Towards more scalable and privacy-preserving distributed asset transfer systems

Arthur Rauch

COLLEGE DOCTORAL BRETAGNE
MATHS, TELECOMS
INFORMATIQUE, SIGNAL
SYSTEMES, ELECTRONIQUE

Université de Rennes

THÈSE DE DOCTORAT DE

L'UNIVERSITÉ DE RENNES

ÉCOLE DOCTORALE N° 601
Mathématiques, Télécommunications, Informatique, Signal, Systèmes, Électronique
Spécialité : *Informatique*

Par

Arthur RAUCH

Towards more scalable and privacy-preserving distributed asset transfer systems

Thèse présentée et soutenue à Rennes, France, le 18.12.2024
Unité de recherche : IRISA (UMR 6074)
Thèse N° :

Rapporteurs avant soutenance :

Christian CACHIN Full Professor, University of Bern, Switzerland
Vincent GRAMOLI Associate Professor, University of Sydney, Australia

Composition du Jury :

Attention, en cas d'absence d'un des membres du Jury le jour de la soutenance, la composition du jury doit être revue pour s'assurer qu'elle est conforme et devra être répercutée sur la couverture de thèse

Président :	Prénom NOM	Fonction et établissement d'exercice *(à préciser après la soutenance)*
Examinateurs :	Christian CACHIN	Full Professor, University of Bern, Switzerland
	Vincent GRAMOLI	Associate Professor, University of Sydney, Australia
	Maria POTOP-BUTUCARU	Professeur des Universités, Sorbonne Université - LIP6, France
	Guillaume PIERRE	Professeur des Universités, Université de Rennes, France
Dir. de thèse :	Davide FREY	Chargé de recherche hors classe Inria, Rennes, France
Co-dir. de thèse :	Emmanuelle ANCEAUME	Directrice de recherche CNRS, Rennes, France

Invité(s) :

Prénom NOM Fonction et établissement d'exercice

Résumé en français

Les systèmes distribués pair-à-pair ont connu une popularité croissante au cours des dernières années avec la promesse d'échanges sans intermédiaires de confiance. Parmi ces systèmes, la blockchain s'est particulièrement distinguée. Les partisans des systèmes basés sur la blockchain, tels que les cryptomonnaies, ont souvent vanté l'anonymat qu'ils offrent comme une promesse de protection de la vie privée, et les blockchains ont par conséquent été envisagées comme un moyen de stocker des données de manière sûre et sécurisée. Cet argument s'inscrit dans un contexte où les utilisateurs sont de plus en plus préoccupés par la protection de leur vie privée et la confiance dans les institutions de la société moderne s'érode.

Les applications fondées sur la technologie blockchain n'ont pas seulement suscité l'intérêt des cercles technologiques et financiers, mais aussi l'attention des gouvernements, des régulateurs et des législateurs à un niveau plus politique. Par exemple, en 2018, les États membres de l'UE ont signé une déclaration commune pour un partenariat européen de la Blockchain, dans le but de développer une infrastructure distribuée capable d'améliorer les services numériques transfrontaliers au sein du marché unique numérique, en mettant l'accent sur la confiance et les valeurs. De même, dans ses conclusions du 9 juin 2020 intitulées "Façonner l'avenir numérique de l'Europe", le conseil de l'Union Européenne a identifié la blockchain comme l'un des piliers de la stratégie numérique de l'Union Européenne.

L'Observatoire et Forum Européen de la Blockchain décrit les plateformes blockchain comme un outil prometteur pour effectuer des transactions en temps réel et libérer d'importants avantages économiques. McKinsey, quant à eux, évoquent leurs utilisations gouvernementales. Par exemple, l'Estonie utilise une blockchain permissionnée pour stocker des signatures de hachage garantissant l'intégrité des données stockées sur les serveurs gouvernementaux. De même, la Suède a développé un système de gestion immobilière basé sur la blockchain capable d'enregistrer les transferts de propriété. Toutes ces initiatives témoignent de l'intérêt significatif porté à la technologie blockchain et démontrent son potentiel pour de nombreux scénarios d'application.

Malheureusement, la nature décentralisée, entièrement répliquée et immuable de la

blockchain entre en conflit avec des exigences légales clés imposées au stockage des données personnelles par plusieurs textes français et européens, notamment la Convention européenne des droits de l'homme du Conseil de l'Europe, la Charte des droits fondamentaux et le Règlement général sur la protection des données (RGPD).

Outre les questions juridiques, le modèle de réplication complète compromet la privacité des utilisateurs et introduit des surcoûts inutiles, causant ainsi des problèmes de scalabilité, même pour les applications actuelles à faible stockage telles que les cryptomonnaies. De nombreuses questions demeurent ouvertes tant au niveau juridique que technique.

Défis

L'un des principaux atouts des blockchains réside dans leur nature distribuée. Du point de vue des systèmes distribués, la plupart des blockchains existantes adoptent un modèle de réplication complète. Cela offre une bonne tolérance aux pannes, au prix d'un coût de stockage élevé. La cohérence des données dans les systèmes blockchain est garantie par de fortes contraintes de synchronisation entre les processus (par exemple, le consensus), ce qui a un impact majeur sur les performances. Par exemple, Bitcoin traite environ 7 transactions par seconde et Ethereum environ 15 transactions par seconde. Obtenir des systèmes efficaces pouvant être mis à l'échelle nécessite des solutions capables d'assurer la tolérance aux pannes avec des niveaux de réplication et de synchronisation plus raisonnables.

Les propriétés d'immutabilité et de réplication complète des blockchains semblent également entrer en conflit avec le droit français et européen, ce qui pose d'importants défis non seulement en termes de confidentialité et de contraintes légales, mais aussi au niveau technique, en particulier lorsqu'il s'agit de traiter des données personnelles. Le défi consiste donc à fournir de nouvelles abstractions plus adaptées aux opérations sur des données sensibles en matière de confidentialité. Ces abstractions devraient, par exemple, masquer le contenu des comptes des utilisateurs et de leurs transferts, et améliorer l'anonymat en cachant le lien émetteur-récepteur des échanges des utilisateurs. En même temps, les utilisateurs devraient toujours pouvoir s'appuyer sur la blockchain pour garder une trace de tous les accès aux données stockées et prévenir lds comportements malveillants.

Contributions

Les principales contributions de cette thèse sont les suivantes :

1. SplitChain, un protocole destiné à soutenir la création de blockchains évolutives basées sur la preuve d'enjeu et les comptes sans compromettre la décentralisation et la sécurité. Ceci est réalisé en utilisant le sharding, c'est-à-dire en divisant la blockchain en plusieurs chaînes plus légères gérées par leurs propres ensembles disjoints de validateurs appelés shards. Ces shards traitent des ensembles disjoints de transactions en parallèle et ne stockent que les transactions et les comptes utilisateurs de leur propre chaîne. Le protocole est conçu pour adapter dynamiquement le nombre de shards à la charge du système afin d'éviter les problèmes de surdimensionnement rencontrés dans les solutions statiques basées sur le sharding.
2. Un nouveau système de transfert d'actifs (cryptomonnaie) tolérant aux pannes byzantines asynchrones avec trois propriétés remarquables: quasi-anonymat, légèreté et absence de consensus. Le quasi-anonymat signifie qu'aucune information n'est divulguée concernant les destinataires et les montants des transferts d'actifs. La légèreté signifie que les schémas cryptographiques sous-jacents sont succincts et que chaque processus ne stocke que ses propres transferts. L'absence de consensus signifie que le système ne repose pas sur un ordre total des transferts d'actifs. L'algorithme proposé est le premier système de transfert d'actifs qui remplit simultanément toutes ces propriétés dans le modèle asynchrone avec des fautes byzantines. Pour cela, nous adoptons une approche modulaire combinant un nouvel objet distribué appelé preuves d'accord et des primitives cryptographiques telles que les commitments, les accumulateurs universels et les preuves à divulgation nulle de connaissance.

Ces contributions visent à résoudre les défis majeurs auxquels sont confrontés les systèmes blockchain et de transfert d'actifs distribués, en particulier en termes de scalabilité, de confidentialité et d'efficacité. Elles ouvrent la voie à des systèmes plus performants, plus respectueux de la vie privée et plus conformes aux exigences légales, tout en préservant les avantages fondamentaux de la technologie blockchain.

Plan du document

Le contenu de ce manuscrit est organisé comme suit :

— Le Chapitre 1 est consacré au contexte et aux travaux connexes de cette thèse. Il offre une vue d'ensemble des concepts fondamentaux des systèmes distribués et de la blockchain, ainsi qu'une revue détaillée de l'état de l'art dans les domaines pertinents.

— Le Chapitre 2 présente les solutions développées pour résoudre le problème de scalabilité des systèmes de transfert d'actifs distribués et des blockchains. Il détaille en particulier le protocole SplitChain et son approche innovante du sharding dynamique.

— Le Chapitre 3 se concentre sur les problèmes de confidentialité et d'anonymat dans les systèmes distribués. Il explore à la fois des primitives cryptographiques connues et nouvelles, et présente une conception efficace de transfert d'actifs préservant la confidentialité. Ce chapitre met en lumière l'importance croissante de la protection de la vie privée dans les systèmes blockchain et propose des solutions concrètes pour y répondre.

Table of Contents

List of publications

— Emmanuelle Anceaume, Davide Frey, and Arthur Rauch, "Sharding in Permissionless Systems in Presence of an Adaptive Adversary", in: Proceedings of the 31st International Colloquium on Structural Information and Communication Complexity, SIROCCO 2024, pp. 481-487, DOI: 10.1007/978-3-031-60603-8_26, URL: https://doi.org/10.1007/978-3-031-60603-8_26.

— Timothé Albouy, Emmanuelle Anceaume, Davide Frey, Mathieu Gestin, Arthur Rauch, Michel Raynal and François Taiani, "Asynchronous BFT Asset Transfer: Quasi-Anonymous, Light, and Consensus-Free", Submitted to PODC 2024, ArXiv: https://arxiv.org/abs/2405.18072.

— Arthur Rauch, "Student Research Abstract: SplitChain: Blockchain with fully decentralized dynamic sharding resilient to fast adaptive adversaries", in: Proceedings of the 38th ACM/SIGAPP Symposium on Applied Computing, SAC 2023, pp. 280-283, DOI: 10.1145/3555776.3577207, URL: https://doi.org/10.1145/3555776.3577207.

— Emmanuelle Anceaume, Davide Frey, and Arthur Rauch, "Sharding in Permissionless Systems in Presence of an Adaptive Adversary", in: Proceedings of the 12th International Conference on Networked Systems, NETYS 2024, Vol. 14783, pp. 1–31. Springer. URL: https://doi.org/10.1007/978-3-031-67321-4_1.

Abstract

Since 2018, there has been a notable increase in the popularity of peer-to-peer distributed systems. In particular, blockchain technology has seen the emergence of numerous applications, spanning from cryptocurrency to digital identity, and healthcare systems. Several authors have dedicated efforts to analyzing the tensions between decentralized systems such as the Blockchain, and privacy regulations. Most existing blockchains adopt a full replication model. From a legal perspective, the fully replicated nature of blockchains means that personal data is likely to be stored throughout the world, on blockchain nodes distributed across different countries. From a technical perspective, full replication provides good fault tolerance at the cost of scalability. To achieve scalability, we must develop solutions that can ensure fault-tolerance with more reasonable levels of replication, while protecting privacy and avoiding clashes with national or cross-border regulatory laws.

To address these issues, we propose two systems. The first is based on horizontal partitioning (sharding) of the blockchain to better distribute the costs of data storage and processing among subsets of peers. The second doesn't rely on consensus. It can therefore process independent transactions concurrently. Furthermore, it uses a set of cryptographic primitives to anonymize users' data exchanges and verify their legitimacy, without revealing or storing sensitive data.

Chapter 1

Introduction

Distributed, peer-to-peer systems have been gaining popularity in recent years with the promise of exchange without the need for trusted intermediaries. Among these systems, blockchain has been particularly prolific. Proponents of blockchain-based systems, such as cryptocurrencies, have often touted the anonymity they offer as a promise of privacy protection, and blockchains have consequently been envisioned as a way to store data in a safe and secure manner. This argument is in line with users' growing concerns about the protection of their privacy as well as the deterioration of trust in institutions within modern society. Applications based on the blockchain technology have not only attracted interest from technological and financial circles, but also at a more political level from governments, regulators and lawmakers. For instance, in 2018, EU member states signed a joint declaration for a European Blockchain Partnership with a "view to developing a blockchain infrastructure that can enhance value-based, trusted, user-centric digital services across borders within the Digital Single Market." Similarly, in its 9 June 2020's conclusions entitled "Shaping Europe's digital future", the Council of the European Union identified blockchain as one of the pillars of the European Union's digital strategy. The EU Blockchain Observatory and Forum describes blockchain platforms as a promising tool for transacting in real time, and unleashing vast economic benefit, while McKinsey [Che+] discusses their governmental uses. For example, Estonia has been using a permissioned blockchain to store hash signatures that guarantee the integrity of data stored on governmental servers. Similarly, Sweden has developed a blockchain-based real-estate management system that can store property transfers. All these initiatives show the significant interest in blockchain technology and demonstrate the potential it holds for a number of application scenarios. Unfortunately, the decentralised, fully replicated and immutable nature of the blockchain clashes with key legal requirements imposed on the storage of personal data by a number of French and European texts, including the *European Convention on Human Rights* of the council of Europe, the *Charter of Fundamental Rights* and the *General Data Protection Regulation* (GDPR). In addition to legal issues,

the full replication model compromises privacy and introduces unnecessary overhead and scalability issues, even for current low storage applications such as cryptocurrencies. A number of questions remain open both at the legal and at the technical level.

1.1 Challenges

One of the main assets of blockchains lies in their distributed nature. From a distributed systems perspective, most existing blockchains adopt a full replication model. This provides good fault tolerance, at the cost of extremely high storage space. Data consistency in blockchain systems is guaranteed by strong synchronisation constraints between processes (*e.g.*, consensus), which has a major impact on performance. For example, Bitcoin processes around 7 transactions per second and Ethereum around 15 transactions per second. Achieving scalability requires solutions that can achieve fault-tolerance with more reasonable levels of replication and synchronisation.

The immutability and full-replication properties of blockchains also seem to clash with French and European law, which poses important challenges not only in terms of privacy and legal constraints, but also at the technical level, particularly when dealing with personal data. The challenge therefore consists in providing new abstractions that can be more suitable for operations on privacy sensitive data. These abstractions should, for example, hide the content of user accounts and their transfers and improve anonymity by hiding the sender-receiver link in user interactions. At the same time, users should still rely on the blockchain to keep track of all the accesses to the stored data and prevent malicious behaviours.

1.2 Contributions

The main contributions of this thesis are:

— SplitChain, a protocol intended to support the creation of scalable proof-of-stake and account-based blockchains without undermining decentralization and security. This is achieved by using sharding, i.e. by splitting the blockchain into several lighter chains managed by their own disjoint sets of validators called shards. These shards process disjoint sets of transactions in parallel and only store the transactions and user accounts of their own chain. The protocol is designed to dynamically adapt the

number of shards to the system load to avoid over-dimensioning issues encountered in static sharding-based solutions.

— A new asynchronous Byzantine-tolerant asset transfer system (cryptocurrency) with three noteworthy properties: quasi-anonymity, lightness, and consensus-freedom. Quasi-anonymity means no information is leaked regarding the receivers and amounts of the asset transfers. Lightness means that the underlying cryptographic schemes are *succinct*, and each process only stores its own transfers. Consensus-freedom means the system does not rely on a total order of asset transfers. The proposed algorithm is the first asset transfer system that simultaneously fulfills all these properties in the presence of asynchrony and Byzantine processes. To obtain them, we adopt a modular approach combining a new distributed object called agreement proofs and cryptographic primitives such as commitments, universal accumulators and zero-knowledge proofs.

1.3 Outline

The contents of this document are organized as follows. Chapter 2 is dedicated to the background and related work of this thesis. Chapter 3 presents the solutions developed to address the scalability problem of distributed asset transfer and blockchain systems. Finally, Chapter 4 focuses on privacy and anonymity issues in distributed systems with both known and novel cryptographic primitives and an efficient privacy-preserving asset transfer design.

1.4 Acknowledgements

This work was partially supported by the French ANR project PriCLeSS (ANR-10-LABX-07-81), devoted to the modular design of building blocks for large-scale Byzantine-tolerant multi-users applications.

Chapter 2

Background and related work

2.1 Background

2.1.1 Common model for distributed computing

Processes. A distributed system comprises n processes denoted by $p_1, ..., p_n$. Depending of the system, the number of processes may change over time. Each process p_i has a unique identity. Up to t processes can be Byzantine, where a Byzantine process is a process whose behavior does not follow the code specified by its algorithm [LSP82; PSL80]. Byzantine processes may collude to fool non-Byzantine (a.k.a correct or valid) processes.

Process faults and Adversary. Processes in a distributed system can be subject to malfunctions known as faults. These faults can be unintentional (software or hardware) or malicious and orchestrated by an "adversary". A variety of models exist for characterising process failures in distributed systems. The most common fault models are crash faults and Byzantine faults. The crash failure model refers to processes that permanently stop participating in the system at an arbitrary point in time. The Byzantine fault model is more flexible. A process is said to be Byzantine if it deviates from the standard protocol and behaves in an arbitrary way. The adversary seeks to compromise the correctness of the system by controlling process faults and the scheduling of messages in the asynchronous model. We denote the adversary as Adv. The resilience of a distributed algorithm against the adversary is expressed as a maximum threshold for the number of malicious processes, expressed as a fraction of the total number of processes.

Network. We focus on the communication model called "message passing" in which processes exchange information in the form of messages. The model is said to be asynchronous if the messages can have arbitrary reception times. The communication channels between the processes are assumed to be reliable and authenticated, *i.e.*, the network does not corrupt, drop, duplicate, or create messages and there is no loss or alteration of messages and when a message is received, the recipient knows the identity of the sender. Moreover,

it is not possible for a process to usurp the identity of other processes.

Let MSG be a message type and v be the value contained in the message. Each process is provided with an active and a callback operation. The operation "send MSG(v) **to** p_j" is used for sending, and the callback "**when** MSG(v) **is** received **from** p_i" is used for receiving. 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.

2.1.2 Distributed primitives

2.1.2.1 Byzantine reliable broadcast

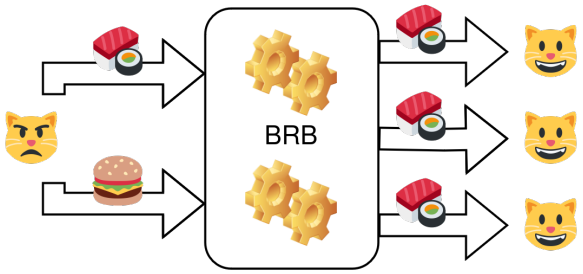


Figure 2.1 – Example of byzantine reliable broadcast execution

Byzantine reliable broadcast is an important distributed primitive for communication in fault-tolerant distributed systems. First defined in 1982 by Lamport, Shostak and Pease as the Byzantine generals problem [LSP82], it was applied to the asynchronous communication model by Bracha and Toueg in 1985 [BT85]. The objective of reliable broadcast is to construct a secure broadcast distributed operation in the presence of failures (crash or Byzantine). All processes are provided with the distributed operation brb_broadcast() and the callback brb_deliver(), each implemented using several invocations of the unreliable operation broadcast and its callback received. The two distributed operations are defined by the following properties [Ray18].

— **Validity**: If a correct process brb-delivers a message MSG from a correct process p_i, then p_i has brb-broadcasted MSG.

— **Integrity**: A correct process can only brb-deliver the same message once.

— **No duplicity**: No two correct process brb-deliver distinct messages from p_i.

— **Termination 1 (local delivery)**: If the sender p_i is correct, all correct processes eventually brb-deliver the same message.

— **Termination 2 (global delivery)**: If a correct process brb-delivers a message from p_i (regardless of p_i being faulty or not), then all correct processes brb-deliver a message from p_i.

In Figure 2.1, we can see that despite the contradictory values sent by the Byzantine cat, the honest cats all deliver the same message.

2.1.2.2 Consensus

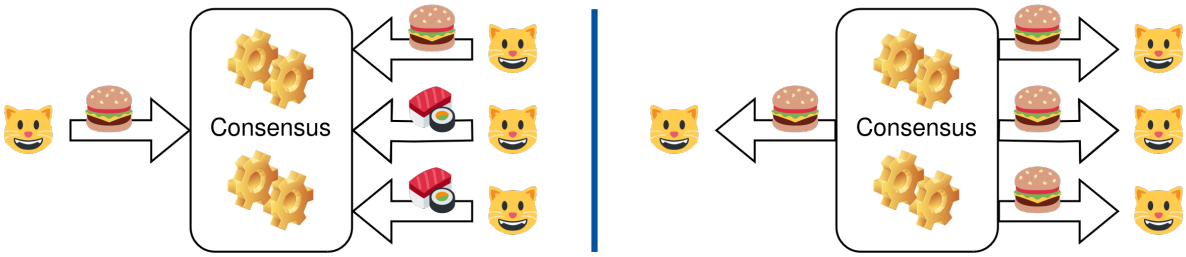


Figure 2.2 – Example of consensus execution

Consensus is one of the most important problems in distributed computing. It can be stated as follows. Each process is provided with a distributed operation "propose" that allows it to submit a value and returns the result r of the consensus. The process is said to "decide" the value r. It is assumed that all processes invoke the propose operation. Any consensus algorithm must respect three properties [LF82]:

— **Validity**: If a correct process decides a value v, then v was one of the proposed values.

— **Agreement**: If a correct process decides a value v, then all correct processes decide v.

— **Termination**: All correct processes decide on a value in a finite amount of time (the propose operation terminates).

In Figure 2.2, we can see that each cat proposes a value and that, despite proposing different values, they all deliver the same value at the end of the execution of the consensus algorithm.

Consensus in the asynchronous model. Fischer, Lynch and Paterson proved in their famous FLP impossibility theorem [FLP83] that the consensus problem cannot be solved deterministically in the asynchronous message passing model if at least one process may crash. The intuition behind this impossibility stems from the inability to distinguish a crashed process from a slow process or one with which the communication delay is high. Let p be a process waiting for a message from a process q. To ensure that the termination property is satisfied, p must decide whether it can stop waiting for q's message. If p stops waiting for q's message even though it has not crashed, safety (validity and agreement) can be compromised. On the other hand, if p decides to keep waiting for q to respond when q has crashed, then liveness (termination) is violated.

Many compromises have been made in the literature to get around FLP. For example, some distributed systems use stronger synchrony assumptions (e.g. semi-synchrony) [Gil+17; Luu+16] or weaken consensus guarantees. In particular, blockchain systems [Nak08a; ZMR18; Gil+17; Luu+16] favour the probabilistic consensus approach by allowing safety properties to be temporarily compromised with low probability.

2.1.3 Blockchains

Blockchain technology was first introduced by Nakamoto in 2008 [Nak08a]. A blockchain system relies on a large number of computation and storage processes known as miners or validators. These validators are responsible for receiving user requests (*e.g.*, transactions) submitted to the system, verifying them, grouping them together inside blocks and then adding them in order to a distributed append-only database called ledger. Several authors have attempted to propose a unified specification of blockchains. In particular, [Anc+19] proposes a unified framework for blockchains as an append-only tree (BlockTree) instead of a linked list and provides a hierarchy of consistency criteria that formally characterizes the histories admissible for distributed programs that operate on the blockchain. This captures the eventual convergence process in blockchain systems, which may allow the creation of forks, *i.e.*, concurrent branches of the BlockTree competing to become the main history of the blockchain.

Formally, a BlockTree is a directed rooted tree whose each vertex is a block and any edge points backward to the root, called genesis block. This genesis block designates a initial block common to all validators and containing a set of initial values for the blockchain. To interact with the BlockTree, processes can perform two operations, $\mathsf{append}(b)$ which appends a new block b to the BlockTree and $\mathsf{read}()$, which returns a sequence of blocks

(a.k.a. the blockchain) of the BlockTree determined by a selection function f. For example, in the case of Bitcoin, f returns the longest branch.

In addition to a set of transactions, a block of height n contains a fingerprint (*e.g.*, a Merkle root) of the state of the system, i.e. the result of applying the sequence of transactions stored in the first n blocks of the blockchain. Note that the change of state from block n to $n+1$ is performed by applying the transactions of block $n+1$ to the state of the system at block n. A blockchain therefore expresses the history of all state changes of a replicated state machine. This allows new validators to join the system trustlessly by replaying all state changes in the blockchain since the genesis block and obtain the latest state of the system.

Blockchains being distributed protocols, the use of an agreement algorithm is of course necessary for validators to agree on the content of blocks. Such an algorithm can be implemented by various mechanisms, the most popular among blockchains being consensus which provide a total ordering of the sequence of transactions. However, we will see later that it is possible to use weaker notions of order to construct an asset transfer system from weaker agreement algorithms, such as Byzantine reliable broadcast.

2.1.3.1 Identity management of validators

Distributed systems can be divided into two categories depending on how they manage validator identities: permissionless (or open) systems and permissioned (or closed) systems. Before the popularisation of large-scale peer-to-peer systems (*e.g.*, Gnutella, Bittorrent, Tor...), research on distributed systems focused mainly on permissioned systems, where the number and identity of participating processes are static and publicly known. In practice, the most common solution to defend against Sybil attacks is based on identity certification. However, this mechanism relies on a central, trusted certification authority that ensures that each process is associated with a single, unique identity. When the number of processes is known, it is easy to express the adversary's power in terms of the fraction or number of corrupted processes, and to apply the many results from the literature on distributed permissioned systems.

