**.** *For any global history H capturing an execution of $\mathscr{A}$, for any invocation* $\mathsf{transfer}_i(\star, j)$ *contained in H where p_i and p_j are correct processes, then, with high probability, p_j is indistinguishable among all the correct active processes of H.* (Proof in Appendix B.2.)

Lemma 6.4 (QAAT-Confidentiality)**.** *For any global history H capturing an execution of $\mathscr{A}$, for any invocation* $\mathsf{transfer}_i(v, j)$ *contained in H where p_i and p_j are correct processes, then, with high probability, v is indistinguishable among any value in $\mathbb{R}^+$.* (Proof in Appendix B.2.)

Succinctness. The overall succinctness of our system stems from the succinctness of our Agreement Proof implementation (Appendix D.5), the constant size of the digests and proofs of RSA accumulators (Appendix C.2) and of commitments (Appendix C.3), and of the succinctness of Spartan zk-SNARKs (Appendix C.4). By definition, the proofs of computation produced by the prover of a succinct ZKP scheme are succinctly verified. Therefore, primitives whose verification is encapsulated inside the ZKP predicate only need to be constant size for our scheme to be succinct (although accumulator (non-)membership proof verification and commitment openings can, in fact, be succinct).

Storage of $O(\lambda + (|T|/n)\log n + n)$ bits per correct process. In our system, each correct process p_i only stores its local transfer details, the sequence number of each of the n processes, and a few cryptographic structures, yielding a storage of $O(\lambda + (|T|/n)\log n + n)$, where T is the set of all transfers in the system. Let us consider the size of p_i's local variables. By definition, bal_i and sn_i are constant-size (see Section 6.2.3). Moreover, bal_c_i and bal_o_i have $O(\lambda)$ bits with a constant size commitment scheme (Appendix C.3), A_i has $O(\lambda)$ bits with the constant size RSA accumulator implementation (Appendix C.2), and σ_i has $O(\lambda)$ bits with our implementation of agreement proofs (Appendix D). Moreover, our AP algorithm stores $O(\lambda + n)$ bits per correct process (Appendix D). Finally, the T_i set is of size $O(|T|/n)$ (where T is the set of all transfers of the system) and contains transfer details of size $O(\log n)$ bits (because of process IDs). This amounts to a total storage cost of $O(\lambda + (|T|/n)\log n + n)$.

Communication of $O(n\lambda)$ bits overall. In our system, a $\mathsf{transfer}_i(\star, j)$ invocation by a correct process p_i entails the following communications: *(i)* the ap_prove invocation by the sender p_i at line 44, *(ii)* the ra_send of the transfer details from the sender to the receiver at line 29, and *(iii)* the ap_prove invocation by the receiver p_j at line 44 (which is only guaranteed to happen if p_j is correct). As our Agreement Proof implementation (Appendix D) communicates only $O(\lambda n)$ bits overall (Appendix D.7), *(i)* and *(iii)* incur an overall communication of $O(\lambda n)$. For *(ii)*, the receiver-anonymous ra_send operation can be implemented with an overall communication cost of $O(\lambda n)$, as discussed in Section 3. As a result, our system's overall communication cost is $O(\lambda n)$.

Consensus-freedom. This directly comes from our distributed Agreement Proof implementation (Appendix D), which does not rely on strong agreement such as consensus (Appendix D.4).

6.2.5 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 an optimization method for ZK proofs known as *folding* [32] in Appendix F.

Reducing local storage (public key rotation). The QAAT system presented in this article is light, *i.e.*, the storage cost per process p_i is in $O(\lambda + (|T|/n)\log n + n)$, where $|T|$ is the set of all transfers (debits and credits) of the system. 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 remove the need to store past transfers. 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). 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, and obtains a storage cost of $O(\lambda + n)$. 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 her. 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. This method is briefly mentioned by Zef [3] as a solution to safely delete the data of unused accounts. To circumvent the incompatibility of its asynchronous model and the handshake, Zef assumes that each transfer is initiated in a synchronous environment outside the system where the receiver transmits its public key to the sender.

Towards full anonymity. Our system is not fully anonymous, as it guarantees Receiver Anonymity and Confidentiality, but not that the transfer issuer remains anonymous (*i.e.*, 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 some well-chosen moments). The design of the heuristics to choose the moments to issue empty transfers is left to future work.

7 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 light 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 $O(\lambda|T|/n)$ storage cost, where T is the set of all transfers of the system. 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.

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 that systems with low verification, storage, and network costs can still guarantee strong privacy features. 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 [33].

A Underlying Building Blocks: Full Definitions

This appendix section presents the full definitions of the cryptographic schemes used in our QAAT system, namely Commitments, Universal Accumulators (UA), and Zero-Knowledge Proofs (ZKP). More precisely, this section provides the operations and properties of these schemes.

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 $\mathbb{F}$ (see Table 3). 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, the schemes presented in this section do not have explicit termination properties, as their operations only consist of local mathematical computations.

We further denote by $\varepsilon(\lambda)$ a positive number between 0 and 1 that can be made arbitrarily small by increasing λ.

Set	Meaning
$\mathbb{D}$ (for "data")	Input set (values) of the proving operations of the cryptographic schemes
$\mathbb{F}$ (for "finite")	Output set (proofs) of the proving operations of the cryptographic schemes

Table 3: Input and output sets of the cryptographic schemes of Appendix A.

A.1 Commitment schemes

A commitment scheme is a cryptographic primitive that enables a prover to succinctly commit to a value v in the form a small digest c (the commitment) and compute a proof o (the opening) that the commitment "opens" to v. Commitment schemes must be binding, *i.e.*, the prover cannot find another value that matches the digest. Most schemes also have a hiding property, meaning that the digest does not reveal any information about the committed value. Some schemes allow digests to be combined to perform operations (*e.g.*, addition or multiplication) on committed values, such as homomorphic schemes. There are many variants of commitments to handle different types of data, such as polynomials, vectors, or even functions. In our system, we use a simple *commitment* scheme (C) to a bounded integer.

Commitment operations.

- $\mathsf{c\_commit}(v)/\langle c,o\rangle$ takes a value $v \in \mathbb{D}$ and outputs the corresponding commitment c and opening o.
- $\mathsf{c\_verify}(c,v,o)/r$ outputs $r = \texttt{true}$ if o is a valid opening for the commitment c and value $v \in \mathbb{D}$, and $r = \texttt{false}$ otherwise.

Commitment properties.

- **C-Correctness.** $\forall\, v \in \mathbb{D}$, if $\mathsf{c\_commit}(v)/\langle c,o\rangle$ then $\mathsf{c\_verify}(c,v,o)/\texttt{true}$.

  Informally, a commitment created from a value opens to this value (using an opening).
- **C-Binding.** $\forall\, c,o,o' \in \mathbb{F}, \forall\, v,v' \in \mathbb{D} : \Pr\big(\mathsf{c\_verify}(c,v,o)/\texttt{true} \wedge \mathsf{c\_verify}(c,v',o')/\texttt{true} \wedge v' \neq v\big) < \varepsilon(\lambda)$. Informally, a commitment opens to only one value (w.h.p.).

- **C-Hiding.** $\forall\, c_1, c_2 \in \mathbb{F}, \forall\, v \in \mathbb{D}, \mathsf{c\_commit}(v)/\langle c, \star \rangle : |\Pr(c = c_1) - \Pr(c = c_2)| < \varepsilon(\lambda)$.

  Informally, a commitment does not leak any information on its committed value (w.h.p).

A.2 Universal Accumulators (UA)

An *accumulator* scheme (notion introduced in [4]) produces a short commitment to a set of elements E. 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. RSA accumulators [7] are a notable implementation of this scheme. We assume that the empty accumulator of each process is created during the system's setup phase (see Appendix E).

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

- $\mathsf{ua\_is\_empty}(A)/b$: Takes A and outputs $b = \texttt{true}$ if $E = \varnothing$, and $b = \texttt{false}$ otherwise.
- $\mathsf{ua\_add}(A, v)/A'$: Takes A and a value v and outputs the updated accumulator A' containing v.
- $\mathsf{ua\_prove\_mem}(A, E, v)/r$: Takes A, E, and a value v and outputs a membership proof $r = w$ of value v in accumulator A if $v \in E$, and $r = \texttt{abort}$ otherwise.
- $\mathsf{ua\_prove\_non\_mem}(A, E, v)/r$: Takes A, E, and a value v and outputs a non-membership proof $r = u$ of value v in A if $v \notin E$, and $r = \texttt{abort}$ otherwise.
- $\mathsf{ua\_verify\_mem}(A, v, w)/r$: Outputs $r = \texttt{true}$ if w is a correct membership proof of value v for A, and $r = \texttt{false}$ otherwise.
- $\mathsf{ua\_verify\_non\_mem}(A, v, u)/r$: Outputs $r = \texttt{true}$ if u is a correct non-membership proof of value v for A, and $r = \texttt{false}$ otherwise.

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

- **UA-Soundness.** $\forall\, v \notin E, \mathsf{ua\_prove\_mem}(A, E, v)/r : \Pr(r \neq \texttt{abort}) < \varepsilon(\lambda)$ and $\forall\, v \in E, \mathsf{ua\_prove\_non\_mem}(A, E, v)/r : \Pr(r \neq \texttt{abort}) < \varepsilon(\lambda)$.

  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.** $\forall\, v \in E, \mathsf{ua\_verify\_mem}(A, v, \mathsf{ua\_prove\_mem}(A, E, v))/r : \Pr(r = \texttt{true}) > 1 - \varepsilon(\lambda)$ and $\forall\, v \notin E, \mathsf{ua\_verify\_non\_mem}(A, v, \mathsf{ua\_prove\_non\_mem}(A, E, v))/r$: $\Pr(r = \texttt{true}) > 1 - \varepsilon(\lambda)$.

  Informally, all honestly accumulated values are verified as true with their corresponding membership proof with a negligible probability of error.

- **UA-Undeniability.** $\forall\, v \in \mathbb{D}, \forall\, w, u \in \mathbb{F}$:
  $\Pr(\mathsf{ua\_verify\_mem}(A, v, w) \wedge \mathsf{ua\_verify\_non\_mem}(A, v, u)) < \varepsilon(\lambda)$.

  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 E is leaked from its associated accumulator A, membership proofs w or non-membership proofs u.[9]

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

[9] For a more formal definition of Indistinguishability, we refer the interested reader to [17].

A.3 Zero-Knowledge Proofs (ZKP)

We refer to the last scheme used in this article as *zero-knowledge proofs* (*ZKP*), which correspond to proofs that a prover knows some secret data without divulging any other information on it to the verifier. Specifically, we use zero-knowledge Succinct Non-Interactive Arguments of Knowledge (zk-SNARKs) [42]. The difference between proof and argument systems comes from the strength of the soundness property. A proof system must be perfectly sound (*i.e.*, withstand a computationally unbounded adversary) whereas an argument system guarantees computational soundness (*i.e.*, against a PPT adversary with a very high probability). The use of arguments is necessary as it was proven by Fortnow [20] that no perfect soundness zero-knowledge proof systems exist for NP-complete problems, while there exist perfect zero-knowledge arguments systems for NP-complete problems (*e.g.*, zk-SNARKs).

