Consensus Byzantin et blockchain : Modèles unifiés et nouveaux protocoles

Antoine Durand

INSTITUT POLYTECHNIQUE DE PARIS

NNT : 2021IPPAT025

Thèse de doctorat

TELECOM Paris

IP PARIS

Byzantine consensus and blockchain : Models unification and new protocols

Thèse de doctorat de l'Institut Polytechnique de Paris
préparée à Télécom Paris

École doctorale n°626 Institut Polytechnique de Paris (EDIPP)
Spécialité de doctorat: Informatique

Thèse présentée et soutenue à Palaiseau, le 24 Novembre, par

ANTOINE DURAND

Composition du Jury :

Petr Kuznevtsov Professeur, Télécom Paris (INFRES)	Président
Maria Potop-Butucaru Professeure, Sorbonne Université (LIP6)	Rapportrice
Pierre Jouvelot Chercheur, Mines-Paristech (CRI)	Rapporteur
Joaquin Garcia-Alfaro Professeur, Télécom SudParis (LINCS)	Examinateur
Sara Tucci-Piergiovanni Cheffe de laboratoire, CEA LIST	Examinatrice
Emmanuelle Anceaume Directrice de recherche, CNRS (IRISA)	Examinatrice
Gérard Memmi Professeur, Télécom Paris (INFRES)	Directeur de thèse

Acknowledgements

This thesis has been supported by IRT-SystemX and Atos-Bull

I am thankful to all the great people that supported me directly or indirectly during my PhD.

First of all, I had the chance of having a wonderful thesis director, Prof. Gérard Memmi. I would also like to thank my friendly colleagues, Kei Brousmiche, Nicolas Heulot, Omar Dib, Khalifa Toumi, Hanna-Mae Bisserier, Lucas Benmouffok, Natkamon Tovanich and all the other people from the BST team and IRT-SystemX.

I am especially grateful to David Leporini, Guillaume Hebert and Thomas Domingos from Atos for the numerous interesting discussions and continuous help during this thesis.

My thanks goes to all the researchers I had the chance to interact with, and particularly to Emmanuelle Anceaume and Romaric Ludinard for their invaluable collaboration, which gave shape to this thesis.

I am grateful to all the people from the BART research team; in particular Petr Kuznetsov, Matthieu Rambaud, Emilio J. Gallego Arias, thanks to whom I really became part of the research community.

Many thanks goes to the reviewers, Maria Potop-Butucaru and Pierre Jouvelot, who went beyond their role to help improve this thesis, and as well as all the jury members.

Finally I would like to mention Louis Martin-Pierrat, Matthieu Regueira, Loïc Bryl and Marianna Baziz for their help and fruitful discussions.

List of Abbreviations

ABC	Atomic BroadCast
BA	Byzantine Agreement
BFT	Byzantine Fault Tolerance
(B/V/W/S)C	(Binary / Vector / Weak / Strong) Consensus
CRS	Common Reference String
DHT	Distributed Hash Table
GST	Global Stabilisation Time
IoT	Internet of Things
PKI	Public Key Infrastructure
PoS	Proof of Stake
PoW	Proof of Work
(T)RBC	(Terminating) Reliable BroadCast
SMR	State Machine Replication
UTXO	Unspent Transaction Output
VRF	Verifiable Random Function

Résumé

Avec l'avènement récent du Bitcoin et des algorithmes basés sur la blockchain, il y a eu un regain d'intérêt autour de l'implémentation de protocoles de consensus dans des domaines applicatifs tels que la finance, la santé, la chaîne logistique, la traçabilité, l'internet des objets, etc. Les contraintes de conception de protocoles capables de supporter de telles applications sont très strictes, et incluent notamment la capacité à rejoindre et quitter dynamiquement le système, à tolérer la présence de participants malveillants (fautes Byzantines), et à passer à l'échelle en fonction du nombre de participants sans surcharger le réseau. Simultanément, l'attention portée à ces protocoles a été une source de confusion, submergeant un sujet déjà vaste et complexe avec des assertions imprécises et une terminologie variable, limitant ainsi la possibilité de fonder des discussions de design sur des bases bien formelles. Dans cette thèse, nous visons à améliorer cet état de fait de deux manières complémentaires.

Premièrement, nous proposons une unification des fondamentaux de la blockchain, grâce à une formalisation cohérente des modèles, spécifications avec réductions, et théorèmes d'impossibilité d'intérêt pour les protocoles de consensus tolérant les fautes Byzantines. Plus précisément, nous capturons différents niveaux de synchronie réseau et d'hypothèses cryptographiques, nous formalisons certaines variantes du consensus et du broadcast Byzantin, et nous recadrons les théorèmes qui excluent l'implémentation d'une spécification dans un ensemble de modèles. En particulier, nous faisons l'argument qu'Atomic Broadcast est une spécification appropriée pour les protocoles de blockchain, et que le choix d'une définition appropriée pour la variante Byzantine du consensus devrait faire l'objet de considérations supplémentaires. Au-delà de l'unification des résultats existants, notre formulation généralise les modèles et les théorèmes d'impossibilité existants pour couvrir de manière transparente le cas des protocoles basées sur la blockchain, ce qui inclut notamment le mécanisme de preuve de travail (Proof-of-Work), chose manquante jusqu'à présent. Grâce à cette approche systématisée, nous sommes en mesure de comparer les

modèles de plusieurs blockchains de référence malgré leurs différences fondamentales, ainsi que de procéder à une évaluation fine de leurs caractéristiques de performance. En effet, de telles comparaisons ont fait l'objet d'un intérêt de recherche significatif, bien qu'elles ne soient généralement pas fondées sur des bases formelles.

Ensuite, nous faisons une proposition de blockchain, StakeCube. La sécurité de StakeCube est basée sur le modèle de preuve d'enjeu (Proof-of-Stake), et sa capacité de passage à l'échelle repose sur un principe du partitionnement (sharding), qui est mis en œuvre par une table de hachage distribuée, PeerCube [ALRB08]. StakeCube s'appuie sur un système à deux niveaux où les shards exécutent un protocole de consensus pour gérer leur état local, et nous tirons avantage d'un consortium de shards pour exécuter un protocole de consensus d'ensemble qui détermine l'extension de la chaîne de blocs. Cette approche permet de nouveaux compromis sur la distribution du coût en communication grâce à une taille de shard et de consortium configurable, contrastant ainsi avec des conceptions plus simples basées sur un seul nœud leader ou comité de nœuds. Nous avons également implémenté une version restreinte de StakeCube et évalué ses performances, validant ainsi ses propriétés théoriques et notamment celle de passage à l'échelle. En effet, comme StakeCube troque la preuve de travail contre la preuve d'enjeu sans sacrifier le passage à l'échelle, ce protocole est particulièrement bien adapté aux applications IoT. Pour mieux démontrer cet aspect, nous avons implémenté une application IoT de place de marché d'énergie dans StakeCube et nous avons pu tester avec succès sa viabilité lorsque exécuté sur un matériel limité (Raspberry Pi Zero) dans un réseau de 500 nœuds.

Abstract

With the recent advent of Bitcoin and blockchain-based algorithms, there has been a renewed interest around the implementation of agreement protocols within applicative fields, such as finance, health, traceability, the internet of things, etc. The design constraints for protocols that are able to support such applications are very tight, including notably the ability to dynamically join and leave the system, the tolerance of malicious participants (a.k.a. Byzantine faults), and the ability to scale with the number of participants while avoiding network overload. Simultaneously, the attention to these protocols has been a source of misunderstanding, flooding an already large and complex subject with vague claims and different terminology, thus limiting the possibility to formally discuss these design constraints. In this thesis, we aim to improve this state of affairs in two ways.

First, we make a unifying view of the blockchain landscape through a consistent formalisation of models, specifications with reductions, and impossibility theorems of interest for Byzantine tolerant agreement protocols. Specifically, we capture varying levels of network synchrony and cryptographic assumptions, we formalise a few BFT variants of consensus and broadcast, and we reframe the thoerms that exclude the implementation of a specification under a range of model. Importantly, we argue that Atomic Broadcast is a suitable specification for blockchain protocols, and that more consideration should be spent towards choosing an appropriate definition for the BFT variant of consensus. Beyond the unificaiton of existing results, our formulation generalises models and impossibility theorems to seamlessly cover new blockchain-based design, including protocols based on Proof-of-Work. Thanks to this systemizing approach, we are able to compare the model of several prominent blockchains despite their fundamental differences, as well as making a fine-grained assessment of their performance characteristics. Indeed, such comparisons are the subject of a significant research interest, but they typically are not based on formal grounds.

Then, we make a proposal for a scalable blockchain, StakeCube. StakeCube's security

is based on the Proof-of-Stake model, and its scalability relies on the sharding paradigm, implemented through a distributed hash table named PeerCube [ALRB08]. StakeCube relies on a two-tierd system where shards run agreement protocols to manage their own state, and we take advantage of a consortium of shards to another agreement protocol driving the addition of blocks to the chain. This approach offers new trade-offs for distributing communication load thanks to a configurable shard and consortium size, in contrast to simpler designs based on a single leader node or committee. We also implemented (a restricted version of) StakeCube and evaluated its performance, thus validating its scalability property. Notably, because StakeCube trades Proof-of-Work for Proof-of-Stake without sacrificing scalability, it is particularly well suited for IoT applications. To further demonstrate this aspect, we implemented an energy marketplace IoT application in StakeCube and were able to successfully test its viability when executed on limited hardware (Raspeberry Pi Zero) with 500 nodes.

Contents

Chapter 1

Introduction

The blockchain ecosystem has a remarkable history, as it was developed by individuals, industrials, and academics concurrently.

Bitcoin and blockchain technology in general experienced a surge of interest in the recent years, first as a new financial paradigm [Swa15], then as a distributed tool with promising applications for a wide range of sectors [JS19], such as health, supply chain [LH19], identity management [DP18, SP18b], transportation [AGMS20], energy [BAD+18], smart cities [SP18a], and the internet of things [CK19, PTM+18, BBG+17].

Such promises naturally attracted a large number of competing enterprises and projects, thus each of them attempting to differentiate from each other. Although this can boost innovation, it also encouraged the creation of new terminology and abstractions, even when unnecessary. As a result, the overall blockchain ecosystem is quite fragmented, which is especially visible when it comes to having a common terminology.

On the academic side, there is a proficient literature on distributed systems and most particularly around the notion of consensus that largely predates blockchain technology [NC17]. There are numerous variations on the problem definition, models, building blocks and assumptions that make the scope of the topic already too large to be considered in its entirety [GR06, Lyn96, Ray13]. More recently, a large research effort has been initiated to analyse and improve blockchain technology in itself and its applications [GK20, DBD+18, Shi20, Gra20]. This research effort has also spawned an impressive number of research directions and concurrent works.

This proliferation contrasts with the widely recognized difficulty of designing and proving secure distributed cryptographic algorithms. In turn, this complicates the task

of making a correct and thorough assessment of byzantine tolerant protocols, including blockchain.

As a result, the blockchain landscape can be felt quite confusing, lacking a consistent formalisation [Pot20] and numerous sources using terminology with different meaning [GS18, NC17]. In practice, one should consider the meaning of the word "blockchain" depending on context. To this regard, we provide here first a few definitions to make a rough disambiguation:

- *Blockchain data structure*: An ordered set of data, commonly named *blocks*. Each block includes the hash of the previous one, thus chaining the whole set. This is the most literal definition, from which emerged the word "block-chain".

- *Blockchain protocols*: This definition aims at considering the notion of blockchain as a distributed system, however it is one most prone to interpretations. On the one hand, it can be used to designate any distributed algorithm that uses a blockchain data structure, regardless of its goal. On the other hand, it can refer to the algorithm that aims at maintaining a consistent state across multiple participants.

- *Blockchain applications*: Because a blockchain protocol is not an end in itself, there is an application making use of the underlying blockchain protocol to achieve a specific, arbitrary goal. This encompasses cryptocurrencies, but also frameworks, platforms and systems that make use of a modular "engine", *i.e.*, a protocol.

- *Blockchain technology*: A wider term to designate anything related to blockchain data structure, protocols, applications, including their environment and ecosystem. The name *distributed ledger technology* is sometimes equivalently used.

In this thesis, we will be most concerned about blockchain protocols. More specifically, we will favor the approach that defines blockchain protocols in terms of what they are trying to achieve rather than how they are implemented. To remove ambiguity, it is possible to use the term "distributed ledger" for the former, and "blockchain-based protocols" for the latter.

Our quest for disambiguation does not end here. Importantly, the trustlessness aspect is deemed essential to blockchain protocols. Technically, this translates to a security property, *i.e.*, the ability to function even when a subset of participants are misbehaving in

arbitrary, malicious ways, also known as Byzantine faults. Such protocols are therefore called Byzantine Fault Tolerant (BFT) protocols. This property implies that each participant does not have to trust any other participant to correctly execute the protocol, since any of them may be malicious. Hence, asserting that blockchains are trustless should amount to requiring this property in the definition. Crucially, this trustlessness aspect cannot be absolute either, as some degree of trust in the participants as a whole must be present. This is apparent through the necessity of an assumption limiting the proportion of malicious participants in the network, *e.g.*, by requiring an "honest (super)[1] majority". However, several prominent blockchain applications only tolerate crashing participants (not malicious ones) [ABB+18], and therefore cannot qualify as blockchains in that specific sense. In this thesis, we will focus on trustless protocols, although we remain neutral on whether the canonical "blockchain" definition should require it.