2.1.3.2 Sybil-resistance in permissionless systems

In a permissionless system subject to churn, *i.e.*, where the number and identity of processes varies, it is necessary to implement mechanisms for managing the identity of

processes. In particular, it is necessary to protect against Sybil attacks, where the adversary creates fictitious process identities in order to take control of the system. To achieve this, the creation of a process identity is linked to a limited resource, making the financial cost of a Sybil attack impractical. This resource may be based on the hardware of the validators (computing power, memory or disk storage capacity, etc.), such as Bitcoin's Proof of Work. In this case, the dissuasive factor against Sybil attacks is the cost of obtaining significant hardware capacity in the system. However, these methods are criticised for their energy consumption. For example, the University of Cambridge assessed in 2018 that the energy consumption of Bitcoin was comparable to that of Switzerland. Alternatively, the scarce resource can be directly integrated into the system in the form of a virtual asset. Dissuasion then comes from the investment required to obtain these assets, and the Byzantine adversary can be quantified as a fraction of the total number of assets in the system. This solution includes the cryptocurrency-based Proof of Stake and its many variants.

2.1.3.3 Scalability issues

The immutable nature of this ledger implies a constant growth of its storage requirements, with every block being retained for eternity. Not only does this impact storage, but it also increases the communication requirements for a new honest party to join the system as it needs to download the entire blockchain from the network in order to have a consistent view of the transaction history. Blockchain suffers from a scalability problem, both in terms of the total volume of transactions and the number of independent participants involved in processing them, which prevents its effective use for low-latency or high-speed applications. Various solutions have been explored to solve this problem, which can be broadly classified into three categories [Yu+20]: reducing overhead, vertical scaling and horizontal scaling (sharding). Reducing the overhead of duplicated communication and computation can be achieved by developing new blockchain consensus protocols. However, this type of solutions cannot reduce the overhead beyond $O(n)$, where n is the number of participants, as they must exchange and store messages during each consensus. Solutions for vertical scaling, i.e. adding resources to a single node, have also been explored. These include the augmentation of the maximum number of transactions per block or the organization of the transactions into directed acyclic graphs (DAGs) to parallelise their confirmation. This improves transaction throughput, however as the security of the blockchain depends heavily on the consensus of the entire decentralized network, network

growth entails a higher bandwidth requirement. Vertical scaling solutions cannot improve throughput indefinitely, limiting the blockchain to the throughput of resource constrained nodes.

2.1.3.4 Sharding

Sharding is a scaling solution involving splitting the blockchain into several smaller blockchains, called "shards", each processed and stored by its respective set of validators. Allocating validators to separate shards better balances communication and storage loads, accommodating more efficient network growth to achieve near-linear throughput scalability as the number of shards increases. However, sharding poses several challenges.

Firstly, since the system is partitioned, a shard can be compromised using a smaller malicious proportion of the total number of validators compared to what is required for classic blockchains. In order to ensure shard safety, it is necessary to implement an unbiased and verifiable random allocation of validators to prevent the adversary from targeting a particular shard. It is also necessary to periodically relocate validators to prevent the adversary from amassing corrupted validators inside a shard. Secondly, it is necessary to ensure both the verification and atomicity of cross-shard transactions. A cross-shard transaction refers to a transaction made between users managed by different shards. As each shard only knows the state of the users stored in its blockchain, it is necessary to ensure that *(i)* an invalid transaction will not be accepted by any shard, and *(ii)* a valid transaction partially accepted by the involved shards can be aborted so that funds cannot be locked up indefinitely or duplicated. To solve both problems there must exist some minimal amount of synchronization among shards.

2.1.3.5 Consensus-freeness

Until recently [Gue+22; Gup16; Auv+20], all asset-transfer systems relied on a consensus primitive. Fundamentally, an asset-transfer system must be double-spending free, that is, it must be impossible to spend a unit of value more than once. Consensus was thought to be necessary to avoid double spending, but as shown in [Gue+22; Auv+20], process order is sufficient when each account is owned by a single process. In systems where accounts can have multiple owners, total ordering is needed [Gue+22; Gup16]. If some account issues two conflicting transfers spending the same funds, the rest of the network, and especially the corresponding creditors, have to settle on which transfer (if any) is correct. This relaxed, per-account transfer ordering can be obtained by communication

primitives weaker than consensus, such as *reliable broadcast* [Bra87]. This has direct practical implications because consensus imposes sequential processing of the asset transfers (typically block by block in the case of blockchains). On the other hand, consensus-free asset-transfer systems can process transfers concurrently, resulting in higher throughput [BDS20; Col+20]. Consensus-freedom also allows deterministic algorithms to achieve progress in an asynchronous setting. 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 reader is invited to read [NK22], in which possibility and impossibility results for payment channels in asynchronous asset transfer systems have been addressed.

2.1.3.6 Legal compliance

With the promise of a secure peer-to-peer exchange technology with no trusted intermediaries, blockchain has become a popular tool for individuals and businesses alike. However, its use to store and process data raises concerns about privacy and compliance with current regulations, notably the GDPR. At the legal level, the GDPR forces data controllers to put in place appropriate technical and organisational measures, in particular concerning the collection and usage of personal data [BB20]. But who is the data controller in the context of a blockchain? Similarly, the GDPR also states that data subjects must give an informed consent about the processing of their personal data. On the one hand, we can assume that a data subject that stores his or her own data on the blockchain falls into the category of household exemption (i.e. Blockchain users would be seen as a group of private individuals, not performing any public activities). On the other hand, the involvement of personal data into smart contracts managed by third parties can easily clash with consent requirements, particularly when processing is automated. A simple solution can consist in only storing metadata on the blockchain. But some authors observe that the use of hashed signatures of personal data may, in some cases, retain a personal-data nature [BKP14]. This relates to the not-always-clear distinction between anonymity and pseudonymity as in the case of the de-anonymisation of blockchain users. In response to these concerns, two primary approaches have emerged.

Identification of participants: the permissioned approach

More focused on businesses, private or permissioned blockchains restrict access to the distributed ledger to a set of participants that have been authenticated by a trusted authority. This allows control over which participants can store or process a specific piece

of data and which users can access it. As a result, companies can easily designate data controllers and data processors for their blockchain. Because the identity of the participants of a permissioned blockchain is centralized, certain traditional properties of the public blockchain can be manipulated. For example, the immutability of stored data can be circumvented by causing participants to delete data. This raises questions about the reliability and robustness of these solutions in the face of malicious actors when compared with the decentralized and untrusted model of the public blockchain.

Data desensitisation: anonymity and confidentiality

On the other hand, the preservation of privacy in public blockchains focuses on the anonymity of users, by masking the content of transactions and the link between their senders and recipients. By using cryptographic commitments instead of storing data on the blockchain and making users anonymous, it is possible to remove the personal nature of the data stored on the public blockchain and escape the scope of the GDPR. However, this means that users cannot demand the application of their rights from a designated data controller, such as the explicability of the processing of their data in the event of theft, which reinforces the mistrust of companies towards these solutions.

2.1.4 Notions of cryptography

This section aims to offer a simplified view of the some fundamentals of modern cryptography. This includes the notion of computational security and hash functions. We refer the interested reader to the book [KL14] for a more in-depth approach to modern cryptography.

2.1.4.1 A simple approach to security

In order to prove that a scheme is secure, it is first necessary to have a formal definition of what "secure" is. This also allows protocols to be compared in a sensible way. Generally speaking, a definition of safety comprises the adversary's objective (a successful attack) and a threat model describing the power of the adversary and the actions it can perform.

Let us consider an arbitrary shared-key encryption scheme used by two parties to communicate secretly, with a key that can be used to encrypt and decrypt their messages. The safety guarantee would then be that a ciphertext does not leak any information about the corresponding message (or plaintext) to the adversary, no matter what information the adversary already has. We still need to characterise the power of the adversary. Numerous

options exist in the literature. In particular, the following examples of threat models are common.

— **Know-plaintext attack:** The adversary is able to learn one or more plaintext/ciphertext pairs generated using a key k and attempts to deduce information about the plaintex of some other ciphertexts produced using k.

— **Chosen-plaintext attack:** The adversary can obtain plaintext/ciphertexts for plaintext of its choice.

Perfect security Let us consider an arbitrary encryption algorithm $\mathcal{A}$ taking as input a message m and producing a ciphertext c. We say that $\mathcal{A}$ is perfectly secure if, for all messages m, m‘, the probability that a ciphertext c corresponds to m is equal to the probability that c corresponds to m', without relying on any additional assumption. Note that this definition is very powerful. It guarantees that the adversary cannot obtain information about a plaintext, even if it has infinite computing power. Unfortunately, it was proved by Shannon [Sha49] that such an encryption algorithm requires the keys, plaintexts and ciphertexts to be of the same size if optimal, which makes these algorithms highly impractical.

Computational security For most applications, perfect security is unnecessarily strong. In practice, it is acceptable for an encryption algorithm to disclose a fragment of information with a low probability to an adversary with bounded computing power. For example, in 2020, the largest factorised RSA number known to the public was 829 bits long [Bou+20]. It took around 2,700 CPU-years to factor it using a state-of-the-art distributed implementation. In practice, RSA keys are generally between 3072 and 4096 bits long. The notion of computational security thus involves two compromises that make it possible to overcome the limitations of perfect security:

— Security is only guaranteed against efficient adversarial algorithms that run for a reasonable length of time.

— The attack can succeed with a very low probability.

Let $\lambda \in \mathbb{N}_{>0}$ be the security parameter of a cryptographic scheme. The security parameter is chosen by honest parties during the initialisation of the scheme and parameterise both the running time of the adversary and the probability of a successful attack. An efficient adversary is modelled as a probabilistic poly-time algorithm (PPT), i.e. an algorithm executed by a non-deterministic Turing machine in time polynomial in λ (class

$\mathcal{NP}$). The probability of success $epsilon(\lambda)$ of an attack is said to be negligible, i.e. it is smaller than any inverse polynomial in *lambda*.

2.1.4.2 Cryptographic hash functions

Many algorithms are based on the existence of one-way functions, *i.e.*, functions that are efficient to compute but difficult to invert. In particular, a hash function is used to map an entry of arbitrary size to a unique digest of a chosen smaller fixed size parameterised by λ. We consider fundamental concepts of cryptographic hash function security, namely collision-resistance and pre-image resistance.

Definition 1 (Hash function)**.** *Let $H : 0,1^* \rightarrow 0,1^l$ be a hash function taking an arbitrarily long bit-string m as input and returning a short l-bit digest string h. H should possess the following properties:*

- *The computation of H should be efficient.*
- *The inversion of H should be difficult (H is preimage resistant), i.e., for a given $h = H(m)$, finding m is infeasible for a PPT adversary.*
- *H should be collision resistant, i.e., the probability that a PPT adversary finds two input bit-strings m, m' such that $H(m) = H(m')$ is negligible.*

It is also common to model a hash function as a random oracle, i.e. a function whose output is unpredictable. This additional property is used in many protocols. This is known as the random oracle model assumption.

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

2.2 Related Work

In this section, we will discuss some key solutions from the literature on blockchain sharding and asset transfer.

2.2.1 Blockchain sharding

In 2016, Elastico [Luu+16] introduced the concept of sharded blockchain. The execution of the system is partitioned into epochs. At the beginning of each epoch, validators are divided into disjoint subsets called shards. To determine its designated shard, each validator must solve a PoW problem. The least significant bits of the PoW are then used to identify the shard. Each shard of validators thus obtained is responsible for processing a disjoint subset of transactions and issuing the corresponding intermediate block of transactions. A final shard is then responsible for collecting these intermediate transaction blocks and aggregating them into a global transaction block which is added to the system's blockchain. Although this work provides an initial answer to the use of sharding to improve the scalability of blockchain systems, it raises a number of issues. Is it possible to overcome Elastico's assumption of a synchronous network? While renewing shards after each global block ensures strong safety against adaptive adversaries, it is done at the expense of synchronization and storage costs, i.e. all validators must store the global state and reconstruct their shard's overlay after each epoch. Attributing validators to shards using the output of PoW is subject to bias as validators can discard PoW results until they obtain their desired shard allocation [BCG15; Kok+18]. Elastico cannot ensure atomicity in cross-shard transactions [Kok+18], leading to permanent fund locking. Its attribution protocol coupled with its small shard size (approx. 100 validators) yields a very high corruption probability of 2.76% per shard per block, thus leading to a failure probability of 97% for 16 shards after only 6 epochs [Kok+18].

Omniledger [Kok+18] improves upon Elastico. It uses a more scalable consensus algorithm, increasing the size of its shards, thus reducing the probability of their corruption. It features shards with separate ledgers to better distribute storage costs and adapts classic distributed checkpointing principles to prune shard ledgers. It provides a synchronous lock/unlock client-driven mechanism to handle cross-shard transactions, although at the expense of lightweight-client compatibility and significant latency and safety issues [Son+20]. The UTXO model is not adapted to sharding, as its cross-chain transactions can consume UTXOs stored in different chains, having a significant impact on throughput as the number of shards increases. Omniledger reduces the cost of shard reconfiguration by bounding the number of validators shuffled in each of its day-long epochs, however, Omniledger requires a global blockchain for validator allocation.

Zilliqa [Tea17] is an account-based sharded blockchain that handles smart contracts. It inherits all the problems of Elastico except for its ability to handle validators on a separate

chain. It does not shard transactions storage. Moreover, it cannot provide atomicity for cross-shard transactions.

RapidChain [ZMR18] is a UTXO-based sharded blockchain that distinguishes itself from Omniledger by featuring a transaction routing protocol inspired by Kademlia, enabling message routing in log n steps without any special client interaction. To reduce the size of its shards, RapidChain uses a synchronous consensus algorithm tolerating a proportion of 1/2 Byzantine nodes. RapidChain thus inherits problems of Elastico and Omniledger, namely the use of PoW for validator enrollment (Elastico) and the use of the UTXO model (Omniledger).

Monoxide [WW19] is the first sharded blockchain to implement the use of PoW for shard consensus. It allows each miner to finalize and propose new transaction blocks to multiple shards simultaneously, amplifying and distributing their mining capabilities within the system. Its cross-shard transactions are handled in a lock-free manner. However, to guarantee the safety of Monoxide, the majority of miners are required to work for most if not all shards. This causes centralization problems and contradicts the load-distribution principle of sharding, requiring large throughput, storage and computation costs from miners. This behavior causes Monoxide to resemble more to a parallelization solution than to an actual sharded blockchain.

2.2.2 Consensus-free asset transfer systems

Following the seminal result of [Gue+22], several papers have proposed payment systems based on reliable broadcast only, namely AFRT20 [Auv+20], Astro [Col+20], and FastPay [BDS20]. More precisely, AFRT20 introduced the first specification of asset transfer as a concurrent object, presented thorough correctness proofs, and showed that asset transfer is a weaker problem than implementing read/write registers. Astro and FastPay, on the other hand, presented distributed implementations and associated benchmarks. Astro has been later extended to permissionless environments with Pastro (permissionless Astro) [Kuz+23], by combining weighted quorums and proof-of-stake. Building on FastPay, Zef [Bau+23] provides confidential and untraceable transfers. More precisely, Zef is a UTXO-based system where clients store their coins locally, while the network tracks spent coins. This construction uses a specific type of threshold signature that protects sender and receiver privacy to keep track of created coins. When a transfer operation occurs, the transfer recipient receives pieces of signatures that can be aggregated to construct a signature "on behalf of the whole network". This proof can be used to prove that the

coin owned by the client has been lawfully created. When the client transfers a coin, the network stores an identifier associated with it, so the coin can no longer be transferred. The identifier is stored in a so-called *deny list*. Because those operations are conducted without revealing the sender and receiver's identities, no element can be removed from the spent coin list. Therefore, the size of this protocol is linear as a function of the total number of transfers.

2.2.3 Anonymity in asset transfer systems

Anonymous Asset Transfer (AAT) systems have been studied since the introduction of Bitcoin [Nak08b] and can be categorized as either *token-based* or *account-based*. Token-based AAT uses two types of Zero Knowledge Proofs (ZKP): ZKP of token existence and ZKP that a token has not been spent yet. To transfer a token, the sender proves that it possesses a token that has not yet been added to a list of spent tokens. It then "destroys" this token (by adding it to the list of spent coins) and creates a new one belonging to the transfer's receiver. Zcash [HBW16] is an example of this first category. Account-based AAT is mainly represented by Quisquis [Fau+19] and Zether [Bün+20]. They only store commitments to different accounts. For each transfer, the sender proves that its account has enough funds, and full anonymity is provided by randomly selecting all or a subset of the system accounts.

All previously cited systems [HBW16; Fau+19; Bün+20] provide full anonymity (*i.e.*, they ensure sender and receiver anonymity), but they also require consensus. 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 [Bau+23] 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 [FGR23] has shown that consensus is necessary to guarantee full anonymity in a system where asset transfers from correct processes never fail. Zef [Bau+23] circumvents this impossibility by weakening the anonymity requirement. To achieve consensus freedom despite this impossibility, it is possible to weaken the anonymity guarantees of asset transfer, as done in Zef [Bau+23]. Note that when the anonymity of only the sender or the recipient is hidden, we talk about quasi-anonymity. In particular, Zef is receiver anonymous.

Chapter 3

Scaling issues in widely distributed systems

3.1 Introduction

Blockchain technology is well known to provide a tamper-proof "append-only" distributed-ledger abstraction. The immutable nature of this ledger implies a constant growth of its storage requirements, with every block being retained for eternity. Not only does this impact storage, but it also increases the communication requirements for a new honest party to join the system as it needs to download the entire blockchain from the network in order to have a consistent view of the transaction history. Furthermore, the monolithic aspect of blockchains, where all participants must agree on the current state of the system by consensus, leads to inefficiencies and low transaction throughput, which gets worse as the number of participants grows. To address these issues, we propose two solutions. The first relies on sharding to dynamically divide the blockchain into a variable number of smaller blockchains, achieving performance gains as the number of participants increases. The second enables parallel processing of user transactions, allowing them to be verified without consensus using weaker notions of agreement. The two solutions are not mutually exclusive and can be used in conjunction with one another.

3.2 Blockchain sharding

Sharding is a scaling solution involving splitting the blockchain into several smaller blockchains, called “shards”, each processed and stored by its respective set of validators. Allocating validators to separate shards better balances communication and storage loads, accommodating more efficient network growth to achieve near-linear throughput scalability as the number of shards increases. However, sharding poses several challenges.

3.2.1 Challenges

Firstly, since the system is partitioned, a shard can be compromised using a smaller malicious proportion of the total number of validators compared to what is required for classic blockchains. In order to ensure shard safety, it is necessary to implement an unbiased and verifiable random allocation of validators to prevent the adversary from targeting a particular shard. It is also necessary to periodically relocate validators to prevent the adversary from amassing corrupted validators inside a shard. Secondly, it is necessary to ensure both the verification and atomicity of cross-shard transactions. A cross-shard transaction refers to a transaction made between users managed by different shards. As each shard only knows the state of the users stored in its blockchain, it is necessary to ensure that *(i)* an invalid transaction will not be accepted by any shard, and *(ii)* a valid transaction partially accepted by the involved shards can be aborted so that funds cannot be locked up indefinitely or duplicated. To solve both problems there must exist some minimal amount of synchronization among shards. Most sharded-based solutions, including Elastico [Luu+16], Omniledger [Kok+18], RapidChain [ZMR18] or Ethereum 2.0, achieve this using a global synchronization blockchain maintained by every validator in addition to their shard's blockchain. In addition, sharding solutions based on a fixed number of shards (static sharding) are penalized by an uneven distribution of transactions, as revealed by our experiments on Ethereum.

3.2.2 Contributions

We propose SplitChain, a fully decentralized state sharding solution such that,

— Shards progress at their own pace without requiring the maintenance of a synchronization blockchain or any heavy synchronization mechanisms;

— Shards keep a loosely synchronized view of each other's state to guarantee the processing of any cross-shard state updates in a bounded number of consensus executions;

— Shards are tolerant to a fast adaptive adversary controlling less than a third of all validators by relying on a novel distributed attribution mechanism;

— Shards self-adapt to the current payload of the system by self merging and splitting the set of accounts, while adapting to the presence of hot-spots;

— Shards efficiently forward transactions by leveraging the properties of hypercubic routing protocols.

3.2.3 Model

3.2.3.1 Accounts and validators

Splitchain uses the account model to provide durable identities to its users and to enforce single-input single-output transactions for easier transaction management. Accounts are divided into two categories: user accounts and validator accounts. Users can send and receive transactions, while validators participate in SplitChain's protocol.

Definition 2 (Accounts)**.** *Accounts are persistent balances identified by a public key hash.*

Definition 3 (Validators)**.** *Validator accounts are special accounts created by users to participate in SplitChain's protocol. All validator accounts possess the same constant amount of currency called stake.*

Any user can create a validator by submitting the corresponding amount of stake through a transaction. Users may possess multiple validators concurrently. Stake serves as a limited resource to render the cost of Sybil attacks impractical. Users specify a (bounded) number of "*cue*" blocks (see Definition 7) for the existence of their validator, after which the validators expires and its stake is returned.

3.2.3.2 Network model.

We consider a peer-to-peer network of validators that self-divide into several chains. Both the number of chains and validators vary over time to withstand a dynamic and open environment. As in the model presented in Section 2.1.1 of the background, we adopt the asynchronous communication model.

3.2.3.3 Threat model.

Definition 4 ((μ,δ)-adaptive adversary)**.** *We consider a Byzantine adversary that never controls more than a fraction* $0 \leq \mu < 1/3$ *of validators.*[1] *Furthermore, the adversary is adaptive over a period* $\delta \geq 3$*, in the sense that if the adversary chooses to corrupt some newly attributed validator, this corruption will be effective after* δ *successive consensus executions.*

1. Given that validators all hold the same amount of stake, this equates to controlling less than μ of the total validation stake.

3.2.3.4 Cryptographic functions.

Beyond common cryptographic functions, i.e. hash functions and asymmetric signatures, validators use verifiable random functions (VRF) [MRV99] to generate pseudo-random hashes. VRFs are computed privately using a validator's private key, however their result is publicly verifiable using the validator's public key. Validators also use Merkle trees to prove the inclusion of data in a dataset without knowing its content. Every leaf of the Merkle tree is labelled with the hash of a data item, and every node other than the leaves is labelled with the hash of the concatenation of the labels of its child nodes. The root of the tree serves as the commitment to the dataset, and the Merkle path of a data, i.e. the hashes of the sibling nodes of the nodes that connect a leaf to the root of the Merkle tree, serves as the proof of the inclusion of the data. The (μ, δ)-adversary's computational power is restricted to match standard cryptographic assumptions.

3.2.4 Structure of the blocks

There are two types of blocks in SplitChain: *chaining blocks* and *transaction blocks*. Each transaction block is paired with a chaining block: the two constitute the outcome of a single consensus execution. The index of a block refers to its position in the chain.

Definition 5 (Transaction blocks)**.** *Transaction blocks contain all the transactions accepted by validators during a consensus execution.*

Definition 6 (Chaining blocks)**.** *Chaining blocks store the hash of the previous chaining block of their chain and the Merkle root of the transaction block produced during the same consensus execution.*

Definition 7 (Cue block)**.** *Let N_{cue} be some positive integer. $\forall k \geq 0$, every chaining block at position $k \times N_{cue}$ of a chain is called a cue block. A cue block points to both the previous chaining block and the previous cue block.*

Chaining blocks are akin to traditional block headers. A chaining block of a chain C contains a list of hashes. In particular, it provides the Merkle roots of the transaction block produced during their consensus execution, and of the accounts managed by the chain as well as the fingerprints needed for cross-chain transaction verification. It also contains the list of the latest known chaining block indices of the chains other than C, which we refer to as the fingerprint of the block. The interested reader may refer to Figure 3.1. For

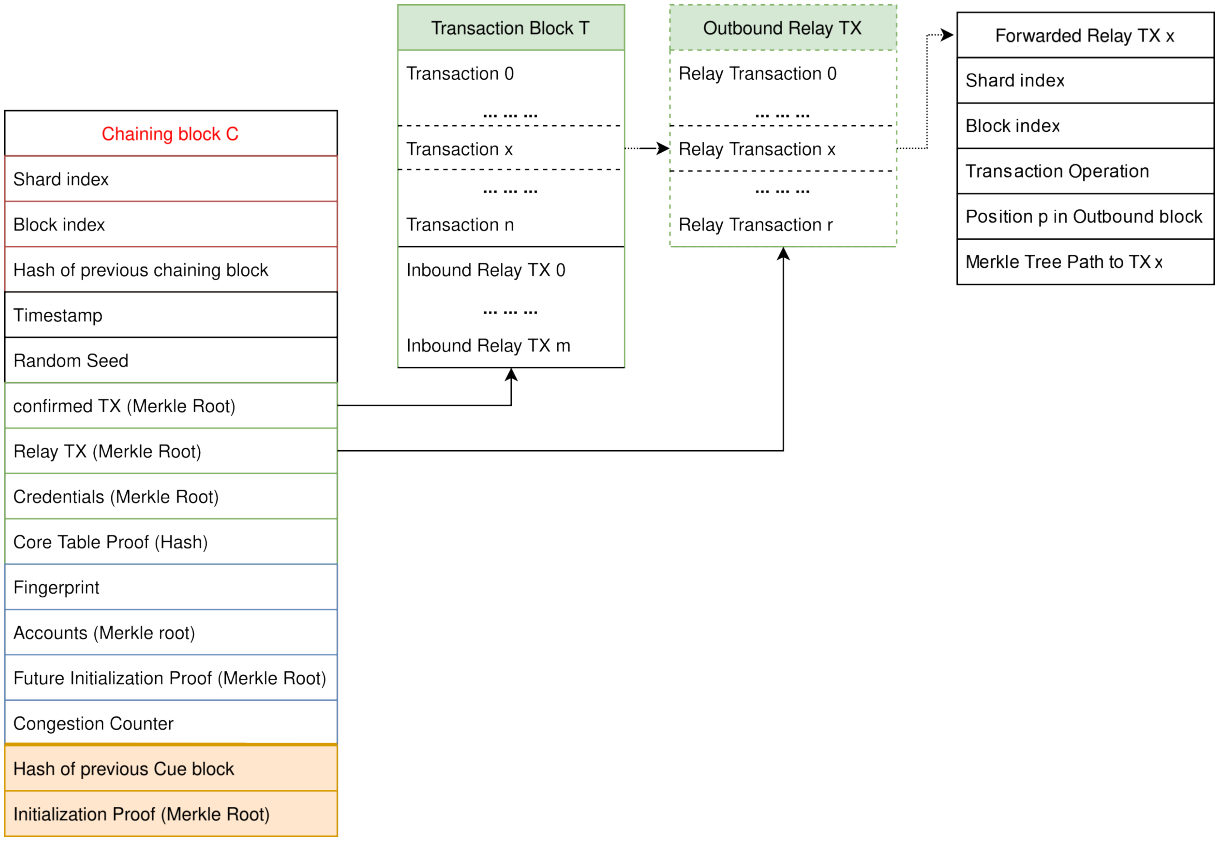


Figure 3.1 – Composition of a chaining block (bottom fields only belong to cue blocks.)

synchronization purposes, chaining blocks are disseminated to all chains using SplitChain's dedicated routing protocol described in Section 3.2.6. In contrast, transaction blocks of chain C are only stored by the validators participating in C's consensus executions.