ZKP predicate. Each object of a ZKP scheme is set up with a specific predicate P_{ZK}, called a *ZKP predicate*, taking as parameters a *public_data* (known both by the prover and verifier) and a *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 π_i is correctly generated only if $P_{ZK}(\textit{public\_data}, \textit{secret\_data}) = \texttt{true}$. Much like the P_A predicate (Section 6.1), the P_{ZK} predicate is typically passed to the implicit setup operation of the ZKP scheme.

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

- Obj_{ZK}.zkp_prove$(\textit{public_data}, \textit{secret_data})/r$: 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.
- Obj_{ZK}.zkp_verify$(\pi, \textit{public_data})/r$: 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 Obj_{ZK} set up with a ZKP predicate P_{ZK}.

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

B Proofs of our QAAT System (Algorithms 1 to 3)

In this appendix section, we provide full derivations on the proof of correctness (Appendix B.1) and quasi-anonymity (Appendix B.2) of our QAAT system (Algorithms 1–3). Throughout the section, we rely on the full property definitions of the commitment, UA, and ZKP schemes given in Appendix A.

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

B.1 Proof of correctness

B.1.1 Preliminaries

We first introduced a number of preliminary results and definitions that we will use to prove the AT-Sequentiality of Algorithm 3 in Appendix B.1.2. In the following, σ_i^{sn} denotes an AP (Agreement Proof) that is valid at a sequence number *sn* for a process p_i (correct or faulty).

Definition B.1 (Valid UA at a sequence number *sn* for a process p_i)**.** *We say that a Universal Accumulator (UA) A is valid at a sequence number sn for a process p_i (correct of faulty) if there exists a valid Agreement Proof (AP) σ_i^{sn} that is valid at sn for p_i and verifies AccountUpdate*.ap_verify$(\sigma_i^{sn}, \langle A, \star \rangle, sn, i)$. *As a shortcut, we might interchangeably say that A was* issued *by p_i at sequence number sn. When this holds, we note A's sequence number and AP issuer as superscript and subscript, respectively, i.e., $A = A_i^{sn}$.*

Lemma B.2 (Unicity of valid UAs at a sequence number *sn* for a process p_i)**.** *There is at most one UA that is valid at a sequence number $sn \geq 0$ for a process p_i (correct of faulty), i.e., if A and A' are both valid UAs at sn for p_i according to Definition B.1, then $A = A'$.*

Proof. This trivially follows from Definition B.1 and the AP-Agreement property of Agreement Proofs. □

Lemma B.3 (Sequence of preceding UAs of a valid UA)**.** *Consider a valid UA A_i^{sn} at sequence number $sn \geq 0$ issued by a process p_i (correct or faulty). The following holds.*

- *A_i^{sn} can be associated with a unique sequence of preceding UAs, denoted $precedingUAs(A_i^{sn}) = (A_i^0, A_i^1, .., A_i^{sn})$, so that each A_i^k is a valid UA issued by p_i at sequence number k, with $0 \leq k \leq sn$.*
- *The sequences of UAs produced for UAs issued by the same process p_i are prefix-ordered, i.e., if $A_i^{sn_1}$ and $A_i^{sn_2}$ are two valid UAs issued by p_i at sequence numbers sn_1 and sn_2 respectively such that $sn_1 \leq sn_2$, then $precedingUAs(A_i^{sn_1})$ is a prefix of $precedingUAs(A_i^{sn_2})$.*

Proof. Let us consider a valid UA A_i^{sn} at sequence number $sn \geq 0$ issued by a process p_i (correct or faulty).

- If $sn = 0$, we pick $precedingUAs(A_i^0) = (A_i^0)$.
- If $sn > 0$, by Definition B.1, A_i^{sn} is associated with some AP σ_i^{sn} issued by p_i that is valid at sequence number *sn*. By AP-Knowledge-Soundness, p_i knows some $data = \langle \sigma_i', \star, \star \rangle$ s.t. $P_A(\star, \langle \sigma_i', \star, \star \rangle, sn)$ is true. Line 8 of the code of Predicate P_A (Algorithm 2) implies that σ_i' is a valid AP at sequence number $sn - 1$ from p_i for some payload $\langle A_{pvr}, \star \rangle$. A_{pvr} fulfills Definition B.1, and is therefore a valid UA at sequence number $sn - 1$ from p_i. By induction, we obtain that, for each sn' s.t. $0 \leq sn' \leq sn$ there is a valid UA $A_i^{sn'}$ at sn' from p_i. We denote this sequence as $precedingUAs(A_i'^{sn}) = (A_i^0, A_i^1, .., A_i^{sn})$.

By Lemma B.2, $precedingUAs(A_i'^{sn})$ is unique. We say that the sequence of valid UAs $(A_i^{sn'})_{0 \leq sn' \leq sn}$ is the *sequence of preceding UAs* of A_i^{sn}. Prefix ordering similarly follows from Lemma B.2. □

Definition B.4 (Valid transfer τ at a sequence number *sn* for a process p_i)**.** *We say that that a transfer $\tau = \langle snd, v, rcv, sn_{snd} \rangle$ is valid at a sequence number $sn > 0$ for a process p_i (correct of faulty) if there exist a non-membership proof u and two valid UAs A_i^{sn} and A_i^{sn-1} issued by p_i at sequence numbers sn and $sn - 1$ respectively, such that:*

$$\mathsf{ua\_verify\_non\_mem}(A_i^{sn-1}, \tau, u) \wedge \mathsf{ua\_add}(A_i^{sn-1}, \tau) = A_i^{sn}. \tag{1}$$

When this holds, we note τ's sequence number and UA issuer[11] *as superscript and subscript, respectively, i.e., $\tau = \tau_i^{sn}$.*

When the context is clear, for simplicity, we might abbreviate Equation (1) using a set notation into $A_i^{sn} = A_i^{sn-1} \uplus \{\tau\}$, where $\uplus$ denotes the disjoint set union.

Lemma B.5 (Transfer associated with a valid UA)**.** *Consider a valid UA A_i^{sn} at sequence number $sn > 0$ issued by a process p_i (correct or faulty)*[12]*. The existence of A_i^{sn} implies that there exists a valid transfer τ_i^{sn} issued by p_i at sequence number sn (Definition B.4). Furthermore this transfer is unique for a given p_i and sn. We say that A_i^{sn} is associated with τ_i^{sn}.*

Proof. Consider a valid UA A_i^{sn} at sequence number $sn > 0$ issued by a process p_i (correct or faulty). By Definition B.1, there exists a valid AP σ_i^{sn} at sequence number sn for p_i so that $AccountUpdate.\mathsf{ap\_verify}(\sigma_i^{sn}, \langle A_i^{sn}, \star\rangle, sn, i)$. By AP-Knowledge-Soundness, and by definition of the P_A predicate for *AccountUpdate* objects (Algorithm 2), the prover p_i must know some $data = \langle \star, \pi_i, old\_state\_data\rangle$ satisfying $P_A(\langle A_i^{sn}, \star\rangle, data, sn)$. Due to line 6 of the code of Predicate P_A (Algorithm 2), this implies that $TransferValidity.\mathsf{zkp\_verify}(\pi_i, old\_state\_data \oplus \langle A_i^{sn}, \star, sn\rangle)$ is true, where $old\_state\_data = \langle A_i^{sn-1}, \star\rangle$. As a result, by ZKP-Knowledge-Soundness and by construction of the P_{ZK} predicate of the ZKP object *TransferValidity* (Algorithm 2), the prover p_i must know some *secret_data* containing a transfer $\tau = \langle snd, v, rcv, sn'\rangle$ variable (fifth parameter at line 11 of Algorithm 2) such that $P_{ZK}(\langle A_i^{sn-1}, \star\rangle \oplus \langle A_i^{sn}, \star, sn\rangle, secret\_data)$ is true (if $i = snd$, then $sn = sn'$ due to the check at line 17 of Algorithm 2, otherwise if $i = rcv$, sn and sn' can be different). Furthermore, due to line 8, and by Lemma B.2, A_i^{sn-1} is the valid UA at sequence number $sn-1$ for process p_i. This observation, together with line 13 of Algorithm 2 implies that τ is a valid transfer issued by p_i at sequence number sn. Because A_i^{sn} and A_i^{sn-1} are unique for p_i at their respective sequence number, line 13 further yields that τ is the sole transfer that is valid for p_i and sn. □

Corollary B.6 (Sequence of preceding transfers of a valid UA)**.** *Consider a valid UA A_i^{sn} at sequence number $sn \geq 0$ issued by a process p_i (correct or faulty). A_i^{sn} can be associated with a unique sequence of preceding transfers, denoted $precedingTrans(A_i^{sn}) = (\tau_i^1, .., \tau_i^{sn})$, so that each τ_i^k is a valid transfer issued by p_i at sequence number k, with $1 \leq k \leq sn$.*

Proof. The corollary follows from Lemmas B.3 and B.5. □

Lemma B.7 (Prefix-ordering of of preceding transfers)**.** *The sequences of transfers produced for UAs issued by the same process p_i are prefix-ordered, i.e., if $A_i^{sn_1}$ and $A_i^{sn_2}$ are two valid UAs issued by p_i at sequence numbers sn_1 and sn_2 respectively such that $sn_1 \leq sn_2$, then $precedingTrans(A_i^{sn_1})$ is a prefix of $precedingTrans(A_i^{sn_2})$.*

Proof. This follows from the unicity of a valid transfer for a given process and sequence number, as stated in Lemma B.5. □

For simplicity, in the following, we equate valid transfers with their corresponding transfer invocation. More precisely, if τ_i^{sn} is a valid transfer at sequence number sn for a process p_i, we denote the transfer invocation corresponding to a τ_i^{sn} by $\mathsf{transfer}_{snd}^{sn'}(v, rcv)$, where snd and sn' are respectively the id of the sender and the sequence number of the transfer contained in τ. In this case, we say that $\mathsf{transfer}_{snd}^{sn'}(v, rcv)$ is the transfer invocation for prover p_i at sequence number sn.

Definition B.8 (Sending UAs, receiving UAs, and null UAs)**.** *We distinguish two sorts of valid UAs that are produced in our system: sending UAs, and receiving UAs.*

[11] We note that the UA issuer of transfer τ is not always the sender of τ.

[12] The sequence number $sn = 0$ is excluded as the genesis AP σ_i^0 does not have a corresponding transfer.