Obviously, the given definition remains very vague; there is no precision on what "a consistent state across multiple participants" is. An appealing approach is to define consistency such that the participants globally emulate a single node running a centralized application. This leads to the State Machine Replication (SMR) paradigm [Sch90b], where an application is represented as a state machine, and each participant answers queries from clients as if they were all forwarded to a single centralized state machine. In that sense, blockchains protocols can be regarded as a way to transform a trusted third party into a weakly trusted set of nodes. To achieve this, each participant locally maintains a copy of the state machine, and the protocol makes them coordinate to execute the same sequence of inputs, resulting in the same sequence of local state for all nodes. Consensus can roughly be seen as the "one off" variation of SMR, in which participants only agree on their input once. More generally, it is considered to be the theoretical core that quantifies the amount of synchrony required by a distributed algorithm [Her91]. In fact, it comes with no surprise that the possibility to reach consensus at all (and by extension, SMR) heavily depends on the synchronicity of the network itself. So far, most academic work on blockchain has adopted the State Machine Replication paradigm, or an equivalent formulation [AM17]. Nevertheless, much work has been devoted to satisfy properties that go further than what is strictly required by SMR, to improve usability and efficiency. Examples include the possibility to have nodes dynamically joining or leaving, smaller communication cost, optimised latency and memory footprint, stronger and finer-grained threat model, *etc.*

[1] A supermajority of nodes contains at least two thirds of the network.

One may think that being able to relate blockchain terminology with its academic counterpart would be sufficient to let someone search by himself what is possible -or not- about their favorite blockchain concept. However the fact is that there is much more work involved to be able to make precise statements without doubt. For instance, we mentioned the necessity of some form of honesty assumption for nodes. Indeed one may regularly encounter statements such as "consensus is solvable if and only if less of one third of participants are misbehaving", but this is far from the complete picture. First, we will point out in this thesis that it implicitly assumes some specific structure on misbehaving of nodes, which is ill-suited to a generic description of blockchain protocol, *e.g.*, Nakamoto consensus, the protocol that underlies Bitcoin [Nak08a]. But more importantly, this claim is not true in the case of a strongly synchronous network with digital signatures. Furthermore, one should pay attention to the definition of consensus used, because as we will see they are not all strictly equivalent. For instance, the previous statement is not true in the case of strong multi-valued consensus. And as a final detail, the cited statement also depends on how nodes misbehave, *e.g.*, maliciously or by crashing. This example shows that such theorems can be quite misleading if taken without context. This is why we set out to draw a more complete picture, where results are stated explicitly with full models and definitions.

It took some time to have Bitcoin's properties fully understood [GKL15, GKL17], and it is easy to see why they could seem surprising to a distributed systems researcher. The protocol works in a network orders of magnitude larger than what was previously possible, with dynamic participants that do not even need to register[2], all while seemingly sidestepping impossibility results on the number of Byzantine participants tolerated, such as the one in the example above. This thesis is making the point that the model, properties and working principles of Bitcoin are not so different from the existing literature. We argue that Proof-of-Work (PoW) in essence only provides a novel way to count and authenticate nodes. This change, although somewhat straightforward, is quite fundamental, and the specificities of Proof-of-Work-based authentication is what gives Bitcoin its surprising properties. In a similar manner, Proof-of-Stake blockchains count and authenticate nodes with respect to their stake, which has the additional advantage of being maintained by the protocol itself. In fact, as we will see in Chapter 2, blockchains in general cannot deal with arbitrary numbers of malicious nodes, and therefore needs a *trust assumption* to limit their

[2] Although we will see in Section 3.2 that this statement is more nuanced.

presence among honest nodes. In other words, they are not Sybil-resistant and need to assume the existence of a Sybil resistant resource associated with nodes. This observation motivates the view that for any X, "Proof-of-X" naturally designates protocols whose Sybil resistance is based on the scarcity of the resource X, and nodes' ability to prove ownership of X. This definition however is not entirely in line with the term usage, especially within commercial projects. Beyond Proof-of-Work, other Sybil resources include:

- Proof-of-Storage protocols, such as FileCoin[3]. Thanks to a cryptographic proof system [FBGB18] nodes can demonstrate ownership of storage space and thus use it as a Sybil resistant resource. This is especially interesting to implement applications such as decentralized "cloud" storage marketplaces, but less for general-purpose applications.

- Solutions based on Trusted Execution Environments (TEEs), such as Proof-of-Elapsed Time. TEEs are hardware processors that are able to provide a cryptographic proof of their correct execution. This system justifies the assumption that these processors cannot be corrupted, hence providing a particularly strong Sybil resistance. On the other hand, the whole system relies on the correctness of the TEE system.

- "Classical" BFT protocols, *i.e.*, those that use the typical assumption of a bounded proportion of malicious nodes. In this case, the security relies on the difficulty to corrupt a node. However this concept is easily generalizable by adding a "weight" to each participant [GB11b]. Thus, as long as the nodes weights are agreed upon beforehand (*i.e.*, they are part of the nodes identity), the security assumption can be translated into a "majority honest weight" assumption, independently from *how* the weights are defined.

Having related Proof-of-Work blockchains like Bitcoin with existing agreement protocols, the natural follow up question is whether it is possible to leverage that knowledge to lift up roadblocks and unlock new applications. It has been conjectured that the use of blockchains could be highly beneficial within IoT ecosystems, with the appealing promise of a solid infrastructure handling the needs of billions of interconnected devices, integrating trust and security by design [PSAG20]. However interpolating the benefits of a large-scale blockchain such as Bitcoin to an IoT setting is not straightforward [QQM+18b].

[3] `https://filecoin.io`

Indeed, distributing the trust down to the end user means bringing the participating blockchain nodes into the smallest device possible. In turn, this removes the possibility of Proof-of-Work (PoW) blockchains due to their excessive computational cost and the security risk coming with an insufficient total hashrate. The straightforward way to work around this issue is to change the trust assumption in order to remove the need for mining and PoW. Among other alternatives, Proof-of-Stake (PoS) is particularly appealing due to its high versatility and better scalability. Indeed, because PoS blockchains' computational burden mainly consists in generating and verifying signatures, their energy consumption may be considered insignificant. But following the literature on BFT protocols, their ability to scale with the number of nodes is limited, which presents a challenge for IoT applications based on PoS blockchains.

In a nutshell, deterministic protocols have a minimal quadratic communication cost (in the number of nodes) they must bear to achieve agreement for a set of nodes. However it is possible to reduce this cost using probabilistic protocols. Typical designs rely on some kind of election mechanism, where a few nodes are randomly selected to execute communication-heavy tasks in place of others. The selected nodes solve consensus[4], and diffuse the result as the output of the overall protocol. As a result, the communication heavy protocol is run only within a small set of nodes, and the communication cost is dominated only by the linear cost of broadcasting. Importantly, the number of elected nodes can be independent from the total number of participants (*i.e.*, constant), because only the proportion of honest nodes in the elected set matters, and that proportion approaches the one of the overall set of participant as the size of subset grows. Hence, the size of the subset only has to be big enough to guarantee a correct proportion of honest nodes within (with some required probability). In simpler words, the subset size is a security parameter, which has to be high enough to guarantee that the elected subset is representative of the participants.

As a corollary, the overall proportion of honest nodes can only be strictly higher than the one of the subset. If the targeted proportion of the subset is the same as the optimal one for the overall protocol, then it cannot be optimal, only arbitrarily close to optimality.

[4]In general the protocol that they run is not precisely consensus but a specific task that drives agreement for the overall protocol.

1.1 Thesis results

In Chapter 2 we start off with a broad overview of the existing models in the literature.

The first contribution of this thesis is to give a precise and coherent meaning to various definitions and assumptions used in the context of Byzantine consensus and blockchain. Concretely, we formulate in Chapter 3 a framework in which we faithfully capture the most relevant models for blockchain protocols. As a first application we give the definitions of distributed problems related to blockchain and characterise their strength by giving the reductions from each other. We argue that even stating precisely the various definitions for Byzantine consensus is subject to a small amount of interpretation, and that the strength relationships between these definitions are not as clearly laid as they are fundamental. Besides, using our framework, we are the first to make a comparison of such a wide range of protocols based on a common model of execution. This allows us for instance to catch some discrepancies in the terminology used in the literature.

We then extend these results, by giving lower and upper bounds in our model. The lower bounds give the models in which consensus or its variants is not solvable, for instance if the number of faulty nodes is too high. The upper bounds show the existence of protocols. That is, we analyse a few prominent blockchain protocols, giving their execution model within our framework, and evaluating their asymptotic performances in terms of communication complexity and latency. Notably, we find out that that many blockchain protocols including Bitcoin implement State Machine Replication and therefore can be compared on this basis. As a result, we are the first to make a comparison of such a wide range of protocols based on a common model of execution. We are also able for instance to catch some discrepancies in the terminology used in the literature to designate synchronicity assumptions.

Then, in Chapter 4, we present StakeCube, a proposal for a scalable Proof-of-Stake blockchain. StakeCube's scalability is also attained using an election mechanism, instantiated by its sharding structure. Roughly, StakeCube use a Byzantine resistant distributed hash table, PeerCube [ALRB08], to gather participants into shards of constant size. Then, a few number of shards are able to cooperate to create the new block to be appended to the blockchain. PeerCube works by assigning a random, periodically renewed identifier to nodes, which determines the grouping into shards. The shard size and node sojourn time

are computed to prevent Byzantine nodes from subverting the agreement protocols executed within shards. In turn, the common randomness required by PeerCube is generated along each new block. We design a novel agreement subprotocol for StakeCube, tailored for ensuring consistency across shards despite the presence of a limited number of corrupt shards. We implemented a prototype of StakeCube from scratch, and we show results from experimental evaluation of its performance, demonstrating a communication cost in line with theoretical analysis. At last, we validate StakeCube's viability to run large scale IoT applications, by implementing an energy marketplace [BAD+18] on top of StakeCube. In this application, one node running on a Raspberry Pi Zero was able to participate without effort in a network of 500 nodes.

1.2 Co-authored articles

During this thesis I worked on [BDH+18, DHLM19, DAL19, DHMA20, DBD+18]. Furthermore, the Chapter 3 has been submitted for publication.

Chapter 2

Background

In this chapter, we first start by making an overview of the models relevant to blockchain protocols and consensus in the literature, to give an overview of the different concepts we wish to make more precise and relate. This include different security notions, approaches to define specifications, faults, trust assumptions, network and clocks synchrony, and cryptography in general. Then we make a quick tour of the definitions made in the literature to capture the problem solved by blockchain systems.

Terminology Before going further, we start by clarifying some basic common ground.

A *node* is a computational device, generally Turing-equivalent, executing an algorithm. Other synonyms include "machine", "processor", "party" or "player". An *algorithm* has the same meaning as "program". A *distributed environment* is a system including multiple machines or nodes running algorithms, potentially (concurrently) interacting with each other and other components[1]. A *distributed* algorithm is an algorithm fitting for execution in a distributed environment. We use the term *protocol* as a synonym for distributed algorithm. An *execution* or *run* is a description of the evolution of a distributed system for a fixed set of inputs. We call *execution model* the mathematical formalism used to make that description. An *implementation* is an algorithm aiming at satisfying a specification. A *specification* is the set of properties that an algorithm must satisfy. We sometimes use the term *problem* instead, in the sense that a protocol is a solution to the problem of satisfying the properties.

[1] The other components are sometimes called *oracles*.

Execution models Before describing any model, we must first make precise the notion of protocol execution and what it means to have a correct implementation, *i.e.* what we will call the execution model.

Early on, the security of cryptographic primitives has been interpreted as properties of algorithms. Naturally, any specification can be expressed in terms of algorithms properties, and pursuing this idea in a straightforward approach leads to *game-based definitions*. In that case the security is defined in terms of a specific set of executions of the primitive (*i.e.*, the games) that must have a correct result. Most known examples include semantic security, IND-CCA, IND-CPA for encryption, universal, selective or existential forgery for signatures [KL14].

A second approach to security definition emerged from the need to conveniently express fine-tuned privacy requirements [Gol97]. Instead of requiring all implementations to have some abstract properties, specifications can be given in terms of an ideal component, and the implementation must behave similarly to the ideal one. The precise rule to compare the ideal and actual executions varies with the security notion considered, with typical examples including computational indistinguishability, statistical indistinguishability, or equality for perfect security. This naturally expresses the idea that the cryptographic algorithm must leak exactly as much information as the ideal component, a task that has proven to be difficult using ad-hoc properties to be satisfied. Another major advantage of this approach is the possibility to make primitives *composable* [Can01], *i.e.*, having the protocol remaining secure even when the ideal components are swapped with an implementation. This approach is called *simulation-based* or the *ideal-real paradigm*.

With the advent of executions models with malicious participants, many subjects in the literature on distributed systems have largely blended with cryptography. In that regard, the specifications for consensus and other problems were first approached with game-based definitions, where some properties must hold for each protocol execution [LSP82]. Indeed, to this day the primary definition for *e.g.*, consensus [CVNV11] remains expressed as various requirements on the output of nodes for all executions. Concretely, the statement "protocol P satisfies specification S" has can be seen in general as a statements of the form "$\forall e \in EXEC(P), S(e)$", where $EXEC(P)$ is the set of all executions of P, and S is a requirement such as consistency, liveness, validity (see next Section). In the probabilistic case, the same statement can be expressed similarly by making $EXEC$ into a probability

space of executions, and requiring that S holds with a sufficiently high probability[2].