3.2.5 Handling multiple chains

3.2.5.1 Assigning validators to chains

Each block of each chain is created via the local execution of a consensus algorithm. Consensus committees are periodically renewed (at each consensus execution) and their members are selected via two stages. The rationale of both stages is to prevent the (μ, δ)-adversary from predicting, and thus manipulating, members of consensus committees. Briefly, starting from their *initialization chains*, validators are uniformly distributed over their *reference chains* (first stage), and then uniformly distributed over their *consensus chains* (second stage). Validators stay forever in their initialization chain (that is as long

Algorithm 1 Initializing a validator account

input : The validator account registering transaction tx.

function `initialize`(tx, pk_{val})
- send(tx).to_any()
- $b \leftarrow$ `wait_for_validation`(tx)
- $C_{\text{init}} \leftarrow b.chain$
- $b_{\text{init}} \leftarrow$ `wait_for_cue_block`(C_{init})
- $tx_{\text{init}} \leftarrow (\text{"}init\text{"}, pk_{val})$
- $ack, \pi_{\text{init}} \leftarrow$ `wait_for_ack`(tx_{init})
- $core\_table[C_{\text{init}}] \leftarrow ack.table$
- **return** $\pi_{\text{init}}, b_{\text{init}}, C_{\text{init}}, core\_table$

output: The validator's initialization proof and cue block and its initialization chain's label and core table.

as they want to actively participate in the creation of SplitChain's blocks), they stay for N_{cue} blocks (see Definition 7) in their reference chain, and finally they stay for no more than the duration of three consensus executions in their consensus chains. Hence, for any chain C_i, and for any validators v, v' and v'' of Splitchain, at time t, C_i can be the initialization chain of v, while it is the reference chain of v' and finally the consensus chain of v''.

3.2.5.2 Initialization chain

Any user u wishing to actively participate in block creation must first register a *validator account*, with a given amount of stake. Validator accounts allow users to participate in different consensus executions. A single user can create multiple validator accounts if they own enough currency. Let $v = (sk_v, pk_v)$ be u's validator. User u submits a transaction to instantiate v's stake on what we call v's *initialization chain*. This chain, denoted by $C^{\text{init}(v)}$, is the chain whose label is the closest to $addr_v = H(pk_v)$ (by closest we mean the chain whose label minimizes the numerical value of the xor between $addr_v$ and the chain's label. The routing mechanism is described in Section 3.2.6). Initialization chains provide a stable anchor point for validators and allow them to prove the existence of their accounts. However, they do not contribute to SplitChain's security. Indeed, some malicious validator v' can iteratively invoke the cryptographic hash function to sit on some targeted chain $C^{\text{init}(v')}$. The current consensus committee of $C^{\text{init}(v)}$ updates the list L of new validators with v and inserts L's Merkle root into the chaining block under construction (in Figure 3.1, this is the "Future Initialization Proof" entry). When the next cue

Algorithm 2 Attribution algorithm

```
input : The list of validators L to attribute to chains, the number N of chains,
        a random seed seed.
function attribution(L, N, seed)
    L ← shuffle(L, seed)
    sets ← []
    α ← ⌊|L|/N⌋
    r ← |L| − N × α
    for i ← 0 to N − 1 do
        sets.append([])
        for j ← 0 to α − 1 do
            sets[i].append(L.pop())
        if r > 0 then
            sets[i].append(L.pop())
            r ← r − 1
    return sets
```

block of $C^{\text{init}(v)}$ is created (see Definition 7), the consensus committee in charge of that cue block will assign each new validator of L to their *reference chain* (see Algorithm 2). Specifically, Algorithm 2 shuffles L (using the seed of the cue block), partitions L into N sub-lists of $\lfloor |L|/N \rfloor$ validators, where N is the current number of chains of SplitChain, and assigns each random sub-list to one of the N chains of SplitChain. The Merkle root of L and the proof of validator assignment to their reference chain is included in the cue block (see Algorithm 1). The chain to which v will be assigned is called v's *reference chain* and is denoted by $C^{\text{ref}(v)}$. At every cue-block creation, Algorithm 1 is executed so that v is periodically re-assigned to a new reference chain.

3.2.5.3 Reference chain

Each reference chain $C_1, \ldots, C_N$ of SplitChain assigns uniformly at random its V/N referenced validators to the N *consensus chains* of SplitChain, where V is the total number of referenced validators in SplitChain. It is important to note that to face a (μ, δ)-adaptive adversary, validator v can not be attributed to the consensus committee of $C^{\text{cons}(v)}$ more than $\delta - 1$ consecutive times. Hence to be assigned to its consensus chain $C^{\text{cons}(v)}$, validator $v = (sk_v, pk_v)$ generates its consensus credential as a new key pair $(sk_{\text{cons}(v)}, pk_{\text{cons}(v)})$ and sends a signed credential storage request to the current consensus committee of $C^{\text{ref}(v)}$.

Algorithm 3 Referencing a validator

function `reference`($\pi_{\text{init}}, core\_init$)
  $pk_{\text{cons}}, sk_{\text{cons}} \leftarrow$ `generate_keys()`
  $tx \leftarrow$ ("$credential$", $\pi_{\text{init}}, pk_{\text{cons}}$)
  send(tx).to($core\_init$)
  $b, \pi_{\text{ref}} \leftarrow$ `wait_for_validation`(tx)
  $C_{\text{ref}} \leftarrow b.chain$
  return $\pi_{\text{ref}}, pk_{\text{cons}}, sk_{\text{cons}}, C_{\text{ref}}$

output: The validator's referencing proof.

Algorithm 4 Join a chain for consensus

input : The reference proof π_{ref}.
function `join_consensus`($\pi_{\text{ref}}, core\_table$)
  $tx \leftarrow$ ("$join$", π_{ref})
  send(tx).to_core(C_{ref})
  $ack \leftarrow$ `wait_for_ack`(tx)
  $core\_table[C_{\text{cons}}] \leftarrow ack.table$
  The validator bootstraps itself to the chain using the list of nodes provided in the ack message
  return $core\_table$

Validator v regenerates its credentials every δ attributions. Algorithm 3 describes the process of submitting credentials. This signed request contains $addr_v$, ref_v and $H(pk_{\text{cons}})$, proving the legitimacy of v's request. If the request is valid, v's consensus credential is added by the current consensus committee of $C^{\text{ref}(v)}$ to the list L of validators that will be used by Algorithm 2 to build N sub-lists of $\lfloor |L|/N \rfloor = V/N^2$ validators each. Such a sub-list is called a *reference set*, and each sub-list is assigned uniformly at random to one of the N chains os SplitChain. The chain to which v is assigned is called v's consensus chain and is denoted by $C^{\text{cons}(v)}$. The Merkle root "Credentials" (see Figure 3.1) of this list is added to the chaining block. Merkle paths are sent to the referenced validators of $C^{\text{ref}(v)}$ to serve as credential proof. The current consensus committee of $C^{\text{ref}(v)}$ sends to each consensus chain a transaction that contains the size of the reference set they have allocated to it. This allows validators to know the total amount of stake inside their consensus committee.

3.2.5.4 Consensus chain

To join the consensus committee of $C^{\text{cons}(v)}$, v issues a "join" request to $C^{\text{cons}(v)}$ using SplitChain's dedicated routing protocol. Upon receipt of the request, the core validators of $C^{\text{cons}(v)}$ directly reply to v with the list of core validators (see C3.2.6.2) and a sub-list of the current consensus committee members chosen at random. We call these validators the bootstrapping validators of v. The number of bootstrapping validators is large enough so that, with high probability $1-\varepsilon$, with $\varepsilon \in (0,1)$, at least one of them is honest. As chaining blocks cannot be forged by the (μ, δ)-adversary without taking over a chain's consensus, the proofs they contain are sufficient to verify the authenticity of the data provided by the bootstrapping validators. Thus, only one honest bootstrapping validator is sufficient for a successful bootstrap of v to $C^{\text{cons}(v)}$. Bootstrapping validators will provide v with $C^{\text{cons}(v)}$'s state. This state corresponds to the latest state of $C^{\text{cons}(v)}$'s user accounts, the proofs of attribution of the current consensus committee and the list of core validators. Note that the fingerprint of the latest chaining block b of $C^{\text{cons}(v)}$ contains the latest chaining-block index of each chain of SplitChain known to the consensus committee of $C^{\text{cons}(v)}$ when b was created. Therefore, the latest chaining block b contains the chaining block index of each reference chain used to create the current consensus committee of $C^{\text{cons}(v)}$. From this list, v establishes the list of attribution proofs of the committee, requesting the missing chaining blocks from its bootstrapping validators if necessary. Algorithm 4 presents the detailed pseudo-code of v's attribution to its consensus chain.

3.2.5.5 Creation of the blocks of a chain

As described above, each chain C_i of Splitchain plays the role of a consensus chain. By construction the consensus committee of C_i contains V/N validators. (Note that Section 3.2.5.6 presents a partial attribution policy that can replace the one presented in Section 3.2.5.1, when SplitChain is made of sufficiently many chains). For performance reasons, V/N validators cannot all be involved in each consensus execution. The solution we propose relies on a particular asynchronous consensus called the merge consensus algorithm [Sau+20]. This algorithm leverages the cryptographic sortition lottery introduced by Algorand [Gil+17] to elect, in a non-interactive and private way, a subset of the committee members. This subset has a bounded expected size that is large enough to handle adversarial behaviors, but independent of the size of the consensus committee. The consensus algorithm runs a series of asynchronous rounds, such that at each round,

a new subset of validators is elected via cryptographic sortition. The algorithm ensures, with high probability (whp), that all the transactions proposed by honest validators at the beginning of a consensus execution are included in a block after a finite number of rounds.

3.2.5.6 Leveraging a large number of chains

As explained in Section 3.2.5.4, all reference chains send their latest chaining block containing the Merkle root of their reference set to all the consensus chains. This allows the consensus committee of each chain to be updated with V/N^2 attributed validators. Once these sets have been received, consensus executions can be triggered (see Section 3.2.5.5). We call this policy the *total attribution policy*. It ensures that consensus committees cannot be compromised by a (μ, δ)-adversary with overwhelming probability (see Theorem 3.2.2). However, it introduces significant delays in the presence of a large number of chains.

When the number of chains is large enough (i.e., $N > 10$) the following *partial attribution* policy is more appropriate. A random subset of S reference chains is selected among the N chains of SplitChain. Each consensus committee contains $S(V/N^2)$ validators instead of $N(V/N^2)$. The S new reference sets of the next consensus committee are selected as follows: Let L be the list of chains included in the latest fingerprint of C_i. List L is randomly shuffled using the random seed of the latest chaining block of C_i. The first S chains in L become the reference chains providing the new reference sets of the next consensus of C_i.

For safety reasons, partial attribution cannot be used in the presence of only a few reference chains, as the (μ, δ)-adversary can concentrate its power to manipulate these few chains. Theorem 3.2.3 provides a lower bound S_{min} on S as a function of the total number of reference chains. Due to asynchrony and because the consensus committee of a chain C_i only waits for S reference sets to initiate the consensus, the chaining block of a chain C_j listed inside the fingerprint of the latest chaining block of C_i may be older than the actual latest chaining block of C_j. We call $\rho(N, S)$ the difference between the latest chaining block index of C_j and the latest chaining block index of C_j mentioned inside the fingerprint of C_i. To prevent the adversary from taking advantage of $\rho(N, S)$ to corrupt the new reference sets of C_i, we require its adaptivity to be reduced to $\delta + \rho(N, S)$ and the duration of a validator's membership to be reduced from $\delta - 1$ to $\delta - \rho(N, S)$. So, for the partial attribution policy, we consider a $(\mu, \delta + \rho(N, S))$-adaptive adversary. To bound the value of $\rho(N, S)$ with high probability, the consensus committee of C_i is

required to wait for the delivery of any new block b appearing in the fingerprint of the chaining blocks of one of the S new reference committees. Similarly, any new chaining block inside b's fingerprint must be delivered by C_i's committee before initiating a new consensus execution. Theorem 3.2.4 provides an upper bound on $\rho(N, S)$.

3.2.5.7 Pruning and verifying transactions

It is important to limit the amount of data stored by validators to prevent bootstrapping costs from linearly increasing with the number of blocks in a chain. We propose a pruning method that guarantees near-constant bootstrapping costs. Specifically, each validator v of the consensus committee of $C_i^{\text{cons}(v)}$ stores only all the cue blocks of $C_i^{\text{cons}(v)}$ as checkpoints and only the latest k chaining and transaction blocks of $C_i^{\text{cons}(v)}$. This allows v to respond to users that wish to verify old transactions that were validated in the latest k blocks. Users wishing to prove the inclusion of old transactions (i.e., those belonging to transaction blocks $b_i, \dots b_\ell$, that have been deleted) must locally store the chaining blocks between $b_i, \dots b_\ell$ and the next cue block. Cryptographic links between consecutive chaining blocks are then enough to recursively prove the existence of a pruned chaining block. The inclusion of the transaction is proven as per usual by the user against the Merkle root of the transactions contained in the chaining block. Each validator also stores the latest cue block of all the other chains: this supports the validator attribution strategy described in Section 3.2.5.1, as cue blocks contain the initialization proofs of validators.

3.2.5.8 Cross-chain transactions

Splitchain's transactions take place between any two user accounts.

Definition 8 (Cross-chain transactions)**.** *When both accounts of a transaction are not managed by the same chain, a transaction is said to be cross-chain.*

The chain in charge of handling a transaction is the one whose label prefixes the emitter account (denoted by C_{send} in the following). Cross-chain transactions are handled in two steps: the withdrawal operation and then the deposit operation. The withdrawal operation occurs when the transaction is inserted in a transaction block of C_{send}. During the creation of the new transaction and chaining blocks, the consensus committee members of C_{send} determine the list of cross-chain transactions inside the transaction block and generate the corresponding *relay transactions*, which they organize as a Merkle tree T. A relay transaction contains the user's initial transaction and the Merkle path of the

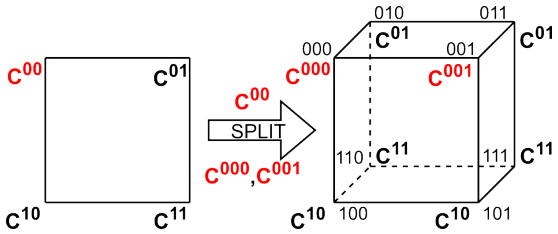


Figure 3.2 – Mapping of vertices and chain labels after a chain split causing an increase in hypercube dimension.

relay transaction inside T. The Merkle root "relay TX" (see Figure 3.1) of T is included into the new chaining block under construction. Consensus committee members of C_{send} then send the relay transaction to the receiver's account chain. As chaining blocks are propagated to all chains, the consensus committee members of the receiver's chain can verify the completion of the withdrawal operation and include the relay transaction in the next transaction block of their chain, which confirms the deposit operation, completing the cross-chain transaction. Note that transactions are always redirected to the chain whose label prefixes the identifier of the emitter account, whereas relay transactions are redirected to the chain of the receiver account.

3.2.5.9 Grouping user accounts

Users exchanging on a frequent basis may want to avoid the latency introduced by cross-chain transactions. Thus, we introduce the notion of group accounts that allow users of the same group to always be part of the same chain. A group account is a collection of accounts sharing the same group identifier, i.e. a key hash. When a user u wishes to open an account inside a group gid, u only needs to submit a transaction to a non-existent account (gid, pk_u). Similarly, when a user wishes to withdraw from a group, they can simply submit a transaction to transfer all their account funds elsewhere.

3.2.6 Routing

3.2.6.1 Hypercube, merging and splitting operations

SplitChain initially starts with a single chain C whose label is the empty chain of bits ε. If during N_{cue} consecutive blocks, the average number of transactions per block exceeds some threshold T_{split}, then C triggers a split. Splitting is an operation that allows a chain of label l to be replaced by two chains of labels $l|0$ and $l|1$, called siblings, each taking over

half of the accounts of the original chain. If the original chain C is labeled ε, then both new chains will be labeled 0 and 1 respectively. Conversely, if during N_{cue} consecutive blocks the average number of transactions per block is lower than a threshold T_{merge}, the chain initiates a merge with its sibling chain. Both chains merge in a single one whose label corresponds to the maximal prefix of their previous labels. By design, each chain label is unique. Our topology is inspired by hypercubic networks [ALS12]. A hypercube of dimension d contains 2^d vertices. Each vertex is assigned a d-bit label. Two vertices are neighbors if the numerical value of the xor of their labels is equal to a power of 2, i.e. if their labels differ by only one bit. For example, in Figure 3.2, vertex 000 is a neighbor of vertices 001, 010 and 100. The number of hops between two vertices can be determined using the Hamming weight of the xor of their labels. The diameter of the hypercube, i.e. the maximum number of hops between the two most distant vertices, is d. We use the following approach to match each chain to one or more vertices of a hypercube: Let k be the number of bits of the longest chain label. Splitchain conforms to a subgraph of a hypercube of dimension k, where each chain whose label has exactly k bits is mapped to the vertex of the same label, and each chain whose label contains strictly less than k bits is mapped to all vertices whose labels are prefixed by the chain's label. Thus, in Figure 3.2, chain 01 is mapped to vertices 010 and 011. As a result, when all chain labels are k bits long, the network corresponds to a hypercube of dimension k. A hypercube of dimension k can be constructed recursively by connecting two hypercubes of dimension $k-1$. This allows the dimension of the hypercube to be changed dynamically with k: if k increases, each vertex of label l of size $k-1$ is divided into two vertices of labels $l|0$ and $l|1$. Figure 3.2 shows the transition from a hypercube of dimension 2 to a hypercube of dimension 3 after chain 00 splits into (i.e., is replaced by) two chains 000 and 001. Similarly, if k decreases, vertices are merged in the same way as for chains. This ensures that no vertex represents two chains simultaneously.

3.2.6.2 Core validators

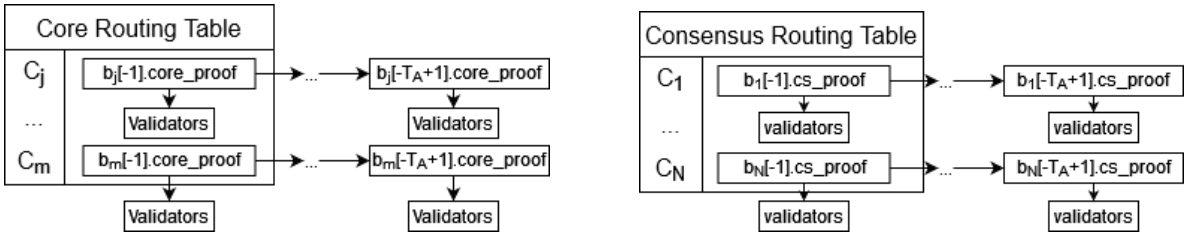
To limit the number of messages transmitted by validators and hence to improve the usage of network resources, we introduce the notion of core validators. The number of core validators is chosen to ensure that at least one honest validator belongs to the core. Election of core validators is as follows. When validator v is attributed to the consensus committee of $C^{\text{cons}(v)}$, v triggers twice the cryptographic sortition lotery at round 0. Once for determining whether it will executes round 0 of the merge-consensus algorithm

(see Section 3.2.5.5) and a second time to determine whether it will be part of $C^{\text{cons}(v)}$'s core. Both invocations of the cryptographic sortition lotery use the same parameters except for parameter τ, which represents the expected number of successful winners, so that core validators represent a subset of the validators elected for round 0 of the merge consensus. If successful, v is part of the core of $C^{\text{cons(v)}}$ until its consensus credentials expire, which corresponds to the duration that represents $\delta - 1$ consecutive consensus executions (v tries to be elected in the core of $C^{\text{cons(v)}}$ only once during the lifespan of its credentials). Once elected, core validators send their proofs of election along with their consensus messages, guaranteeing with any high probability that all committee members of $C^{\text{cons}(v)}$ will agree on the list of core validators at the end of the merge consensus execution. The hash of the list of new core validators will be included in the chaining block decided as the outcome of the merge consensus (see Figure 3.1). Core validators are in charge of executing the routing protocol. They maintain two routing tables: A core routing table containing the list of the core validators of neighboring chains for cross-chain communication, and a consensus table containing the list of their consensus committee members. By using their core routing table, core validators forward transactions to the core validators of the neighboring chain whose label is closest to the transaction's destination. On the other hand, the consensus table is used to broadcast messages within their consensus committee. Only core validators keep track of the list of committee members.

3.2.6.3 Core routing table

Definition 9 (Core routing table)**.** *The core routing table of chain C_i, $1 \leq i \leq N$, contains the list of the core validators of C_i's neighbouring chains.*

Core routing tables, denoted by RT^{core}, are maintained by core validators and contain the lists of core validators of all of its neighboring chains in the hypercube. For any two neighboring chains C_i and C_j, $RT_i^{core}[j]$ contains a linked list made of at most $\delta - 1$ elements. The k-th element of $RT_i^{core}[j]$, $k \leq \delta - 1$, contains the k-th most recent core proof of C_j and points to the core validators it proves (see Figure 3.3a). Upon receipt of a new chaining block b of C_j, core validators of C_i create and insert in block order a new element containing the core proof of b. Note that the core validators of C_j send separately the set of new core validators and chaining block b. Once received, it is verified using the core proof of b and then linked to the new element.



(a) Structure of the core routing table

(b) Structure of the consensus routing table

Figure 3.3 – Routing tables

3.2.6.4 Consensus table

Definition 10 (Consensus routing table)**.** *The consensus routing table of a chain C_i contains the list of validators of the consensus committee of C_i.*

Core validators also keep track of the consensus committee members. All intra-chain communication are handled by core validators, reducing the number of messages within a chain. This also isolates intra-chain communications from the overall network, allowing chains to be added without overloading SplitChain. A consensus routing table, as shown in Figure 3.3b, contains the list of consensus committee members sorted by reference chains and chronological order of attribution proof (the Merkle root called "Credentials"). The structure is composed of N linked lists of at most $\delta - 1$ elements, the i-th element of the j-th linked list containing the i-th most recent attribution proof of the validators referenced by the chain C_j. Each element points to the list of attributed validators. This list is updated upon receipt of validators "join" requests. When a core validator receives a new chaining block b from some chain C, it inserts in block order a new element containing b's attribution proof and removes the $\delta-1$ element if needed. This allows the gradual replacement of validators with old credentials, to prevent the (μ, δ)-adversary from compromising consensus committees. In addition, organizing validators with the proof of attribution of the chaining blocks ensures that when a proof expires, all the validator credentials proven by the proof also expire, whether these validators ever participated to the merge consensus executions or not. This prevents the adversary from withholding credentials to build up enough consensus credentials to take over a chain.

3.2.7 Security Analysis

In this section we analyze the security of SplitChain. Specifically we first show that an adversary cannot tamper or predict randomness of block seeds, and then we evaluate the

probability of corruption of a chain's consensus committee as well as the probability of corruption of a chain's routing core. For the purpose of SplitChain, we can assume that at any time the total number of validators is arbitrarily large.

3.2.7.1 Randomness creation

The security analysis of SplitChain relies on the assumption that for any chain C of SplitChain, and for any instantiation $k \geq 1$ of the merge consensus, the seeds of the chaining blocks preceding the k-th one of C have been generated in an unbiased and unpredictable way. Theorem 3.2.1 is essential to prevent the adaptive adversary from manipulating the outcome of the cryptographic sortition lotery.