- *A valid A_i^{sn} is a sending UA if the issuer p_i of A_i^{sn} is also the sender of the corresponding transfer* $\mathsf{transfer}_i^{sn}(\star, j)$ *(where i and j can be different). If the issuer p_i is correct, the sending UA has been produced at line 38 of Algorithm 3, during the* process_transfer() *call at line 27.*

- *A valid A_i^{sn} is a receiving UA if the issuer p_i of A_i^{sn} is also the receiver of the corresponding transfer* $\mathsf{transfer}_j^{sn'}(\star, i)$ *(where i and j can be different, and sn and sn' can be different). If the issuer p_i is correct, the receiving UA is produced at line 38 of Algorithm 3, during the* process_transfer() *call at line 33.*

Recall 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 UA A_i^{sn} is both a sending UA and a receiving UA. In this case, A_i^{sn} is said to be a null UA, and* $\mathsf{transfer}_i^{sn}(v,i)$ *is said to be a null* transfer *invocation.*

Lemma B.9 (Matching receiving UA to a sending UA)**.** *Any valid receiving UA A_i^{sn} for some transfer $\tau_i = \langle j, \star, i, sn_{\tau_i} \rangle$ from a receiver p_i (correct or faulty) can be matched to a valid sending UA $A_j^{sn_{\tau_i}}$ for the same transfer τ_i from the sender $p_j \neq p_i$ (correct or faulty).*

Proof. Consider a valid receiving UA A_i^{sn} (Definition B.8) at sequence number sn for process p_i corresponding to some receiving transfer $\tau_i = \langle j, \star, i, sn_{\tau_i} \rangle$ whose sender is p_j and receiver is p_i. Two cases arise depending on whether A_i^{sn} is a null or non-null UA (cf. Definition B.8).

- If A_i^{sn} is a null UA, it is both a sending UA and a receiving UA, and corresponds to a transfer $\tau_i = \langle i, \star, i, sn \rangle$ from p_i to itself, thus fulfilling the lemma.

- If A_i^{sn} is a non-null receiving UA, this A_i^{sn} is associated to a valid AP σ_i^{sn} (Definition B.1). Lines 20 and 21 of P_{ZK} (Algorithm 2) imply the existence of a valid UA $A_j^{sn_{\tau_i}}$ for p_j at sequence number sn_{τ_i}, with $j \neq i$ (since A_i^{sn} is non-null), and $\tau_i \in A_j^{sn_{\tau_i}}$.

  Consider the sequence $precedingUAs(A_j^{sn_{\tau_i}}) = (A_j^k)_{0 \leq k \leq sn_{\tau_i}}$ of UAs preceding $A_j^{sn_{\tau_i}}$ (Lemma B.3), and the sequence $precedingTrans(A_j^{sn_{\tau_i}}) = (\tau_j^k)_{0 < k \leq sn_{\tau_i}}$ of transfers preceding $A_j^{sn_{\tau_i}}$ (Corollary B.6). By construction of the two sequences, each transfer τ_j^k is valid for p_j at sequence number k, which implies

  $$\forall k \in [1..sn_{\tau_i}] : A_j^k = A_j^{k-1} \uplus \{\tau_j^k\}. \tag{2}$$

  Line 7 of P_A (Algorithm 2) implies that A_j^0 is empty. This observation and the above equation yield that $A_j^{sn_{\tau_i}}$ contains exactly the transfers present in the sequence $precedingTrans(A_j^{sn_{\tau_i}})$.

  Since $\tau_i \in A_j^{sn_{\tau_i}}$, there exists some $k_0 \in [1..sn_{\tau_i}]$ such that $\tau_i = \tau_j^{k_0}$. As the sender of τ_i is p_j, line 17 of P_{ZK} (Algorithm 2) implies that $k_0 = sn_{\tau_i}$, and that $\tau_i = \tau_j^{sn_{\tau_i}}$ by the unicity of a valid transfer for a given process and sequence number, as stated in Lemma B.5. □

Notations. For the remaining proofs, we introduce the following notations to manipulate the different notions we have discussed. All notations are defined with respect to a particular global history H.

- $transfer(A_i^{sn}) \stackrel{\text{def}}{=} \mathsf{transfer}_{snd}^{sn'}(v, rcv)$ is the transfer invocation corresponding to a valid UA A_i^{sn} when $sn > 0$ (see Lemma B.5). A_i^{sn} might be a sending UA (in which case $i = snd$ and $sn = sn'$) or a receiving UA (in which case $i = rcv$).

- $sendingUA(\mathsf{transfer}_i^{sn}(v, j)) \stackrel{\text{def}}{=} A_i^{sn}$ is the sending UA issued by p_i corresponding to a valid transfer invocation $\mathsf{transfer}_i^{sn}(v, j)$ performed by p_i towards p_j. By definition of a valid transfer and Lemma B.9, this sending UA always exists, and by AP-Agreement, it is unique. As a shortcut, we further say that sn (the sequence number of the UA A_i^{sn}) is the sequence number of the transfer invocation $\mathsf{transfer}_i^{sn}(v, j)$.

- $sendingUA(A_i^{sn}) \stackrel{\text{def}}{=} A_j^{sn'}$ is by extension the sending UA issued by p_j corresponding to a valid receiving UA A_i^{sn} issued by p_i. $sendingUA(A_i^{sn})$ is a shortcut for $sendingUA(transfer(A_i^{sn}))$. In case A_i^{sn} is a null UA (Definition B.8), we have $sendingUA(A_i^{sn}) = A_i^{sn}$.

- $receivingUA(\mathsf{transfer}_i^{sn}(v,j)) \stackrel{\text{def}}{=} A_j^{sn'}$ is, when it exists, the receiving UA issued by p_j corresponding to a valid transfer invocation $\mathsf{transfer}_i^{sn}(v,j)$ performed by p_i towards p_j. When both the sender p_i and the receiver p_j are correct, this receiving UA always exists because of the network's reliability, and because the internal operation $\mathsf{process\_transfer}(..)$ of Algorithm 3 (lines 34–45) does not contain any blocking operation.

  When $\mathsf{transfer}_i^{sn}(v,j)$ has no corresponding receiving UA (this can happen when either the sender p_i or the receiver p_j are Byzantine), by convention we define $receivingUA(\mathsf{transfer}_i^{sn}(v,j)) \stackrel{\text{def}}{=} \bot$.

- 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 $\mathsf{transfer}_i^{sn}(\star,\star)$.

B.1.2 AT-Sequentiality of Algorithm 3

Let us consider any execution of Algorithm 3, captured as a global history $H = (L_1, ..., L_n)$. To prove AT-Sequentiality (Lemma 6.1), we must construct a mock history $\widehat{H}$ of H that preserves the local histories of correct processes and replaces the local histories L_j of Byzantine processes p_j by mock local histories $\widehat{L}_j$ so that $\widehat{H}$ can be AT-sequenced. We construct $\widehat{H}$ and the corresponding AT-sequence S_i incrementally by introducing several intermediary definitions and lemmas.

In the following, C is the set of correct processes, and B the set of Byzantine processes. $C \cup B = \{p_1, .., p_n\}$. We construct the mock local histories $\{\widehat{L}_j\}_{p_j \in B}$ of Byzantine processes as follows.

Definition B.10. *We define the set Σ of UAs defined recursively by the following rules.*

- *Σ contains all the UAs issued by correct processes at line 38 of Algorithm 3. This includes both sending and receiving UAs.*

- *If some receiving UA A belongs to Σ, then $sendingUA(A)$ also belongs to Σ,*

$$\big\{sendingUA(A) \mid A \text{ is a valid receiving } UA \wedge A \in \Sigma\big\} \subseteq \Sigma.$$

- *If some UA A belongs to Σ, then all UAs by the same issuer that precedes A also belong to A,*

$$\bigcup_{A \in \Sigma} \mathsf{set}(precedingUAs(A)) \subseteq \Sigma. \tag{3}$$

Σ can be seen as the causal transitive closure of the UAs issued by correct processes: it contains all the UAs that can be traced back to some UA issued by a correct process, either through a sending/receiving relationship as captured by Lemma B.9, or through local precedence as captured by Lemma B.3.

Lemma B.11 (Validity of UAs in Σ)**.** *All UAs contained in Σ are valid.*

Proof. The remark holds because all UAs issued by correct processes are valid by construction, and results from the application of Lemma B.9 and Lemma B.3 when defining $sendingUA(\cdot)$ and $precedingUAs(\cdot)$ respectively. □

[13] Recall that we do not consider $\mathsf{balance}()$ invocations from Byzantine processes.

Definition B.12 (Mock history $\widehat{H}$)**.** *For a Byzantine process $p_j \in B$, we construct the set Θ_j of transfer invocations whose sender is p_j, and for which both a sending and a receiving UA can be found in Σ. More formally, we have*

$$\Theta_j \stackrel{\text{def}}{=} \left\{ \mathsf{transfer}_j^{sn}(\star,\star) \;\middle|\; \begin{array}{c} \exists A_1, A_2 \in \Sigma : A_1 \textit{ is a sending UA } \wedge A_2 \textit{ is a receiving UA } \wedge \\ \textit{transfer}(A_1) = \textit{transfer}(A_2) = \mathsf{transfer}_j^{sn}(\star,\star) \end{array} \right\}.$$

Let us note SN_j the set of the sequence numbers of the transfer invocations present in Θ_j. SN_j is a set of strictly positive integers (since p_j's genesis UA A_j^0 created at line 1 of Algorithm 1 does not have a corresponding transfer invocation), $SN_j \subseteq \mathbb{N}^*$. Note that, by AP-Agreement, each sequence number in SN_j corresponds to a single transfer invocation in Θ_j. As a result, we can define the mock local history $\widehat{L}_j$ of p_j as the sequence of transfer invocations present in Θ_j ordered by their sequence numbers,

$$\widehat{L}_j \stackrel{\text{def}}{=} \left(\mathsf{transfer}_j^{sn}(\star,\star) \in \Theta_j\right)_{sn \in SN_j}.$$

Combining the mock local histories $(\widehat{L}_j)_{j\in B}$ of Byzantine processes with the local histories $(L_i)_{i\in C}$ of correct processes yields a mock history $\widehat{H}$.

Definition B.13 (Set $\mathscr{S}_i$ of operation perceived by a correct process p_i)**.** *Consider a correct process p_i, and the set $\mathscr{S}_i$ containing all invocations made by p_i (to the operations* transfer *and* balance*), and all transfer invocations by other processes (correct of faulty) present in $\widehat{H}$.*

$$\mathscr{S}_i = \mathsf{set}(L_i) \cup \bigcup_{j \in C \setminus \{i\}} \mathsf{transferSet}(L_j) \cup \bigcup_{j \in B} \mathsf{transferSet}(\widehat{L}_j), \tag{4}$$

where C and B are the sets of correct and Byzantine processes as defined above, respectively, and $\mathsf{transferSet}(\cdot)$ *is the notation introduced in Section 4 to denote the set of transfer invocations contained in a sequence.*

To produce a sequence S_i from the elements of $\mathscr{S}_i$, we construct a partial order $\rightsquigarrow_i$ on $\mathscr{S}_i$. We do so incrementally, by introducing two binary relations, $\overset{\mathbf{1}}{\rightsquigarrow}_i$ and $\overset{\mathbf{2}}{\rightsquigarrow}_i$.

- $\overset{\mathbf{1}}{\rightsquigarrow}_i$ totally orders the invocation of p_i and ensures that incoming transfers received by a process p_j are ordered before the subsequent balance invocations (if $j = i$) and outgoing transfers issued by p_j.
- $\overset{\mathbf{2}}{\rightsquigarrow}_i$ focuses specifically on p_i to guarantee that incoming transfers received by p_i are totally ordered with respect to p_i's local invocation to $\mathsf{balance}(\cdot)$.