In the following, the specifications of interest for blockchain and consensus will be described with the intent to follow classical game-based definitions. Of course, other models do exist with a different representation of what it means to execute a protocol. Prominently, the *shared-memory* model [Lyn96] is concerned with processors that access the same memory, by opposition with the *message passing* model described in this chapter. With the Heard-of model [CS09], Charron-Bost makes the key observation that separating node failure from network behavior makes the models unnecessarily specific, and thus having specifications purely expressed in terms of received messages is both more adequate and simpler. On the side of the ideal-real paradigm, the Universal Composability framework [Can01] by Canetti defines executions through Turing Machines that are enriched to be able to spawn and interact with new machines. Alternatively, a similar approach but with Task restricted Probabilistic Input/output Automata (task-PIOA) [CCK+08a] has been proposed by Canetti and Lynch, with a notable treatment of long-lived protocols such as blockchains [CCK+08b]. Other frameworks with similar goals includes the Reasctive Simulatability (RSIM) framework [BPW07], the Inexhaustible Interactive Turing Machine (IITM) framework [KTR20], Constructive Cryptography [Mau11].

Our execution model The remaining of this chapter is based on the message passing model, where each node $p \in \Pi$ communicates by exchanging messages. The nodes in Π are Turing-equivalent devices that execute an algorithm P, *i.e.*, the protocol, which determines their interactions within their environment. We assume that local computation costs (*i.e.*, time and memory) are insignificant. A problem or specification is expressed through some properties on the protocol execution. For a more formal execution model, see Section 3.1.

2.1 The adversary

With a formal statement of correctness like $\forall e \in EXEC(P), S(e)$, the entire model of execution is represented by the set (or distribution space) $EXEC$. In a sense, it represents

[2]Because there is no privacy consideration to define consensus, one may argue that the only appeal of simulation-based definitions in this case is composability.

everything that the protocol is guaranteed to withstand, and in real executions we can only assume that reality is correctly modelled by one of these theoretical executions.

However the literature makes prominent usage of the notion of an "adversary" who decides specific events that happen in an execution, such as delaying messages and corrupting nodes. In essence, the adversary is only a construct that describes a set of executions. For instance, the previous statement could be written instead : "$\forall \mathcal{A} \in ADV, S(EXEC(P,A))$", where ADV is the set of all adversarial strategies, and $EXEC(P,A)$ is the execution of P with A's strategy.

It is clear that the difference with the first statement is only a matter of notation, however there are mainly two advantages.

- The formalism with an adversary is more intuitive and can help with reasoning. Typically, sentences of the form "The adversary can..." are generally clearer than "It is possible that...". There is an intuitive appeal to think of a game between the protocol designer and an adversary which can make a specific set of actions at each step of the execution.

- Some security mechanisms may be constructed with some notions of adversary. For instance, a common assumption requires that all adversarial actions are the result of a polynomially bounded Turing machine, therefore giving life to a malicious entity [DY83].

The latter assumption characterizes the computational power of the adversary. Without any assumption on its computational power, we are in the information-theoretic (IT) setting. Even in the IT setting the adversary may still be allowed to break the protocol with a small probability[3]. In this case, the probability must be a negligible function of a global *security parameter* denoted by λ.

The "computational security" setting is obtained when the adversary is polynomial in λ. Note that if the protocol is deterministic and secure *for all* of its executions then it is IT secure, and computational security cannot even be defined because there is no security parameter λ.

Expectedly, the most common use of the different assumptions regarding adversarial power is to take advantage of some cryptographic primitives, such as digital signatures

[3] *Perfect security* requires that probability to be 0.

and hash functions. In conjunction with a trusted setup, they enable protocols to be built in situations where impossibility results would otherwise apply. For instance, the use of digital signatures is a workaround for the fundamental impossibility to distinguish a liar among two nodes [PSL80].

2.1.1 Fault type

The primary objective of a protocol is to give guarantees despite to presence of *corrupt* or *faulty* nodes that do not necessarily follow the protocol instructions. Typically, corrupt nodes are assumed to stay corrupt forever. Although it may seem overly restrictive, this is due to the idea that protocol design should not rely on faulty nodes becoming honest again to function, whereas the question of how to safely recover nodes can be treated as a practical (implementation dependent) one. Nonetheless, this possibility may also be formally considered.

In any case, the effect of corruption on nodes behavior must be specified, which is the *fault type* of the model. Among the most common options.

- Byzantine [LSP82]: There is no restriction at all on the actions of a Byzantine node, or in other words, their behavior is arbitrary. Another formulation is to say that, upon corruption, the adversary learns the node state and answers in its place, effectively creating a perfectly synchronized coalition with all corrupt nodes. This is, the strongest possible adversary regarding fault type, since there are no restrictions at all. Historically, this setting has been considered for critical distributed systems where correct execution is required even in unexpected failures scenarios. However another motivation is that nodes may actually act maliciously, possibly by the action of its operator but more importantly in case where another actor take controls of the node. Additionally, as evoked in the introduction, the ability to tolerate Byzantine faults is an argument to qualify the system as being "trustless".

- Crash faults [PSL80]: The node stops responding altogether upon corruption. It does not send any message nor does it interact with its environment in any way. This setting is better suited in a context of distributed applications that must continue to work even in the presence of hardware/software errors. In other words, compared to Byzantine faults, the security aspect is absent.

- Honest but curious / passive / semi-honest adversary [KL14]: Corrupt nodes follow protocol, but send their state and all data to the adversary. This model is appealing for applications with third parties that are relied upon but cannot be audited, *e.g.* for privacy concerns.
- Omission failures [PT86]: Nodes may fail to send or receive a message.
- Covert Adversary [AL07]: Corrupt nodes can behave arbitrarily, except that they will not take any action that would lead them to be caught, *e.g.*, if an honest node collects and diffuse two votes by the same node (also known as an equivocation proof).
- Rational [IML05, Amo20]: Nodes follow a strategy that aims at optimizing some gain function. This type of fault is motivated by a game-theoretic approach to security. This option can be motivated by the need of a more realistic fault model.

It is generally desirable to analyse protocols under multiple fault types. For instance, Bitcoin is proven secure with up to half of the hashpower Byzantine [GKL20a], but that assumption can be removed by instead assuming that nodes try to maximize their monetary gains [BGM+18], although some other simplifying assumptions are used to prove this result. Ouroboros [KRDO16, BGKR18] is shown secure under Byzantine faults, but offers better performance with a covert adversary. Furthermore, all these fault types may be combined, resulting in a refined model taking into account the possible interactions between the multiple faults types. Examples include the BAR model (Byzantine Altruistic Rational) [AAC+05], but also XFT (Byzantine and crash faults) [LVC+16]. In this thesis we will focus on Byzantine faults alone.

2.1.2 Corruption threshold

We will see in Section 3.5 that not much can be achieved if the adversary can corrupt any node it wants. Only reliable broadcast, which is effectively the weakest protocol we consider can tolerate an arbitrary number of faults, and only with a fairly strong model[4]. Therefore an assumption limiting the quantity of corrupt nodes is necessary to

[4] See Section 3.5 theorems 4 and 7.

work around impossibility theorems. Fundamentally, it means that these protocols (including blockchains) are not *Sybil resistant* [Dou02], implying that a notion of identity is required for the participants, and the protocol cannot tolerate an adversary able to artificially create many identities.

Classically, each node as a physical machine represents a single identity, and the impossibility proof exhibits an attack on any consensus protocol or some other problem, only assuming that up to t nodes can be corrupt with t that must be sufficiently high with respect to n, the total number of nodes [FR03]. Roughly speaking, the optimal value for t is: n for reliable broadcast in synchronous network, $\lfloor \frac{n}{2} \rfloor$ for consensus in synchronous networks, and $\lfloor \frac{n}{3} \rfloor$ otherwise. For more precise statements, see Section 3.5.

Interestingly, the notion of identity formulated in the impossibility proof has few requirements and does not need to be specifically tied to physical machines, public keys, or something else. Indeed the whole reason to have node identities at all is to formalise Sybil attacks and the impossibility to resist such attacks. Therefore, the only requirement for protocols is to make their assumptions with identities that adequately prevent the impossibility result to apply. As shown in Section 3.5, the precise condition to prevent agreement is the existence of a k-sized partition of the node set such that the adversary is able to corrupt any of the partition sets. As a result, any kind of (non-duplicable) resource whose ownership is provable is suitable to escape Sybil attacks, because assuming that at most $1/k$ of that resource is owned by malicious nodes prevents the existence of any k-sized partition. This mechanism is how protocols based on Proof-of-Work (or any other) falsify the premise of the impossibility result, by assuming a sufficiently high fraction of the "work" (*i.e.*, hashpower) is owned by honest nodes. Therefore we feel justified to put forth a definition for the definition of a "Proof-of-X protocol", which is the presence of an assumption limiting the ownership of X by malicious nodes.

2.1.3 Adversarial adaptivity

Having specified what happens upon corruption, we can further restrict *when* corruption can take place, which is related to the notion of the adversary *adaptivity* [CDD+01].

- Static: Nodes must be corrupted before any of them makes an input to the protocol, and the set of corrupted nodes cannot change during execution.

- Adaptive/dynamic : Honest nodes can be corrupted during protocol execution. This means that the adversary can corrupt nodes depending on the protocol inputs, and it can wait for relevant information before choosing which nodes to corrupt. In this setting it may be needed to add the assumption that honest nodes can forget some part of their state, to gain the ability to force a secret information to expire.

- Weakly dynamic: Same as dynamic, except that corruption takes some prescribed time to be effective, or more generally, waits for some precondition. This is particularly useful in context of committee election, as mentioned in the introduction. If the correctness of the protocol relies for a short duration on a set of nodes that could be corrupted, then by taking this set at random and with a weakly dynamic adversary we can ensure that it will not be able to corrupt those nodes before they finish their task. If the overall protocol is long-lasting, *e.g.* like a blockchain, the weakly dynamic adversary is an attractive design trade-off, capturing the evolution of the set of corrupted nodes while letting honest ones complete some short lived actions.

- After-the-fact-removal (implies dynamic), also known as strongly dynamic: When an honest node sends a message on the network, depending on its content the adversary is able to corrupt the node immediately and remove the message from the network before it is received by any other node. One may argue that after-the-fact-removal makes more sense than dynamic alone, because if the adversary is able to instantly corrupt a node right at the moment after sending a message, it seems implausible that the adversary would be unable to prevent the message from being further dispatched in the network. As an interesting note, this is the only assumption regarding adversarial adaptivity that explicitly depends on the existence of a network.

With dynamic adversaries comes the possibility for transient faults, a.k.a. mobile or recovery faults. Instead of staying corrupted for the remaining of the protocol, nodes can become honest again. This may seem a weakening of the adversary because it does not seem to gain anything from it, but depending on the protocol the opposite may be true. If the adversary is limited by the corruption threshold, it can work around that limitation by recovering some nodes that aren't needed anymore. With transient faults, also comes the question of what is the state of nodes after recovery. Indeed re-joining an existing network

often requires trust assumptions *e.g.*, to contact at least one honest node, and possibly a costly procedure. Furthermore, if the adversary is not weakly dynamic, an assumption on the rate at which corruptions change may be required, to prevent the unrealistic situation where the adversary would corrupt every node at a high frequency to effectively control all of them without exceeding the corruption threshold.

2.2 Synchronization

2.2.1 Clocks

To keep track of the passing of time, each node has a local clock that regularly ticks. This can only give meaningful information if their clocks have some level of consistency, hence the existence of clock synchrony assumptions. Even without explicit timeouts in the protocol, the possibility to have nodes blocked due to processing times requires considerations on how synchronized is time keeping. The assumptions are stated in terms of ϕ, the bound (over all executions) of the maximum factor between any two nodes clock rate. In the landmark paper by Dwork, Lynch and Stockmeyer, three possible assumptions are identified [DLS88].

- Strong: ϕ exists and is known to the nodes, *i.e.*, given to the protocol as a public parameter.
- Partial: ϕ exists, however it is not given to the protocol, and it is generally not possible to infer it during execution.
- Asynchronous : No assumptions at all, there is no ϕ. Or equivalently, $\phi = \infty$.

More recently, and particularly in the context of blockchain, the issue of clock synchronization is less of a concern, and roughly only one setting is considered, "loosely synchronized clocks" or "roughly synchronized clocks". There are two equivalent formulations:

- There is a known bounded offset between any two node's clocks, *i.e.*, an absolute bound over all executions.

- Only assume strong synchrony, but with a protocol that terminates in finite time.

The justification is that hardware performance makes the issue of clock drift mostly moot, and then being able to obtain some correct date information once is a reasonable assumption. But perhaps confusingly, the latter assumption is stronger than strong clock synchrony assumption.

Because clocks are local to each node, there is no use in having clocks at all if they are fully asynchronous.

2.2.2 Network

The network provides a send/receive interface for messages between nodes. Its basic functionality is to deliver sent messages to some recipient node. A wide range of network topologies have been studied, for instance depending on connectivity and in ring networks. In this thesis we focus on the simplest model and most relevant to blockchains in general: point to point complete communication graphs. That is, any node may send a message to any other node. One interesting alternative is if honest nodes can only multicast to all nodes, because they don't need to know who is participating to send messages.

First, let us give some assumptions that define the basic functionality of a communication network.

- Eventual delivery : A message will eventually be delivered to its recipient.
- Reliable channel : Messages are not duplicated, created nor modified.

Those two assumptions together form a *secure channel*.