Theorem 3.2.1. *Let $\epsilon \in (0, 1)$ be the security parameter of SplitChain. The seed of any chaining block cannot be tampered with or predicted in advance by the adversary with any high probability $1 - \epsilon$.*

Proof. We assume the existence of a preceding chaining block $C[k-1]$ with a proper pseudorandom seed s_0. This block may be the genesis block of SplitChain if k is the first consensus of the system. Similarly, if the system consists of multiple chains, we suppose that the seeds of the preceding blocks of these chains are also pseudorandom and unpredictible.

The attribution of the new validators of the consensus committee of $C[k]$ is based on the last seeds of its reference chains. As these seeds are both pseudorandom and unpredictable, the result of the attribution of the new validators of $C[k]$ using these seeds is also pseudorandom and unpredictable. As the adversary needs δ blocks of delay to corrupt validators, it is not possible for it to target the validators of the reference committees quickly enough to influence the randomness of the attribution. Furthermore, in the case of partial attribution, the adversary cannot predict which of the system's chains will serve as the reference chains of the consensus committee of $C[k]$, as they are also randomly chosen using the seed of $C[k-1]$. Thus, if the preceding seeds are unpredictable, the adversary cannot temper the randomness of the attribution of new validators for $C[k]$'s consensus committee.

Furthermore, the verifiable random function (VRF) used in the pseudorandom sortition algorithm (see Section 3.2.5.5) run by new validators of the consensus committee to determine if they are part of the core uses the seed of $C[k-1]$ and the private key of the validators. As the seed used by the sortition function is both pseudorandom and

unpredictable, and the core contains at least one honest validator with high probability $1 - \epsilon$, the new seed of $C[k]$ obtained by hashing the concatenation of the core validator's public keys is both pseudorandom and unpredictable. Thus, the adversary cannot predict or manipulate the value of the new seed in advance. □ □

3.2.7.2 Variation in size of reference sets

Lemma 3.2.1 shows how fairly the attribution algorithm of SplitChain (i.e., Algorithm 2) distributes validators between the different consensus committees.

Lemma 3.2.1. *Let $E_1, E_2, ..., E_N$ be the reference sets produced by Algorithm 2. Then $\forall j, k \in \{1, ..., N\}$, $| \, |E_j| - |E_k| \, | \leq 1$.*

Proof. Let N and L be respectively the number of reference sets to be produced and the set of validators to be attributed. The algorithm first assigns $\alpha = \lfloor |L|/N \rfloor$ validators to each set. This makes $r = |L| - \alpha N < N$ validators to be attributed. Then for the r remaining validators, the algorithm assigns each of them to a different set. This results in $0 \leq r < N$ sets of $\alpha + 1$ validators and $N - r$ sets of α validators. □ □

3.2.7.3 Safety of consensus committees

To ensure that the (μ, δ)-adversary cannot take over the consensus execution of any consensus chain C, the consensus committee of C must contain less than a third of corrupted validators. We examine the probability of corruption of the consensus committee of an arbitrary chain C for both the total attribution policy and the partial one. Recall that μ represents the proportion of malicious validators in SplitChain and N represents the current number of chains of SplitChain.

Let $A = \mu V$ be the total number of corrupted validators in SplitChain. Let A_i be the number of corrupted validators in the reference set of C_i, $1 \leq i \leq N$. We have $A = \sum_{i=1}^{N} A_i$. Recall that the attribution of any validator lasts for $\delta - 1$ consecutive consensus. Let A_i^k be the random variable that represents the maximal number of malicious validators attributed by the $(\ell + k)$-th instance of Algorithm 2 executed in C_i, for any $\ell \geq 1$ and $k \in [1, \delta - 1]$. We have $A_i = \sum_{k=1}^{\delta-1} A_i^k$.

Each reference chain C_i, $1 \leq i \leq N$, sends a new reference set of validators to a chain C_j to update the consensus committee membership of C_j. Let $X_{i,j}$ be the random variable that represents the number of corrupted validators assigned by C_i to the consensus committee of C_j. We have $0 \leq X_{i,j} \leq A_i$. Specifically, the number of new corrupted

validators $X_{i,j}^k$ attributed by C_i to the $(\ell' + k)$-th consensus committee of C_j, for any $\ell' \geq 1$ and $k \in [1, \delta - 1]$, is less than or equal to A_i^k and we have $X_{i,j} = \sum_{k=1}^{\delta-1} X_{i,j}^k$.

Let Y_j be the random variable that represents the number of corrupted validators in the consensus committee of a chain C_j. We have that Y_j is the sum of the corrupted validators of all the reference sets attributed to C_j by its reference chains.

Lemma 3.2.1. $X_{i,j}^k$ follows a hypergeometric distribution with parameters $V/(N(\delta-1))$, $V/((\delta-1)N^2)$ and A_i^k.

Proof. The reference set attributed by the $(\ell + k)$-th instance of Algorithm 2 in C_i to the consensus committee of C_j, for any $\ell \geq 1$ and $k \in [1, \delta - 1]$, is a dichotomous population of size $V/(N(\delta-1))$ composed of A_i^k corrupted validators and $(V/(N(\delta - 1))) - A_i^k$ honest validators. Each validator in this population can only be allocated to one consensus committee, hence the sampling is drawn without replacement. Finally, validators are evenly allocated between the N consensus chains, so each attribution subset contains $V/((\delta-1)N^2)$ validators. Hence, $X_{i,j}^k \sim H(V/(N(\delta-1)), V/((\delta-1)N^2), A_i^k)$. □ □

3.2.7.4 Total attribution

As long as the proportion μ of corrupted validators in SplitChain is less than 1/3, Lemma 3.2.2 shows that the (μ, δ)-adaptive adversary must distribute its corrupted validators evenly in the reference sets.

Lemma 3.2.2. *The probability of corruption of a consensus committee by the (μ, δ)-adaptive adversary is maximized when corrupted validators are equally distributed among all reference chains, i.e., $\forall i$, $\forall k$, $A_i^k = A/(N(\delta-1))$.*

Proof. By definition of Y_j and by applying lemma 3.2.1, we have

$$Y_j = \sum_{i=1}^{N} X_{i,j} = \sum_{i=1}^{N} \sum_{k=1}^{(\delta-1)} H\left(\frac{V}{N(\delta-1)}, \frac{V}{(\delta-1)N^2}, A_i^k\right), \tag{3.1}$$

and the expectation of Y_j is given by

$$E(Y_j) = \sum_{i=1}^{N} \sum_{k=1}^{(\delta-1)} \left(\frac{V}{(\delta-1)N^2}\right)\left(\frac{(\delta-1)NA_i^k}{V}\right) = \sum_{i=1}^{N} \sum_{k=1}^{(\delta-1)} \frac{A_i^k}{N} = \frac{A}{N}. \tag{3.2}$$

The average proportion of the (μ, δ)-adaptive adversary in a consensus committee is therefore $(A/N)/(V/N) = A/V = \mu$. Note that this value is independent from A_i^k. Therefore, the adversary can only maximize its chances of corrupting the consensus committee of C_j by increasing the variance of Y_j. The variables $X_{i,j}^k$ are drawn from disjoint populations, and are therefore independent. Thus, the variance of Y_j is given by summing the variances of random variables $X_{i,j}^k$:

$$\begin{aligned}
\mathrm{Var}(Y_j) &= \sum_{i=1}^{N} \sum_{k=1}^{(\delta-1)} \frac{V}{(\delta-1)N^2} \frac{(\delta-1)NA_i^k}{V} \left(1 - \frac{(\delta-1)NA_i^k}{V}\right) \frac{\frac{V}{N(\delta-1)} - \frac{V}{(\delta-1)N^2}}{\frac{V}{N(\delta-1)} - 1} \\
&= \sum_{i=1}^{N} \sum_{k=1}^{(\delta-1)} \frac{A_i^k}{N} \left(1 - \frac{(\delta-1)NA_i^k}{V}\right) \frac{\frac{V(N-1)}{(\delta-1)N^2}}{\frac{V}{N(\delta-1)} - 1} \\
&= \sum_{i=1}^{N} \sum_{k=1}^{(\delta-1)} \left(\frac{A_i^k}{N} - \frac{(\delta-1)A_i^{k\,2}}{V}\right) \frac{V\,(N-1)}{N(\delta-1)\,(V-N)} \\
&= \frac{V\,(N-1)}{N(\delta-1)\,(V-N)} \left(\frac{1}{N} \sum_{i=1}^{N} \sum_{k=1}^{(\delta-1)} A_i^k - \frac{(\delta-1)}{V} \sum_{i=1}^{N} \sum_{k=1}^{(\delta-1)} A_i^{k\,2}\right) \\
&= \frac{V\,(N-1)}{N(\delta-1)\,(V-N)} \left(\frac{A}{N} - \frac{(\delta-1)}{V} \sum_{i=1}^{N} \sum_{k=1}^{(\delta-1)} A_i^{k\,2}\right). \qquad (3.3)
\end{aligned}$$

To maximize $\mathrm{Var}(Y_j)$, the (μ, δ)-adversary must choose the A_i^k so that to minimize $\sum_{i=1}^{N} A_i^{k\,2}$, with $0 \leq A_i^k \leq V/(N(\delta-1))$ and $\sum_{i=1}^{N} \sum_{k=1}^{(\delta-1)} A_i^k = A$. Intuitively, $\sum_{i=1}^{N} \sum_{k=1}^{(\delta-1)} A_i^{k\,2}$ is minimal when $A_i^k = A/(N(\delta-1))$. We formulate this hypothesis by the following inequality, where v_i^k represents the deviation of A_i^k from $A/(N(\delta-1))$, such that

$-A/(N(\delta-1)) \leq v_i^k \leq V/(N(\delta-1)) - A/(N(\delta-1))$ and $\sum_{i=1}^{N}\sum_{k=1}^{(\delta-1)} v_i^k = 0$. We have:

$$\begin{aligned}
&\sum_{i=1}^{N}\sum_{k=1}^{(\delta-1)} A_i^{k2} \leq \sum_{i=1}^{N}\sum_{k=1}^{(\delta-1)} (\frac{A}{N(\delta-1)})^2 \\
&< \sum_{i=1}^{N} (\frac{A}{N(\delta-1)} + v_i^1)^2 + (\frac{A}{N(\delta-1)} + v_i^2)^2 + ... + (\frac{A}{N(\delta-1)} + v_i^{(\delta-1)})^2 \\
&N(\delta-1)(\frac{A}{N(\delta-1)})^2 < \sum_{i=1}^{N}(\delta-1)(\frac{A}{N(\delta-1)})^2 + \sum_{k=1}^{(\delta-1)} v_i^{k2} + 2\frac{A}{N(\delta-1)}\sum_{k=1}^{(\delta-1)} v_i^k \\
&\frac{A^2}{N(\delta-1)} < \frac{A^2}{N(\delta-1)} + \sum_{i=1}^{N}\sum_{k=1}^{(\delta-1)} v_i^{k2} \\
&0 < \sum_{i=1}^{N}\sum_{k=1}^{(\delta-1)} v_i^{k2}
\end{aligned} \tag{3.4}$$

We notice that inequality 3.4 stands when any deviation v_i^k is different from 0, that is, any other distribution than $A_i^k = A/(N(\delta-1))$ results in a greater value of the sum and reduces the variance of Y_j. Therefore, to maximize the variance of Y_j, i.e., to maximize the corruption probability of the consensus committee of C_j, malicious validators must be evenly distributed across the k executions of Algorithm 2. □ □

Lemma 3.2.2. The probability p_c that a consensus committee of chain C_j is corrupted by the adversary is given by

$$p_c = P\left(Y_j > \lfloor\frac{c}{3}\rfloor\right) = 1 - \sum_{l=0}^{\lfloor\frac{c}{3}\rfloor}\binom{c}{l}(\mu)^l(1-\mu)^{c-l} \tag{3.5}$$

Proof. A consensus committee is corrupted if the sum of the adversary's validators is greater than one third of the chain. By applying lemma 3.2.2 to equation 3.1, we can express the probability of corruption p_c of a consensus committee as:

$$\begin{aligned}
p_c &= P\left(Y_j > \lfloor\frac{V}{3N}\rfloor\right) \\
&= 1 - P\left(\sum_{i=1}^{N}\sum_{k=1}^{(\delta-1)} H\left(\frac{V}{N(\delta-1)}, \frac{V}{(\delta-1)N^2}, \frac{A}{N(\delta-1)}\right)\right) \\
&\leq \left\lfloor\frac{V}{3N}\right\rfloor
\end{aligned} \tag{3.6}$$

The binomial distribution is a good approximation of the hypergeometric distribution when the population is sufficiently larger than the sample size, which is commonly accepted as a sampling fraction of 0.1 [Mon09]. That is, $H(m,n,p) \approx B(n,p/m)$ when $n/m < 0.1$. In our case, $(V/((\delta-1)N^2))/(V/(N(\delta-1))) < 0.1$ imposes that $N > 10$. Thus, we assume that the number of chains N is always greater than 10. By applying this approximation to equation 3.6, we can simplify our equation:

$$\begin{aligned}
P(Y_j \leq \lfloor\frac{V}{3N}\rfloor) &= \lim_{V\to\infty} P(\sum_{i=1}^{N}\sum_{k=1}^{(\delta-1)} H(\frac{V}{N(\delta-1)}, \frac{V}{(\delta-1)N^2}, \frac{A}{N(\delta-1)}) \leq \lfloor\frac{V}{3N}\rfloor) \\
&= P(\sum_{i=1}^{N}\sum_{k=1}^{(\delta-1)} B(\frac{V}{(\delta-1)N^2}, \frac{A}{V}) \leq \lfloor\frac{V}{3N}\rfloor) \\
&= P(B(\sum_{i=1}^{N}\sum_{k=1}^{(\delta-1)} \frac{V}{(\delta-1)N^2}, \mu) \leq \lfloor\frac{V}{3N}\rfloor) \\
&= P(B(\frac{V}{N}, \mu) \leq \lfloor\frac{V}{3N}\rfloor) \\
&= \sum_{k=0}^{\lfloor\frac{V}{3N}\rfloor} \binom{\frac{V}{N}}{k} (\mu)^k (1-\mu)^{\frac{V}{N}-k}
\end{aligned} \tag{3.7}$$

Let c be the constant representing the minimal number of validators needed to ensure the proper and safe execution of a chain. By replacing V/N with c in equation 3.7, we can calculate the probability of corruption of a chain by the adversary:

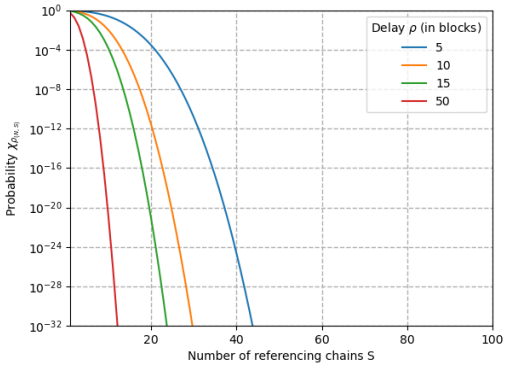
$$P(Y_j > \lfloor\frac{c}{3}\rfloor) = 1 - \sum_{k=0}^{\lfloor\frac{c}{3}\rfloor} \binom{c}{k} (\mu)^k (1-\mu)^{c-k}$$

□ □

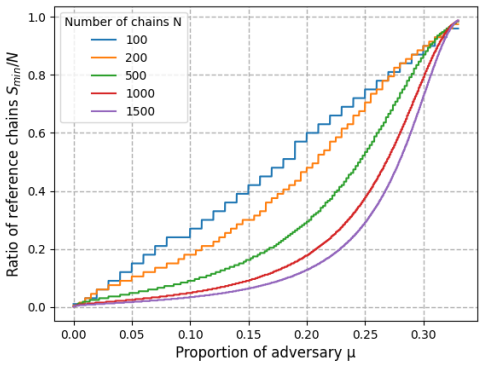
Theorem 3.2.2 provides the probability p_c of corruption of a consensus committee when the total attribution policy is applied.

Theorem 3.2.2. *For any security parameter $\epsilon \in (0,1)$, for any proportion of corrupted validators $\mu < 1/3$, there exists a finite consensus committee size c_{min} such that $\forall c \geq c_{min}$, the probability p_c that a committee of c validators is corrupted satisfies $p_c < \epsilon$, where $p_c = 1 - \sum_{\ell=0}^{\lfloor c/3\rfloor} \binom{c}{\ell} \mu^\ell (1-\mu)^{c-\ell}$.*

Proof. Probability p_c is given by lemma 3.2.2. Let $c = V/N$ be the number of validators inside a consensus committee, and $\mu = A/V$ the proportion of corrupted validators in



(a) Probability $\chi_{\rho_{(N,S)}}$ as a function of the number of reference chains S and the delay in blocks ρ for $N = 100$ chains.



(b) Required proportion of reference chains to achieve partial attribution with probability $1 - 10^{-6}$ as a function of μ for different values of N.

Figure 3.4

SplitChain, with $\mu < 1/3$. We know from Equation 3.2 that the average number $E(Y_j)$ of corrupted validators in a consensus committee is $A/N = \mu c$. We also know from lemma 3.2.2 that $p_c = 1 - P(Y_j \leq \lfloor c/3 \rfloor)$, with $Y_j \sim BinomialDistribution(c, \mu)$.

Let $\mu = 1/3$. Then $E(Y_j) = c/3$. By definition of the statistical mean, we have $\lim_{c\to\infty} P(Y_j \leq E(Y_j)) = 0.5$. Thus, when $\mu = 1/3$, $\lim_{c\to\infty} p_c = 0.5$. However, SplitChain strictly requires that $\mu < 1/3$. As $E(Y_j) < c/3$ when $\mu < 1/3$, $\lim_{c\to\infty} P(Y_j \leq \lfloor c/3 \rfloor) = 1$ and $\lim_{c\to\infty} p_c = 0$. By definition, the cumulative distribution function is monotone increasing, so p_c is monotone decreasing. Thus, there exists a value c_{min}, such that for any $\epsilon < 1$ and $c \geq c_{min}$, $p_c < \epsilon$. □ □

Figure 3.5a illustrates Theorem 3.2.2 for different values of μ and ϵ.

3.2.7.5 Partial attribution

We prove that the partial attribution is secure in presence of a $(\mu, \delta + \rho)$-adversary. Let S represent the number of randomly selected reference chains among the N chains of SplitChain. Theorem 3.2.3 gives the probability of corruption p_S of a partially attributed consensus committee. It provides a lower bound S_{min} on the number of reference chains S such that a consensus committee built with the partial attribution policy using $S \geq S_{min}$ reference chains has probability less than ϵ to be corrupted.

Let S be the number of reference sets required for a chain C_j to update its $(\ell + k)$-

th consensus committee before initiating a new consensus execution. Let $R_a^k \geq N$ be the number of referencing sets corrupted by the adversary among the N reference sets proposed to C_j by the chains of SplitChain at some attribution execution $(\ell' + k)$. The number of corrupted reference attributed to the $(\ell + k)$-th consensus committee of C_j is the random variable $S_a^k \leq S$.

Lemma 3.2.3. The number of corrupted validators Y_j is maximized when the corrupted validators are equally distributed among the R_a reference chains, i.e. $\forall C_i \in R_a, A_i = \mu' V/N$, where μ' is the new proportion of corrupted validator inside the S_a corrupted reference chains, such that $\mu' \geq \mu$.

Proof. This result is obtained by applying lemma 3.2.2 with S referencing chains instead of N and μ' instead of μ. □ □

Lemma 3.2.4. S_a follows the hypergeometric distribution with parameters $H(N, R_a, S)$.

Proof. The reference sets correspond to a dichotomous population of size N composed of R_a corrupted sets and $N - R_a$ honest sets. Each set in this population can only be allocated once to the same consensus committee, hence the sampling is drawn without replacement. Finally, only S sets are chosen at random, thus the number of randomly selected corrupted sets S_a follows the hypergeometric distribution $H(N, R_a, S)$. □ □

Lemma 3.2.5. The proportion of malicious validators inside a consensus committee $S_a \times (\mu'/S)$ is maximized when the reference chains targeted by the adversary are fully corrupted, i.e. when $\mu' = 1$.

Proof. The average value of S_a is: $E(S_a) = S\frac{R_a}{N} = S\frac{(\mu/\mu')N}{N} = S\frac{\mu}{\mu'}$. This corresponds to an average adversary proportion of $E(S_a)\frac{\mu'}{S} = \mu$ corrupted reference validators. Note that this mean value doesn't depend on μ'.

The adversary proportion's variance is:

$$\begin{aligned}
V_a(S_a\frac{\mu'}{S}) &= \frac{\mu'^2}{S^2} S\frac{R_a}{N}\frac{N-R_a}{N}\frac{N-S}{N-1} \\
&= \frac{\mu'^2}{S}\frac{\frac{\mu}{\mu'}N}{N}\frac{N-\frac{\mu}{\mu'}N}{N}\frac{N-S}{N-1} \\
&= \frac{\mu'^2}{S}\frac{\mu}{\mu'}(1-\frac{\mu}{\mu'})\frac{N-S}{N-1} \\
&= \frac{\mu}{S}(\mu'-\mu)\frac{N-S}{N-1}
\end{aligned}$$

The derivative function of the adversary proportion's variance is $\frac{\partial}{\partial \mu'} V_a(S_a \frac{\mu'}{S}) = \frac{\mu}{S} \frac{N-S}{N-1}$, whose roots are $N = S$ and $\mu = 0$. If $S = N$, then all the available reference validators are used and the proportion of corrupted reference validators is μ. For $\mu > 0$, $\mu' \leq 1$ and $0 < S < N$, the derivative is always positive. Hence, the adversary proportion's variance is ascending for $\mu \leq \mu' \leq 1$ and reaches its maximum when $\mu' = 1$. □ □

Lemma 3.2.6. (Probability of corruption of a partially attributed consensus committee).

$$p_c = P(Y_j > \lfloor c/3 \rfloor) = P(S_a > \lfloor S/3 \rfloor) = 1 - \sum_{k=0}^{\lfloor S/3 \rfloor)} \binom{S}{k} (\mu)^k (1-\mu)^{n-k} \tag{3.8}$$

Proof. A consensus committee is corrupted if it contains more than one third of malicious validators. As the committee is composed of S reference sets, this corresponds to $S(V/3N^2)$ validators. By definition of Y_j and by applying lemma 3.2.3, we have:

$$p_c = P(\sum_{i=1}^{S_a} H(\frac{V}{N}, \frac{V}{N^2}, \frac{V}{N}\mu') > \lfloor S\frac{V}{3N^2} \rfloor) \tag{3.9}$$

Let $\lim_{V,N\to\infty} \frac{V}{N^2} = c$, c being the fixed number of validators within an attribution subset.

$$\begin{aligned}
\lim_{V,N\to\infty} P(Y_j > \lfloor S\frac{V}{3N^2} \rfloor) &= \lim_{V,N\to\infty} P(\sum_{i=1}^{S_a} H(\frac{V}{N}, \frac{V}{N^2}, \frac{V}{N}\mu') > \lfloor S\frac{V}{3N^2} \rfloor) \\
&= \lim_{V,N\to\infty} P(\sum_{i=1}^{S_a} H(\frac{V}{N}, c, \frac{V}{N}\mu') > \lfloor S\frac{c}{3} \rfloor) \\
&= P(\sum_{i=1}^{S_a} B(c, \mu') > \lfloor S\frac{c}{3} \rfloor) \\
&= P(B(S_a c, \mu') > \lfloor S\frac{c}{3} \rfloor)
\end{aligned}$$

We know from lemma 3.2.5 that this value is maximized when $\mu' = 1$, thus we obtain $p_c = P(S_a c > \lfloor S(c/3) \rfloor) = P(S_a > \lfloor S/3 \rfloor)$. As a reference set contains at least one validator, i.e. $c \geq 1$, the minimum number of consensus validators of a chain is S. From

lemma 3.2.4, we know that $S_a \sim H(N, R_a, S)$. As $\mu' = 1$, $R_a = \mu N$. Thus:

$$\begin{aligned} p_c &= \lim_{V,N\to\infty} P(S_a > \lfloor S/3 \rfloor) \\ &= \lim_{N\to\infty} P(H(N, \mu N, S) > \lfloor S/3 \rfloor) \\ &= P(B(S, \mu) > \lfloor S/3 \rfloor) \\ &= 1 - \sum_{\lfloor S/3 \rfloor)}^{k=0} \binom{S}{k} (\mu)^k (1-\mu)^{S-k} \end{aligned}$$

□ □

Theorem 3.2.3. *For any safety parameter $\epsilon \in (0, 1)$, for any proportion of corrupted validators $\mu < 1/3$, there exists a finite number of reference sets S_{min} such that $\forall S \geq S_{min}$, the probability p_S that a consensus committee composed of S reference sets is corrupted satisfies $p_S < \epsilon$, with $p_S = 1 - \sum_{k=0}^{\lfloor S/3 \rfloor)} \binom{S}{k} \mu^k (1-\mu)^{S-k}$.*

Proof. We know from lemma 3.2.6 that the number of fully corrupted reference sets in a committee is $p_c = P(S_a > \lfloor S/3 \rfloor) = 1 - P(S_a \leq \lfloor S/3 \rfloor)$, with $S_a \sim B(S, \mu)$. The average number of fully corrupted reference sets in a consensus committee is $E(S_a) = S\mu$.