Distinguishing between $\overset{\mathbf{1}}{\rightsquigarrow}_i$ and $\overset{\mathbf{2}}{\rightsquigarrow}_i$ as intermediary steps towards $\rightsquigarrow_i$ will in turn make it easier to prove that the $\rightsquigarrow_i$ is acyclic (a necessary condition to show that it is a partial order).

Definition B.14 (Binary relations $\overset{\mathbf{1}}{\rightsquigarrow}_i$, $\overset{\mathbf{2}}{\rightsquigarrow}_i$, and $\rightsquigarrow_i$ on $\mathscr{S}_i$)**.** *We define $\overset{\mathbf{1}}{\rightsquigarrow}_i$ on $\mathscr{S}_i$ as follows.*

- **Inclusion of p_i's local order.** *$\overset{\mathbf{1}}{\rightsquigarrow}_i$ respects the local process order $\rightarrow_i$ of operations invoked by p_i, i.e., $\rightarrow_i \subseteq \overset{\mathbf{1}}{\rightsquigarrow}_i$.*
- **Ordering of incoming transfers.**
  - *For any process p_j, $t_{\triangleright j} = \mathsf{transfer}_\star^\star(\star, j) \in \mathscr{S}_i$ a transfer invocation whose receiver is p_j, and $t_{j\triangleright} = \mathsf{transfer}_j^\star(\star,\star) \in \mathscr{S}_i$ a transfer invocation by p_j such that $t_{\triangleright j} \neq t_{j\triangleright}$, if $\textit{receivingUA}(t_{\triangleright j}) \in \textit{precedingUAs}(\textit{sendingUA}(t_{j\triangleright}))$, then $t_{\triangleright j} \overset{\mathbf{1}}{\rightsquigarrow}_i t_{j\triangleright}$.*

– *For transfer invocation* $t_{\triangleright i} = \mathsf{transfer}^{\star}_{\star}(\star, i) \in \mathscr{S}_i$ *whose receiver is* p_i*, and any invocation* $b_i = \mathsf{balance}_i()/v$ *performed by* p_i*, if the execution of line 38 that generated* $\mathit{receivingUA}(t_{\triangleright i})$ *occurred before that of line 23 corresponding to* b_i *, then* $t_{\triangleright i} \overset{\mathbf{1}}{\rightsquigarrow}_i b_i$.

We define $\overset{\mathbf{2}}{\rightsquigarrow}_i$ *on* $\mathscr{S}_i$ *as follows.*

- **Ordering of p_i's balance.** *For any invocation* $b_i = \mathsf{balance}_i()/v$ *performed by* p_i*, and any transfer invocation* $t_{\triangleright i} = \mathsf{transfer}^{\star}_{\star}(\star, i) \in \mathscr{S}_i$ *whose receiver is* p_i*, if the execution of line 23 corresponding to* b_i *occurred before that of line 38 that generated* $\mathit{receivingUA}(t_{\triangleright i})$*, then* $b_i \overset{\mathbf{2}}{\rightsquigarrow}_i t_{\triangleright i}$.

$\rightsquigarrow_i$ *is defined as the transitive closure of the union of* $\overset{\mathbf{1}}{\rightsquigarrow}_i$ *and* $\overset{\mathbf{2}}{\rightsquigarrow}_i$*, i.e.,* $\rightsquigarrow_i = \left(\overset{\mathbf{1}}{\rightsquigarrow}_i \cup \overset{\mathbf{2}}{\rightsquigarrow}_i \right)^{+}$.

Lemma B.15. $\overset{\mathbf{1}}{\rightsquigarrow}_i$ *is acyclic.*

Proof. The distributed system is implicitly embedded in some physical time, which we assume is Newtonian and linear.[14] We associate each element x of $\mathscr{S}_i$ with a timestamp $\tau(x)$ from this physical time as follows:

- if $x = \mathsf{balance}_i()/v$ is a balance invocation by p_i, $\tau(x)$ is the time point at which line 23 of Algorithm 3 is executed;
- if $x = \mathsf{transfer}^{\star}_{\star}(\star, \star)$ is a transfer invocation, $\tau(x)$ is the time point at which $\mathit{sendingUA}(x)$ was created.

Consider $x, y \in \mathscr{S}_i$ such that $x \overset{\mathbf{1}}{\rightsquigarrow}_i y$. At least one of the following cases holds.

- *Case 1 (Inclusion of p_i's local order):* x and y are both invoked by p_i with $x \rightarrow_i y$. Because p_i is sequential, by definition of $\rightarrow_i$ we have $\tau(x) < \tau(y)$.
- *Case 2 (Ordering of incoming transfers):*
  - $x = \mathsf{transfer}^{\star}_{\star}(\star, j) \in \mathscr{S}_i$ is a transfer invocation whose receiver is some process p_j, and $y = \mathsf{transfer}^{\star}_{j}(\star, \star) \in \mathscr{S}_i$ a transfer invocation by p_j, such that $\mathit{receivingUA}(x) \in \mathit{precedingUAs}(\mathit{sendingUA}(y))$. This last inclusion means that $\mathit{sendingUA}(y)$ can be linked recursively to $\mathit{receivingUA}(x)$ using proofs issues by p_j, the predicates P_A and P_{ZK}, and AP-Knowledge-Soundness and ZKP-Knowledge-Soundness. AP-Knowledge-Soundness and ZKP-Knowledge-Soundness both imply that p_j must know the secret data required to issue a proof *before* the proof is generated. As a result $\mathit{receivingUA}(x) \in \mathit{precedingUAs}(\mathit{sendingUA}(y))$ implies that $\tau(x) < \tau(y)$.
  - $x = \mathsf{transfer}^{\star}_{\star}(\star, i) \in \mathscr{S}_i$ is a transfer invocation whose receiver is p_i, $y = \mathsf{balance}_i()/v$ is a balance invocation performed by p_i, such that the execution of line 38 that generated $\mathit{receivingUA}(x)$ occurred before that of line 23 corresponding to y. The same reasoning as above yields that $\tau(x) < \tau(y)$.

We conclude that $\overset{\mathbf{1}}{\rightsquigarrow}_i$ respects the timestamps assigned by $\tau(\cdot)$. Assuming physical time is acyclic, this implies that $\overset{\mathbf{1}}{\rightsquigarrow}_i$ is also acyclic. □

[14] Note that the proof also holds in the context of special relativity, even though there may not be an absolute global clock ordering the events of the system. This is because any observer in the system still sees the events in a linear (total) order that may, however, differ from one observer to the next. For this proof, it suffices to take any observer (*e.g.*, p_i) as a frame of reference to construct the timestamps. But because of the causality principle, two causally-linked events are necessarily seen in the same order, no matter the chosen frame of reference.

Lemma B.16. $\overset{2}{\rightsquigarrow}_i$ *is acyclic.*

Proof. $\overset{2}{\rightsquigarrow}_i$ is a simple bipartite binary relation between balance invocations on one side, and transfer invocations on the other, hence it cannot contain cycles, and is therefore acyclic. □

Lemma B.17. $\overset{1}{\rightsquigarrow}_i \cup \overset{2}{\rightsquigarrow}_i$ *is acyclic.*

Proof. The proof is by contradiction. Assume $\overset{1}{\rightsquigarrow}_i \cup \overset{2}{\rightsquigarrow}_i$ contains at least one cycle. Consider the shortest such cycle $\mathscr{C}_{\mathsf{min}}$. Because $\overset{1}{\rightsquigarrow}_i$ and $\overset{2}{\rightsquigarrow}_i$ are both acyclic (Lemmas B.15 and B.16), $\mathscr{C}_{\mathsf{min}}$ must involve both $\overset{1}{\rightsquigarrow}_i$ and $\overset{2}{\rightsquigarrow}_i$.

Consider one of the $\overset{2}{\rightsquigarrow}_i$ ordering in $\mathscr{C}_{\mathsf{min}}$. By definition of $\overset{2}{\rightsquigarrow}_i$, this ordering is of the form $b_i \overset{2}{\rightsquigarrow}_i t_{\triangleright i}$, where b_i is a balance invocation by p_i and any $t_{\triangleright i}$ is a transfer invocation whose receiver is p_i. Consider the successor y of $t_{\triangleright i}$ in $\mathscr{C}_{\mathsf{min}}$. By construction of $\overset{2}{\rightsquigarrow}_i$, $t_{\triangleright i} \not\overset{2}{\rightsquigarrow}_i y$, which implies $t_{\triangleright i} \overset{1}{\rightsquigarrow}_i y$.

By definition of $\overset{1}{\rightsquigarrow}_i$, $t_{\triangleright i} \overset{1}{\rightsquigarrow}_i y$ leads to two cases.

- *Case 1:* $t_{\triangleright i} \rightarrow_i y$. In this case, $y \in L_i$ by definition of $\rightarrow_i$, p_i's local process order.
- *Case 2:* $t_{\triangleright i} \not\rightarrow_i y$. In this case, $t_{\triangleright i} \overset{1}{\rightsquigarrow}_i y$ resulted from the rule *Ordering of incoming transfers*. y is therefore either a balance or transfer operation invoked by p_i, *i.e.*, we also have $y \in L_i$.

We therefore have $b_i \overset{2}{\rightsquigarrow}_i t_{\triangleright i} \overset{1}{\rightsquigarrow}_i y$, with $b_i, y \in L_i$. The definitions of $\overset{2}{\rightsquigarrow}_i$ and $\overset{1}{\rightsquigarrow}_i$ imply that the invocation of b_i by p_i precedes in time the generation of $\mathit{receivingUA}(t_{\triangleright i})$ also by p_i, which itself precedes the invocation of y still by p_i (either directly in the case of $\overset{2}{\rightsquigarrow}_i$, and if y is a balance invocation, or using the same argument on cryptographic proofs as in Lemma B.15 if y is a transfer). By definition of the process order $\rightarrow_i$, this leads to $b_i \rightarrow_i y$, and hence $b_i \overset{1}{\rightsquigarrow}_i y$. We can, therefore, remove $t_{\triangleright i}$ from $\mathscr{C}_{\mathsf{min}}$, and replace $b_i \overset{2}{\rightsquigarrow}_i t_{\triangleright i} \overset{1}{\rightsquigarrow}_i y$ by $b_i \overset{1}{\rightsquigarrow}_i y$, thus creating a cycle $\mathscr{C}'_{\mathsf{min}}$ in $\overset{1}{\rightsquigarrow}_i \cup \overset{2}{\rightsquigarrow}_i$ with one less element than $\mathscr{C}_{\mathsf{min}}$. This is a contradiction as $\mathscr{C}_{\mathsf{min}}$ was assumed to be the shortest. □

Corollary B.18. $\rightsquigarrow_i$ *is a partial order on* $\mathscr{S}_i$.