Generally the adversary can read message contents; secrecy can be assumed but this is useful mostly when participants have to transmit secrets to specific parties, and it's often not the case with agreement protocols. If the adversary cannot spoof the message sender then the channels are said to be authenticated, or equivalently the messages are "oral messages" [LSP82]. This should not be confused with solving the authenticated version of a problem, *i.e.*, when digital signatures are available. The difference can be summarized as below.

- Digital signatures are transferable, while with authenticated channels a node cannot prove to a third party that some other node sent a message. This is important because this change affects solvability results.

- Although information-theoretic security is impossible for signatures, we can model an ideal signature scheme that has information-theoretic security even if it cannot be implemented in the plain model. This technique can roughly be seen as equivalent to an ideal transferable-authenticated channel. For a practical application, unconditionally secure *pseudo-signatures* may be employed for the same purpose [FWW04, PW96].

The definitions for network synchronicity are analogous to clock ones. They are stated in terms of Δ, the bound (over all executions) of the maximum time it takes for a message to be delivered [DLS88].

- Strong synchrony : Δ exists and is known to the nodes.

  Alternatively, a "round" or "step-lock" formalism is possible : Each node has access to a global "round" clock, and sent messages are guaranteed to be received before the end of the current round. The equivalence between the two formalism is subject to caveats, see Section3.2.2.

- Partial synchrony [DLS88]. There are two formulations.

  - Δ exists, but it is not given to the protocol and it isn't possible in general to infer it. This is analogous to the definition for the clock partial synchrony assumption.
  - Δ exists and is known to the nodes, but it holds after some arbitrary Global Stabilization Time (GST), unknown to the protocol. This formulation is also called eventual synchrony.

  Both definitions are generally seen as equivalent, although there are also some small technical caveats, see Section 3.2. In practice, the GST model is useful to design protocols as if they had a synchronous round structure, simply by proving that the protocol remains safe if the network is asynchronous. The underlying reason is than many protocols rely on an estimation mechanism for Δ, whose functioning details

are irrelevant to the protocol design. Hence, the GST formalism is a practical way to simply consider the resulting round structure.

- Asynchronous : there is no Δ, or $\Delta = \infty$. The difference between partial synchrony and asynchrony only concerns infinite runs : A finite part of an execution is always partially synchronous. Another wording is to say that, in an asynchronous network, the delivery time of new messages may increase forever.

The synchrony assumptions depends on the notion of clocks. Indeed, Δ is intended to be expressed in "real time steps" which has meaning only if clocks give some. Because clock delays can equivalently be seen in terms of network delay, only the weakest assumption of the two will prevail [DLS88]. If the protocol doesn't make use of timeouts, then the formulation of clocks can be embedded in the network functionality.

In the context of crash faults, an alternative formulation can be made in terms of failure detectors [FGK11]. Using this abstraction, the network is asynchronous but there is an additional oracle that nodes can access to learn about the state of other nodes (crashed or not). Depending on the reliability and accuracy of that information, the failure detector encodes varying strengths of assumptions.

On dealing with strong synchrony The asynchronous model has a lot of very strong negative results, but on the other hand strong synchrony is contriving important assumption, and consequently partial synchrony is an attractive middle ground to these extremes. However another alternative is to try to mitigate the issues with strong synchrony directly.

- Even in real implementations, it is difficult to obtain hard guarantees on the Δ bound, which generally remains a hopeful assumption. This contrasts with the devastating effects of violating this bound, potentially breaking safety for all nodes. To answer to this issue, the *sluggish* type of faults may be added [GPS19], where sluggish nodes follow the protocol honestly but may violate the Δ bound. This does not raise the overall corruption threshold, but sluggish nodes will still be guaranteed safety. In a sense this is similar to allowing the adversary to mark some corrupt nodes as being "honest, partially synchronous" nodes.

- Because the Δ bound has to always hold, even in worst-case execution, it has to be overestimated. This can be mitigated with *responsive* protocols [PS17b], which

are protocols whose latency is independent of Δ. Of course strongly synchronous protocols cannot be always responsive, otherwise they would simply be partially synchronous. Therefore responsive protocols are only so under some conditions, *i.e.*, *optimistically*.

- Some flexibility can be added by giving each node the ability to choose its own Δ independently from the other nodes. Furthermore, at least in the case of Proof-of-Work protocols, it is possible to make all the online phase of the protocol independent from Δ, such that its value is only needed to extract the output from data generated in the online phase. For instance, Spectre and Phantom [SLZ16, SZ18] let nodes generate a blockchain through mining without knowing Δ, and knowing its value let compute each transaction' probability of being ever invalidated.

  This approach also provides graceful security degradation. Let $confidence_\Delta(tx)$ be the function that evaluates the probability for a transaction tx to be reverted. If $confidence$ only tends to 0 when Δ goes to infinity, then violation of the bound only results in a higher-than-expected probability of failure, not total breakdown.

2.3 Randomness

Historically, protocols have been divided between and deterministic and probabilistic ones. A probabilistic protocol P is not required to be secure for all executions, rather, it must satisfy some properties *with probability p*.

Investigation of probabilistic protocols has been motivated in no small part due to the FLP impossibility [FLP85], which states that consensus cannot be solved with one crash fault by a deterministic protocol in an asynchronous network. Because the FLP impossibility shows the existence of non-terminating executions, probabilistic protocols circumvent this result by ensuring that these executions have probability 0. This is possible if, for instance, honest nodes periodically makes a coinflip that always has a non-zero chance of leading to a terminating execution. In that case, the protocol may even have probability 1 of being secure; although the other side of the coin is that any bound on the execution length must hold with probability < 1, because any step from a non-terminating execution can be reached with non-zero probability.

As a result, and also due to the existence of other bounds on probabilistic protocols[5], they have acquired a reputation of being inefficient, despite the possibility of *expected* constant execution time [CKPS01]. However, as cryptography becomes overwhelmingly used, especially in the BFT setting, modern protocols tends to be almost always probabilistic, since cryptographic primitives are most often considered in a computational setting where the adversary has a (negligible) probability of winning. Therefore, any protocol that uses, *e.g.*, hash functions may be considered as probabilistic, in the sense that they satisfy their properties with some probability p (which in this case cannot even be 1).

Lastly, we point out that the definition of probabilities in distributed systems is not as straightforward as it may seem. The probability space of executions is dependent on the actions of the adversary, however as illustrated in the beginning of this chapter, the adversary is nothing more than a universally quantified set. Such details on the formalism are sometime stepped aside, *e.g.*, by considering the probability space "over all players and the adversary random coins" [Can01].

2.4 Assumptions and Setup

Commonly for cryptographic protocols, some additional assumptions may be required. Broadly speaking, they may fall in three categories:

- The modelling of a real-world components, such as hardware security modules or the time required to solve Proof-of-Work challenges.
- Cryptographic assumptions, typically the computational hardness of some problem.
- Trusted setups, when some interactive program is run before any intervention of the adversary, and obviously before the protocol input is known.

These assumptions are not solely the concern of cryptography, because they also influence whether consensus (or any other problem) is solvable [FR03].

Regarding setups for BFT protocols, there are a few common assumptions of interest.

[5] See for instance Section 3.5 and Theorem 6.

- Public parameters, Common Reference String (CRS), correlated randomness: A trusted dealer samples a string from a known distribution and sends it to all parties. This assumption is typically used by protocols based on zero-knowledge proofs, where generating the string implies the knowledge of a trapdoor breaking the scheme.

- Public Key Infrastructure (PKI) model: Every node generates a public/private key pair and sends the public part to every other node. Note that new nodes joining the network must also transmit their public key, which is akin to an "online extension" of this setup assumption. Additionally, one can see the public key of a node as also playing the role of network address, either because it has an address attached, or because the network is able to resolve a public key.

More generally, any idealized cryptographic primitive can be assumed to be available for the protocol to work. For instance, oracles, setup and computational assumptions are often used only to build up cryptographic primitives. In this case, it is our impression that directly assuming the required primitive in its idealized form is preferable, for several reasons:

- It simplifies presentation, allowing to focus on the protocol semantics instead of implementation details.

- Perhaps paradoxically, a weaker model without setup and computational assumptions may be simpler to describe and work with, *e.g.* by focusing purely on combinatorial arguments.

- The assumptions needed for the actual protocol may be hidden by the stronger cryptographic ones. Thus this approach can strictly strengthen the proven results.

- Specifying a given implementation for cryptographic primitives may be unnecessarily restrictive, and there is no issue in suggesting an implementation with desirable properties, to describe a complete instantiation of the overall protocol.

- More generally, this approach essentially boils down to favoring fine-grained modular designs.

2.5 Problems

Having reviewed the existing models in which BFT protocols executes, we now turn our attention to the specifications that they satisfy. These specifications acts as an abstraction of the object implemented by protocols, and in this case in particular we are interested in the definition of a blockchain as a distributed object.

Blockchain To define what a blockchain is, we evoked the State Machine Replication (SMR) approach in Chapter 1, in which a blockchain protocol aims at emulating a centralized stateful application. To make things more precise, the definition we will focus on in this thesis is *atomic broadcast* (ABC), also known as total-order broadcast [DSU04a].An atomic broadcast protocol lets any node initiate the A-broadcast of a message (*i.e.*, transactions in the blockchain case) at any moment, locally A-delivers messages, and guarantees the following. In the context of blockchain, messages can directly be valid transactions.

- Liveness and validity: All messages A-broadcast by honest nodes must eventually be A-delivered by all honest nodes.
- Consistency : All honest nodes A-delivers the same sequence of messages.

This definition predates the blockchain era, and since then other abstractions have been built to better fit its specificities.

The first[6] and most prominent effort to define a blockchain is due to Garay and Kiayias [GKL15, GKL20a] through the *Transaction Ledger* abstraction. Although the execution model of the transaction ledger is the UC framework [Can01], the properties themselves are stated in an ad-hoc manner not compatible with other UC definitions[7]. This gap has been fixed [KZZ16] with a formalisation of the transaction ledger in the GUC [CDPW07], an extension of the UC framework.

Definition 1 (Transaction Ledger)**.** A protocol that maintains a list of blocks of transactions is a Transaction Ledger if it satisfies the two following properties.

[6] At their level of formalisation.

[7] Concretely, the transaction ledger is *not* a UC functionality.

- *Persistence:* Once a node of the system proclaims a certain transaction tx as stable, the remaining nodes, if queried, will either report tx in the same position in the ledger or will not report as stable any transaction in conflict to tx. Here the notion of stability is a predicate that is parametrized by a security parameter k; specifically, a transaction is declared stable if and only if it is in a block that is more than k blocks deep in the ledger.

- *Liveness:* If all honest nodes in the system attempt to include a certain transaction then, after the passing of time corresponding to u slots (called the transaction confirmation time), all nodes, if queried and responding honestly, will report the transaction as stable.

These properties are parametrized by k and u, and the definition is loosely equivalent[8] to atomic broadcast with $k = \text{poly}(\lambda)$ and latency $O(\Delta u)$. The Liveness property of the transaction ledger may seem slightly weaker than ABC because *all* nodes must input a transaction to guarantee its inclusion. However, it is still possible to satisfy ABC liveness using a transaction ledger, the input message m of the a-broadcast is multicasted to all other nodes, and upon reception they submit m to the transaction ledger. In practice, for instance with Bitcoin, this mechanism is instantiated by the transaction pool.

This definition is sometimes broken up in three parametrized properties on the blockchain data structure: Common Prefix (CP), Chain Growth (CG) and Chain Quality (CQ). Informally, Common Prefix states that pruning k_p blocks from a chain results in the same chain for all honest nodes. Chain Growth (CG) states that every k_g consequent rounds, each node adds at least τ blocks, *i.e.*, τ is a round/block conversion parameter. Chain Quality (CQ) states that in every chain segment of length k_q, there is at least a proportion μ of honest blocks. k_p-persistence is straightforward to prove from k_p-CP. For liveness, the proof from all three properties yields $max(k_g, \frac{k_q+k_p}{\tau})$-liveness.

Anceaume *et al.* proposed a blockchain abstraction [APL+19a] where they considered a block tree structure with *read* and *append* operations. Then they define a weak and a strong consistency criteria on the operations history to capture the blockchain properties. The strong criterion implies a common prefix on the tree which we expect to be equivalent to the atomic broadcast. Anta *et al.* gave an abstraction of a Distributed Ledger Object

[8] See Section 3.3 for the definition of loose reduction and equivalence.

(DLO) [AKGN18] with a similar approach, and they show how their abstractions reduces to ABC.

Related problems Beyond a specific target application, the consensus problem is considered to be a fundamental primitive for distributed algorithms. Its executions are finite which makes it conceptually simpler, and its equivalence with atomic broadcast[9] means that any distributed application can still be implemented through consensus. A solution to the consensus problem [Fis83] is a protocol that lets all nodes start with one input value, and guarantees the following properties.

- Liveness : All honest nodes eventually deliver a value, exactly once.
- Consistency : All honest nodes deliver the same value.
- Validity : The delivered value must be the input value of one honest node.

Beyond the other definitions that we are considering in Section 3.3, there are many variations which may be considered [CVNV11]. Liveness can be weakened by removing the completeness requirement, to only require that *some* honest node eventually delivers a value, exactly once. Consistency can be weakened by making it approximate [DLP$^+$86] w.r.t. some distance function: The honest nodes output must be closer to each other than a given ε. Another option is the k-set version [Cha93] with a weaker agreement property : The number of different honest outputs is at most k.