Let $\mu = 1/3$. Then $E(S_a) = S/3$. By definition of the statistical mean, we have $\lim_{S\to\infty} P(S_a \leq E(S_a)) = 0.5$. Thus, when $\mu = 1/3$, $\lim_{S\to\infty} p_c = 0.5$. However, SplitChain strictly requires that $\mu < 1/3$. As $E(S_a) < S/3$ when $\mu < 1/3$, $\lim_{S\to\infty} P(S_a \leq \lfloor S/3 \rfloor) = 1$ and $\lim_{S\to\infty} p_c = 0$. Because the cumulative distribution function is by definition monotone increasing, p_c is monotone decreasing. Thus, there exists a value S_{min} such that for any $\epsilon > 0$ and $S > S_{min}$, $pc < \epsilon$. □ □

Figure 3.4b displays the value of S_{min}/N for $\epsilon = 10^{-6}$ as a function of the adversary proportion μ in SplitChain and for different values of N. The value of S_{min} decreases when the number N of chains of the system increases. However, when the adversary proportion μ becomes very close to 1/3, S_{min}/N tends to 1 and total attribution is needed. Our experiments run on Ethereum show that both policies can be safely applied on Ethereum (see Section 3.2.9).

3.2.7.6 Resistance against an adaptive adversary

The objective of this section is to analyze whether the adversary can benefit from the partial attribution policy to corrupt a targeted consensus committee via the manipulation

of a given reference set S_{j*} of a chain C_{j*}. Specifically, let $\mathcal{C}(t)$ be the consensus committee of chain C at time t, and $\mathcal{B}(t) = \{S_1, \ldots, S_{j*}, \ldots, S_N\}$ be the set of the last reference sets sent by the N chains of SplitChain that have been received by $\mathcal{C}(t)$. From the partial attribution policy (see Section 3.2.5.6), $\mathcal{C}(t)$ randomly selects S sets from $\mathcal{B}(t)$ to determine the composition of the next consensus committee $\mathcal{C}(t+1)$ at time $t+1$. Consensus committee $\mathcal{C}(t+1)$ on its turn will randomly select S sets from $\mathcal{B}(t+1) = \{S_1^{(1)}, \ldots, S_{j*}, \ldots, S_N^{(1)}\}$, where $S_i^{(1)}$ represents the last reference set received by $\mathcal{C}(t+1)$ and sent by C_i. Note that, by the asynchrony of the system, all communications from the chains may be arbitrarily long, delaying accordingly the receipt of reference sets, and in particular the new reference set of C_{j*}. Thus $\mathcal{C}(t+1)$ may still consider the obsolete reference set S_{j*} as the latest reference set of C_{j*}. Let $\rho_{(N,S)}$ be the upper bound on the difference between the chaining block index of the obsolete reference set S_{j*} and the latest chaining block index of C_{j*}. Consensus committee $\mathcal{C}(t+\rho_{(N,S)})$ will randomly select S reference sets from $\mathcal{B}(t+\rho_{(N,S)}) = \{S_1^{(\rho_{(N,S)})}, \ldots, S_{j*}, \ldots, S_N^{(\rho_{(N,S)})}\}$. Suppose that reference set S_{j*} is never selected during the first $\rho_{(N,S)}-1$ random selections. Recall that for the partial attribution policy (see Section 3.2.5.6), we assume a $(\mu, \delta + \rho_{(N,S)})$-adaptive adversary. Were reference set S_{j*} selected after $\rho_{(N,S)}$ validator attributions, the adversary could already have corrupted the validators of S_{j*}. Let $\chi_{\rho_{(N,S)}}$ be the probability that a consensus committee uses a reference set that has been obsolete for at least $\rho_{(N,S)}$ consecutive validator attributions. Theorem 3.2.4 shows that $\chi_{\rho_{(N,S)}}$ is less than the security parameter ϵ. Figure 3.4a illustrates that $\rho_{(100,24)} = 5$.

Lemma 3.2.3. *Let $\chi_{\rho_{(N,S)}}$ be the probability that a consensus committee uses a reference set that has been obsolete for at least $\rho_{(N,S)}$ consecutive validator attributions. Then $\chi_{\rho_{(N,S)}} \leq P(H(N,S,1)=0)\Big(P(H(N,S,1)=0)p_{2,S} + P(H(N,S,1)=1)p_{2,S-1}\Big)$, where H is the hypergeometric distribution and*

$$p_{i,k} = \begin{cases} 1, & \text{if } i > \rho_{(N,S)} \\ 0, & \text{if } k \geq N \\ P(H(N,S,1)=0)\sum_{j=0}^{min(S,k)}\Big(P(H(N,S,k)=j) & \\ (P(H(N,S,1)=0))^j p_{i+1,k+S-j}\Big) & \end{cases}$$

Proof. We first assume that all chains are up to date, i.e. at time t, each chain knows the penultimate block $t-1$ of all other chains. Because of the asynchrony, we adopt the

worst-case scenario and assume that the index of chain C_1 is only updated by a chain C_2 if C_1 is selected for the partial attribution of C_2, i.e. a new chaining block of C_1 is required to initiate the consensus of C_2.

Let χ_1 be the probability that C does not draw C'. Since all chains are up to date, none of the drawn chains contain block indices unknown to C, therefore $\chi_1 = P(H(N,S,1) = 0$. Given that C can draw itself, the number of new chain block indices is equal to S with probability $P(H(N,S,1) = 0)$ or equal to $S-1$ with probability $P(H(N,S,1) = 1)$. If a chain whose block index is greater than $t-1$ is selected, it may contain a new block index of C' or any other new chain block indices greater than $t-1$. We approximate this probability by considering only the probability that the chain drew C' during its last draw, i.e. $P(H(N,S,1) = 0)$. Thus, we obtain:

$$\chi_2 \leq \chi_1 \Bigg(P(H(N,S,1)=0)\Big(\sum_{j=0}^{S} P(H(N,S,S)=j)(P(H(N,S,1)=0))^j\Big) + P(H(N,S,1)=1)\Big(\sum_{j=0}^{S-1} P(H(N,S,S-1)=j)(P(H(N,S,1)=0))^j\Big)\Bigg)$$

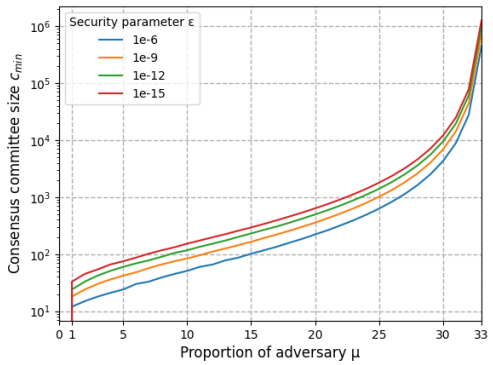
Let $p_{2,k} = P(H(N,S,1) = 0)\sum_{j=0}^{min(S,k)} P(H(N,S,k) = j)(P(H(N,S,1) = 0))^j$. Then $\chi_2 \leq \chi_1\Big(P(H(N,S,1)=0)p_{2,S} + P(H(N,S,1)=1)p_{2,S-1}\Big)$. Taking into account the case where $k \geq N$, i.e. all chains have been drawn at least once (so the probability that C' has not been drawn is 0), we can apply the formula of $p_{i,k}$ iteratively until $i = \rho_{(N,S)}$. □ □

Theorem 3.2.4. *$\forall N, \forall S \leq N$, $\exists \rho(_{(N,S)} > 0$ such that $\chi_{\rho_{(N,S)}} < \epsilon \in (0,1)$.*

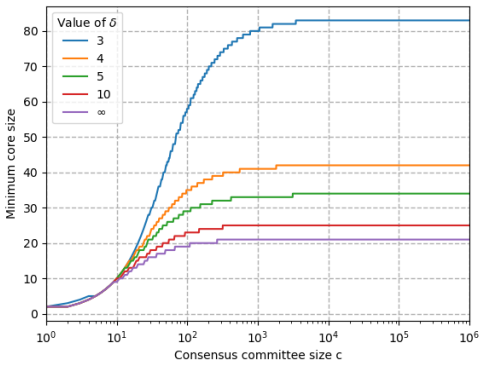
Proof. Direct from Lemma 3.2.3 □ □

3.2.7.7 Safety of the election of core validators

Validators of the core of a chain are crucial for intra- and cross-chain communications. There must always be at least one honest core validator to ensure the routing of messages in a chain. In the following, T represents the lifespan of consensus credentials, $\mu' = \mu(T)/(T-1)$ refers to the maximum proportion of corrupted validators eligible for the core and c is the number of validators of any consensus committee. We use $1 \leq r \leq T$ to designate the T last consensus of a chain and refer to the number of validators that joined the consensus committee during consensus r as c_r. Theorem 3.2.5 shows that there exists



(a) Minimum number of validators c_{min} of a consensus committee as a function of μ for different safety probabilities $1 - \epsilon$.



(b) Minimum core size τ_{min} as a function of the consensus committee size c for different values of adversary adaptivity δ

Figure 3.5

a finite core size above which the probability p_{core} that the core is corrupted is smaller than ϵ, for any $\epsilon \in (0, 1)$.

Definition 3.2.11. *The core of a chain is composed of the new core validators of the last $\delta - 2$ consensus. Let $X = X^A + X^H$ be the number of validators in the core, where X^A is the number of malicious core validators and X^H the number of honest core validators. We respectively denote X_r^A and x_r^H the number of corrupt and honest new core validators at the r-th last consensus, with $X_r = X_r^A + X_r^H$.*

Lemma 3.2.7. The probability that the core of a chain is corrupted is given by

$$p_{core} = (1 - \tau/c)^{(1-\mu')(c-c_{(\delta-1)})} \tag{3.10}$$

Proof. New core validators are elected by sortition during each consensus. A validator can only be elected as a core validator during its first consensus. Furthermore, the list of new core validators is only determined at the end of the consensus execution. Thus, core validators can only join the core for $\delta - 2$ consensus executions. The probability of a validator u being elected as a core validator is $B(w_u, \tau/w)$[Sau+20], where B is the binomial distribution, w is the total validation stake of the consensus committee and w_u is the stake of validator u. In SplitChain, all validators have the same validation stake,

thus $w_u = 1$ and:

$$\begin{aligned} X &\sim \sum_{(\delta-2)}^{r=1} \sum_{c_r}^{i=1} B(1, \tau/c) \\ &\sim B(\sum_{(\delta-2)}^{r=1} c_r, \tau/c) \\ &\sim B(c - c_{(\delta-1)}, \tau/c) \end{aligned} \tag{3.11}$$

We consider the worst case where the adversary concentrates the corrupted validators of the consensus committee among the new validators of the last $\delta-2$ consensus. Following the same logic as with equation 3.11, with a proportion of $\mu' = \mu(\delta-1)/((\delta-2))$ corrupted validators, we obtain $X^H \sim B((1-\mu')(c-c_{(\delta-1)}), \tau/c)$. Thus, the probability of corruption p_{core} of the core is:

$$\begin{aligned} p_{core} &= P(B((1-\mu')(c - c_{(\delta-1)}), \tau/c) = 0) \\ &= \binom{(1-\mu')(c - c_{(\delta-1)})}{0} (\tau/c)^0 (1-\tau/c)^{(1-\mu')(c-c_{(\delta-1)})-0} \\ &= (1-\tau/c)^{(1-\mu')(c-c_{(\delta-1)})} \end{aligned}$$

□ □

Theorem 3.2.5. *For any safety parameter $\epsilon \in (0,1)$, for an adaptive adversary with $\delta \geq 3$, there exists a finite core size τ_{min} such that $\forall \tau \geq \tau_{min}$, the probability p_{core} that a core of τ validators is corrupted satisfies $p_{core} < \epsilon$, with $p_{core} = (1-\tau/c)^{(1-\mu')(c-c_T)}$.*

Proof. We know from Lemma 3.2.7 that $X^H \sim B((1-\mu')(c - c_{(\delta-1)}), \tau/c)$, with $\mu' = \mu((\delta-1))/((\delta-2))$. The expected proportion of honest core validators is $E(B((1-\mu')(c - c_{(\delta-1)}), \tau/c)) = \tau(1-\mu') + \tau(c_{(\delta-1)}/c)(\mu'-1)$. As μ' decreases when δ increases, with $\lim_{\delta\to\infty} \mu' = \mu$, $\forall \delta \geq 3$, $\mu < 1/3 \Rightarrow \mu' < 2/3$. If $\mu = 1/3$ and $\delta = 3$, then we have $E(B((1-\mu')(c - c_{(\delta-1)}), \tau/c)) = \tau/6$. As p_{core} decreases with the value of δ and τ, $\exists \tau_{min}$ such that $\forall \delta \geq 3$ and $\forall \tau \geq \tau_{min} > 0$, we have $p_{core} \leq \epsilon$, where $\epsilon < 0.5$ is the security parameter. □ □

Figure 3.5b plots τ_{min} as a function of the consensus committee size c for different values of δ. As observed, τ_{min} quickly stabilizes as c increases. We also notice that τ_{min} decreases very rapidly when δ increases.

3.2.8 Comparison to other sharding solutions

While several sharded blockchains have been proposed in recent years, we focus on previous works that have introduced new concepts that have influenced the design of SplitChain.

Elastico [Luu+16] is the first sharded blockchain. Its validators are divided into shards, each one creating a block of transactions, which are then aggregated into a "global-block" to add to the system's unique blockchain. The shards are renewed after each global-block, ensuring strong safety against adaptive adversaries at the expense of synchronization and storage costs, i.e., validators store the entire system. As a safeguard against Sybil attacks, each validator must solve a PoW puzzle, whose solution also determines which shard it belongs to. Elastico cannot ensure atomicity in cross-shard transactions [Kok+18], leading to permanent fund locking. Its attribution protocol coupled with its small shard size (approx. 100 validators) yields a very high corruption probability of 2.76% [Kok+18] per shard per block. In comparison, SplitChain splits computation and storage between the different chains at the cost of a slower adaptive adversary. SplitChain implements a pruning mechanism for the blocks of the chains and gradually renews the validators of the chains, limiting the bootstrapping overhead of new validators. SplitChain supports cross-chain transactions natively and includes a routing protocol designed for chains to route transactions in a logarithmic number of hops in proportion to the total number of chains in the system. Finally, we do not rely on PoW, which requires a synchronous communication model and whose use is controversial because of its excessive energy consumption.

Omniledger [Kok+18] improves upon Elastico. It uses a more scalable consensus algorithm, increasing the size of its shards, thus reducing the probability of their corruption. It features shards with separate ledgers to better distribute storage costs and adapts classic distributed checkpointing principles to prune shard ledgers. It provides a synchronous lock/unlock client-driven mechanism to handle cross-shard transactions, although at the expense of lightweight-client compatibility and significant latency and safety issues [Son+20]. The UTXO model is not adapted to sharding, as its cross-chain transactions can consume UTXOs stored in different chains, having a significant impact on throughput as the number of shards increases. Omniledger reduces the cost of shard reconfiguration by bounding the number of validators shuffled in each of its day-long epochs, however, Omniledger requires a global blockchain for validator allocation. In contrast, SplitChain shuffles a small number of validators of a chain after each block to withstand an adversary adaptive over a small, configurable number of consensus instances

and does not require any global blockchain to manage validator identities. Validators can handle the management and routing of cross-chain transactions autonomously, preserving lightweight-client compatibility.

Zilliqa [Tea17] is an account-based sharded blockchain that handles smart contracts. It inherits all the problems of Elastico except for its ability to handle validators on a separate chain. It does not shard transactions storage. Moreover, it cannot provide atomicity for cross-shard transactions.

RapidChain [ZMR18] is a UTXO-based sharded blockchain that distinguishes itself from Omniledger by featuring a transaction routing protocol inspired by Kademlia, enabling message routing in log n steps without any special client interaction. To reduce the size of its shards, RapidChain uses a synchronous consensus algorithm tolerating a proportion of 1/2 Byzantine nodes. RapidChain thus inherits problems of Elastico and Omniledger, namely the use of PoW for validator enrollment (Elastico) and the use of the UTXO model (Omniledger). On the other hand, Splitchain is asynchronous, does not require PoW and routes transactions through a small subset of validators (the core) thereby significantly reducing message load.

Monoxide [WW19] is the first sharded blockchain to implement the use of PoW for shard consensus. It allows each miner to finalize and propose new transaction blocks to multiple shards simultaneously, amplifying and distributing their mining capabilities within the system. Its cross-shard transactions are handled in a lock-free manner. However, to guarantee the safety of Monoxide, the majority of miners are required to work for most if not all shards. This causes centralization problems and contradicts the load-distribution principle of sharding, requiring large throughput, storage and computation costs from miners. This behavior causes Monoxide to resemble more to a parallelization solution than to an actual sharded system like SplitChain.

3.2.9 Experiments

To evaluate the efficiency of SplitChain's dynamic sharding compared to static sharding, we replay the entire transaction history of Ethereum from its launch in July 2015 to block $19,353,052$ in March 2024. This amounts to a total of more than 2.280 billion transactions. Blocks and transactions are extracted from a go-ethereum node using ethereum-etl, then stored and ordered in an SQLlite3 database. Our experiment is implemented in C++. Figure 3.6 displays the average number of transactions for each group of consecutive $1K$ blocks.

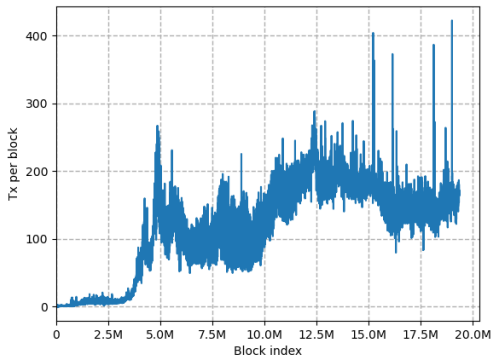
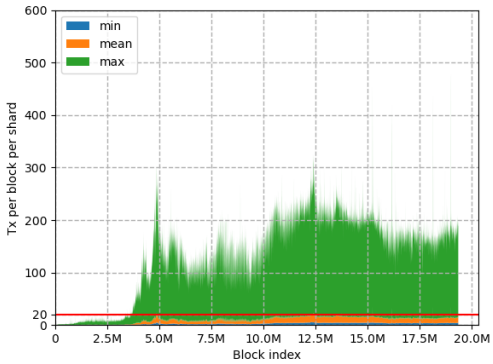


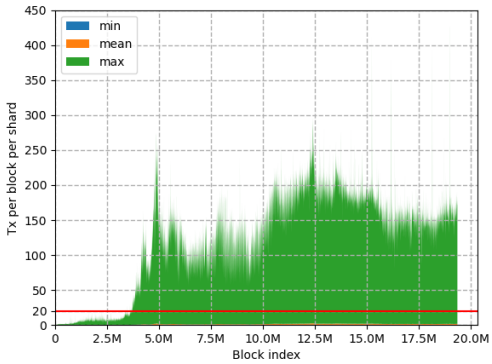
Figure 3.6 – Number of transactions per block (Ethereum)

To assess the efficiency of transaction allocation for static sharding and SplitChain, we evaluate, for each $1K$ block segment, the minimum, average and maximum number of transactions processed by the chains in each of the two models. Figures 3.7a and 3.7b display the distribution of transactions for a static sharding solution of 32 and 256 chains. In both cases, the difference between the average and the maximum is very high: At block 12.5 billion, the maximum is 17 times higher than the average for 32 chains, and 131 times higher for 256 chains. In particular, we notice that the maximum load is almost identical in the two cases, indicating that although multiplying the number of chains by 8 reduces the average load on the chains, the distribution of transactions remains uneven.

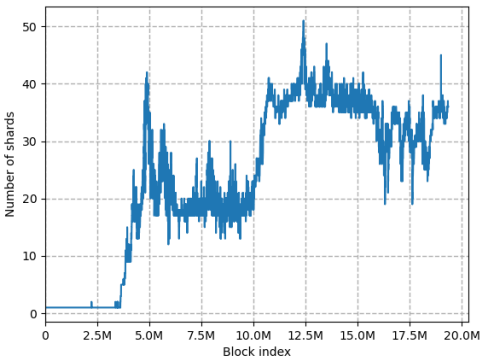
Figure 3.7c shows the variation of the number of chains in SplitChain. We have opted for a split threshold of 20 transactions and a merge threshold of 5 transactions per block. Having a low split threshold reduces computation time and the volume of information transmitted between validators within a chain, resulting in faster consensus and lower transaction-confirmation latency. Chains decide whether to split or merge every $1K$ blocks. We observe a strong correlation between the number of chains and the number of transactions. The number of chains used by SplitChain is also low, reaching 51 at most. Finally, Figure 3.7d shows the distribution of transactions in SplitChain. We notice that once a given total number of transactions has been reached (around 100 transactions per block), the average load quickly stabilizes at around 10 transactions per block, with a maximum load that is only 2.5 times higher at block 12.5 billion. Indeed, splitting chains efficiently spreads the load of the busiest chains, while creating far fewer chains than static sharding for the same label length. We conclude from this experiment that SplitChain's dynamic sharding better distributes transactions across different chains than static sharding.



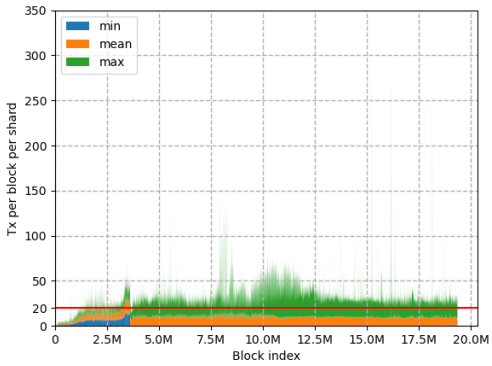
(a) Number of transactions per block per chain, 32-chains static sharding



(b) Number of transactions per block per chain, 256-chains static sharding



(c) Number of chains per 1K blocks segment, SplitChain



(d) Number of transactions per block per chain, SplitChain

Figure 3.7 – Efficiency of transaction distribution between chains.

3.2.9.1 Hotspot addresses

Ethereum's history includes hotspot accounts, i.e. accounts producing a very large number of transactions during one or several $1,000$-block segments. Given that a user account cannot be spread over multiple chains, a hotspot address causes repeated splits in the chain handling it, resulting in the creation of a multitude of unused chains that cannot be merged. The following method is used to detect hotspots and cancel a split operation.

Let m be our hotspot-address detection parameter. Let us divide the account space of a chain C into 2^m subsets. Each subset corresponds to the account space of a descendant chain of C after m divisions. We say that C contains a hotspot if one of the subset of

C registers more transactions than the chain splitting threshold T_{split}. If C contains a hotspot, it does not split even if the threshold T_{split} is exceeded. For our experiment, we chose $m = 10$, which corresponds to 1024 subsets.

3.2.10 Limitations

Sharding provides an effective way to address the challenges associated with storage in a large-scale distributed system such as the blockchain. But even in sharded blockchains, the data within each shard is still fully replicated to provide adequate fault tolerance. While the dataset of a shard is smaller than that of a full blockchain and does not consume as much storage resources, it still unnecessarily exposes potentially sensitive data to a large number of involved parties. The storage of data within each shard could still be improved by introducing more frugality and privacy, for example by using the techniques presented in Chapter 4.

3.3 Asynchronous, consensus-free BFT asset transfer with light storage

3.3.1 Introduction

This second part of the chapter is dedicated to our new asynchronous Byzantine-tolerant asset transfer system (cryptocurrency) with three noteworthy properties: quasi-anonymity, lightness, and consensus-freedom. In particular, we focus on consensus-freedom with the concurrent specification of asset transfer. This specification describes the weak ordering of the transfer operations performed by the processes and emphasises that each process may observe a different sequence of operations locally. The implementation of asset transfer and the other properties are discussed in the next chapter.

3.3.2 Model

We adopt the model presented in Section 2.1.1 with the following modifications. Up to $t < n/3$ processes can be Byzantine and 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

Notation	Meaning
λ, $\epsilon(\lambda)$, Adv	Security parameter, Negligible real number in $[0;1]$, Adversary
AT, QAAT	Asset transfer object, Quasi-anonymous asset transfer object
AP, σ, P_A	Agreement proof scheme, Agreement proof, AP predicate
C, c, o	Commitment scheme, Commitment, Opening
UA, A, w, u	Universal accumulator scheme, Accumulator, Membership proof, Non-membership proof
ZKP, π, P_{ZK}	Zero-knowledge proof scheme, Zero-knowledge proof, ZKP predicate
$\mathbb{D}$ (for "data")	Input set (values) of the proving operations of Section 4.2.1's schemes
$\mathbb{F}$ (for "finite")	Output set (proofs) of the proving operations of Section 4.2.1's schemes
τ:$\langle snd, v, rcv, sn\rangle$	Asset transfer details: sender, amount, receiver, sequence number

Table 3.1 – Key notations used in QAAT.

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 [RSG98]. Note that both these solutions have a communication complexity of at most $O(\lambda n)$.

Notations. The *invocation* by a process of an operation op with input parameter *in* and output result *out* is denoted $\mathsf{op}(in)/out$. The "$\star$" symbol means that the corresponding value is left unspecified, *e.g.*, $\mathsf{op}_i(\star)$ refers to an invocation of op by p_i with an arbitrary input. A pair made up of a and b is denoted $\langle a, b\rangle$. Table 3.1 summarizes key notations used throughout this section and the next chapter.

3.3.3 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 [Auv+20]. 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. We assume that each account is owned by a single process.