Proof. $\rightsquigarrow_i$ is transitive by construction. By Lemma B.17, it is the transitive closure of an acyclic relation and is, therefore, also acyclic. □

Lemma B.19. *Each element of the poset* $(\mathscr{S}_i, \rightsquigarrow_i)$ *only has a finite number of predecessors.*

Proof. This follows from the sequential nature of processes (and in particular that all processes have a starting point in their execution), the link between $\overset{1}{\rightsquigarrow}_i$ and $\overset{2}{\rightsquigarrow}_i$ and physical time (Lemmas B.15 and B.16), and the (implicit) assumption that processes only take a finite number of local steps per unit of time. □

Definition B.20 (Sequence S_i)**.** *By Lemma B.19 the poset* $(\mathscr{S}_i, \rightsquigarrow_i)$ *can be sorted topologically into a sequence. We define* S_i *as one of the topological sorts of* $(\mathscr{S}_i, \rightsquigarrow_i)$*. We note* $\rightarrow_{S_i}$ *the total order induced by* S_i *on its elements.*

In the following, we will use the function $\mathit{total}(\cdot,\cdot)$ introduced in Section 4 to compute the account balance of a process resulting from the transfer it sends and receives. Given a process p_j and a set of operations invocations O, $\mathit{total}(j, O)$ is defined as

$$\mathit{total}(j,O) = \mathtt{init}_j + \left(\Sigma_{\mathsf{transfer}_\star(v,j)\in O}\, v\right) - \left(\Sigma_{\mathsf{transfer}_j(v,\star)\in O}\, v\right).$$

Lemma B.21. $\forall\, b_i = \mathsf{balance}_i()/v \in S_i : v = \mathit{total}(i, \{o \in S_i \mid o \rightarrow_{S_i} b_i\})$.

Proof. Consider $b_i = \mathsf{balance}_i()/v \in S_i$ a balance operation performed by a correct process p_i that returns a value v. Define the sets I_i and R_i as follows:

- I_i is the set of transfers *invoked* locally by p_i before p_i executed b_i. By construction of Algorithm 3, p_i generated a sending UA at line 38 for each of these transfers.
- R_i is the set of transfers *received* by p_i whose receiving UA was generated by p_i (at line 38) before invoking b_i.

Because p_i is correct, it executes Algorithm 3, which ensures by construction (in particular due to lines 23, 35 and 45) that

$$v = total(i, I_i \cup R_i), \tag{5}$$

taking into account that the effects of null transfers on i cancel out in the definition of *total*.

Consider now $T_{i\triangleright}$ the subset of transfers present in $\mathscr{S}_i$ that p_i issued, and $T_{\triangleright i}$ the subset of transfers present in $\mathscr{S}_i$ that p_i received, *i.e.*,

$$T_{i\triangleright} \stackrel{\text{def}}{=} \{\mathsf{transfer}_i(\star,k) \in \mathscr{S}_i \mid i \neq k\}, \tag{6}$$

$$T_{\triangleright i} \stackrel{\text{def}}{=} \{\mathsf{transfer}_k(\star,i) \in \mathscr{S}_i \mid i \neq k\}. \tag{7}$$

By definition of $total(\cdot,\cdot)$, we have similarly

$$total\big(i, \{o \in S_i \mid o \rightarrow_{S_i} b_i\}\big) = total\big(i, \{o \in S_i \mid o \rightarrow_{S_i} b_i\} \cap (T_{i\triangleright} \cup T_{\triangleright i})\big). \tag{8}$$

Because p_i's local process order is included in $\rightsquigarrow_i$, it is also included in $\rightarrow_{S_i}$. As a result, the transfers invoked by p_i before invoking b_i are exactly those transfers that precedes b_i in $\rightarrow_{S_i}$,

$$\{o \in S_i \mid o \rightarrow_{S_i} b_i\} \cap T_{i\triangleright} = I_i.$$

Furthermore, due to the rule *Ordering of incoming transfers* of $\overset{\mathbf{1}}{\rightsquigarrow}_i$ and the definition of $\overset{\mathbf{2}}{\rightsquigarrow}_i$, b_i is totally ordered w.r.t. to the transfers received by p_i in $\rightsquigarrow_i$ and therefore in $\rightarrow_{S_i}$. As a result, we have

$$\{o \in S_i \mid o \rightarrow_{S_i} b_i\} \cap T_{\triangleright i} = R_i.$$

These last two equalities together with Equations (5) and (8) yield the lemma.[15] □

Lemma B.22. $\forall\, j \in [0..n], \forall\, t_{j\triangleright} = \mathsf{transfer}_j(v,\star) \in S_i : v \leq total\big(j, \{o \in S_i \mid o \rightarrow_{S_i} t_{j\triangleright}\}\big)$.

Proof. Consider a process p_j (correct or faulty) and $t_{j\triangleright} = \mathsf{transfer}_j(v,\star) \in S_i$ a transfer invocation from j that is present S_i of amount v to some process.

Let us define the sets $I^{S_i}_{t_{j\triangleright}}$ and $R^{S_i}_{t_{j\triangleright}}$ as follows.

- $I^{S_i}_{t_{j\triangleright}}$ is the set of transfers whose sender is p_j that appear before $t_{j\triangleright}$ in sequence S_i and includes $t_{j\triangleright}$,

$$I^{S_i}_{t_{j\triangleright}} = \big\{\mathsf{transfer}^\star_j(\star,\star) \in S_i \mid \mathsf{transfer}^\star_j(\star,\star) \rightarrow_{S_i} t_{j\triangleright} \vee \mathsf{transfer}^\star_j(\star,\star) = t_{j\triangleright}\big\}.$$

- $R^{S_i}_{t_{j\triangleright}}$ is the set of transfers whose receiver is p_j that appear before $t_{j\triangleright}$ in sequence S_i.

$$R^{S_i}_{t_{j\triangleright}} = \big\{\mathsf{transfer}^\star_\star(j,\star) \in S_i \mid \mathsf{transfer}^\star_\star(j,\star) \rightarrow_{S_i} t_{j\triangleright}\big\}. \tag{9}$$

Remark B.22.1. $total(j, I^{S_i}_{t_{j\triangleright}} \cup R^{S_i}_{t_{j\triangleright}}) = total(j, \{o \in S_i \mid o \rightarrow_{S_i} t_{j\triangleright}\}) - v$.

[15] Let us recall that, given 3 sets A, B, and C, we always have $(A \cap B) \cup (A \cap C) = A \cap (B \cup C)$.

Proof. This directly follows from the definition of $total(\cdot,\cdot)$, $I^{S_i}_{t_{j\triangleright}}$ and $R^{S_i}_{t_{j\triangleright}}$. □

Consider $SN^{S_i}_{t_{j\triangleright}}$ the set of the sequence numbers of the transfers present in $I^{S_i}_{t_{j\triangleright}}$, *i.e.*,

$$SN^{S_i}_{t_{j\triangleright}} = \{sn \in \mathbb{N}^* \mid \mathsf{transfer}^{sn}_j(\star,\star) \in I^{S_i}_{t_{j\triangleright}}\}. \tag{10}$$

Because $t_{j\triangleright} \in I^{S_i}_{t_{j\triangleright}}$, $SN^{S_i}_{t_{j\triangleright}}$ contains the sequence number of $t_{j\triangleright}$. However, because $\rightsquigarrow_i$ does not impose strong constraints on the order of the transfer invocations issued by a process other than p_i, $I^{S_i}_{t_{j\triangleright}}$ might also contain transfer invocations that have a higher sequence number than $t_{j\triangleright}$. Furthermore, note that, by construction, $SN^{S_i}_{t_{j\triangleright}}$ only contains sequence numbers of transfers whose sender is p_j.

Define $sn^{\mathsf{max}}_{j\triangleright}$ as the highest sequence number contained in $SN^{S_i}_{t_{j\triangleright}}$,

$$sn^{\mathsf{max}}_{j\triangleright} = \mathsf{max}\big(SN^{S_i}_{t_{j\triangleright}}\big), \tag{11}$$

and $A^{\mathsf{max}}_{j\triangleright}$ as the UA issued by j with the sequence number $sn^{\mathsf{max}}_{j\triangleright}$, *i.e.*, $A^{\mathsf{max}}_{j\triangleright} = A^{sn^{\mathsf{max}}_{j\triangleright}}_j$. (By AP-Agreement, this UA is unique.)

Consider $\Sigma^{\mathsf{max}}_{j\triangleright}$ the set of UAs issued by p_j that precede $A^{\mathsf{max}}_{j\triangleright}$ according to the predicates P_A and P_{ZK} (Lemma B.3),

$$\Sigma^{\mathsf{max}}_{j\triangleright} = \mathsf{set}\big(precedingUAs(A^{\mathsf{max}}_{j\triangleright})\big). \tag{12}$$

Based on $\Sigma^{\mathsf{max}}_{j\triangleright}$, let us define the sets $I^{\mathsf{max}}_{t_{j\triangleright}}$ and $R^{\mathsf{max}}_{t_{j\triangleright}}$ as follows.

- $I^{\mathsf{max}}_{t_{j\triangleright}}$ is the set of transfers whose sender is p_j and whose sending UA appear in $\Sigma^{\mathsf{max}}_{j\triangleright}$,

$$I^{\mathsf{max}}_{t_{j\triangleright}} = \big\{\mathsf{transfer}^\star_j(\star,\star) \mid sendingUA(\mathsf{transfer}^\star_j(\star,\star)) \in \Sigma^{\mathsf{max}}_{j\triangleright}\big\}. \tag{13}$$

- $R^{\mathsf{max}}_{t_{j\triangleright}}$ is the set of transfers whose receiver is p_j and whose receiving UA appear in $\Sigma^{\mathsf{max}}_{j\triangleright}$, excluding the transfer invocation associated with $A^{\mathsf{max}}_{j\triangleright}$ in case it is a null UA (*i.e.*, both a sending and receiving UA),

$$R^{\mathsf{max}}_{t_{j\triangleright}} = \left\{ \mathsf{transfer}^\star_\star(j,\star) \;\middle|\; \begin{array}{l} receivingUA(\mathsf{transfer}^\star_\star(j,\star)) \in \Sigma^{\mathsf{max}}_{j\triangleright} \wedge \\ \mathsf{transfer}^\star_\star(j,\star) \neq transfer(A^{\mathsf{max}}_{j\triangleright}) \end{array} \right\}. \tag{14}$$

Let v_{max} denote the amount of the transfer invocation $transfer(A^{\mathsf{max}}_{j\triangleright})$, *i.e.*, $transfer(A^{\mathsf{max}}_{j\triangleright}) = \mathsf{transfer}^{sn^{\mathsf{max}}_{j\triangleright}}_j(v_{\mathsf{max}},\star)$. (The notation $transfer(A^{\mathsf{max}}_{j\triangleright})$ was defined at the end of Appendix B.1.1)

Remark B.22.2. $total\left(j, I^{\mathsf{max}}_{t_{j\triangleright}} \cup R^{\mathsf{max}}_{t_{j\triangleright}}\right) = total\left(j, \left(I^{\mathsf{max}}_{t_{j\triangleright}} \setminus \left\{transfer(A^{\mathsf{max}}_{j\triangleright})\right\}\right) \cup R^{\mathsf{max}}_{t_{j\triangleright}}\right) - v_{\mathsf{max}}$.