Another fundamental problem in distributed systems is the Reliable Broadcast [LSP82]. A reliable broadcast protocol is parametrized by a sender node S. It lets an honest node R-broadcast a message to all nodes, while guaranteeing that even a corrupt sender cannot force other nodes to R-deliver different messages. Reliable broadcast is strictly weaker that consensus, can be implemented in an asynchronous network, and is commonly used as a building block in practical protocols.

- Liveness : If the sender S is honest, then all honest nodes eventually R-deliver a message exactly once.
- Consistency : All honest nodes R-deliver the same message.

[9] This affirmation is not without caveats, see Section 3.3.

- Validity : If the sender S is honest, then the R-delivered message is the R-broadcast of S.

In regard to cryptocurrencies, there is a somewhat overlooked but crucial hindsight that solving consensus is not necessary to build a payment system [GKM+19]. Therefore it is possible to build a cryptocurrency application using primitives that are strictly weaker than consensus, such as Lattice Agreement [LAQ20] or Reliable Broadcast [CGK+20]. The resulting protocol benefits from less restrictive bounds and thus can weaken its assumption on network synchrony, improve fault tolerance, and improve performance. On the other hand, Turing-complete smart contracts are incompatible with this technique since they are equivalent to State Machine Replication.

2.5.1 Extensions

The problems presented here as well as the ones added in Section 3.3 can be seen as a description of the core functionality provided by the distributed objects. Different algorithms may provide others desirable properties. Although they are technically a part of the problem definition as solved by the protocol, it may be more intuitive to consider them as some kind of "added feature".

Support for reconfiguration Reconfigurable protocols add the possibility to have members that can be changed during execution. This is mostly useful for protocols that are expected to be long-lived, such as Atomic Broadcast. In fact, the possibility to have nodes easily leave and re-join the system at any moment is one of the main characteristics of *permissionless* environments.

With a common assumption that all nodes know the set of participants, the straightforward way to support reconfiguration is to solve consensus to agree on an updated set of members, while taking into account that the "members update" operation also impacts how other operations complete. Additionally Kuznetsov and Tonkikh showed that for Lattice agreement protocols, reconfiguration itself can be solved with Lattice agreement [KT20].

Fairness There are several related notions of fairness that can be considered. Regarding the SMR abstraction, fairness can seek to address that the adversary can choose the final ordering of commands [KZGJ20]. Another concern is to bound the proportion of commands

that are issued by corrupt nodes [PS17a]. This definition in particular can be relevant in the context of rational[10] players, to ensure that players are rewarded in accordance with their effort.

Requirements on performance Protocols can be designed with specific conditions on *e.g.* the number of exchanged bits, the time to finish execution, *etc*. Furthermore, optimistic protocols [KAD+07, PS18] are more efficient under some conditions. Elette Boyle, Ran Cohen and Aarushi Goel [BCG20] presented a consensus protocol with a "balanced" communication cost which must be the same for all parties.

Improved bootstrapping For protocols with dynamic participation, it is expected that nodes will commonly (re-)join after being out of the system for extended periods, therefore requiring to recover an up-to-date state to resume execution. Besides, this operation fundamentally requires trust assumptions to reconnect to the network (contacting at least one honest node). This is complicated by the possibility of honest nodes being corrupted after leaving the network. Some protocols may offer the possibility to do so without having to download lengthy transcripts, *e.g.* through check-pointing or zero-knowledge proofs [KK20, LSGZ19], while minimizing the assumptions required for a successful join.

Guaranteeing different properties under different conditions/models This gives rise to "degrading" versions of a problem, *e.g.* when the guarantees change with different corruption thresholds [HKL20].

Verifiability A "validated" version of some problem is required to produce a certificate along the output that can be checked for validity. The verifier, given the protocol instance and public parameters, should be convinced that the protocol took place as specified. In the case of "public verifiability" the verifier needs not to be a protocol member.

[10] See Section 2.1.1.

Chapter 3

Formalisation of distributed components

The primary objective of this chapter is to establish a library of definitions of the *primitives* and problem *specifications* commonly used by BFT protocols. More importantly, these definitions must be based on a common model of execution, thus ensuring a consistent terminology. To illustrate how this contribution helps to clarify the blockchain landscape, we show some uses of this library. First, we relate the definitions we just made with each other, through reductions and impossibility theorems from the literature, yielding a formalised map of the BFT landscape that is both clear and precise[1]. This map highlights fundamental differences in the definitions of BFT consensus, an observation that we believe has been largely ignored within existing research. Then, we map our library to existing implementations of blockchain protocols, *i.e.*, we analyse their model and specification which we interpret using the components we have defined. As a bonus, we are also able to use our formalism to compare performance metrics.

We do not aim at making a framework that would be explicitly used in subsequent work to make security proofs. Rather, we want to make a simple and coherent description of models and specifications that constitute the base of recent protocols, to ensure that any ambiguity in terminology can be resolved at least in principle. As an example of this contribution, we argue in Section 3.6.1 that the network model of two prominent papers [GKL20a, BGKR18] are stated inaccurately, partial synchrony is claimed, whereas an interpretation in our framework requires a strongly synchronous network.

[1] See Figure 3.2

Related work The main motivation of this work is the absence of a coherent collection of definitions for both the primitives and the specifications of distributed components. To make those definition, we first provide a minimal formalism (in Section 3.1) that serves as an abstraction layer for an execution model. This execution model is by no means intended to be novel or to improve other state of the art frameworks such as the Universal Composability (UC) framework [Can01], I/O Automatas [CCK+08a, CCK+08b], the RSIM framework [BPW07], Constructive Cryptography [Mau11], the MOSS framework [LHSW20] and the IITM framework [KTR20]. On the contrary, we intend for future work to replace our execution model with an existing one[2].

Indeed, existing work regarding the formalisation of BFT protocols has largely focused on either the execution model alone, *e.g.* with one of the framework mentioned above, or making definitions of primitives and specifications specifically for one protocol in isolation; thus they are far from forming a complete characterisation of the state-of-the-art. However, there has been some effort to make standard definitions of common distributed components for existing execution model. Liu-Zhang and Maurer [LM20] build on the constructive cryptography framework to formalise a synchronous network, and they provide a specification for reliable broadcast as well. Kiayias *et al.* [KZZ16] have provided definitions for a synchronous network and a fair distributed ledger in the GUC framwork [CDPW07], an extension of the UC framwork. To the extent of our knowledge, there are no formalised definition of Byzantine consensus within one of these framework.

Garay and Kiayias [GK20] took the UC framework as an execution model to classify several protocols, Their work is similar to what is done in Section 3.6, since they are interpreting protocols in a common set of models. Independently of any framework, Milosevic*et al.* [MHS11] related a wide variety of consensus definitions to atomic broadcast in the Byzantine case, which is similar to our contributions in Section 3.3. For those last two papers, our contributions goes on step further by making our definitions significantly more formal and complete. Additionally, we combine the impossibility theorems with the reductions to complete the picture, and we make precise metrics evaluation of existing protocols.

Regarding the UC framework, it should be noted that some existing blockchain protocols described within this framework [GKL20a, BGKR18, DPS16] uses its execution model but formalizes the blockchain abstractions using ad-hoc properties instead of UC

[2] see Section 5.1.

functionalities that are covered by the UC notion of security. In a sense, the resulting approach is very similar to ours, but based on a substantially more complex formalism.

In Section 3.1, we start by formalising a basic execution model that will serve as a common ground for the following sections. We give definitions for various adversaries and primitives that are used as model in Section 3.2; we give definitions for protocols specifications (*i.e.* properties they must satisfy) in Section 3.3, and we relate them by their strength. Then, in Section 3.4 we provide common performance metrics, namely latency and communication cost, in our framework. In Section 3.5 we lay out necessary trust assumptions and minimal costs to solve the problems from Section 3.3, and reinterpret the proofs in our model. Then, in Section 3.6 we analyse a few prominent blockchain protocols, by giving their model and metrics, as interpreted in our framework.

3.1 Preliminary work : A simple execution model

In this section we make a small and simple execution model that will be the basis for the definitions in the remaining of the chapter. In the interest of systematisation, we made the overall framework as modular as reasonably possible, the intent being that a specific model can be simply described by picking the desired components.

Overview We first define an abstract "execution" object which gives an ordered list of events. Then we define an abstract container "module" object that represents interfaces which the nodes can connect to, with their properties. The properties themselves are stated in terms of admissible executions. The modules will be used to define clocks, network, randomness, setups, ideal signatures, random oracle, PoW oracle, as well as all the problem definitions (*i.e.*, specifications) in the following section. Then, given a distributed algorithm P that uses a module list A, we define the set of all executions of P in A, admissible with respect to adversary abilities. Finally, we say that a protocol P meets its specification B if all its executions are admissible executions of B.

To simplify presentation we first cover the deterministic case and then update the definitions to capture probabilistic executions. In Table 3.1 we give a reference to most notations used in this section.

Notation	Meaning
$P \in \mathcal{P}[M]$	protocol P
$p, q \in \Pi$	nodes p and q from the set of nodes Π
$m \in M$	module m from set of modules M
E_i, $i \in E^I$	event E_i at index i from execution E
$\mathcal{E}[X]$, $\mathcal{T}[X]$	executions and execution trees over X
$\mathbb{P}_T[\phi]$	probability of ϕ within probability space induced by T
$\_$	any value (existentially quantified)
$\text{Events}(\cdot)$	Event set (of a protocol or module)
$\mathcal{M}_P^{\mathcal{A}} \models S$, $\mathcal{M}_P^{\mathcal{A}} \models^{\lambda} S$	protocol P with adversary $\mathcal{A}$ (statistically) satisfies module S
A_m,	admissibility predicate of module m
I, O	sets of input and output values
C_i, H_i	set of corrupt and honest nodes at index i
$\mathcal{C}$	corruption structure
Δ, Φ,	network delay, nodes clocks offset
λ	security parameter
$\# \cdot$	set cardinality
$\text{negl}(\cdot)$, $\text{poly}(\cdot)$	a negligible function, any polynomial function
$O(\cdot)$, $\Omega(\cdot)$, $\Theta(\cdot)$	upper, lower, and exact asymptotical bound

TABLE 3.1: Summary of notations

Notations For naturals a and b, $[\![a,b]\!]$ is the set of natural numbers between a and b, and $[x,y]$ is the set of reals between the reals x and y. For a set X, $\mathcal{P}(X)$ is the power set of X, #X is the cardinality of X. $A \times B$ is the Cartesian product of the sets A and B, which we also call *tuples*. Function abstraction is noted $x \mapsto f(x)$. The logical truth and false are noted $\top$ and $\bot$, respectively.

3.1.1 Deterministic modules and protocol execution

We start off by introducing the basic objects of our formalism, sequences, modules, events and protocols. First, let Π be a finite set, representing all the nodes in the system, with $n := \#\Pi$ the number of nodes. We use sequences of events as the basic object to describe executions, with events being tuples that contains information about an input/output to the nodes, or a corruption.

Definition 2 (Sequence)**.** For a given a set X, a sequence E over X is a tuple (E^I, E^F), where $E^I \in \bigcup_{k\in\mathbb{N}}\{[\![0,k]\!]\} \cup \{\mathbb{N}\}$ and $E^F : E^I \to X$. For simplicity, we will write E_i in place of $E^F(i)$ for any $i \in E^I$.

For a given set X, we note $\mathcal{E}[X]$ the set of sequences over X. We then call X the *event set* of E.

Let $E \in \mathcal{E}[X]$. The preimage of E is noted E^{-1}, *i.e.*, $E^{-1}(x) := \{i \in E^I \mid E_i = x\}$ for some $x \in X$. In the following we extensively use this preimage notation as way to select specific events from an execution, *i.e.*, a filter operation. To this end, we also introduce the notation "_", which stands for an arbitrary placeholder in the component of a tuple. Formally it is defined as follows. First, the notation E^{-1} is extended in the standard way to sets, *i.e.*, for a set $X' \subseteq X$ we have $E^{-1}(X') := \bigcup_{x\in X'} E^{-1}(x)$. Then, we interpret tuples containing "_" as the set of tuples with all possible values for those components. Let $\Pi_{i\in I}A_i$ be a set of tuples, then the tuple $(a_0,\dots,a_i,\dots) \in \Pi_{i\in I}(A_i \cup \{\_\})$, is a shorthand for the set

$$\{(a'_0,\dots,a'_i,\dots) \in \Pi_{i\in I}A_i \mid \forall i \in I, a_i \neq \_ \Rightarrow a_i = a'_i\}$$

Those two notations combined placeholder stands for an existentially quantified variable over the appropriate domain. For instance, if $E \in \mathcal{E}[A \times B]$ and $a \in A$, then $E^{-1}(a,\_)$ is a shorthand for $\{i \in E^I \mid \exists b \in B,\ E_i = (a,b)\}$.

We now define a module as an object that interact with the nodes through inputs outputs, and that guarantees additional properties as expressed by a predicate on sequences.

Definition 3 (Module)**.** A module m is a tuple (l_m, I_m, O_m, A_m), such that:

- l_m is a finite string called *label* of m, which serves as the interface name used by nodes[3].
- I_m and O_m are countable sets, called *input set* and *output set* respectively.
- $A_m \subseteq \mathcal{E}[X]$ is called the *admissibility predicate* of m. Here, X is a parameter to A_m that may be any countable set such that $\text{Events}(m) \in X$, with $\text{Events}(m)$ from definition 4 below. A_m is seen as a predicate over $\mathcal{E}[X]$, *i.e.*, for a sequence E, we write $A_m(E)$ instead of $E \in A_m$.