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

— $\mathsf{balance}()/v$ returns the balance v of the calling process' account according to its current local vision.

— $\mathsf{transfer}_i(v, j)/r$ transfers the amount v from the account of the calling process

p_i to the account of p_j, returns r = `commit` if the account of p_i has enough funds, r = `abort` otherwise. For simplicity, if r = `commit`, we omit writing 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 [Auv+20] 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 non-negative value denoted $\texttt{init}_i$ (in our QAAT implementation of Algorithm 7, $\texttt{init}_i$ 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 [Auv+20]. 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.

AT-sequence. Given a process ID i and a set of operation invocations O (performed by p_i and by other processes), the function $total(i, O)$ returns the account balance 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):

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

A sequence S of invocations is an *asset-transfer-sequence* (*AT-sequence*) if and only if:

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

Informally, these conditions mean that the balance operation must return the balance of the process' 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 $\mathcal{A}$ 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 $\mathcal{A}$ (balance and transfer) by correct processes terminate.

— **AT-Sequentiality.** For any global history H capturing an execution of $\mathcal{A}$ there exists a mock history $\widehat{H}$ of H that can be AT-sequenced.

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

— **QAAT-Receiver-Anonymity.** For any global history H capturing an execution of $\mathcal{A}$, invocation $\mathsf{transfer}_i(v, \star)$ contained in H where p_i and p_j are correct processes, and $j' \in [1..n]$ a process identity chosen by Adv attempting to guess j, then $\Pr(j = j') \leq \min((1/n) + \epsilon(\lambda), 1)$.

— **QAAT-Confidentiality.** For any global history H capturing an execution of $\mathcal{A}$, invocation $\mathsf{transfer}_i(v, j)$ contained in H where p_i and p_j are correct processes, and $v' \in \mathbb{R}^+$ an amount chosen by the adversary Adv attempting to guess v, then $\Pr(v = v') \leq \epsilon(\lambda)$.

Informally, if we consider an adversary Adv with absolutely no information on a transfer $\mathsf{transfer}_i(v, j)$, the best this adversary can do is to pick $v \in \mathbb{R}^+$ and $j \in [1..n]$ uniformly at random. Doing so, Adv will guess j correctly with a probability of $1/n$, and v with a probability of 0.[2] The above two properties capture that a Quasi-Anonymous Asset Transfer can be made arbitrarily close to this ideal situation by increasing the security parameter λ of the cryptographic schemes.[3]

3.4 Conclusion

We have presented SplitChain, a protocol supporting the creation of scalable account-based blockchains without undermining decentralization and security. SplitChain distinguishes itself from other sharded blockchains by minimizing the synchronization constraints among shards while maintaining security guarantees. Specifically, SplitChain is the first permissionless sharded blockchain that does not require a dedicated shard or a global blockchain to attribute validators to their consensus chain. This avoids the need for a global reconfiguration of the shards each time a new batch of validators is added to the system. A dedicated routing protocol enables transactions to be redirected between shards with a low number of hops and messages. Finally, SplitChain dynamically adapts the number of shards to the system load to avoid over-dimensioning issues encountered in static sharding-based solutions.

We have also presented a concurrent specification of asset transfer, highlighting the possibility of using agreement algorithms weaker than consensus, such as byzantine reliable broadcast. Consensus-freedom makes it possible to process asset transfers concurrently, resulting in higher throughput. Furthermore, building a total order in the presence of

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

asynchrony and failures is bound by the impossibility of the FLP theorem [FLP85], stating that consensus cannot be implemented deterministically in an asynchronous system prone to process faults. In comparison, agreement using weaker ordering can be implemented in asynchronous systems even in the presence of Byzantine faults [Bra87]. This makes consensus-free asset-transfer systems more resilient because they deterministically guarantee liveness in environments where their consensus-based counterparts cannot.

Chapter 4

Anonymity and confidentiality in distributed systems

4.1 Introduction

In this final chapter, we will introduce several cryptographic primitives, and then show how they can be used to implement an efficient and privacy-friendly asset transfer. In particular, we will describe a new asynchronous Byzantine-tolerant asset transfer system (cryptocurrency) with three noteworthy properties: quasi-anonymity, lightness, and consensus-freedom [1]. 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. The proposed algorithm is the first asset transfer system that simultaneously fulfills all these properties in the presence of asynchrony and Byzantine processes.

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

4.2.1 Privacy-enabling cryptographic primitives

Before moving to our actual contributions in Section 4.2.3, we present in this section the cryptographic schemes upon which our QAAT system builds, namely, *Commitments* (*C*), *Universal Accumulators* (*UA*), and *Zero-Knowledge Proofs* (*ZKP*). These schemes all require an implicit *setup* operation that takes as input the security parameter λ and

1. 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 [Gue+22; Gup16], *i.e.*, consensus or atomic broadcast.

outputs the public parameters of the scheme, which are known to all processes, including the adversary. For instance, the public parameters of a cryptographic scheme based on RSA groups would include the corresponding RSA modulus. All the cryptographic schemes that follow implicitly use some public parameters (although these parameters are not exposed explicitly in the specifications and algorithms). The domains of definition for the input values and output proofs of our schemes are respectively called $\mathbb{D}$ and $\mathbb{F}$. Each scheme features a proving algorithm that aims to convince a verifying algorithm of some claim through a proof. In the following, the proving algorithm is called the prover, while the verifying algorithm is called the verifier. Formally, both the prover and the verifier are PPT algorithms.

4.2.1.1 Succinctness

A cryptographic proving scheme is succinct if and only if its proof size and verification time are at most polylogarithmic (and thus sublinear) in the "size of the problem" [Bit+22]. This "size of the problem" depends on the scheme. For instance, it may refer to the number of signatures to aggregate in an aggregated signatures scheme [BLS04], 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 communication and verification costs do not become prohibitively high).

4.2.1.2 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 and compute a proof o 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.

C 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.

C 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.** $\forall\ c, o, o' \in \mathbb{F}, \forall\ v, v' \in \mathbb{D} : \Pr\Big(\mathsf{c\_verify}(c, v, o)/\texttt{true} \wedge \mathsf{c\_verify}(c, v', o')/\texttt{true} \wedge v' \neq v\Big) < \epsilon(\lambda)$. Informally, a commitment opens to only one value (w.h.p.).

— **C-Hiding.** $\forall\ c_1, c_2 \in \mathbb{F}, \forall\ v \in \mathbb{D}, \mathsf{c\_commit}(v)/\langle c, \star\rangle : |\Pr(c = c_1) - \Pr(c = c_2)| < \epsilon(\lambda)$.
Informally, a commitment does not leak any information on its committed value (w.h.p).

4.2.1.3 Cryptographic accumulators

An *accumulator* scheme (notion introduced in [BM93]) produces a short commitment to a set of elements S. A *universal accumulator* (UA) is a special kind of accumulator that can prove both the inclusion or non-inclusion of an element in the set by using *membership* or *non-membership proofs*, respectively. RSA accumulators [BBF19] 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 A.3).

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

Informally, the probability of computing a membership proof for a non-accumulated element or a non-membership proof for an accumulated element is negligible.

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

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

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

Informally, the probability of computing both a membership and non-membership proof for the same element is negligible.

— **UA-Indistinguishability.** No information on some accumulated set E is leaked from its associated accumulator A, membership proofs w or non-membership proofs u. [2]

For simplicity and compliance with the traditional definitions given in the cryptography literature [BBF19], we do not give properties for the $\mathsf{ua\_add}$ 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.

4.2.1.4 Arguments of Knowledge

By abuse of notation, we refer to this scheme 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

2. For a more formal definition of Indistinguishability, we refer the interested reader to [DHS15].

Arguments of Knowledge (zk-SNARKs) [Set20]. 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 in [For87] 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 $P_{ZK}(public\_data, secret\_data) =$ `true`. Much like the P_A predicate (Section 4.2.2), 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$(public_data, secret_data)/r$: Returns a result r, which can be either a zero-knowledge proof $r = \pi$ if the input parameters *secret_data* and *public_data* satisfy P_{ZK}, or $r =$ `abort` otherwise.

— Obj_{ZK}.zkp_verify$(\pi, public_data)/r$: Returns the validity $r \in \{$`true`, `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$(\pi, public_data)$ is true, then the prover knows some *secret_data* such that Obj_{ZK}.zkp_prove$(public_data, secret_data)/\pi$ and $P_{ZK}(public\_data, secret\_data)$ is true w.h.p.

— **ZKP-Completeness.** For any pair $\langle public\_data, secret\_data \rangle$ such that $P_{ZK}(public\_data, secret\_data)$ is true, we must have Obj_{ZK}.zkp_verify$(Obj_{ZK}$.zkp_prove$(public_data, secret_data), public_data)$ is true w.h.p.

— **ZKP-Zero-Knowledge.** No information on some *secret_data* is leaked by its as-

sociated *public_data* and zero-knowledge proof π.[3]

4.2.2 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 4.2.3.3, leverages APs to prevent double-spending. In the following, we provide a specification of the AP scheme, and we provide a consensus-free implementation with constant storage complexity and a communication complexity of $O(n\lambda)$ in Appendix A.2.

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 $P_A(v, data, sn_i) =$ `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 4.2.3.1 and discuss implementation details in Appendix A.3.)

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

- Obj_A.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 $r =$ `abort` otherwise.
- Obj_A.ap_verify$(\sigma_j, v, sn_j, j)/r$: The operation returns the validity $r \in \{$`true`, `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*

3. For a more formal definition of Zero-Knowledge, we refer the interested reader to [Gol01].

from p_i" if and only if any invocation of Obj_A.ap_verify(σ, 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.ap_prove$(v, \star)/\sigma_i$ as its sn_i^{th} invocation of Obj_A.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.ap_verify$(\sigma_i, v, sn_i, i) = Obj_A$.ap_verify$(\sigma'_i, v', sn_i, i) =$ `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 $P_A(v, data, sn_i)/$`true`.

— **AP-Termination.** Given a correct process p_i that executes Obj_A.ap_prove$(v, data)/r$ with value v and a *data*, if $P_A(v, data, sn_i) =$ `true` where sn_i is the sequence number of the current ap_prove invocation by p_i, then $r = \sigma_i$ where σ_i is a valid AP for v at sn_i from p_i. If $P_A(v, data, sn_i) =$ `false` then $r =$ `abort`.

4.2.3 Quasi-anonymous Asset Transfer (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.

4.2.3.1 Setup and initialization of process variables

We present in Algorithm 5 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 $bal_i =$