Proof. Similarly to Remark B.22.1, the remark follows from the fact that $transfer(A^{\mathsf{max}}_{j\triangleright}) \in I^{\mathsf{max}}_{t_{j\triangleright}}$ and $transfer(A^{\mathsf{max}}_{j\triangleright}) \notin R^{\mathsf{max}}_{t_{j\triangleright}}$, the definition of v_{max}, and that of $total(\cdot,\cdot)$. □

Remark B.22.3. $total\left(j, \left(I^{\mathsf{max}}_{t_{j\triangleright}} \setminus \left\{transfer(A^{\mathsf{max}}_{j\triangleright})\right\}\right) \cup R^{\mathsf{max}}_{t_{j\triangleright}}\right) \geq v_{\mathsf{max}}$.

Proof. This follows from a recursion on the sequence of (valid) UAs $precedingUAs(A^{\mathsf{max}}_{j\triangleright})$ that precede $A^{\mathsf{max}}_{j\triangleright}$ (Lemma B.3), as well as the checks and updates contained in the P_{ZK} predicate (Algorithm 2). This includes the check at line 15 that a sending process must have enough funds, and the updates performed on the sender's balance at line 16 (for null UAs), line 17 (for non-null sending UAs), and line 19 (for non-null receiving UAs). □

Remark B.22.4. $total\left(j, I^{\mathsf{max}}_{t_{j\triangleright}} \cup R^{\mathsf{max}}_{t_{j\triangleright}}\right) \geq 0$.

Proof. This is a direct consequence of Remarks B.22.2 and B.22.3. □

Remark B.22.5. $I^{S_i}_{t_{j\triangleright}} \subseteq I^{\mathsf{max}}_{t_{j\triangleright}}$.

Proof. Consider $t'_{j\triangleright} = \mathsf{transfer}^{sn'}_j(\star,\star) \in I^{S_i}_{t_{j\triangleright}}$, a transfer invocation whose sender is p_j at sequence number sn' that belongs to $I^{S_i}_{t_{j\triangleright}}$. By definition of $SN^{S_i}_{t_{j\triangleright}}$ and $sn^{\mathsf{max}}_{j\triangleright}$ (Equations (10) and (11)), we have

$$sn' \leq sn^{\mathsf{max}}_{j\triangleright}.$$

By AP-Agreement, this inequality yields $sendingUA(t'_{j\triangleright}) \in \mathsf{set}(precedingUAs(A^{\mathsf{max}}_{j\triangleright}))$, and by definition of $\Sigma^{\mathsf{max}}_{j\triangleright}$ and $I^{\mathsf{max}}_{t_{j\triangleright}}$ (Equations (12) and (13)), that $t'_{j\triangleright} \in I^{\mathsf{max}}_{t_{j\triangleright}}$, proving the remark. □

Remark B.22.6. $R^{\mathsf{max}}_{t_{j\triangleright}} \subseteq R^{S_i}_{t_{j\triangleright}}$.

Proof. Consider $t'_{\triangleright j} = \mathsf{transfer}^{\star}_{\star}(j,\star) \in R^{\mathsf{max}}_{t_{j\triangleright}}$, a transfer invocation whose receiver is p_j that belongs to $R^{\mathsf{max}}_{t_{j\triangleright}}$. By definition of $\Sigma^{\mathsf{max}}_{j\triangleright}$ and $R^{\mathsf{max}}_{t_{j\triangleright}}$ (Equations (12) and (14)),

$$receivingUA(t'_{\triangleright j}) \in \mathsf{set}\big(precedingUAs(A^{\mathsf{max}}_{j\triangleright})\big) \wedge t'_{\triangleright j} \neq transfer(A^{\mathsf{max}}_{j\triangleright}). \tag{15}$$

Furthermore, because $transfer(A^{\mathsf{max}}_{j\triangleright}) \in \mathscr{S}_i$, we have $t'_{\triangleright j} \in \mathscr{S}_i$ (either because the sender of $t_{\triangleright j}$ is correct, or by construction of Σ if it is faulty, see Equations (3) and (4)). Remember that $A^{\mathsf{max}}_{j\triangleright}$ is associated with sequence number $sn^{\mathsf{max}}_{j\triangleright} \in SN^{S_i}_{t_{j\triangleright}}$ (Equation (11)). Because $SN^{S_i}_{t_{j\triangleright}}$ only contains the sequence numbers of transfers whose sender is p_j, $transfer(A^{\mathsf{max}}_{j\triangleright})$ must be some transfer $\mathsf{transfer}^{sn^{\mathsf{max}}_{j\triangleright}}_j(\star,\star)$ sent from p_j. As a result, $t'_{\triangleright j} \in \mathscr{S}_i$ together with Equation (15) mean that the condition for the rule *Ordering of incoming transfers* in the definition of $\overset{\mathbf{1}}{\rightsquigarrow}_i$ is met, yielding $t'_{\triangleright j} \overset{\mathbf{1}}{\rightsquigarrow}_i transfer(A^{\mathsf{max}}_{j\triangleright})$, and therefore $t'_{\triangleright j} \rightsquigarrow_i transfer(A^{\mathsf{max}}_{j\triangleright})$, and

$$t'_{\triangleright j} \rightarrow_{S_i} transfer(A^{\mathsf{max}}_{j\triangleright}). \tag{16}$$

Because $transfer(A^{\mathsf{max}}_{j\triangleright}) \in R^{S_i}_{t_{j\triangleright}}$, by definition of $R^{S_i}_{t_{j\triangleright}}$ (Equation (9)), $transfer(A^{\mathsf{max}}_{j\triangleright}) \rightarrow_{S_i} t_{j\triangleright}$, which with Equation (16) leads to $t'_{\triangleright j} \rightarrow_{S_i} t_{j\triangleright}$, and therefore $t'_{\triangleright j} \in R^{S_i}_{t_{j\triangleright}}$. □

Remark B.22.7. $total(j, I^{S_i}_{t_{j\triangleright}} \cup R^{S_i}_{t_{j\triangleright}}) \geq total\left(j, I^{\mathsf{max}}_{t_{j\triangleright}} \cup R^{\mathsf{max}}_{t_{j\triangleright}}\right)$

Proof. This follows from Remarks B.22.5 and B.22.6 and the definition of $total(\cdot,\cdot)$. □

The lemma follows from Remarks B.22.1, B.22.4 and B.22.7. □

Lemma 6.1 (AT-Sequentiality)**.** *For any global history H capturing an execution of* Algorithm 3*, there exists a mock history $\widehat{H}$ of H that can be AT-sequenced.*

Proof. The lemma is proved by using $\widehat{H}$ provided by Definition B.12, the sequence S_i constructed in Definition B.20 for each correct process p_i, and considering Lemmas B.21 and B.22 that show that S_i is an AT-Sequence. □

B.1.3 AT-Termination of Algorithm 3

Lemma 6.2 (AT-Termination)**.** *All operation invocations 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 27). This ap_prove operation terminates by AP-Termination, so the transfer operation also terminates. □

B.2 Proof of quasi-anonymity

Lemma 6.3 (QAAT-Receiver-Anonymity)**.** *For any global history H capturing an execution of $\mathscr{A}$, for any invocation* $\mathsf{transfer}_i(\star, j)$ *contained in H where p_i and p_j are correct processes, then, with high probability, p_j is indistinguishable among all the correct active processes of H.*

Proof. To determine if the adversary *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 $data = \langle \sigma_i, \pi, sn_i + 1, \langle A_i, A'_i, bal\_c_i, bal\_c'_i \rangle \rangle$;
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 using the ra_send operation, which ensures receiver anonymity and confidentiality (see Section 3). Therefore, only accumulator updates (item 1) need to be considered. C-Hiding guarantees that the commitments bal_c_i and $bal\_c'_i$ 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 $\min((1/n) + \varepsilon(\lambda), 1)$, where ε is the statistical negligible function and λ is the security parameter of the system. □

Lemma 6.4 (QAAT-Confidentiality)**.** *For any global history H capturing an execution of $\mathscr{A}$, for any invocation* $\mathsf{transfer}_i(v, j)$ *contained in H where p_i and p_j are correct processes, then, with high probability, v is indistinguishable among any value in $\mathbb{R}^+$.*

Proof. The proof follows the same reasoning as that of Lemma 6.3. □

C Implementations of the Schemes of Section 5 / Appendix A

C.1 Preliminaries

C.1.1 Groups of unknown order

Our choice of UA [7] implementation requires the use of groups of unknown order (as defined in [15]) in which the Strong RSA assumption, the Low Order assumption and the Adaptive Root assumption holds. Concrete examples of groups of unknown order are RSA groups, ideal class groups and hyperelliptic Jacobians. However, only class groups and hyperelliptic Jacobians are efficient in the trustless setup model. In particular, [18] provides algorithms to trustlessly setup genus-3 hyperelliptic Jacobians of unknown order with 3392-bits field elements for 128 bits of security, which is close to the 3072-bits elements of RSA groups generated using a trusted setup for the same bit-security. They also provide a new compression algorithm for class groups, reaching 128-bits of security with 5088-bits elements.

$\|S\|$	Addition[1]	Membership proof		Non-membership proof	
		Generation[2]	Verification	Generation	Verification
100	0.337 ms	31.462 ms	0.351 ms	31.835 ms	0.687 ms
500	0.368 ms	155.366 ms	0.345 ms	151.720 ms	0.687 ms
1000	0.342 ms	299.615 ms	0.339 ms	298.825 ms	0.675 ms
1500	0.337 ms	447.879 ms	0.338 ms	444.978 ms	0.663 ms
2000	0.360 ms	607.562 ms	0.345 ms	605.435 ms	0.668 ms

[1] This also outputs the membership proof of the new element.

[2] Unused, as we rely on the proof generated during addition.

Table 4: Performance of accumulator proof generation and verification for various set sizes $|S|$.

C.2 Accumulator

For our accumulator (specified in Appendix A.2), we opt for the trustless extension of universal RSA accumulators in groups of unknown order from [7]. While RSA accumulators typically require the accumulated elements to be prime numbers, one can use a hash function with prime domain to accumulate arbitrary elements. The scheme features constant size public parameters, accumulator digest and proofs of (non-)membership. Moreover, (non-)membership proofs can be verified in constant time. The scheme can thus be described as succinct. Note that when adding a new element e to an accumulator A, the membership proof of e is the value of A before e was added. Therefore, as a process only needs the membership proof of a transfer when updating the accumulator and creating the ZKP of its account update, membership proof generation is done in constant time in Algorithm 3.

C.2.1 Hashing to primes

An RSA accumulator can only contain prime numbers. To overcome this limitation, one must use a hash-to-prime function $H_{\mathsf{Primes}} : \{0,1\}^* \to \mathsf{Primes}(\lambda)$, where $\mathsf{Primes}(\lambda)$ is the set of primes smaller than 2^λ. H_{Primes} must be deterministic and collision-resistant to prevent an element from being mapped to multiple primes or multiple elements from being hashed into the same prime. A common method of hashing to prime numbers is to combine a pseudo-random function (*e.g.*, hash function) with a probabilistic primality test. The resulting function consists of successively hashing the concatenation of the data to hash with an incremented counter until finding the smallest nonce that yields a prime number.