We call $\text{Events}(m)$, the *event set* of m that contains all events related to m, the constant $\mathfrak{m}$ indicates an event related to a module. It contains input events indicated by $\mathfrak{i}$, with the node making the input as well as its value; output events indicated by $\mathfrak{o}$, with the node receiving the output as well as its value; and corruption events, with the node being corrupt. It also contains corruption events, with the node being corrupt, as indicated by the constant $\mathfrak{c}$.

Definition 4 (Module event set)**.**

$$\begin{aligned}\text{Events}(m) :=& \{(\mathfrak{m}, \mathfrak{i}, l_m)\} \times \Pi \times I_m \\ &\cup \{(\mathfrak{m}, \mathfrak{o}, l_m)\} \times \Pi \times O_m \\ &\cup \{\mathfrak{c}\} \times \Pi\end{aligned}$$

Additionally, we use a dot notation for tuple projection. For an event $e \in \text{Events}(m)$, $e.p \in \Pi$ is the node component (whether it is an input, output or corruption event), and $e.v \in I_m \cup O_m$ is the output or input value.

Concretely, A_m can rely on the existence of a total order of events, among which are its own and corruptions events. $A_m(E)$ may hold depending the presence of elements in

[3] When the module represents a subprotocol, the label does play a role of session identifier.

the sequence that are *not* in Events(m), but it can only do so "blindly", *i.e.*, based on their order, independently from their content.

Given a module list M, we note $\mathcal{P}[M]$ the set of protocols that uses M. A protocol $P \in \mathcal{P}[M]$ is modelised as a state machine that accepts top-level inputs or outputs from modules, and returns top-level outputs and inputs to modules.

Definition 5 (Protocol)**.** A protocol $P \in \mathcal{P}[M]$ is a tuple $(I_P, O_P, S_P, S_P^0, F_P)$, such that:

- I_P and O_P are countable sets, called *input set* and *output set* respectively.
- S_P is a countable set called the *states* set and $S_P^0 \in S_P$ is called the *initial state*.
- $F_P : S \times (I \bigcup_{m \in M} O_m) \to S_P \times (O_P \bigcup_{m \in M} I_m)^*$ is a computable function called the *transition function*. Its input domain is a state and an input in I_P or output from M's output set. Its output domain is a resulting state and list of input values in M or output from O_P.

We define the event set of P similarly to modules, that is:

$$\begin{aligned} \forall P \in \mathcal{P}[M],\ \text{Events}(P) := &\{(\mathfrak{p}, \mathfrak{i})\} \times \Pi \times I_P \\ &\cup \{(\mathfrak{p}, \mathfrak{o})\} \times \Pi \times O_P \\ &\bigcup_{m \in M} \text{Events}(m) \end{aligned}$$

Given $P \in \mathcal{P}[M]$, we call the *executions* of P the set $\mathcal{E}[\text{Events}(P)]$.

As can be seen from definitions 4 and 5, an event's first component is its type which can be either $\mathfrak{m}$ for module, $\mathfrak{p}$ for protocol or $\mathfrak{c}$ for corruption. For a module or protocol event, its second component is either $\mathfrak{i}$ for input or $\mathfrak{o}$ for output.

In terms of physical interpretation, an event E_i can be seen as an atomic time step, and an execution E contains all the information to describe the evolution of the system. However we emphasize that this interpretation only serves as a thinking tool, and does not have any formal implication.

Finally, we will call a *model* $\mathcal{M}$ of P a predicate on P's executions. This predicate defines the executions of P that are deemed to be compliant with the model in which P runs. It includes basic requirements such as the fact that honest nodes follows the protocol, as well as some of the adversary capabilities, such as a weak adaptivity.

Definition 6 (Protocol models - deterministic version)**.** Given protocol $P \in \mathcal{P}[M]$, a model of P is a unary predicate $\mathcal{M} \subseteq \mathcal{E}[\text{Events}(P)]$. Like with other predicates, we write $\mathcal{M}(E)$ to mean $E \in \mathcal{M}$.

In section 3.2, after extending our framework to probabilistic executions, we will be making definitions of models $\mathcal{M}^{\mathcal{A}}$ for each type of adversary $\mathcal{A}$. For this deterministic version, we give an example of requirement for $\mathcal{M}$, which expresses that all executions of P must be admissible for all its modules.

Example (simple model requirement)**.**

$$\mathcal{M}_P(E) := \forall m \in M,\ A_m(E)$$

To complete this introduction we finally give the security statement that P correctly implement a specification module S. This is done simply by requiring the admissible executions of P to be also admissible for S. However we first we need to slightly modify P's execution to make them fit for A_S.

Definition 7 (Stripping function)**.** Let S be module, P a protocol such that $I_P = I_S$ and $O_P = O_S$. Then $\mathcal{S}_m(E)$ is a function that takes $E \in \mathcal{E}[\text{Events}(P)]$ and removes all modules inputs/outputs from the execution, *i.e.*, events in $E^{-1}(\mathfrak{m}, \_, \_, \_, \_)$; and prefixes the protocol's inputs/outputs with m's label l_m, *i.e.*, it maps $(\mathfrak{p}, d, p, v)$ to $(\mathfrak{m}, d, l_m, p, v)$.

Definition 8 (Implementation correctness - deterministic version)**.** Let $P \in \mathcal{P}[M]$ be a protocol in model $\mathcal{M}_P$. P is secure with respect to (or satisfies) module S iff the stripped admissible executions of P are also admissible for S, *i.e.*,

$$\forall E \in \mathcal{E}[\text{Events}(P)],\ \mathcal{M}_P(E) \Rightarrow A_S(\mathcal{S}_S(E))$$

Note that definitions 6 and 8 are only for illustration purposes and will not be used otherwise, however they both have an extended version in the full framework, *i.e.*, definitions 14 and 16 respectively.

Remark The definition of P's executions look very similar to the definition of a module, with input and output set, event set, and admissible executions. This is expected because the end goal is to state that executions of P are consistent with some specification module.

The reason to not define the protocol as a module itself is that it would have to be dependent on the module list it uses, which would in turn considerably complicate the definition of modules. In general, any notion that changes the overall protocol execution cannot be formalised as a module.

3.1.2 Extension to the probabilistic case

Next we extend our model to allow for random executions, which is not a straightforward addition. The current approach with our model is to define all the possible executions, and then state the admissibility of each execution separately. Mixing probabilistic events that relates multiple executions with combinations of unrelated events is tricky. Our approach is based on the key hindsight by Lynch [Lyn96, Section 4.1.2] that there is one probability space for each adversarial strategy. We first define *execution trees* that captures any possible probability spaces over executions. Then we treat each probability space as a single execution like in the deterministic case, such that admissibility predicates now qualify over executions trees.

First, we introduce the notations for probability spaces.

Definition 9 (discrete probability space)**.** A discrete probability space $\mu \in \mathcal{D}$ is a tuple $(\Omega_\mu, \mathbb{P}_\mu)$ where Ω_μ is a finite set called the *outcome set* of μ, and $\mathbb{P}_\mu : \mathcal{P}(\Omega_\mu) \to [0,1]$ is a σ-additive function called the *probability measure* of μ. A σ-additive function is a function $\mathbb{P}_\mu$ such that $\forall A, B \subseteq \Omega_\mu, \mathbb{P}_\mu(A \cup B) = \mathbb{P}_\mu(A) + \mathbb{P}_\mu(B)$. Furthermore, we use the standard notation $\mathbb{P}_\mu[\Phi]$ to mean $\mathbb{P}_\mu(\{x \in \Omega_\mu \mid \Phi\})$.

Probabilistic events We add the possibility for a module to have a probability space associated to its output set. That is, instead of producing an output event during an executions, modules may instead produce a probability distribution over the output set. The definition of an execution does not change, only the event set of modules is modified to allow for *probabilistic output events*.

Definition 10 (Probabilistic module)**.** A probabilistic module m is a tuple (l_m, I_m, O_m, A_m) exactly as from definition 3, except that:

- $A_m \subseteq \mathcal{T}[X]$. Here, X is a parameter to A_m that may be any countable set such that $\text{Events}(m) \in X$, with $\mathcal{T}[X]$ from definition 11 below.

- The event set of m is *probabilistic*:

$$\begin{aligned}\text{Events}(m) := &\{\mathfrak{c}\} \times \Pi \\ &\cup \{(\mathfrak{m}, l_m, \mathfrak{i})\} \times \Pi \times I_m \\ &\bigcup_{\substack{\mu \in \mathcal{D} \\ \Omega_\mu \subseteq O_m}} \{(\mathfrak{m}, l_m, \mathfrak{o})\} \times \Pi \times \Omega_\mu \times \{(\Omega_\mu, \mathbb{P}_\mu)\}\end{aligned}$$

We also keep the same dot notations for the event set of probabilistic modules, *i.e.*, $e.p \in \Pi$ and $e.v \in I_m \cup O_m$ for $e \in \text{Events}(m)$, and we add $E.\mu \in \mathcal{D}$.

For simplicity, all modules outputs have an associated probability space. We say that the output $(\mathfrak{m}, \mathfrak{o}, p, v, \mu)$ is *deterministic* iff $\Omega_\mu = \{v\}$ and *probabilistic* otherwise. Without loss of generality, we will consider all modules to be probabilistic modules and we only write "module" instead.

Given that each probabilistic event also give the resulting outcome in the execution, they have an associated probability which is simply the probability function $\mathbb{P}_\mu$ applied to the outcome. By extension, we can associate a probability to each execution, which is the product of probability of all their probabilistic events, *i.e.*, for $E \in \mathcal{E}[\text{Events}(P)]$ the probability of E is $\Pi_{i \in E^I} \mathbb{P}_{E_i.\mu}(\{E_i.v\})$.

Execution trees We can see that although we have assigned probabilities to executions, we do not have a clear notion of probability space over executions. Given that if two executions differs by some non-probabilistic event they are considered completely independent, it is not clear to see whether two executions should be considered to be part of the same probability space, or if they are to be considered separately. To fill this gap, we define the notion of execution tree:

Definition 11 (Execution tree)**.** For a given set X, the set of execution trees over X is noted $\mathcal{T}[X]$. A tree T is a subset of $\mathcal{E}[X]$, such that:

- For any probabilistic event E_i in an execution $E \in T$, for every outcome $v_2 \in \Omega_\mu$ in the probability space E_i, there is another execution $E' \in T$ such that, $E_j = E'_j$ for any

$j < i$ and E'_i is the same as E_i except that its outcome is v instead. That is:

$$\forall E \in T, \forall i \in E^{-1}(\mathfrak{m}, \mathfrak{o}\_, \_, \_, \_), \text{ let } (\mathfrak{m}, \mathfrak{o}, l, p, v_1, \mu) := E_i, \forall v_2 \in \Omega_\mu, \exists E' \in T,$$
$$i \in E'^I \wedge (\forall j < i, E_j = E'_j) \wedge E'_i = (\mathfrak{m}, \mathfrak{o}, l, p, v_2, \mu)$$

- Any two executions in T must have the same prefix up until some probabilistic event E_i with a different outcome. That is:

$$\forall E, E' \in T, \exists i \in E^{-1}(\mathfrak{m}, \mathfrak{o}\_, \_, \_, \_), \text{ let } (\mathfrak{m}, \mathfrak{o}, l, p, v_1, \mu) := E_i,$$
$$i \in E'^I \wedge (\forall j < i, E_j = E'_j) \wedge \exists v_2 \neq v_1,\ E'_i = (\mathfrak{m}, \mathfrak{o}, l, p, v_2, \mu)$$

For a given protocol P, each probability tree T induces a probability space for the executions of P, with executions as outcomes and.

Definition 12 (Execution trees as probability space)**.** The discrete probability space associated to an execution tree $T \in \mathcal{T}[\text{Events}(P)]$ is[4] $(\Omega_T, \mathbb{P}_T) \in \mathcal{D}$.

$$\Omega_T := T$$
$$\forall E \in T, \mathbb{P}_T(\{E\}) := \Pi_{i \in E^I} \mathbb{P}_{E_i.\mu}(\{E_i.v\})$$

Intuitively, an execution tree represents one possible probabilistic execution, which fits the view that every system run (probabilistic or not) is entirely independent from one another. Interestingly, we can see that two executions may have multiple execution trees in common, although their respective probabilities remains the same within each tree.

Adversaries and models Given that in Section 3.2 we will define a unique model $\mathcal{M}_P^{\mathcal{A}}$ for a given adversary $\mathcal{A}$ and protocol P, we introduce the notation immediately, which lets us define security with respect to an adversary $\mathcal{A}$, instead of with respect to an arbitrary model $\mathcal{M}$.

Definition 13 (Adversary)**.** An adversary $\mathcal{A}$ is a tuple with three elements.

1. Either "computational" or "unbounded" (see Section 3.2.1).

[4] This is a slight abuse of notation, since technically T is not a member of $\mathcal{D}$.