Algorithm 5 Initialization of the variables of Algorithm 7 (code for p_i).

```
1 initialize(bal_i ← init_i; sn_i ← 0; ⟨bal_c_i, bal_o_i⟩ ← c_commit(init_i); T_i ← ∅;
        A_i ← empty universal accumulator of p_i;
        σ_i ← initial valid agreement proof for ⟨A_i, bal_c_i⟩ at sn_i = 0 from p_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 A.3.

4.2.3.2 AP and ZKP predicates

Our system relies on an AP object *AccountUpdate*, and a ZKP object *TransferValidity*, both shown in Algorithm 6. 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 as input in its *data* parameter 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. $old\_state\_data_{pvr}$ 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 () 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 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).

Algorithm 6 AP and ZKP predicates of Algorithm 7.

2 `initialize`($AccountUpdate \leftarrow$ *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}, τ, u_{pvr}) $\wedge$ ua_add(A_{pvr}, τ) $= A'_{pvr}$;

▷ *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}, τ, w_{snd});

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

22 **else return** `false`.

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 to 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 to 21). In mode 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). This condition is needed to support empty transfers, which are used to enhance the sender anonymity of the system. 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 fund 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, that the sender's accumulator contains the transfer, 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).

4.2.3.3 Algorithm

Algorithm 7 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

Algorithm 7 QAAT algorithm (code for p_i).

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

24 **operation** transfer(v, j) **is**
25 **if** $v < 0 \lor v > bal_i$ **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** $\left\{ \begin{array}{l} \tau.rcv = i \land \text{ua\_verify\_mem}(A_j, \tau, w_j) \land \\ AccountUpdate.\text{ap\_verify}(\sigma_j, \langle A_j, bal\_c_j \rangle, \tau.sn, j) \end{array} \right\}$ **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 $sn_i \leftarrow sn_i + 1$; $A_i \leftarrow A'_i$; $bal_i \leftarrow bal'_i$; $bal\_c_i \leftarrow bal\_c'_i$; $bal\_o_i \leftarrow bal\_o'_i$.

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 to 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).

4.2.4 Sketch proofs of the QAAT algorithm

In this section, we skecth the high-level intuition behind the correctness, of our solution. The full correctness developments are given in Section 4.2.5.

4.2.4.1 Correctness proof

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

Theorem 1 (AT-Sequentiality). For any global history H capturing an execution of *Algorithm 7* there exists a mock history $\widehat{H}$ of H that can be AT-sequenced.

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 $\mathcal{S}_i = \mathsf{set}(\widehat{L}_i) \cup \bigcup_{j \neq i} \mathsf{transferSet}(\widehat{L}_j)$. 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 $\mathsf{balance}$ invocations by p_i exactly reflect previous transfers (Property n.1 of an AT-sequence in Section 3.3.3) 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 $\mathsf{balance}$ 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 $(\mathcal{S}_i, \rightsquigarrow_i)$ and prove it fulfills the two properties of AT-Sequences. (Full derivations in Section 4.2.5.1.) □

Theorem 2 (AT-Termination). All operation invocations of *Algorithm 7* (balance and transfer) terminate for correct processes. *(Proof in Section 4.2.5.1.)*

4.2.4.2 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 Section 4.2.5.2.

Theorem 3 (QAAT-Receiver-Anonymity). For any global history H capturing an execution of *Algorithm 7*, invocation $\mathsf{transfer}_i(\star, j)$ in H where p_i and p_j are correct processes, and process identity $j' \in [1..n]$ chosen by *Adv* trying to guess j, we must have: $\Pr(j = j') \leq \min((1/n) + \epsilon(\lambda), 1)$. *(Proof in Section 4.2.5.2.)*

Theorem 4 (QAAT-Confidentiality). For any global history H capturing an execution of *Algorithm 7*, invocation $\mathsf{transfer}_i(v, j)$ in H where p_i and p_j are correct processes, and amount $v' \in \mathbb{R}^+$ chosen by *Adv* trying to guess v, we must have: $\Pr(v = v') \leq \epsilon(\lambda)$. *(Proof in Section 4.2.5.2.)*

4.2.4.3 Succinctness

The overall succinctness of our system stems from the succinctness of our Agreement Proof implementation (Appendix A.2), the conciseness (constant size digests and proofs) of RSA accumulators (Appendix A.1.2) and of commitments (Appendix A.1.3), and of the succinctness of Spartan zk-SNARKs (Appendix A.1.4). 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 concise for our scheme to be succinct (although accumulator (non-)membership proof verification and C openings can, in fact, be succinct).

4.2.4.4 Light storage

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 each of the n sequence numbers in $\vec{sn}$ are constant-size (see Section 4.2.3.3). Moreover, bal_c_i and bal_o_i have $O(\lambda)$ bits with a concise commitment scheme (Appendix A.1.3), A_i has $O(\lambda)$ bits with the (concise) RSA accumulator implementation (Appendix A.1.2), and σ_i has $O(\lambda)$ bits with our implementation of agreement proofs (see Appendix A.2). 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 $O(\lambda+(|T|/n)\log n+n)$. If the processes are not required to keep track of their transactions, this cost can be reduced to $O(\lambda(|T|/n)+n)$ by locally storing and accumulating transaction hashes instead of plaintext transactions ($O(\lambda)$ instead of $O(\log n)$).

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 (presented in Appendix A.2) communicates only $O(\lambda n)$ bits overall, *(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 3.3.2. As a result, our system's overall communication cost is $O(\lambda n)$.

Consensus-freedom. This directly comes from our distributed Agreement Proof implementation, which does not rely on strong agreement such as consensus (Appendix A.2).

4.2.5 Full proofs of our QAAT System (Algorithms 5 to 7)

In this section, we provide full derivations on the proof of correctness (Section 4.2.5.1) and quasi-anonymity (Section 4.2.5.2) of our QAAT system (Algorithms 5 to 7). Throughout the section, we rely on the property definitions of the C, UA, and ZKP schemes given in Section 4.2.1.

4.2.5.1 Proof of correctness

Preliminaries We first introduced a number of preliminary results and definitions that we will use to prove the AT-Sequentiality of Algorithm 7 in Section 4.2.5.1. In the following, σ_i^{sn} denotes an AP (Agreement Proof) that is a valid at a sequence number sn for a process p_i (correct of faulty).

Definition 4.2.1 (Valid UA at a sequence number *sn* for a process p_i.)**.** *We say that an 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 a 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 4.2.1 (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 4.2.1, then $A = A'$.*

Proof. This trivially follows from Definition 4.2.1 and the AP-Agreement property of Agreement Proofs. □

Lemma 4.2.2 (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 a 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 4.2.1, 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 6) 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 4.2.1, 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 4.2.1, $precedingUAs(A_i'^{sn})$ 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 4.2.1. □

Definition 4.2.2 (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 exists 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*

$$\mathsf{ua\_verify\_non\_mem}(A_i^{sn-1}, \tau, u) \wedge \mathsf{ua\_add}(A_i^{sn-1}, \tau) = A_i^{sn}. \tag{4.1}$$

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

When the context is clear, for simplicity, we might abbreviate Equation (4.1) using a set notation into $A_i^{sn} = A_i^{sn-1} \uplus \{\tau\}$, where $\uplus$ denotes the disjoint set union.

Lemma 4.2.3 (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)*[4]*. The existence of A_i^{sn} implies there exists a valid transfer τ_i^{sn} issued by p_i at sequence number sn (Definition 4.2.2). 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 4.2.1, 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 6), 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 6), this implies that $TransferValidity.\mathsf{zkp\_verify}(\pi_i, old\_state\_data \oplus \langle A_i^{sn}, \star, sn \rangle)$ is true. As a result, by ZKP-Knowledge-Soundness and by construction of the P_{ZK} predicate of the ZKP object *TransferValidity* (Algorithm 6), 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 6) such

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

that $P_{ZK}(\langle A_i^{sn-1}, \star\rangle old\_state\_data \oplus \langle A_i^{sn}, \star, sn\rangle, secret\_data)$ is true (if $i = snd$, then $sn = sn'$ due to the check at line 17 of Algorithm 6, otherwise if $i = rcv$, sn and sn' can be different). Furthermore, due to line 8, and by Lemma 4.2.1, 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 6 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 5 (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 4.2.2 and 4.2.3. □

Lemma 4.2.4 (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 a 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 follow from the unicity of a valid transfer for a given process and sequence number, as stated in Lemma 4.2.3. □

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 4.2.3 (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.*

— *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 during the* process_transfer() *call at line 38 of Algorithm 7.*

— *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 during the* process_transfer() *call at line 38 of Algorithm 7.*

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 4.2.5 (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 4.2.3) at sequence number sn_i 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 4.2.3).

— 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 4.2.1). Lines 20 and 21 of P_{ZK} (Algorithm 6), implies 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 4.2.2), 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 5). By construction of the two sequences, each transfer τ_j^k is valid for p_j at sequence number k, which implies

$$\forall k \in [1..sn_{\tau_i}] : A_j^k = A_j^{k-1} \uplus \{\tau_j^k\}. \tag{4.2}$$

Line 7 of P_A (Algorithm 6) 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 6) 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 4.2.3. □

Notations. For the remaining proofs, we provide a recap of introduce the following notations to manipulate the different notions we have introduced. 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 4.2.3). 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 4.2.5, 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 4.2.3), 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 process_transfer(..) of Algorithm 7 (lines 34 to 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 $\mathsf{balance}_i^{sn}()$. [5]

For simplicity, we denote by $\mathsf{op}_i^{sn}()$ any invocation $\mathsf{balance}_i^{sn}()$ or $\mathsf{transfer}_i^{sn}(\star, \star)$.

AT-Sequentiality of Algorithm 7 Let us consider any execution of Algorithm 7, captured as a global history $H = (L_1, ..., L_n)$. To prove AT-Seqentiality (Theorem 1), we

5. Recall that we do not consider balance() invocations from Byzantine processes.

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. $C \cup B = \{p1, .., p_n\}$. We construct the mock local histories $\{\widehat{L}_j\}_{p_j \in B}$ of Byzantine processes as follows.

Definition 4.2.4. *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 7. 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,*

$$\bigcup_{A \in \Sigma} \mathsf{set}(precedingUAs(A)) \subseteq \Sigma. \tag{4.3}$$

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 4.2.5, or through local precedence as captured by Lemma 4.2.2.

Lemma 4.2.6 (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 4.2.5 and Lemma 4.2.2 when defining $sendingUA(\cdot)$ and $precedingUAs(\cdot)$ respectively. □

Definition 4.2.5 (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*

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

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 5 does not have a corresponding transfer invocation), $SN_j \subseteq \mathbb{N}_{>0}$. 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 4.2.6 (Set $\mathcal{S}_i$ of operation perceived by a correct process p_i.)**.** *Consider a correct process p_i, and the set $\mathcal{S}_i$ 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 $\widehat{H}$.*

$$\mathcal{S}_i = \mathsf{set}(L_i) \cup \bigcup_{j\in C\setminus\{i\}} \mathsf{transferSet}(L_j) \cup \bigcup_{j\in B} \mathsf{transferSet}(\widehat{L}_j), \tag{4.4}$$

where C and B are the sets of correct and Byzantine processes as defined above, respectively, and transferSet$(\cdot)$ *is the notation introduced in Section 3.3.3 to denote the set of transfer invocations contained in a sequence.*

To produce a sequence S_i from the elements of $\mathcal{S}_i$, we construct a partial order $\rightsquigarrow_i$ on $\mathcal{S}_i$. 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 insures 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 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 4.2.7 (Binary relations $\overset{\mathbf{1}}{\rightsquigarrow}_i$, $\overset{\mathbf{2}}{\rightsquigarrow}_i$, and $\rightsquigarrow_i$ on $\mathcal{S}_i$)**.** *We define $\overset{\mathbf{1}}{\rightsquigarrow}_i$ on $\mathcal{S}_i$ as follows.*

— **Inclusion of p_i's local order.** $\overset{1}{\rightsquigarrow}_i$ *respects the local process order* $\rightarrow_i$ *of operations invoked by* p_i*, i.e.* $\rightarrow_i \subseteq \overset{1}{\rightsquigarrow}_i$.

— **Ordering of incoming transfers.**

— *For any process* p_j, $t_{\triangleright j} = \mathsf{transfer}^{\star}_{\star}(\star, j) \in \mathcal{S}_i$ *a transfer invocation whose receiver is* p_j*, and* $t_{j\triangleright} = \mathsf{transfer}^{\star}_{j}(\star, \star) \in \mathcal{S}_i$ *a transfer invocation by* p_j *such that* $t_{\triangleright j} \neq t_{j\triangleright}$*, if* $receivingUA(t_{\triangleright j}) \in precedingUAs(sendingUA(t_{j\triangleright}))$ *then* $t_{\triangleright j} \overset{1}{\rightsquigarrow}_i t_{j\triangleright}$.

— *For transfer invocation* $t_{\triangleright i} = \mathsf{transfer}^{\star}_{\star}(\star, i) \in \mathcal{S}_i$ *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* $receivingUA(t_{\triangleright i})$ *occurred before that of line 23 corresponding to* b_i *, then* $t_{\triangleright i} \overset{1}{\rightsquigarrow}_i b_i$.

We define $\overset{2}{\rightsquigarrow}_i$ *on* $\mathcal{S}_i$ *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* $t_{\triangleright i} = \mathsf{transfer}^{\star}_{\star}(\star, i) \in \mathcal{S}_i$ *whose receiver is* p_i*, if the execution of line 23 corresponding to* b_i *occurred before that of line 38 that generated* $receivingUA(t_{\triangleright i})$*, then* $b_i \overset{2}{\rightsquigarrow}_i t_{\triangleright i}$.

$\rightsquigarrow_i$ *is defined as the transitive closure of the union of* $\overset{1}{\rightsquigarrow}_i$ *and* $\overset{2}{\rightsquigarrow}_i$.

$$\rightsquigarrow_i = \left(\overset{1}{\rightsquigarrow}_i \cup \overset{2}{\rightsquigarrow}_i \right)^{+}$$

Lemma 4.2.7. $\overset{1}{\rightsquigarrow}_i$ *is acyclic.*

Proof. The distributed system is implicitly embedded in some physical time, which we assume is Newtonian and linear. We associate each element x of $\mathcal{S}_i$ 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 7 is executed.

— if $x = \mathsf{transfer}^{\star}_{\star}(\star, \star)$ is a transfer invocation, $\tau(x)$ is the time point at which $sendingUA(x)$ was created.

Consider $x, y \in \mathcal{S}_i$ such that $x \overset{1}{\rightsquigarrow}_i y$. 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):*

— $x = \mathsf{transfer}^{\star}_{\star}(\star, j) \in \mathcal{S}_i$ is a transfer invocation whose receiver is some process p_j, and $y = \mathsf{transfer}^{\star}_{j}(\star, \star) \in \mathcal{S}_i$ 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 $\tau(x) < \tau(y)$.

— $x = \mathsf{transfer}^{\star}_{\star}(\star, i) \in \mathcal{S}_i$ 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. □

Lemma 4.2.8. *$\overset{\mathbf{2}}{\rightsquigarrow}_i$ is acyclic.*

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

Lemma 4.2.9. *$\overset{\mathbf{1}}{\rightsquigarrow}_i \cup \overset{\mathbf{2}}{\rightsquigarrow}_i$ is acyclic.*

Proof. The proof is by contradiction. Assume $\overset{\mathbf{1}}{\rightsquigarrow}_i \cup \overset{\mathbf{2}}{\rightsquigarrow}_i$ contains at least one cycle. Consider the shortest such cycle $\mathcal{C}_{\mathsf{min}}$. Because $\overset{\mathbf{1}}{\rightsquigarrow}_i$ and $\overset{\mathbf{2}}{\rightsquigarrow}_i$ are both acyclic (Lemmas 4.2.7 and 4.2.8), $\mathcal{C}_{\mathsf{min}}$ must involve both $\overset{\mathbf{1}}{\rightsquigarrow}_i$ and $\overset{\mathbf{2}}{\rightsquigarrow}_i$.

Consider one of the $\overset{\mathbf{2}}{\rightsquigarrow}_i$ ordering in $\mathcal{C}_{\mathsf{min}}$. By definition of $\overset{\mathbf{2}}{\rightsquigarrow}_i$, this ordering is of the form $b_i \overset{\mathbf{2}}{\rightsquigarrow}_i t_{\triangleright i}$, 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 $\mathcal{C}_{\mathsf{min}}$. By construction of $\overset{\mathbf{2}}{\rightsquigarrow}_i$, $t_{\triangleright i} \overset{\mathbf{2}}{\not\rightsquigarrow}_i y$, which implies $t_{\triangleright i} \overset{\mathbf{1}}{\rightsquigarrow}_i y$.

By definition of $\overset{\mathbf{1}}{\rightsquigarrow}_i$, $t_{\triangleright i} \overset{\mathbf{1}}{\rightsquigarrow}_i y$ 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, $t_{\triangleright i} \overset{\mathbf{1}}{\rightsquigarrow}_i y$ 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 $b_i \overset{\mathbf{2}}{\rightsquigarrow}_i t_{\triangleright i} \overset{\mathbf{1}}{\rightsquigarrow}_i y$, with $b_i, y \in L_i$. The definitions of $\overset{\mathbf{2}}{\rightsquigarrow}_i$ and $\overset{\mathbf{1}}{\rightsquigarrow}_i$ imply that the invocation of b_i by p_i precedes in time the generation of $receivingUA(t_{\triangleright i})$ also by p_i, which itself precedes the invocation of y still by p_i (either directly in the case of $\overset{\mathbf{2}}{\rightsquigarrow}_i$, and if y is a balance invocation, or using the same argument on cryptographic proofs as in Lemma 4.2.7 if y is a transfer). By definition of the process order $\rightarrow_i$, this leads to $b_i \rightarrow_i y$, and hence $b_i \overset{\mathbf{1}}{\rightsquigarrow}_i y$. We can, therefore, remove $t_{\triangleright i}$ from $\mathcal{C}_{\mathsf{min}}$, and replace $b_i \overset{\mathbf{2}}{\rightsquigarrow}_i t_{\triangleright i} \overset{\mathbf{1}}{\rightsquigarrow}_i y$ by $b_i \overset{\mathbf{1}}{\rightsquigarrow}_i y$, thus creating a cycle $\mathcal{C}'_{\mathsf{min}}$ in $\overset{\mathbf{1}}{\rightsquigarrow}_i \cup \overset{\mathbf{2}}{\rightsquigarrow}_i$ with one less element than $\mathcal{C}_{\mathsf{min}}$. This is a contradiction as $\mathcal{C}_{\mathsf{min}}$ was assumed shortest. □

Corollary 6. $\rightsquigarrow_i$ is a partial order on $\mathcal{S}_i$.

Proof. $\rightsquigarrow_i$ is transitive by construction. By Lemma 4.2.9, it is the transitive closure of an acyclic relation and is, therefore, also acyclic. □

Lemma 4.2.10. *Each element of the poset ($\mathcal{S}_i$,$\rightsquigarrow_i$) only has a finite number of predecessors.*

Proof. This follows from the sequential nature of processes, the link between $\overset{\mathbf{1}}{\rightsquigarrow}_i$ and $\overset{\mathbf{2}}{\rightsquigarrow}_i$ and physical time (Lemmas 4.2.7 and 4.2.8), and the (implicit) assumption that processes only take a finite number of local steps per unit of time. □

Definition 4.2.8 (Sequence S_i)**.** *By Lemma 4.2.10 the poset ($\mathcal{S}_i$,$\rightsquigarrow_i$) can be sorted topologically into a sequence. We define S_i as one of the topological sorts of ($\mathcal{S}_i$,$\rightsquigarrow_i$). We note $\rightarrow_{S_i}$ the total order induced by S_i on its elements.*

In the following, we will use the function $total(\cdot,\cdot)$ introduced in Section 3.3.3 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, $total(j, O)$ is defined as

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

Lemma 4.2.11. $\forall\ b_i = \mathsf{balance}_i()/v \in S_i : v = total(i, \{o \in S_i \mid o \rightarrow_{S_i} b_i\})$.

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

— I is the set of transfers *invoked* locally by p_i before p_i executed b_i. By construction of Algorithm 7, p_i generated a sending UA at line 38 for each of these transfers.

— R 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 7, which ensures by construction (in particular due to lines 23, 35 and 45) that

$$v = total(i, I \cup R), \tag{4.5}$$

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

$$T_{i\triangleright} \stackrel{\text{def}}{=} \{\mathsf{transfer}_i(\star, k) \in \mathcal{S}_i \mid i \neq k\} \tag{4.6}$$

$$T_{\triangleright i} \stackrel{\text{def}}{=} \{\mathsf{transfer}_k(\star, i) \in \mathcal{S}_i \mid i \neq k\} \tag{4.7}$$

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

$$total\Big(i, \{o \in S_i \mid o \rightarrow_{S_i} b_i\}\Big) = total\Big(i, \{o \in S_i \mid o \rightarrow_{S_i} b_i\} \cap (T_{i\triangleright} \cup T_{\triangleright i})\Big). \tag{4.8}$$

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 $\rightarrow_{S_i}$,

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

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

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

These last two equalities together with Equations (4.5) and (4.8) yield the lemma.

□

Lemma 4.2.12. $\forall\ j \in [0..n], \forall\ t_{j\triangleright} = \mathsf{transfer}_j(v, \star) \in S_i : v \leq total\Big(j, \{o \in S_i \mid o \rightarrow_{S_i} t_{j\triangleright}\}\Big)$.

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 $I_{t_{j\triangleright}}^{S_i}$ and $R_{t_{j\triangleright}}^{S_i}$ as follows.

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

$$I_{t_{j\triangleright}}^{S_i} = \Big\{\mathsf{transfer}_j^\star(\star,\star) \in S_i \mid \mathsf{transfer}_j^\star(\star,\star) \rightarrow_{S_i} t_{j\triangleright} \wedge \mathsf{transfer}_j^\star(\star,\star) = t_{j\triangleright}\Big\}.$$

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

$$R_{t_{j\triangleright}}^{S_i} = \Big\{\mathsf{transfer}_\star^\star(j,\star) \in S_i \mid \mathsf{transfer}_\star^\star(j,\star) \rightarrow_{S_i} t_{j\triangleright}\Big\}. \tag{4.9}$$

Remark 4.2.6.1. $total(j, I_{t_{j\triangleright}}^{S_i} \cup R_{t_{j\triangleright}}^{S_i}) = total(j, \{o \in S_i \mid o \rightarrow_{S_i} t_{j\triangleright}\}) - v$.

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

Consider $SN_{t_{j\triangleright}}^{S_i}$ the set of the sequence numbers of the transfers present in $I_{t_{j\triangleright}}^{S_i}$

$$SN_{t_{j\triangleright}}^{S_i} = \{sn \in \mathbb{N}_{>0} \mid \mathsf{transfer}_j^{sn}(\star,\star) \in I_{t_{j\triangleright}}^{S_i}\}. \tag{4.10}$$

Because $t_{j\triangleright} \in I_{t_{j\triangleright}}^{S_i}$, $SN_{t_{j\triangleright}}^{S_i}$ 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, $I_{t_{j\triangleright}}^{S_i}$ might also contain transfer invocations that have a higher sequence number than $t_{j\triangleright}$.

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

$$sn_{j\triangleright}^{\mathsf{max}} = \mathsf{max}\Big(SN_{t_{j\triangleright}}^{S_i}\Big), \tag{4.11}$$

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

Consider $\Sigma_{j\triangleright}^{\mathsf{max}}$ the set of UAs issued by p_j that precede $A_{j\triangleright}^{\mathsf{max}}$ according to the predicates P_A and P_{ZK} (Lemma 4.2.2),

$$\Sigma_{j\triangleright}^{\mathsf{max}} = \mathsf{set}\Big(precedingUAs(A_{j\triangleright}^{\mathsf{max}})\Big). \tag{4.12}$$

Based on $\Sigma_{j\triangleright}^{\mathsf{max}}$, let us define the sets $I_{t_{j\triangleright}}^{\mathsf{max}}$ and $R_{t_{j\triangleright}}^{\mathsf{max}}$ as follows.

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

$$I^{\mathsf{max}}_{t_{j\triangleright}} = \left\{ \mathsf{transfer}^{\star}_{j}(\star,\star) \mid \mathit{sendingUA}(\mathsf{transfer}^{\star}_{j}(\star,\star)) \in \Sigma^{\mathsf{max}}_{j\triangleright} \right\}. \tag{4.13}$$

— $R^{\mathsf{max}}_{t_{j\triangleright}}$ is the set of transfers whose receiver is p_j 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),

$$R^{\mathsf{max}}_{t_{j\triangleright}} = \left\{ \mathsf{transfer}^{\star}_{\star}(j,\star) \;\middle|\; \begin{array}{l} \mathit{receivingUA}(\mathsf{transfer}^{\star}_{\star}(j,\star)) \in \Sigma^{\mathsf{max}}_{j\triangleright} \wedge \\ \mathsf{transfer}^{\star}_{\star}(j,\star) \neq \mathit{transfer}(A^{\mathsf{max}}_{j\triangleright}) \end{array} \right\}. \tag{4.14}$$

Let v_{max} denote the amount of the transfer invocation $\mathit{transfer}(A^{\mathsf{max}}_{j\triangleright})$, *i.e.*, $\mathit{transfer}(A^{\mathsf{max}}_{j\triangleright}) = \mathsf{transfer}^{sn^{\mathsf{max}}_{j\triangleright}}_{j}(v_{\mathsf{max}},\star)$.

Remark 4.2.6.2. $\mathit{total}\left(j, I^{\mathsf{max}}_{t_{j\triangleright}} \cup R^{\mathsf{max}}_{t_{j\triangleright}}\right) = \mathit{total}\left(j, \left(I^{\mathsf{max}}_{t_{j\triangleright}} \setminus \left\{\mathit{transfer}(A^{\mathsf{max}}_{j\triangleright})\right\}\right) \cup R^{\mathsf{max}}_{t_{j\triangleright}}\right) - v_{\mathsf{max}}$.

Proof. Similarly to Remark 4.2.6.1, the remark follows from the fact that $\mathit{transfer}(A^{\mathsf{max}}_{j\triangleright}) \in I^{\mathsf{max}}_{t_{j\triangleright}}$ and $\mathit{transfer}(A^{\mathsf{max}}_{j\triangleright}) \notin R^{\mathsf{max}}_{t_{j\triangleright}}$, the definition of v_{max}, and that of $\mathit{total}(\cdot,\cdot)$. □

Remark 4.2.6.3. $\mathit{total}\left(j, \left(I^{\mathsf{max}}_{t_{j\triangleright}} \setminus \left\{\mathit{transfer}(A^{\mathsf{max}}_{j\triangleright})\right\}\right) \cup R^{\mathsf{max}}_{t_{j\triangleright}}\right) \geq v_{\mathsf{max}}$.

Proof. This follows from a recursion on the sequence of (valid) UAs $\mathit{precedingUAs}(A^{\mathsf{max}}_{j\triangleright})$ that precede $A^{\mathsf{max}}_{j\triangleright}$ (Lemma 4.2.2), as well as the checks and updates contained in the P_{ZK} predicate (Algorithm 6). 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 4.2.6.4. $\mathit{total}\left(j, I^{\mathsf{max}}_{t_{j\triangleright}} \cup R^{\mathsf{max}}_{t_{j\triangleright}}\right) \geq 0$.

Proof. This is a direct consequence of Remarks 4.2.6.2 and 4.2.6.3. □

Remark 4.2.6.5. $I^{S_i}_{t_{j\triangleright}} \subseteq I^{\mathsf{max}}_{t_{j\triangleright}}$.

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 (4.10) and (4.11)), 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 (4.12) and (4.13)), that $t'_{j\triangleright} \in I^{\mathsf{max}}_{t_{j\triangleright}}$, proving the remark. □

Remark 4.2.6.6. $R^{\mathsf{max}}_{t_{j\triangleright}} \subseteq R^{S_i}_{t_{j\triangleright}}$.

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 (4.12) and (4.14)),

$$receivingUA(t'_{\triangleright j}) \in \mathsf{set}\Big(precedingUAs(A^{\mathsf{max}}_{j\triangleright})\Big) \ \wedge \ t'_{\triangleright j} \neq transfer(A^{\mathsf{max}}_{j\triangleright}) \tag{4.15}$$

Furthermore, because $transfer(A^{\mathsf{max}}_{j\triangleright}) \in \mathcal{S}_i$ we have $t'_{\triangleright j} \in \mathcal{S}_i$ (either because the sender of $t_{\triangleright j}$ is correct, or by construction of Σ if it is faulty, see Equations (4.3) and (4.4)). This inclusion together with Equation (4.15) 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

$$t'_{\triangleright j} \rightarrow_{S_i} transfer(A^{\mathsf{max}}_{j\triangleright}). \tag{4.16}$$

Because $transfer(A^{\mathsf{max}}_{j\triangleright}) \in R^{S_i}_{t_{j\triangleright}}$, by definition of $R^{S_i}_{t_{j\triangleright}}$ (Equation (4.9)), $transfer(A^{\mathsf{max}}_{j\triangleright}) \rightarrow_{S_i} t_{j\triangleright}$, which with Equation (4.16) leads to $t'_{\triangleright j} \rightarrow_{S_i} t_{j\triangleright}$, and therefore $t'_{\triangleright j} \in R^{S_i}_{t_{j\triangleright}}$. □

Remark 4.2.6.7. $total(j, I^{S_i}_{t_{j\triangleright}} \cup R^{S_i}_{t_{j\triangleright}}) \geq total\left(j, I^{\mathsf{max}}_{t_{j\triangleright}} \cup R^{\mathsf{max}}_{t_{j\triangleright}}\right)$

Proof. This follows from Remarks 4.2.6.5 and 4.2.6.6 and the definition of $total(\cdot, \cdot)$. □

The lemma follows from Remarks 4.2.6.1, 4.2.6.4 and 4.2.6.7. □

Theorem 1 (AT-Sequentiality). For any global history H capturing an execution of *Algorithm 7* 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 4.2.5, the sequence S_i constructed in Definition 4.2.8 for each correct process p_i, and considering Lemmas 4.2.11 and 4.2.12 that show that S_i is an AT-Sequence. □

AT-Termination of Algorithm 7

Theorem 2 (AT-Termination). All operation invocations of *Algorithm 7* (balance and transfer) terminate for correct processes.

Proof. The balance operation of Algorithm 7 terminates trivially. Furthermore, the only blocking instruction of the transfer operation of Algorithm 7 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. □

4.2.5.2 Proof of quasi-anonymity

Theorem 3 (QAAT-Receiver-Anonymity). For any global history H capturing an execution of *Algorithm 7*, invocation $\mathsf{transfer}_i(\star, j)$ in H where p_i and p_j are correct processes, and process identity $j' \in [1..n]$ chosen by *Adv* trying to guess j, we must have: $\Pr(j = j') \leq \min((1/n) + \epsilon(\lambda), 1)$.

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 7. 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 . 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 $\min((1/n) + \epsilon(\lambda), 1)$, where ϵ is the statistical negligible function and λ is the security parameter of the system. □

Theorem 4 (QAAT-Confidentiality). For any global history H capturing an execution of *Algorithm 7*, invocation $\mathsf{transfer}_i(v, j)$ in H where p_i and p_j are correct processes, and amount $v' \in \mathbb{R}^+$ chosen by *Adv* trying to guess v, we must have: $\Pr(v = v') \leq \epsilon(\lambda)$.

Proof. The proof follows the same reasoning as that of Theorem 3. □

4.2.6 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* [KST22] in Appendix A.4.

Constant local storage (public key rotation). The QAAT system presented in this chapter is light, *i.e.*, the storage cost per process p_i is in $O(\lambda|T|/n)$, 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 obtain a system with a constant storage cost. When a process p_i rotates its public key, it changes its old public key for a new one, while transferring all its funds to the account associated to the new public key. Once this is done, p_i can flush its old local data (and, in particular, its accumulator data) so that its storage cost stays bounded by a constant. Thus, the process only has to record information concerning this rotating transfer which can be seen as the genesis state of a newly created account. However, this mechanism would involve one major technical challenge: a sender has to retrieve the receiver's public key before it can send funds to 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 [Bau+23] 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 chose the moments to issue empty transfers is left to future work.

4.3 Conclusion

This chapter 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 chapter 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 chapter 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 computational, storage, and network requirements 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 [Kuz+23].

Conclusion

This thesis addresses key challenges in blockchain technology, focusing on scalability and privacy. Our work tackles specific limitations of existing blockchain systems, particularly the scalability issues arising from full replication models and the use of consensus, and privacy concerns due to the public nature of blockchain data.

To address these challenges, this thesis has explored two novel approaches in distributed asset transfer systems, namely blockchain sharding and consensus-freedom.

SplitChain: Advancing Scalability in Blockchain Systems

The first contribution is SplitChain, a protocol for creating scalable proof-of-stake and account-based blockchains. SplitChain distinguishes itself from existing sharded blockchain solutions in several ways. It operates in a fully asynchronous setting and minimizes synchronization constraints among shards while maintaining strong security guarantees against an adaptive adversary. It is the first permissionless sharded blockchain that does not require a dedicated shard or global blockchain for validator attribution. This central coordinating shard was a bottleneck for the sharded blockchains in the literature, as its restricted capacity limited the total number of shards in the system. It also features a dedicated routing protocol that enables efficient transaction redirection between shards with a low number of hops and messages. Finally, it dynamically adapts the number of shards based on system load, addressing over-dimensioning issues prevalent in static sharding solutions. Our experimental results, based on replaying the entire transaction history of Ethereum, demonstrate that SplitChain's dynamic sharding approach achieves a more balanced distribution of transactions across shards compared to static sharding solutions.

QAAT: Enhancing Privacy and Efficiency in Asset Transfer

The second contribution is QAAT (Quasi-Anonymous Asset Transfer), an asynchronous Byzantine-tolerant asset transfer system. QAAT successfully achieves three critical properties. It ensures quasi-anonymity, preventing information leakage regarding transfer amounts and receivers. The system is light, employing succinct cryptography and minimizing storage costs. Finally, it is consensus-free, operating without reliance on a total order of transfers.

QAAT is the first asset transfer system to simultaneously fulfill these three properties while maintaining Byzantine fault tolerance in asynchronous networks. It demonstrates how strong privacy features can be guaranteed in systems with low computational, storage, and network requirements. The system's design allows each process to store only its own transactions, which are then verified and processed by other processes in the system without revealing any sensitive information about their content or recipients.

Limitations and Future Work

Several areas warrant further research. Exploring the combination of SplitChain and the privacy-preserving techniques developed in our quasi-anonymous asset transfer system could lead to a comprehensive blockchain solution addressing both scalability and privacy concerns simultaneously. Extending QAAT to support smart contracts while preserving its privacy and efficiency properties would greatly enhance its practical applicability.

For QAAT, future research could explore making the system permissionless without sacrificing its consensus-freedom,

Conclusion

This thesis presents two contributions to the ongoing evolution of blockchain technology by addressing key challenges in scalability and privacy. SSplitChain demonstrates that it is possible to achieve scalability in blockchain systems without sacrificing security or decentralization, paving the way for more efficient large-scale blockchain deployments. Our quasi-anonymous asset transfer system showcases how cryptographic techniques can be combined to achieve strong privacy guarantees while maintaining high performance and fault tolerance. Moreover, by focusing on privacy-preserving techniques and data storage, our work makes progress towards aligning blockchain technology with privacy regulations such as GDPR.

BIBLIOGRAPHY

[ALS12] Emmanuelle Anceaume, Romaric Ludinard, and Bruno Sericola, « Performance Evaluation of Large-scale Dynamic Systems », *in*: *Sigmetrics Performance Evaluation Review - SIGMETRICS* 39 (2012).

[Anc+19] Emmanuelle Anceaume et al., « Blockchain Abstract Data Type », *in*: *SPAA*, ACM, 2019, pp. 349–358.

[Auv+20] Alex Auvolat et al., « Money Transfer Made Simple: a Specification, a Generic Algorithm, and its Proof », *in*: *Bulletin of EATCS* 132 (2020), pp. 22–43.

[Bau+23] Mathieu Baudet et al., « Zef: Low-latency, Scalable, Private Payments », *in*: *Proc. 22nd Workshop on Privacy in the Electronic Society (WPES@CCS'23)*, ACM, 2023, pp. 1–16.

[BB20] Alexandra Bensamoun and Brunessen Bertrand, *Le règlement général sur la protection des données : aspects institutionnels et matériels*, Droit et science politique, Mare & Martin, 2020.

[BBF19] Dan Boneh, Benedikt Bünz, and Ben Fisch, « Batching Techniques for Accumulators with Applications to IOPs and Stateless Blockchains », *in*: *Proc. 39th Int'l Cryptology Conference*, Springer LNCS 11692, 2019, pp. 561–586.

[BCG15] Joseph Bonneau, Jeremy Clark, and Steven Goldfeder, *On Bitcoin as a public randomness source*, Cryptology ePrint Archive, Paper 2015/1015, 2015, URL: `https://eprint.iacr.org/2015/1015`.

[BDS20] Mathieu Baudet, George Danezis, and Alberto Sonnino, « FastPay: High-Performance Byzantine Fault Tolerant Settlement », *in*: *Proc. 2nd ACM Conference on Advances in Financial Technologies (AFT'20)*, ACM, 2020, pp. 163–177.

[Bit+22] Nir Bitansky et al., « Succinct Non-Interactive Arguments via Linear Interactive Proofs », *in*: *J. Cryptol.* 35.*3* (2022), p. 15.

[BKP14] Alex Biryukov, Dmitry Khovratovich, and Ivan Pustogarov, « Deanonymisation of Clients in Bitcoin P2P Network », *in*: *CCS*, ACM, 2014, pp. 15–29.

[BLS04] Dan Boneh, Ben Lynn, and Hovav Shacham, « Short Signatures from the Weil Pairing », *in*: *J. Cryptol.* 17.*4* (2004), pp. 297–319.

[BM93] Josh Cohen Benaloh and Michael de Mare, « One-Way Accumulators: A Decentralized Alternative to Digital Signatures (Extended Abstract) », *in*: *Proc. 12th Workshop on the Theory and Application of of Cryptographic Techniques (EUROCRYPT'93)*, vol. 765, Springer LNCS, 1993, pp. 274–285.

[Bou+20] Fabrice Boudot et al., « Comparing the Difficulty of Factorization and Discrete Logarithm: A 240-Digit Experiment », *in*: *CRYPTO (2)*, vol. 12171, Lecture Notes in Computer Science, Springer, 2020, pp. 62–91.

[Bra87] Gabriel Bracha, « Asynchronous Byzantine Agreement Protocols », *in*: *Inf. Comput.* 75.*2* (1987), pp. 130–143.

[BT85] Gabriel Bracha and Sam Toueg, « Asynchronous Consensus and Broadcast Protocols », *in*: *J. ACM* 32.*4* (1985), pp. 824–840, DOI: 10.1145/4221.214134, URL: https://doi.org/10.1145/4221.214134.

[Bün+20] Benedikt Bünz et al., « Zether: Towards Privacy in a Smart Contract World », *in*: *Proc. 24th Int'l Conference on Financial Cryptography and Data Security (FC'20)*, vol. 12059, Lecture Notes in Computer Science, 2020, pp. 423–443.

[Che+] Steve Cheng et al., *Using blockchain to improve data managementin the public sector*, https://www.mckinsey.com/business-functions/mckinsey-digital/our-insights/using-blockchain-to-improve-data-management-in-the-public-sector.

[Col+20] Daniel Collins et al., « Online Payments by Merely Broadcasting Messages », *in*: *Proc. 50th IEEE/IFIP Int'l Conference on Dependable Systems and Networks (DSN'20)*, IEEE, 2020, pp. 26–38.

[Das+23] Sourav Das et al., « Practical Asynchronous High-threshold Distributed Key Generation and Distributed Polynomial Sampling », *in*: *Proc. 32nd USENIX Security Symposium*, USENIX Association, 2023, pp. 5359–5376.

[DGS22] Samuel Dobson, Steven D. Galbraith, and Benjamin Smith, « Trustless unknown-order groups », *in*: *CoRR* abs/2211.16128 (2022), DOI: 10.48550/ARXIV.2211.16128, eprint: 2211.16128.

[DHS15] David Derler, Christian Hanser, and Daniel Slamanig, « Revisiting Cryptographic Accumulators, Additional Properties and Relations to Other Primitives », *in*: *Topics in Cryptology - CT-RSA 2015, The Cryptographer's Track at the RSA Conference 2015, San Francisco, CA, USA, April 20-24, 2015. Proceedings*, ed. by Kaisa Nyberg, vol. 9048, Lecture Notes in Computer Science, Springer, 2015, pp. 127–144, DOI: 10.1007/978-3-319-16715-2_7.

[DK02] Ivan Damgård and Maciej Koprowski, « Generic Lower Bounds for Root Extraction and Signature Schemes in General Groups », *in*: *Advances in Cryptology - EUROCRYPT 2002, International Conference on the Theory and Applications of Cryptographic Techniques, Amsterdam, The Netherlands, April 28 - May 2, 2002, Proceedings*, ed. by Lars R. Knudsen, vol. 2332, Lecture Notes in Computer Science, Springer, 2002, pp. 256–271, DOI: 10.1007/3-540-46035-7_17.

[Fau+19] Prastudy Fauzi et al., « 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)*, vol. 11921, LNCS, Springer, 2019, pp. 649–678.

[FG17] Christian Franck and Johann Großschädl, « Efficient Implementation of Pedersen Commitments Using Twisted Edwards Curves », *in*: *MSPN*, vol. 10566, Lecture Notes in Computer Science, Springer, 2017, pp. 1–17.

[FGR23] Davide Frey, Mathieu Gestin, and Michel Raynal, « The Synchronization Power (Consensus Number) of Access-Control Objects: the Case of AllowList and DenyList », *in*: *Proc. 37th Int'l Symposium on Distributed Computing (DISC'2023)*, vol. 281, LIPIcs, Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2023, 21:1–21:23.

[FLP83] Michael J. Fischer, Nancy A. Lynch, and Mike Paterson, « Impossibility of Distributed Consensus with One Faulty Process », *in*: *PODS*, ACM, 1983, pp. 1–7.

[FLP85] Michael J. Fischer, Nancy A. Lynch, and Mike Paterson, « Impossibility of Distributed Consensus with One Faulty Process », *in*: *J. ACM* 32.*2* (1985), pp. 374–382.

[For87] Lance Fortnow, « The complexity of perfect zero-knowledge », *in*: *SCT*, IEEE Computer Society, 1987, p. 156.

[Gam84] Taher El Gamal, « A Public Key Cryptosystem and a Signature Scheme Based on Discrete Logarithms », *in*: *CRYPTO*, vol. 196, Lecture Notes in Computer Science, Springer, 1984, pp. 10–18.

[Gil+17] Yossi Gilad et al., « Algorand: Scaling Byzantine Agreements for Cryptocurrencies », *in*: *Proceedings of the 26th Symposium on Operating Systems Principles*, SOSP '17, Shanghai, China: Association for Computing Machinery, 2017, pp. 51–68, ISBN: 9781450350853, DOI: 10.1145/3132747.3132757, URL: https://doi.org/10.1145/3132747.3132757.

[Gol01] Oded Goldreich, *The Foundations of Cryptography - Volume 1: Basic Techniques*, Cambridge University Press, 2001, pp. 1–372, ISBN: 0-521-79172-3, DOI: 10.1017/CBO9780511546891, URL: http://www.wisdom.weizmann.ac.il/%5C%7Eoded/foc-vol1.html.

[Gue+22] Rachid Guerraoui et al., « The consensus number of a cryptocurrency », *in*: *Distributed Comput.* 35.*1* (2022), pp. 1–15.

[Gup16] Saurabh Gupta, *A Non-Consensus Based Decentralized Financial Transaction Processing Model with Support for Efficient Auditing (Master Thesis)*, Arizona State University, 2016.

[GWC19] Ariel Gabizon, Zachary J. Williamson, and Oana Ciobotaru, « PLONK: Permutations over Lagrange-bases for Oecumenical Noninteractive arguments of Knowledge », *in*: *IACR Cryptol. ePrint Arch.* (2019), p. 953.

[HBW16] Daira Emma Hopwood, Sean Bowe, and Taylor Hornby Nathan Wilcox, *Zcash Protocol Specification*, 2016.

[KG20] Chelsea Komlo and Ian Goldberg, « FROST: Flexible Round-Optimized Schnorr Threshold Signatures », *in*: *Proc. 27th Int'l Conference on Selected Areas in Cryptography (SAC'20)*, vol. 12804, Lecture Notes in Computer Science, Springer, 2020, pp. 34–65.

[KL14] Jonathan Katz and Yehuda Lindell, *Introduction to Modern Cryptography, Second Edition*, 2nd, Chapman & Hall/CRC, 2014, ISBN: 1466570261.

[Kok+18] Eleftherios Kokoris-Kogias et al., « OmniLedger: A Secure, Scale-Out, Decentralized Ledger via Sharding », *in*: *2018 IEEE Symposium on Security and Privacy (SP)*, 2018, pp. 583–598, DOI: 10.1109/SP.2018.000-5.

[KST22] Abhiram Kothapalli, Srinath T. V. Setty, and Ioanna Tzialla, « Nova: Recursive Zero-Knowledge Arguments from Folding Schemes », *in*: *42nd Annual International Cryptology Conference (CRYPTO'22)*, vol. 13510, LNCS, Springer, 2022, pp. 359–388.

[Kuz+23] Petr Kuznetsov et al., « Permissionless and asynchronous asset transfer », *in*: *Distributed Computing* 36.*3* (2023), pp. 349–371.

[LF82] Leslie Lamport and Michael Fischer, *Byzantine generals and transaction commit protocols*, tech. rep., Technical Report 62, SRI International, 1982.

[LSP82] Leslie Lamport, Robert E. Shostak, and Marshall C. Pease, « The Byzantine Generals Problem », *in*: *ACM Trans. Program. Lang. Syst.* 4.*3* (1982), pp. 382–401, DOI: 10.1145/357172.357176, URL: https://doi.org/10.1145/357172.357176.

[Luu+16] Loi Luu et al., « A Secure Sharding Protocol For Open Blockchains », *in*: *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, CCS '16, Vienna, Austria: Association for Computing Machinery, 2016, pp. 17–30, ISBN: 9781450341394, DOI: 10.1145/2976749.2978389, URL: https://doi.org/10.1145/2976749.2978389.

[Mon09] D.C. Montgomery, *Introduction to Statistical Quality Control*, en, 6th, New York: John Wiley, 2009, p. 96.

[MRV99] S. Micali, M. Rabin, and S. Vadhan, « Verifiable random functions », *in*: *40th Annual Symposium on Foundations of Computer Science (Cat. No.99CB37039)*, 1999, pp. 120–130, DOI: 10.1109/SFFCS.1999.814584.

[Nak08a] Satoshi Nakamoto, *Bitcoin: A Peer-to-Peer Electronic Cash System*, 2008, URL: https://bitcoin.org/bitcoin.pdf.

[Nak08b] Satoshi Nakamoto, *Bitcoin: A peer-to-peer electronic cash system*, 2008.

[NK22] Oded Naor and Idit Keidar, « On Payment Channels in Asynchronous Money Transfer Systems », *in*: *Proc. 36th International Symposium on Distributed Computing (DISC'2022)*, vol. 246, LIPIcs, Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2022, 29:1–29:20.

[PSL80] Marshall C. Pease, Robert E. Shostak, and Leslie Lamport, « Reaching Agreement in the Presence of Faults », *in*: *Journal of the ACM* 27.*2* (1980), pp. 228–234.

[Ray18] Michel Raynal, *Fault-Tolerant Message-Passing Distributed Systems - An Algorithmic Approach*, Springer, 2018, ISBN: 978-3-319-94140-0, DOI: 10.1007/978-3-319-94141-7.

[RSG98] Michael G. Reed, Paul F. Syverson, and David M. Goldschlag, « Anonymous connections and onion routing », *in*: *IEEE J. Sel. Areas Commun.* 16.*4* (1998), pp. 482–494.

[Ruf+22] Tim Ruffing et al., « ROAST: Robust Asynchronous Schnorr Threshold Signatures », *in*: *Proc. ACM SIGSAC Conference on Computer and Communications Security (CCS'22)*, ACM, 2022, pp. 2551–2564.

[Sau+20] Geoffrey Saunois et al., « Permissionless Consensus based on Proof-of-Eligibility », *in*: *2020 IEEE 19th International Symposium on Network Computing and Applications (NCA)*, 2020.

[Set20] Srinath T. V. Setty, « Spartan: Efficient and General-Purpose zkSNARKs Without Trusted Setup », *in*: *Proc. 40th Annual International Cryptology Conference (CRYPTO'20)*, vol. 12172, Lecture Notes in Computer Science, Springer, 2020, pp. 704–737.

[Sha49] C. E. Shannon, « Communication theory of secrecy systems », *in*: *The Bell System Technical Journal* 28.*4* (1949), pp. 656–715, DOI: 10.1002/j.1538-7305.1949.tb00928.x.

[Son+20] Alberto Sonnino et al., « Replay Attacks and Defenses Against Cross-shard Consensus in Sharded Distributed Ledgers », *in*: *2020 IEEE European Symposium on Security and Privacy (EuroS&P)*, 2020, pp. 294–308, DOI: 10.1109/EuroSP48549.2020.00026.

[SS01] Douglas R. Stinson and Reto Strobl, « Provably Secure Distributed Schnorr Signatures and a (t, n) Threshold Scheme for Implicit Certificates », *in*: *Proc. 6th Australasian Conference on Information Security and Privacy (ACISP'01)*, ed. by Vijay Varadharajan and Yi Mu, vol. 2119, Lecture Notes in Computer Science, Springer, 2001, pp. 417–434.

[Tea17] The ZILLIQA Team, *The ZILLIQA Technical Whitepaper*, 2017, URL: https://docs.zilliqa.com/whitepaper.pdf.

[Won+23] Harry W. H. Wong et al., « How (Not) to Build Threshold EdDSA », *in*: *Proc. 26th Int'l Symposium on Research in Attacks, Intrusions and Defenses (RAID'23)*, ACM, 2023, pp. 123–134.

[WW19] Jiaping Wang and Hao Wang, « Monoxide: Scale out Blockchains with Asynchronous Consensus Zones », *in*: *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*, Boston, MA: USENIX Association, Feb. 2019, pp. 95–112, ISBN: 978-1-931971-49-2, URL: https://www.usenix.org/conference/nsdi19/presentation/wang-jiaping.

[Yu+20] Guangsheng Yu et al., « Survey: Sharding in Blockchains », *in*: *IEEE Access* 8 (2020), pp. 14155–14181, DOI: 10.1109/ACCESS.2020.2965147.

[ZMR18] Mahdi Zamani, Mahnush Movahedi, and Mariana Raykova, « RapidChain: Scaling Blockchain via Full Sharding », *in*: *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, CCS '18, Toronto, Canada: Association for Computing Machinery, 2018, pp. 931–948, ISBN: 9781450356930, DOI: 10.1145/3243734.3243853, URL: https://doi.org/10.1145/3243734.3243853.

Appendix A

Implementation of the Quasi-Anonymous Asset Transfer (QAAT)

A.1 Implementations of the Schemes of Section 4.2.1

A.1.1 Preliminaries

A.1.1.1 Groups of unknown order

Our choice of UA [BBF19] implementation requires the use of groups of unknown order (as defined in [DK02]) 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, [DGS22] 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.

A.1.1.2 Notations

In the following sections, we use the notations shown in Table A.1.

A.1.2 Accumulator

For our accumulator (specified in Section 4.2.1.3), we opt for the trustless extension of universal RSA accumulators in groups of unknown order from [BBF19]. While RSA

Notation	Meaning
$a\|\|b$	Concatenation of two bit-strings a and b
$\vec{a}$, a_i	Vector of arbitrary elements. We refer to its i-th element as a_i.
$[n]$	The set of integers $\{1, .., n-1\}$
$x \xleftarrow{\$} S$	Sampling a random element $x \in S$ uniformly
$\mathsf{Primes}(\lambda)$	The set of primes smaller than 2^λ
$\mathsf{Primes}(a, b)$	The set of primes in the interval $[2^a, 2^b - 1]$.
H_{Primes}	A pseudorandom hash-to-prime function $H_{\mathsf{Primes}} : \{0,1\}^* \to \mathsf{Primes}(\lambda)$
$a \mid b$, $a \nmid b$	This stands for "a divides b" and "a does not divide b" respectively
$\mathcal{G} = \{\mathbb{G}, g\}$	A descriptor $\mathcal{G}$ of a group of unknown order $\mathbb{G}$ with a generator $g \in \mathbb{G}$

Table A.1 – Notations for the cryptographic primitives of Appendix A.1.

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 7.

A.1.2.1 Hashing to primes

An RSA accumulator can only contain prime numbers. To overcome this limitation, one must use a hash-to-prime function H_{Primes} that 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.

A.1.2.2 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

|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 A.2 – Performance of accumulator proof generation and verification for various set sizes $|S|$.

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 A.2 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.

A.1.3 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 concise (constant-sized digest) commitment scheme with trustless setup. For example, one could use Pedersen commitments in groups of unknown order (or elliptic curves [FG17]), which are unconditionally hiding and computationally binding. Conversely, other schemes such as ElGamal commitments [Gam84] 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.

A.1.4 Transparent zk-SNARK with time-optimal prover

For our zk-SNARK scheme, we choose Spartan [Set20], which is both transparent (trustless setup) and prover time-optimal.

A.2 A consensus-free quorum-based Agreement Proof implementation

In this section, we provide an implementation of the agreement proof scheme of Section 4.2.2 that does not rely on consensus, is succinct, and has a $O(\lambda)$ storage requirement. Our AP implementation leverages threshold digital signatures (see Appendix A.2.1) to create constant-size agreement proofs σ (*i.e.*, quorums of signatures) and verifying them without the set of all public keys.

A.2.1 Threshold signatures

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

Various asymmetric signature schemes can be transformed into threshold signature schemes [Das+23; KG20; Ruf+22; SS01; Won+23]. 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 global public key. For example, Roast [Ruf+22] implements an asynchronous threshold Schnorr signature scheme that supports arbitrary choices of t, guarantees unforgeability against a dishonest majority ($n > 2t$), and guarantees that as long as $t + 1$ honest signers partici-

pate, a valid signature is outputted the remaining $n - t - 1$ signers are malicious and try to prevent the creation of the signature.

Succinctness of threshold signatures. The most efficient threshold signature implementations (*e.g.*, [Ruf+22]) all guarantee that, for a fixed λ and any number of signers k, the size and verification time of a threshold signature is equivalent to that of a single intermediary signature. Therefore, these implementations are succinct.

Constant size of the verification data in threshold signatures. Unlike classical signature schemes, or even multi-signature schemes such as BLS [BLS04], which require knowing all the identities of the signers of a quorum of signatures, threshold signature schemes only require knowing one global public key for the system for verifying a quorum of signatures. In other words, given the maximum number of signers n in the system, instead of having quorums of signatures of size $O(\lambda + n)$ bits with classic or BLS-style schemes (as these quorums must also include the whole set of signers, that can be encoded in a bitvector of at least n bits), threshold signatures can produce quorums of signatures of size $O(\lambda)$ bits (once a quorum has been generated, the individual identities of the signers are not needed for its verification).

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

A.2.2 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 $O(\lambda)$ bits per correct process.

Algorithm 8 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), $\vec{sn}$ the vector containing at each index $j \in [n]$ the sequence number of process p_j (sn_j is initiated at 0).

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

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

When p_i receives a QUORUMINIT message from a process p_j, it first checks that the provided initialization signature is valid (line 15) and that the provided *data* satisfies the AP predicate (line 16). Then, p_i waits for the provided sn_j to be the next one (in FIFO order) to be processed (line 17). After that, p_i produces its intermediary signature (line 18) and sends it to p_j in a QUORUMSIG message (line 19). Finally, p_i updates the sequence number of p_j in $\vec{sn}$ with its new sn_j (line 20).

When p_i receives a QUORUMSIG message, it saves the provided intermediary signature if it is valid and if p_i has not already produced an AP for the payload (line 22).

Let us remark that, like in our asset transfer algorithm of Algorithm 7, we impose the fact that the size of the sn_i variable must be constant. Let us also note that this implementation of the Agreement Proof scheme has a message complexity of only $O(n\lambda)$ (where we assume for simplicity that the size of the v and *data* parameters of the ap_prove() operation are of constant size).

A.2.3 Proof of Algorithm 8

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 A.2.1 (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 $sn_i{}^{th}$*

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