C.2.2 Experimentations

All our tests are executed sequentially on a core i9-11950H processor. Our implementation is coded in C++ using the GNU MP Bignum library. We instantiated H_{Primes} using SHA3 and a probabilistic Miller-Rabin primality test with 80 rounds, resulting in an error probability of 4^{-80}. The output of SHA3 is truncated to 264 bits, which is sufficient to obtain 128 bits of security as the prime counting function estimates that there are at least 2^{256} primes in $[0, 2^{264}]$. We measure an average time of 9 ms to hash a single element to a 264 bits prime. We also implement a 2048-bit RSA accumulator. Table 4 shows the performance of proof generation and verification. We observe that while (non-)membership proof generation from "scratch" scales linearly with the size of the accumulated set, membership proof generation during element addition is computed in constant time as element accumulation consists of a single exponentiation of the RSA digest by the newly added element. Similarly, we observe that (non-)membership proof verification is constant time, as it also consists of a single group exponentiation.

C.3 Commitment scheme

In our system, each process creates a hiding commitment of a single value (the balance of its account). This commitment can be based on any constant-size commitment scheme with trustless setup. For example, one could use Pedersen commitments in groups of unknown order (or elliptic curves [21]), which are unconditionally hiding and computationally binding. Conversely, other schemes such as ElGamal commitments [24] could be used to obtain commitments that are unconditionally binding but computationally hiding. Note that it is impossible for a commitment scheme to be both unconditionally binding and unconditionally hiding.

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

For our zk-SNARK scheme, we choose Spartan [42], which is both transparent (trustless setup) and prover time-optimal.

D A consensus-free quorum-based Agreement Proof implementation

In this section, we provide an implementation of the agreement proof scheme of Section 6.1 that does not rely on consensus, uses succinct cryptography, and has a storage of $O(\lambda + n)$ bits per process and an overall communication of $O(\lambda n)$ bits. Our AP implementation leverages threshold digital signatures (see Appendix D.1) to create constant-size (*i.e.*, $O(\lambda)$-bit) agreement proofs σ (*i.e.*, quorums of signatures) and to verify them without the set of all public keys.

D.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, classical threshold signature schemes use a protocol called distributed key generation (DKG) to distribute the secret key of a signature scheme across processes. Processes then interact to reconstruct a single signature that can be verified against a single system-wide public key. The computation of the individual shares of the secret key and the derivation of the system-wide public key are typically performed during the setup phase of the system. Naturally, a k-threshold signature for a message m is valid if and only if it aggregates k valid distinct intermediary signatures for m.

Various threshold signature schemes have been built on top of asymmetric schemes such as RSA, Schnorr, BLS or ECDSA [8, 16, 31, 40, 43, 44]. 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 using the system-wide public key. For example, the asynchronous DKG protocol of [16] can be used with DLOG-based threshold cryptosystems such as BLS [8] or Schnorr [5, 40] to provide k-threshold signatures with high-threshold values ($k \in [t, n-t-1]$).

D.1.1 Succinctness of threshold signatures

Efficient high-threshold signature implementations (*e.g.*, [5, 8, 40]) 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. Unlike classical signature schemes and multi-signature schemes, which require knowing the public keys of all 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 quorum signatures of size $O(\lambda n)$ bits with classical signature schemes or of size $O(\lambda + n)$ bits with multi-signature schemes (as the set of signers of a quorum can be encoded in a bit vector), threshold signatures can produce quorum signatures of size $O(\lambda)$ bits (once a quorum has been generated, the individual identities of the signers are not needed for its verification). Therefore, these implementations are of constant size and constant verification time, and thus succinct.

D.1.2 Verifying intermediary signatures without storing all individual public keys

As mentioned 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, it is possible to use a *vector commitment* (*VC*) [13], a generalization of the classical commitment scheme presented in Appendix A.1. Like its name suggests, a VC scheme commits a vector of values, upon which it is then possible to have a "partial" opening of a single value of the vector, without having to open the vector in its entirety. One of the most widely known examples of vector commitment is Merkle trees [35].

In our context, this scheme would allow us to "compress" the entire vector of public keys of the system into a VC of size $O(\lambda)$. This VC would be created during the system setup. In addition to the global VC, each correct process p_i also stores its individual public key, and the partial opening for the i^{th} value of the VC, corresponding to p_i's public key. This way, p_i can prove to another process p_j that it indeed belongs to the system, by sending to p_j its individual public key and partial opening. With this data, p_j can then verify that the i^{th} value of the global VC indeed opens to p_i's public key.

Since each correct process p_i stores the global VC ($O(\lambda)$ bits), its individual public key ($O(\lambda)$ bits), and the partial opening of the i^{th} value of the global VC ($O(\lambda)$ bits), this technique does not impact the overall storage cost of our AP algorithm (Algorithm 4) of $O(\lambda + n)$ bits stored per correct process.

More generally, this technique effectively allows us to have a public key infrastructure (PKI) under a fixed storage cost of $O(\lambda)$ bits per correct process (provided that the set of public keys does not change over time). For simplicity, all this machinery is hidden in the validation of intermediary signatures of Algorithm 4, lines 15 and 22.

D.1.3 Efficiently aggregating intermediary signatures

For constructing a k-threshold signature, a coordinator process has to receive and aggregate k intermediary signatures by k distinct processes. In particular, it has to make sure that it does not aggregate multiple intermediary signatures from the same process.

For the sake of simplicity, we address these issues by a naive approach in Algorithm 4: we require the coordinator p_i to save all the intermediary signatures it receives (and implicitly their signer's identity) in the $sigs_i$ set (line 25). Once the threshold signature has been produced, $sigs_i$ is emptied, and all the temporary data (intermediary signatures and sender identities) are deleted. The $sigs_i$ set contains at most n intermediary signatures ($O(\lambda)$ bits) and signer identities ($O(\log n)$ bits), which incurs a total of $O(n(\lambda + \log n))$ bits. However, as this data is stored only temporarily during the execution of the ap_prove() operation, we consider that it pertains to the *space complexity* of ap_prove(), and not the overall *storage cost* of our system.

Remark that this suboptimal space complexity can be easily reduced to $O(\lambda + n)$ bits using the following method. Instead of aggregating the intermediary signatures all at once at the end of the execution of ap_prove() (line 11), they can be gradually aggregated as they are received (line 21) on an aggregated signature of $O(\lambda)$ bits, which becomes a threshold signature once it aggregates at least k intermediary

signatures. This way, each intermediary signature can be forgotten right after it is received and aggregated. Furthermore, the coordinator can use a bit vector of n bits to efficiently keep track of the processes that already participated in the aggregated signature (the i^{th} bit indicates if the intermediary signature of p_i was already received or not).

D.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 storage cost of $O(\lambda + n)$ bits per correct process, and a communication of $O(\lambda n)$ bits sent overall.[16]

Algorithm 4 describes the code of this implementation for a correct process p_i and a AP predicate P_A. It uses 3 local variables: $sigs_i$ (a set of intermediary signatures used for generating threshold signatures), v_i (the value of the current ap_prove execution), and seq_nums_i (the array containing the sequence number (initialized with 0) of each process p_j, $j \in [n]$).

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 computes its next sequence number sn_i of p_i (line 5). Then, it verifies that the provided v, $data$, and sn_i satisfy the AP predicate (line 6). If it does, p_i then saves the provided value v in the local variable v_i (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 v, $data$, and sn_j satisfy 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 sequence number of p_j in seq_nums_i with its new sn_j (line 20).

When p_i receives a QUORUMSIG message, it checks if the intermediary signature is valid (line 22), if it has not already received a previous intermediary signature from the same signer (line 23), and if it has not already produced an AP for the payload (line 24). If these conditions pass, p_i saves the provided intermediary signature (line 25).

D.3 Proof of Algorithm 4

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

Lemma D.1 (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 Obj_A.ap_prove$(v, \star)/\sigma_i$ as its sn_i^{th} invocation of Obj_A.ap_prove(..).*

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

[16] For these analyses, like in our asset transfer algorithm of Algorithm 3, we impose that sequence numbers are constant-size.

```
1  init: sigs_i ← ∅; v_i ← ⊥; seq_nums_i[1..n] ← [0..0].
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      sn_i ← seq_nums_i[i] + 1;                          ▷increment own sequence number
6      if P_A(v, data, sn_i) = false then return abort;
7      v_i ← v;                                           ▷save v for condition at line 21
8      sig_i ← initialization signature of ⟨v, data, sn_i, i⟩ by p_i;
9      broadcast QUORUMINIT(v, data, sn_i, i, sig_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) 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(v, data, sn_j) = false then return;
17     wait seq_nums_i[j] = sn_j − 1;
18     sig_i ← intermediary signature for ⟨v, sn_j, j⟩ by p_i;
19     send QUORUMSIG(i, sig_i) to p_j;
20     seq_nums_i[j] ← sn_j.                              ▷update p_j's sequence number
21 when QUORUMSIG(j, sig_j) is received do
22     if sig_j is not a valid intermediary signature for ⟨v_i, sn_i, i⟩ by p_j then return;
23     if some intermediary signature by p_j already in sigs_i then return;
24     if some AP σ_i for v_i already produced by p_i then return;
25     sigs_i ← sigs_i ∪ {sig_j}.
```

Algorithm 4: Light and consensus-free quorum-based Agreement Proof algorithm for an AP predicate P_A, and assuming $n > 3t$ (code for p_i).

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 $Obj_A.\mathsf{ap\_prove}(v, \star)/\sigma_i$ execution. □

Lemma D.2 (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. More formally, $Obj_A.\mathsf{ap\_verify}(\sigma_i, v, sn_i, i) = Obj_A.\mathsf{ap\_verify}(\sigma_i', v', sn_i, i) = \mathtt{true}$ implies $v = v'$.*

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, $|A| > \frac{n+t}{2} < |B|$.[17] 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

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

intermediary signatures for some sn_i at line 18 and right after updating the i-th sequence number in their *seq_nums* array 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 D.3 (AP-Knowledge-Soundness)**.** *If σ_i is a valid AP for value v at sequence number $sn_i > 0$ from a prover p_i (correct or faulty), then p_i knows some data such that $P_A(\star, data, sn_i)/\texttt{true}$.*

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 $P_A(\star, data, sn_i)/\texttt{true}$. □

Lemma D.4 (AP-Termination)**.** *Given a correct process p_i that executes Obj_A.ap_prove$(v, data)/r$ with value v as its sn_i^{th} invocation of Obj_A.ap_prove(..), and a data, if $P_A(\star, data, sn_i) = \texttt{true}$ then $r = \sigma_i$, where σ_i is a valid AP for v at sn_i from p_i. If $P_A(\star, data, sn_i) = \texttt{false}$, then $r = \texttt{abort}$.*

Proof. Let us assume that a correct process p_i executes Obj_A.ap_prove$(v, data)/r$ with value v and a *data* as its sn_i^{th} invocation of Obj_A.ap_prove(..). If $P_A(\star, data, sn_i) = \texttt{false}$, then p_i passes the condition at line 6 and return abort. If $P_A(\star, data, sn_i) = \texttt{true}$, 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 6), 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 $seq\_nums_j[i]$ is initialized at line 1 to 0, 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 $\lfloor \frac{n+t}{2} \rfloor + 1$) 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 21, 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. □

D.4 Consensus-freedom of Algorithm 4

The consensus-freedom of Algorithm 4 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.