2. A corruption structure $\mathcal{C} \in \mathcal{P}(\Pi)$ (see Section 3.2.1).

3. Either "static" or a predicate D (see Section 3.2.1).

Definition 14 (Protocols models)**.** The model of P against adversary $\mathcal{A}$ is the unary predicate $\mathcal{M}_P^{\mathcal{A}} \subseteq \mathcal{T}[\text{Events}(P)]$, defined as follows:

$$\begin{aligned}\mathcal{M}_P^{\mathcal{A}}(T) := \text{MODULES}(T) \wedge \text{STRUCTURE}(\mathcal{C})(T) \wedge \text{HONEST}_D(T) \wedge \\ \text{computational} \in \mathcal{A} \Rightarrow \text{COMPUTATIONAL}(T) \wedge \\ \text{static} \in \mathcal{A} \Rightarrow (D = true \wedge \text{STATIC}(T))\end{aligned}$$

Where each component will be defined further in Section 3.2, except MODULES which is below.

The MODULES predicate is better seen as a necessary requirement of our execution model, and thus we state it here. It requires that every protocol execution is admissible for all its modules.

$$\text{MODULES}(T) := \ \forall m \in M,\ A_m(T)$$

Probabilistic security We are now ready to state the probabilistic statement of security. It is analogous to the deterministic version, except that we must quantify over execution trees instead of just executions. However we also need to account for the fact that only modules can make truly random outputs, the protocol output themselves only have a probability that is derived from the random module events that happened so far. To do so, we extend the stripping function to execution trees by converting protocols outputs into probabilistic events.

Definition 15 (Stripping function - generalized)**.** Let S be module, P a protocol such that $I_P = I_S$ and $O_P = O_S$, and let $T \in \mathcal{T}[\text{Events}(P)]$. Then $\mathcal{S}_S(T)$ is an execution tree where, for all $E \in T$, any protocol output event E_i is converted into a probabilistic event by inserting a probability space $\mu \in \mathcal{D}$ in E_i. The probability function S_F is the unique one such that the execution probability in T and $\mathcal{S}_S(T)$ are the same, *i.e.*, $\forall T, \mathbb{P}_T = \mathbb{P}_{\mathcal{S}_S(T)}$. Additionally, $\mathcal{S}_S(T)$ also applies $\mathcal{S}_S(E)$ from definition 7 to each individual execution $E \in T$.

Definition 16 (Implementation Correctness - generalized)**.** Let $P \in \mathcal{P}[M]$. P against adversary $\mathcal{A}$ is secure with respect to module S is noted $\mathcal{M}_P^{\mathcal{A}} \models S$, and defined below.

$$\mathcal{M}_P^{\mathcal{A}} \models S := \ \forall T \in \mathcal{T}[\text{Events}(P)],\ \mathcal{M}_P^{\mathcal{A}}(T) \Rightarrow A_S(\mathcal{S}_S(T))$$

Additionally, we write $\mathcal{M}_P^{\mathcal{A}}[M] \models S$ to mean that $P \in \mathcal{P}[M]$ and $\mathcal{M}_P^{\mathcal{A}} \models S$.

Random sampling Having treated probabilistic run, the definition of the module that allows random sampling is quite simple. It lets any node make an empty input, upon which it immediately returns a random bit. For the sake of clarity, we give a full formalisation this module: We have $I_{\text{rand}} = \{\text{sample}\}$, $O_{\text{rand}} = \{0,1\}$. The admissibility predicate is:

$$\begin{aligned} A_{\text{rand}}(T) := \forall E \in T,\ \forall p \in \Pi,\ \forall i \in E^{-1}(\mathfrak{m}, \mathfrak{i}, \text{rand}, p, \text{sample}), \\ i+1 \in E^{-1}(\mathfrak{m}, \mathfrak{o}, \text{rand}, p, \_, (\{0,1\}, b \mapsto 0.5)) \end{aligned}$$

This definition however only lets us have probabilistic *specifications*, like the rand module, but does not capture probabilistic *protocols* that implements specifications (probabilistic or not).

Statistical security To be deemed secure, a probabilistic protocol must be secure with a sufficiently high probability p. The precise notion of "sufficiently high" requires the existence of a security parameter λ that tunes the protocol security, *i.e.*, such that $1-p$ becomes negligible as λ is increased. In this case, a negligible function is a function that tend to 0 faster than any inverse polynomial,*i.e.*, $\text{negl}(f) := \forall c, \exists N, \forall x > N, |f(x)| < x^{-c}$.

Definition 17 (Statistical Implementation Correctness)**.** Let $P \in \mathcal{P}[M]$ be a probabilistic protocol. P against adversary $\mathcal{A}$ is statistically secure with respect to module (or statistically satisfies) S is noted $\mathcal{M}_P^{\mathcal{A}} \models^{\lambda} S$, and defined below.

$$\begin{aligned} \mathcal{M}_P^{\mathcal{A}} \models^{\lambda} S := \ & \text{params}(\mathbb{N}) \in M \wedge \\ & \text{Let } \theta_\lambda := \{\mathcal{S}_S(T) \mid T \in \mathcal{T}[\text{Events}(P)] \wedge \mathcal{M}_P^{\mathcal{A}}(T) \wedge \\ & \qquad \forall E \in T,\ \forall p \in \Pi,\ E^{-1}(\mathfrak{m}, \mathfrak{o}, \text{params}, p, \lambda) \neq \emptyset\} \\ & \text{negl}(\lambda \mapsto \max_{T \in \theta_\lambda} 1 - \mathbb{P}_T[A_S(T)]) \end{aligned}$$

Additionally, we write $\mathcal{M}^{\mathcal{A}}[M] \models^{\lambda} S$ to mean $\exists P \in \mathcal{P}[M],\ \mathcal{M}_P^{\mathcal{A}} \models^{\lambda} S$.

3.2 Modelling adversaries and primitives

In this section, we build our library of components for BFT protocols, by giving a formal definition for a restricted set of models that we presented in Chapter 2. We start by giving the predicates to capture common adversaries in Section 3.2.1, then we describe a few modules that capture typical ideal primitives in Section 3.2.2; together, they constitute the protocols' models.

Choosing the models We chose to formalise models that constitute a sufficiently complete baseline core for BFT protocols, while staying as simple as possible. We also want to be able to express most of the impossibility proofs that shape the space of blockchain protocols, the hope being that other models used in the literature may mostly be seen as variations from the baseline we present.

At the very least we need to model the three main types of synchrony assumption, as well as probabilistic algorithms. We also need digital signatures since many of the protocols will consider the authenticated setting. As we stated in the previous chapter, we prefer to have an ideal signature module and keep the possibility of information-theoretic security. However, even with ideal signatures, protocols that use *e.g.*, hash functions can only be computationally secure and we must also add this possibility to our model. Consequently, we also formalise the random oracle and obviously the Proof-of-Work oracle for protocols that use them. We include all variations of the assumptions on the adversary's ability to corrupt nodes, through a generalisation of the threshold adversary to arbitrary corruption structures. Protocols that leverage random, public committees to drive agreement such as StakeCube often rely on delayed adversarial corruption to ensure that committee members stay honest for a sufficiently long time. Therefore we also include a mechanism to capture a generic notion of weakened adaptivity. Finally, we must be able to pass some public parameters to the protocol during an initial setup phase. Because we allow to specify any parameter value to any nodes, our formalisation is actually able to express any kind of trusted interactive setup assumptions, including public key infrastructure and common reference string.

In summary, the captured models can be represented by choosing on for each of the following components:

- Adversary computational power: polynomial or unbounded.
- Adversary adaptivity: static, weakly dynamic, or dynamic.
- Corruption structure: configured with the sets of nodes that may be corrupted.
- Randomness: probabilistic, deterministic.
- Network: strong synchrony, weak synchrony, asynchrony.
- Setup: any interactive trusted setup.
- Cryptographic primitives (any combination thereof): local random coin, random oracle, signatures, PoW, verifiable random functions.

3.2.1 Adversary

Computational power

In the model described so far, any combination of events that satisfy the admissibility properties is an execution (tree). In other words, there is no bound on an adversarial computational power and we have captured the notion of information-theoretic security. To capture computationally bounded adversaries, we require executions to be the output of a polynomial-time Turing Machine. To this end, we again add a requirement to A_P.

Definition 18 (Computational adversary)**.** Let $T \in \mathcal{T}[\text{Events}(P)]$ for some protocol P. COMPUTATIONAL(T) holds iff

There is a Turing Machine $\mathcal{A}$ polynomial in λ, such that for every execution $E \in T$ all events in E are output (in sequence) by $\mathcal{A}$, except for probabilistic events, in which case $\mathcal{A}$ only output the event *without* its outcome, and then receives the event outcome as input.

The computational setting always implicitly implies a statistical implementation correctness.

Faults and corruption structure

In the above, we did not yet require honest nodes to follow the protocol. But to do so, we first need to define faults. We do so in two steps, first we define C_i, the set of corrupted nodes at index i, which is computed from the corruption events. Then we add the requirements on the behavior of nodes depending on whether they belong in C_i.

Definition 19 (Corrupt and Honest nodes)**.** Let E be a sequence and $i \in E^I$ an event index.

$$\begin{aligned} C_i &:= \{p \in \Pi \mid \exists j \in E^{-1}(\mathfrak{c}, p),\ j < i\} \\ C &:= \bigcup_{i \in \mathbb{N}} C_i \\ H_i &:= \Pi \backslash C_i \\ H &:= \Pi \backslash C \end{aligned}$$

This definition of C relies on the fact that nodes cannot recover and be removed from C_i. In the following, "honest" and "corrupt" will be synonymous with H and C, respectively. Note that theses definitions are part of the admissibility predicate, and we always require them to be present, in order to let the modules be dependent on their existence.

We now add the admissibility condition to have honest nodes execute the protocol.

Definition 20 (Honest nodes execution)**.** HONEST(T) holds iff,

For all $E \in T$, For any i in $E^{-1}(\mathfrak{i}, p, \_) \cup E^{-1}(\_, \mathfrak{o}, p, \_)$ for some $p \in H_i$, then the events following E_i are all the events returned from the transition function $F_P(S, E_i)$, in any order. The state S used for computation is inductively defined by F_P, *i.e.*, it is the state returned from the previous p output/input to module, or S_P^0 if there are none.

This definition allows the protocol to make multiple input/output at once, but lets the adversary reorder them. This is to capture, for instance, the sending of multiple messages. To have a module represent local computation, one can require the module's output to be the next event after an input.

With the model defined so far, by default corrupt nodes are not restricted in the inputs they make, and we only added the assumption that correct ones are making inputs/outputs according to the protocol. This means that with the current definitions corrupt nodes are in fact Byzantine. To simplify the definitions later in this chapter we add the assumption

that corrupt nodes do not make any protocol input or output. There is no loss of generality here because in any case the protocol inputs or outputs of Byzantine nodes are ill-defined.

$$\forall E \in T, \forall i \in E^I, \forall p \in C_i,\ i \notin E^{-1}(\mathfrak{p}, \_, p, \_)$$

For illustrative purpose, we show how crash fault could be defined within our framework, by requiring that once nodes are corrupt, they won't be making any subsequent input to any module, or outputs to the protocol.

Example (Crash faults)**.**

$$\forall E \in T, \forall i \in E^I, \forall p \in C_i,\ i \notin E^{-1}(\mathfrak{p}, \mathfrak{o}, p, \_) \cup E^{-1}(\mathfrak{m}, \mathfrak{i}, \_, p, \_)$$

Corruption structure To consistently model the limited ability of the adversary to corrupt nodes even based on hashpower or stake ownership, we choose to generalise statements regarding the count of corrupt nodes into sets of corrupt nodes. We use the term *corruption structure* to refer an assumption on the set of nodes that the adversary can corrupt at a fixed point in time.

Because the granularity of corruption is at the node level, any corruption structure can be encoded as a set of subset of nodes that may be corrupted. Hence, given $\mathcal{C} \subseteq \mathcal{P}(\Pi)$, we say that we assume the corruption structure $\mathcal{C}$ to mean that at any time during execution, the set of corrupt nodes C_i is in $\mathcal{C}$. Formally, we add this condition in the protocol execution admissibility predicate:

Definition 21 (Corruption structure)**.** Let $T \in \mathcal{T}[\text{Events}(P)]$ for some protocol P.

$$\text{STRUCTURE}(\mathcal{C}) :=\ \forall i \in E^I,\ C_i \in \mathcal{C}$$

Furthermore, we assume the corruption structure is monotone. This is not necessary strictly speaking but it makes the framing of some theorems slightly more consistent with the literature [FHM99].

$$\forall C \in \mathcal{C},\ \forall C' \subset C,\ C' \in \mathcal{C}$$

Using the terminology from Chapter 2, the assumption on the corruption structure is the formal interpretation of the trust assumption.

Prominently in the literature, the corruption structures considered are of the form $\mathcal{C}_t = \{C \in \mathcal{P}(\Pi) \mid \#C < t\}$ for some threshold $t \in [\![0, n]\!]$. Protocols are said to tolerate an optimal number of corruptions when they assume a corruption structure of this form with the highest possible threshold, which is generally either $t = \lceil \frac{n}{3} \rceil$ or $t = \lceil \frac{n}{2} \rceil$. However, we can see that this can only be optimal *among the corruption structures in* $\mathcal{C}_t$.

This is clearly apparent when considering protocols with different corruption structures, *e.g.*, in Bitcoin, if a single honest node owns more than $1/2$ of the hashpower, all the $n-1$ other nodes may be corrupt, which is more than what the corruption structure $\mathcal{C}_t$ with the optimal t would allow. The generalization of the threshold adversary we give here is in line with the formulation of Byzantine Quroum Systems by Malhki and Reiter [MR98], as well as the work of Maurer, Hirt and Fitzi on generalized adversaries [HM97, FHM99].