```
1  initialize(sigs_i ← ∅; v_i ← ⊥; s⃗n s.t. sn_j ← 0, j ∈ [n];)
2  operation ap_verify(σ_j, v, sn'_j, j) is
3      return { true  if σ_j is a valid (⌊(n+t)/2⌋+1)-threshold signature for ⟨v, sn'_j, j⟩,
               { false otherwise.
4  operation ap_prove(v, data) is
5      if P_A(data) = false then return abort;
6      v_i ← v;                                          ▷save v for condition at line 22
7      sn_i ← sn_i + 1;                                  ▷increment sn_i
8      sig_i ← initialization signature of ⟨v, data, sn_i, i⟩ by p_i;
9      broadcast QUORUMINIT(v, data, sn_i, i, sig_i);
10     wait (sigs_i has strictly more than (n+t)/2 intermediary signatures for ⟨v, sn_i, i⟩);
11     σ_i ← aggregation of sigs_i into a (⌊(n+t)/2⌋+1)-threshold signature for ⟨v, sn_i, i⟩;
12     sigs_i ← ∅; v_i ← ⊥;                              ▷flush temporary data
13     return ⟨σ_i, sn_i⟩.
14 when QUORUMINIT(v, data, sn'_j, j, sig_j, o_j) is received do
15     if sig_j is not a valid initialization signature for ⟨v, data, sn'_j, j⟩ by p_j then return;
16     if P_A(data) = false then return;
17     wait sn_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     sn_j ← sn'_j;                                     ▷update the j-th position of s⃗n
21 when QUORUMSIG(j, sig_j) is received do
22     if sig_j is a valid intermediary signature for ⟨v_i, sn_i, i⟩ by p_j and AP σ_i for v_i not
         already produced by p_i then sigs_i ← sigs_i ∪ {sig_j}.
```

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 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.ap_prove$(v, \star)/\sigma_i$ execution. □

Lemma A.2.2 (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.ap_verify$(\sigma_i, v, sn_i, i) = Obj_A$.ap_verify$(\sigma'_i, v', sn_i, i) =$ `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, $|A| > \frac{n+t}{2} < |B|$.[1] We thus have $|A \cap B| = |A| + |B| - |A \cup B| > 2\frac{n+t}{2} - |A \cup B| \geq 2\frac{n+t}{2} - n = t$. Therefore, at least one correct process p_j must belong both to A and B, and must have signed both $\langle v, sn_i, i \rangle$ and $\langle v', sn_i, i \rangle$ for $v \neq v'$. But by the fact that correct processes produce intermediary signatures for some sn_i at line 18 and right after update the i-th sequence number in their vector $\vec{sn}$ 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 A.2.3 (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 $P_A(\star, data, sn_i)/$`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 $P_A(\star, data, sn_i)/$`true`. □

Lemma A.2.4 (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*

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

data, if $P_A(\star, data, sn_i) =$ `true` *then* $r = \sigma_i$*, where* σ_i *is a valid AP for* v *at* sn_i *from* p_i*. If* $P_A(\star, data, sn_i) =$ `false`*, then* $r =$ `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 $sn_i{}^{\text{th}}$ invocation of Obj_A.ap_prove(..). If $P_A(\star, data, sn_i) =$ `false`, then p_i passes the condition at line 5 and return `abort`. If $P_A(\star, data, sn_i) =$ `true`, p_i continues the execution, produces an initialization signature at line 8, broadcasts a QUORUMINIT message at line 9 and wait for a quorum of signatures from the system at line 10.

Upon receiving this QUORUMINIT message, each correct process p_j passes the conditions at line 15 (as p_i has correctly generated its initialization signature sig_i) and at line 16 (as p_i has verified the satisfaction of P_A by *data* at line 5), and then wait that the received sn_i is the next in FIFO order to be processed at line 17. As p_i uses its sequence numbers in FIFO order, then p_i has necessarily sent a QUORUMINIT message for each $sn'_i \in [1..sn_i]$, and p_j will eventually receive all these QUORUMINIT messages. By induction, p_j will pass the wait statement at line 17 for each $sn'_i \in [1..sn_i]$, because if $sn'_i = 1$, the condition line 17 is satisfied as $\vec{sn}$ is initialized at line 1 to 0 at index i, and if $sn'_i > 1$, then p_j will eventually replace the $sn'_i - 2$ by $sn'_i - 1$ at line 20. Therefore, all correct processes, which are at least $n - t$, will eventually pass the wait statement at line 17. Given the system assumption $n > 3t$, we have $n - t = \frac{2n-2t}{2} > \frac{n+3t-2t}{2} = \frac{n+t}{2}$. Therefore, strictly more than $\frac{n+t}{2}$ (or, equivalently, at least $\lfloor \frac{n+t}{2} \rfloor + 1$) 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 22, 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. □

A.2.3.1 Consensus-freedom of Algorithm 8

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

A.2.3.2 Succinctness of Algorithm 8

The succinctness of Algorithm 8 comes from the succinctness of threshold signatures: as shown in Appendix A.2.1, the best implementations of threshold signatures guarantee that the size and verification time of threshold signatures (and therefore also the agreement proofs σ_i produced by Algorithm 8) are equivalent to that of a single intermediary signature. Therefore, Algorithm 8 is succinct.

A.2.3.3 $O(\lambda)$ storage cost of Algorithm 8

The $O(\lambda)$ storage of Algorithm 8 comes from its local variables: the $sigs_i$ set and v_i values are emptied at the end of each ap_prove execution (line 12), and by definition, sn_i is constant-size. Therefore, each correct process only stores $O(\lambda)$ bits in Algorithm 8.

A.2.3.4 $O(n\lambda)$ communication cost of Algorithm 8

The $O(n\lambda)$ overall communication cost of Algorithm 8 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 8 is $O(n(\lambda+\log n))$. However, as we assume that $\lambda = \Omega(\log n)$ (see Section 3.3.2, ¶ *Security parameter* λ), the previous asymptote simplifies to $O(n\lambda)$ bits sent overall.

A.3 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 A.1).

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

1. Assume public knowledge of $\mathsf{genesis\_data} \leftarrow \bigcup_{i=1}^{n} \langle pk_i, A_i, bal\_c_i \rangle$, 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 ($\mathtt{init}_i$).
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 8) that will serve as the initial agreement proof of p_i for each $v_i = \langle A_i, bal\_c_i \rangle$.
5. Each process p_i can now safely delete genesis_data and output its initial parameters $\langle A_i, bal\_c_i, \sigma_i, bal\_o_i \rangle$.

A.4 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 4.2.6.

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 7. 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 [KST22] or Sangria [GWC19] 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.

COLLEGE DOCTORAL BRETAGNE
MATHS, TELECOMS
INFORMATIQUE, SIGNAL
SYSTEMES, ELECTRONIQUE

Université de Rennes

Titre : Vers des systèmes distribués de transfert d'argent plus évolutifs et préservant la vie privée

Mot clés : Systèmes distribués, Tolérance aux fautes, Cryptographie, Privacité, Transfert d'argent, Asynchronie.

Résumé : Depuis 2018, la technologie blockchain a vu émerger de nombreuses applications, allant de la crypto-monnaie aux systèmes de santé. La plupart des blockchains existantes adoptent un modèle de réplication complète. D'un point de vue juridique, la nature entièrement répliquée des blockchains signifie que les données personnelles sont susceptibles d'être stockées sur des nœuds répartis dans différents pays. D'un point de vue technique, la réplication complète offre une bonne tolérance aux pannes, mais au détriment de la mise à l'échelle. Il est nécessaire de développer des solutions qui peuvent garantir la tolérance aux pannes avec des niveaux de réplication plus raisonnables, tout en protégeant la vie privée et en évitant les conflits avec les réglementations.

Pour répondre à ces problèmes, nous proposons deux systèmes. Le premier est basé sur le partitionnement horizontal (sharding) de la blockchain afin de mieux répartir les coûts de stockage et de traitement des données entre des sous-ensembles de pairs. Le second ne repose pas sur le consensus. Il peut donc effectuer des transactions indépendantes simultanément. Cette solution introduit également un ensemble de primitives cryptographiques dont la combinaison permet d'anonymiser les échanges de données des utilisateurs et de vérifier leur légitimité, sans révéler ni stocker de données sensibles.

Title: Towards more scalable and privacy-preserving distributed asset transfer systems

Keywords: Distributed computing, Fault-tolerance, Cryptography, Privacy, Asset transfer, Asynchrony.

Abstract: Since 2018, blockchain technology has seen the emergence of numerous applications, spanning from cryptocurrency to healthcare systems. Most existing blockchains adopt a full replication model. From a legal perspective, the fully replicated nature of blockchains means that personal data is likely to be stored on nodes distributed across different countries. From a technical perspective, full replication provides good fault tolerance at the cost of scalability. Therefore, we must develop solutions that can ensure fault-tolerance with more reasonable levels of replication, while protecting privacy and avoiding clashes with regulatory laws.

To address these issues, we propose two systems. The first is based on horizontal partitioning (sharding) of the blockchain to better distribute the costs of data storage and processing among subsets of peers. The second doesn't rely on consensus. It can therefore independent transactions concurrently. Furthermore, it uses a set of cryptographic primitives to anonymize users' data exchanges and verify their legitimacy, without revealing or storing sensitive data.