D.5 Succinctness of Algorithm 4

The succinctness of Algorithm 4 comes from the succinctness of threshold signatures: as shown in Appendix D.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 4) are equivalent to that of a single intermediary signature. Therefore, Algorithm 4 is succinct.

D.6 Storage cost of Algorithm 4

The storage of Algorithm 4 is of $O(\lambda + n)$ bits stored by each correct process. It 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)[18], and seq_nums_i is an array of size n containing constant-size sequence numbers (by definition). Lastly, the threshold signature scheme requires the storage of $O(\lambda)$ bits (see Appendix D.1.2). Therefore, each correct process only stores $O(\lambda + n)$ bits in Algorithm 4.

D.7 Communication cost of Algorithm 4

The communication cost of Algorithm 4 is $O(n\lambda)$ bits sent by correct processes overall during an execution of ap_prove. It follows from the message exchanges entailed by an ap_prove$(v, data)$ invocation by a correct prover p_i. Recall that, for simplicity, we assume that the size of the v and $data$ parameters of ap_prove(..) are of constant size.

In a first step, p_i broadcasts a QUORUMINIT$(v, data, sn_i, i, sig_i)$ message at line 9, where v, $data$, and sn_i are constant-size, i has $O(\log n)$ bits, and sig_i has $O(\lambda)$ bits. As this message is broadcast to all n processes, the overall communication cost of this first step is $O(n(\lambda + \log n))$ bits.

In a second step, every correct process of the system p_j receives the QUORUMINIT message, passes the conditions at lines 15 and 16, and sends a QUORUMSIG(j, sig_j) message to p_i at line 19, where j has $O(\log n)$ bits, and sig_j has $O(\lambda)$ bits. This amounts to $O(n(\lambda + \log n))$ bits sent overall by correct processes during this second step.

Therefore, the total number of bits sent by correct processes during the execution of Algorithm 4 is $O(n(\lambda + \log n))$. However, as we assume that $\lambda = \Omega(\log n)$ (see Section 3, ¶ *Security parameter* λ), the previous asymptote simplifies to $O(n\lambda)$ bits sent overall.

E 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, *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 C).

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. Generate a threshold signature σ_i (compatible with Algorithm 4) that will serve as the initial agreement proof of p_i for each $v_i = \langle A_i, bal\_c_i \rangle$.

[18] See Appendix D.1.3 for a discussion on why we consider the storage of these temporary variables to be part of the space complexity of the ap_prove() operation (and not the system's storage cost), and how this space complexity could be easily reduced to $O(\lambda + n)$ bits.

5. Each process p_i can now safely delete genesis_data and output its initial parameters $\langle A_i, bal\_c_i, \sigma_i, bal\_o_i \rangle$.

F 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 6.2.5.

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 and a balance 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 [32] or Sangria [23] allow the creation of 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.

References

[1] A. Auvolat, D. Frey, M. Raynal, and F. Taïani. Money transfer made simple: a specification, a generic algorithm, and its proof. *Bulletin of EATCS*, 132:22–43, 2020.

[2] M. Baudet, G. Danezis, and A. 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.

[3] M. Baudet, A. Sonnino, M. Kelkar, and G. 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.

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

[5] F. Benhamouda, S. Halevi, H. Krawczyk, Y. Ma, and T. Rabin. SPRINT: high-throughput robust distributed schnorr signatures. In *EUROCRYPT (5)*, volume 14655 of *Lecture Notes in Computer Science*, pages 62–91. Springer, 2024.

[6] N. Bitansky, A. Chiesa, Y. Ishai, R. Ostrovsky, and O. Paneth. Succinct non-interactive arguments via linear interactive proofs. *J. Cryptol.*, 35(3):15, 2022.

[7] D. Boneh, B. Bünz, and B. Fisch. Batching techniques for accumulators with applications to IOPs and stateless blockchains. In *Proc. 39th Int'l Cryptology Conference*, LNCS, pages 561–586, 2019.

[8] D. Boneh, B. Lynn, and H. Shacham. Short signatures from the weil pairing. *J. Cryptol.*, 17(4):297–319, 2004.

[9] J. Bonneau, I. Meckler, R. Vanishree, and E. Shapiro. Mina: decentralized cryptocurrency at scale (White paper), 2020.

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

[11] B. Bünz, S. Agrawal, M. Zamani, and D. 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 *LNCS*, pages 423–443, 2020.

[12] B. Bünz, L. Kiffer, L. Luu, and M. Zamani. Flyclient: super-light clients for cryptocurrencies. In *Proc. IEEE Symposium on Security and Privacy*, pages 928–946, 2020.

[13] D. Catalano and D. Fiore. Vector commitments and their applications. Cryptology ePrint Archive, Paper 2011/495, 2011.

[14] D. Collins, R. Guerraoui, J. Komatovic, P. Kuznetsov, M. Monti, M. Pavlovic, Y. Pignolet, D. Seredinschi, A. Tonkikh, and A. Xygkis. Online payments by merely broadcasting messages. In *Proc. 50th IEEE/IFIP Int'l Conf. on Dependable Systems and Networks (DSN'20)*, pages 26–38. IEEE, 2020.

[15] I. Damgård and M. Koprowski. Generic lower bounds for root extraction and signature schemes in general groups. In *Proc. Int'l Conference on the Theory and Applications of Cryptographic Techniques (EUROCRYPT'02)*, volume 2332 of *LNCS*, pages 256–271. Springer, 2002.

[16] S. Das, Z. Xiang, L. Kokoris-Kogias, and L. 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] D. Derler, C. Hanser, and D. Slamanig. Revisiting cryptographic accumulators, additional properties and relations to other primitives. In *Proc. RSA Conference*, volume 9048 of *LNCS*, pages 127–144. Springer, 2015.

[18] S. Dobson, S. D. Galbraith, and B. Smith. Trustless unknown-order groups. *CoRR*, abs/2211.16128, 2022. eprint: `2211.16128`.

[19] P. Fauzi, S. Meiklejohn, R. Mercer, and C. 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] L. Fortnow. The complexity of perfect zero-knowledge. *Adv. Comput. Res.*, 5:327–343, 1989.

[21] C. Franck and J. Großschädl. Efficient implementation of Pedersen commitments using twisted Edwards curves. In *Proc. 3rd Int'l Conference on Mobile, Secure, and Programmable Networking (MSPN'17)*, volume 10566 of *Lecture Notes in Computer Science*, pages 1–17. Springer, 2017.

[22] D. Frey, M. Gestin, and M. 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*, 21:1–21:23. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2023.

[23] A. Gabizon, Z. J. Williamson, and O. Ciobotaru. PLONK: permutations over Lagrange-bases for oecumenical noninteractive arguments of knowledge. *IACR Cryptol. ePrint Arch.*:953, 2019.

[24] T. E. Gamal. A public key cryptosystem and a signature scheme based on discrete logarithms. *IEEE Trans. Inf. Theory*, 31(4):469–472, 1985.

[25] O. Goldreich. *The Foundations of Cryptography - Volume 1: Basic Techniques*. Cambridge University Press, 2001, pages 1–372. ISBN: 0-521-79172-3.

[26] R. Guerraoui, P. Kuznetsov, M. Monti, M. Pavlovic, and D. Seredinschi. The consensus number of a cryptocurrency. *Distributed Comput.*, 35(1):1–15, 2022.

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

[28] D. E. Hopwood, S. Bowe, and T. H. N. Wilcox. Zcash protocol specification, 2016.

[29] A. Jain, E. Anceaume, and S. 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.

[30] A. Kiayias, N. Leonardos, and D. Zindros. Mining in logarithmic space. In *Proc. ACM SIGSAC Conference on Computer and Communications Security (CCS'21)*, pages 3487–3501. ACM, 2021.

[31] C. Komlo and I. Goldberg. FROST: flexible round-optimized schnorr threshold signatures. In *Proc. 27th Int'l Conf. on Selected Areas in Cryptography (SAC'20)*, volume 12804 of *LNCS*, pages 34–65. Springer, 2020.

[32] A. Kothapalli, S. T. V. Setty, and I. 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.

[33] P. Kuznetsov, Y. Pignolet, P. Ponomarev, and A. Tonkikh. Permissionless and asynchronous asset transfer. *Distributed Computing*, 36(3):349–371, 2023.

[34] L. Lamport, R. E. Shostak, and M. C. Pease. The Byzantine generals problem. *ACM Trans. Program. Lang. Syst.*, 4(3):382–401, 1982.

[35] R. C. Merkle. A digital signature based on a conventional encryption function. In *Proc. 7th Int'l Cryptology Conference (CRYPTO'87)*, volume 293 of *LNCS*, pages 369–378. Springer, 1987.

[36] S. Nakamoto. Bitcoin: a peer-to-peer electronic cash system, 2008.

[37] O. Naor and I. Keidar. On payment channels in asynchronous money transfer systems. In *Proc. 36th Symposium on Distributed Computing (DISC'2022)*, volume 246 of *LIPIcs*, 29:1–29:20, 2022.

[38] M. C. Pease, R. E. Shostak, and L. Lamport. Reaching agreement in the presence of faults. *Journal of the ACM*, 27(2):228–234, 1980.

[39] M. G. Reed, P. F. Syverson, and D. M. Goldschlag. Anonymous connections and onion routing. *IEEE J. Sel. Areas Commun.*, 16(4):482–494, 1998.

[40] T. Ruffing, V. Ronge, E. Jin, J. Schneider-Bensch, and D. Schröder. ROAST: robust asynchronous Schnorr threshold signatures. In *Proc. ACM SIGSAC Conference on Computer and Communications Security (CCS'22)*, pages 2551–2564. ACM, 2022.

[41] P. Schindler, A. Judmayer, N. Stifter, and E. R. Weippl. HydRand: efficient continuous distributed randomness. In *Proc. IEEE Symposium on Security and Privacy (SP'20)*, pages 73–89. IEEE, 2020.

[42] S. T. V. Setty. Spartan: efficient and general-purpose zksnarks without trusted setup. In *Proc. 40th Annual International Cryptology Conference (CRYPTO'20)*, volume 12172 of *LNCS*, pages 704–737. Springer, 2020.

[43] D. R. Stinson and R. Strobl. Provably secure distributed schnorr signatures and a (t, n) threshold scheme for implicit certificates. In *Proc. 6th Australasian Conf. Information Security and Privacy (ACISP'01)*, volume 2119 of *LNCS*, pages 417–434. Springer, 2001.

[44] H. W. H. Wong, J. P. K. Ma, H. H. F. Yin, and S. 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.