Thus the generalisation of a threshold adversary to corruption structures begs the question of what is the optimal corruption structure.. In that regard, we show in Section 3.5 that the theorems proving the upper bounds on t can be generalized in a straightforward manner. Concretely, we replace the threshold assumption $n > kt$ by the following predicate:

Definition 22 (k-cover)**.**

$$k\text{-cover}(\mathcal{C}) := \exists \pi_1, \ldots, \pi_k \in \mathcal{C}, \bigcup \pi_i = \Pi$$

For brevity, we omit the $\mathcal{C}$ parameter further in this thesis.

We can easily see that in $\mathcal{C}_t$, $\frac{n}{k} > t$ is equivalent to $\neg k$-cover. We use this predicate in place of $n > kt$ to generalize theorems that are reframed in our model, *i.e.*, Lemma 15 and 14, Theorems 3, 4 and 5.

It is also possible to generalize the $\mathcal{C}_t$ corruption structures into the *weighted* ones [GB11a]. For a set of nodes weights $W : \Pi \to \mathbb{Q}, \Sigma_{p \in \Pi} W(p) = 1$ and fractional threshold $\alpha \in [0; 1]$, the weighted corruption structure is :

Definition 23 (Weighted corruption structure)**.**

$$\mathcal{C}_{(W,\alpha)} := \{C \in \mathcal{P}(\Pi) \mid \Sigma_{p \in C} W(p) < \alpha\}$$

Like with $\mathcal{C}_t$, we can easily see that $k^{-1} > \alpha$ is equivalent to $\neg k$-cover within $\mathcal{C}_{(W,\alpha)}$. In effect these structures generalizes threshold assumptions such as honest majority or supermajority in a Proof-of-Work and Proof-of-Stake context, by setting the weights according

to the nodes hashpower or stake, respectively. As a result, they will be the main tool to modelize the corruption assumption of blockchain protocols.

Corruption adaptivity

We are now armed to define the notion of adversarial adaptability. This is done by changing the behavior of nodes upon corruption. Instead of executing the protocol (*i.e.*, being honest) *right until* there is a corruption event, they do so until there is a corruption event *and* some predicate D is satisfied. Here, D takes an execution $E \in \mathcal{E}[\text{Events}(P)]$, a point in the execution $i \in E^I$ a node p and is holds if p is corrupted at event E_i.

To implement this, we *modify* the definition of C_i to take D as a parameter. Because C_i is part of the definition of the HONEST predicate, it means that HONEST now takes the D argument as well.

Definition 24 (Adversarial D adaptivity)**.**

$$C_i(D) := \{p \in \Pi \mid \exists j < i,\ E_j = (\mathfrak{c}, p) \wedge D(E, i, p)\}$$

The predicate D is used to define how weakly the adversary is adaptive. For a fully dynamic (rushing) adversary, $D = (E, i, p) \mapsto \top$; for a weakly adaptive adversary, D may require for instance that t time elapsed since the corruption event.

However a static adversary requires that all corruptions happen before the beginning of the protocol, which cannot be expressed using only D.

Definition 25 (Static adversary)**.** Let $T \in \mathcal{T}[\text{Events}(P)]$ for some protocol P.

$$\begin{aligned}\text{STATIC}(T) := \ \forall E \in T, \forall k \in E^I, \forall p \in \Pi,\ D(E, i, p) \wedge \\ \forall i \in E^{-1}(\mathfrak{c}, \_), \forall j \in E^{-1}(\mathfrak{p}, \mathfrak{i}, \_, \_),\ i < j\end{aligned}$$

3.2.2 Modules

In this subsection we now give the formalisation of a few modules, including networks, signatures and PoW mining.

Setups

We can express setup assumptions by having modules output some information to the nodes prior to any corruption taking place and prior to any input being made to the protocol. We choose a slightly stronger requirement, forbidding the protocol to make any input to any module before the end of the setup. We define a predicate $SETUP$ that states that a given event E_i happens during setup:

Definition 26 (Setup events)**.**

$$SETUP(i) := \forall j \leq i,\ j \notin E^{-1}(\mathfrak{m}, \mathfrak{i}, \_, \_, \_) \cup E^{-1}(\mathfrak{c}, \_)$$

This predicate in itself does not define a module, it is intended to be used by other modules to express that some events happen before the beginning of the protocol, *i.e.*, during setup. Because any value can be given to the nodes before input, hence it is powerful enough to build modules that capture public parameters, Common Reference String (CRS), Public Key Infrastructure (PKI), or any interactive program.

Network

In this subsection we define four modules, clock, strong_sync, weak_sync and async_net.

Nodes clocks The clock interface does not accept input and simply outputs a "tick" event regularly. Because local computation time is considered insignificant, nodes can build and track any notion of time they wish. We only consider one possible setting, namely loosely synchronised clocks.

Definition 27 (Loosely synchronised clocks)**.** The module "clock" has label $l_{\text{clocks}} :=$ clock and is defined below.

$$I_{\text{clock}} := \emptyset \ ;\ O_{\text{clock}} := \{\text{tick}\}$$

$$
\begin{aligned}
A_{\text{clock}}(T) := {} & \forall E \in T,\ \exists \Phi \in \mathbb{N}, \\
& \forall p \in \Pi, \exists i \in E^{-1}(\mathfrak{m}, \mathfrak{o}, \text{clock}, p, \Phi), SETUP(i) && (3.1) \\
& \wedge \\
& \forall p \in \Pi, \forall i \in E^I, \exists j \in E^{-1}(\mathfrak{m}, \mathfrak{o}, \text{clock}, p, \text{tick}),\ j \geq i && (3.2) \\
& \wedge \\
& \forall p, q \in \Pi, \forall i \in E^I,\ \begin{array}{l} |\#\{j \in E^{-1}(\mathfrak{m}, \mathfrak{o}, \text{clock}, p, \text{tick}) \mid j \leq i\} - \\ \quad \#\{j \in E^{-1}(\mathfrak{m}, \mathfrak{o}, \text{clock}, q, \text{tick}) \mid j \leq i\}| < \Phi \end{array} && (3.3)
\end{aligned}
$$

Equation 3.1 simply states that the offset bound Φ is a public parameter to the algorithm, 3.2 requires clocks to always output new ticks, and 3.3 states that at any time, the tick count of two nodes differs by at most Φ.

Network The network lets any node input a message and a recipient, and output message reception with sender.

$$I_{\text{async\_net}} := \{0,1\}^* \times \Pi \ ;\ O_{\text{async\_net}} := \{0,1\}^* \times \Pi \tag{3.4}$$

We can notice a small issue on this formalism: the value given by the node that represents the recipient is in Π. However, theoretically Π should only exist as a formal object in our model, not an actual value that can be handled by nodes. To be formally correct, the network module should give to each node a representation of Π, for instance during setup, and sending a message to *the representation* of $p \in \Pi$ then implies a reception by p itself. To avoid cluttering, we do not make the representation explicit in this work.

This remark may seem like a trivial technical detail, but it does translate to significant practical issues. Indeed, what plays the role of the node representation in an implementation is virtually never the network address directly, but most likely a public key or a DNS record. In any case the translation to a network address involves non-trivial protocols and is very much a design concern for the algorithm at hand. For instance, in Bitcoin the issue is significantly mitigated by the fact that nodes only need to multicast messages to all other nodes, thus removing the need to address each node specifically. But the issue still remains, because the network module must address the participants as a whole, and it does require a gossip protocol, which is an integral part of Bitcoin's design.

The "Secure channel" assumption can be stated as the existence of an appropriate bijection on the send and receive events:

Definition 28 (Asynchronous secure channel)**.** The module "async_net" has label $l_{\text{async\_net}} :=$ net, its input and output sets are defined by equation 3.4 and $A_{\text{weak\_sync}}$ is stated below.

$$
\begin{aligned}
&A_{\text{async\_net}}(T) := A_{\text{clock}}(T) \wedge \forall E \in T, \\
&\exists \text{rcpt} : E^{-1}(\mathfrak{m}, \mathfrak{i}, \text{net}, \_, \_) \leftrightarrow E^{-1}(\mathfrak{m}, \mathfrak{o}, \text{net}, \_, \_) \text{ such that,} \\
&\forall p, q \in \Pi, \forall m \in \{0,1\}^*, \ \text{rcpt}(\mathfrak{m}, \mathfrak{i}, \text{net}, p, (m, q)) = (\mathfrak{m}, \mathfrak{o}, \text{net}, q, (m, p)) \\
&\forall i \in E^I, \ \text{rcpt}(i) > i \\
&\text{sent} := \text{rcpt}^{-1}
\end{aligned}
$$

The module for a partially synchronous network require the existence of a clock module. Formally speaking, the module itself is parameterized by a clock module and should be noted "weak_net(clocks)", although we omit the parameter from now on. The partially synchronous network is defined the same way as the asynchronous one, except with one added assumption.

Definition 29 (Partial synchrony)**.** The module "weak_sync" has label $l_{\text{weak\_net}} :=$ net, it is defined by $I_{\text{weak\_sync}} := I_{\text{async\_net}}$, $O_{\text{weak\_sync}} := O_{\text{async\_net}}$, and $A_{\text{weak\_sync}}$ below.

$$
\begin{aligned}
A_{\text{weak\_sync}}(T) := A_{\text{async\_net}}(T) \wedge \ & \forall E \in T, \\
& \exists \Delta \in \mathbb{N}, \forall i \in E^{-1}(\mathfrak{m}, \mathfrak{i}, \text{net}, p, (m, q)), \\
& \#\{ j \in E^{-1}(\mathfrak{m}, \mathfrak{o}, \text{clock}, p, \text{tick}) \mid i \leq j \leq \text{rcpt}(i) \} < \Delta
\end{aligned}
$$

This definition fits the model of partial synchrony where Δ exists but is unknown to the protocol. Recall that in the alternative one there is an unknown time (GST) after which a known bound Δ holds. We choose this definition because (I) the GST formalisation is a higher level abstraction, and (II) formalizing GST would have require a small technical twist in the definition to be strictly equivalent to the one we have just given. To see this, consider a protocol that should implement a GST network given a partially synchronous one. To be correct, the GST network should output a Δ_{GST} value at the beginning of the execution (*e.g.*, as a setup), and then provide a network interface that will guarantee a Δ_{GST}

bound after some finite time. It seems obvious that, given that we have no information on the actual $\Delta_{partial}$ of underlying partially synchronous network, we cannot do better than give an arbitrary value, which is unsuccessful in case $\Delta_{GST} > \Delta_{partial}$. The solution in practice is quite known and simple: Output an arbitrary Δ_{GST}, and periodically increase its value. Therefore, we could make a GST definition equivalent by requiring the network provide Δ_{GST} not just once at the protocol onset, but a finite number of times during execution. Then the two formulations would be equivalent, with the only difference being that the GST abstraction encapsulate the mechanism to estimate and update Δ.

Then as well, to define the strongly synchronous network we add the assumption that Δ is given as a public parameter:

Definition 30 (Strong synchrony)**.** The module "strong_sync" has label $l_{\text{sync\_net}} :=$ net, it is defined by $I_{\text{strong\_sync}} := I_{\text{async\_net}}$, $O_{\text{strong\_sync}} := O_{\text{async\_net}}$, and $A_{\text{weak\_sync}}$ below.

$$A_{\text{strong\_sync}}(T) := A_{\text{weak\_sync}}(T) \wedge \forall p \in \Pi, \exists i \in E^{-1}(\mathfrak{m}, \mathfrak{o}, \text{net}, p, \Delta),\ SETUP(i)$$

Once again we choose the formulation closer to reality. The alternative round-based formulation (see Section 2.2.2) being more useful for theoretical analysis. However in this the "loose synchrony assumption" lets us easily bootstrap to a lock-step execution by emulating the round-based network: Let round 0 start at a fixed time, and set the round duration to $\Delta + \Phi$. In the case where we only have strong *clock* synchrony, we must either have rounds of growing (unbounded) length, or to execute an agreement protocol to periodically re-synchronize clocks. Interestingly, this means that the two formalisms are not strictly equivalent without the loose synchrony assumption.

Our definition for strong synchrony matches the case of a rushing adversary, because it is able to make any message (*e.g.*, its own) delivered instantaneously, before any other.

Oracles

In this subsection we introduce a few modules of interest that capture an idealized version of cryptographic primitives, including Proof-of-Work mining. We describe the modules for signatures, the random oracle and verifiable random functions in a slightly less formal manner, since their definition is already unambiguous in the literature.

Public parameters For completeness we describe the "params" module that gives to nodes all the public parameters of the protocol.

Definition 31 (Public parameters module)**.** The module "params" is parametrized by the set of public parameters, PP. We have $I_{\text{params}(PP)} := \emptyset$, $O_{\text{params}(PP)} := PP$, and,

$$A_{\text{params}(PP)}(T) := \forall p \in \Pi, \exists i \in E^I,\ E^{-1}(\mathfrak{m}, \mathfrak{o}, \text{params}, p, \_) = \{i\} \wedge SETUP(i)$$

Ideal signatures The "signatures" module is an ideal abstraction of a signature scheme. It is defined similarly to the certification functionality $\mathcal{F}_{\text{CERT}}$ by Canetti [Can04], who showed how to implement it using a setup (PKI) and an EUF-CMA secure signature scheme. Note that these definition purposefully encapsulate and hide any PKI assumption necessary for the implementation. In short:

Definition 32 (Ideal signatures module)**.** The module "signatures" is parameterized by a countable message space M.

$$I_{\text{signatures}} := \{\text{sign}\} \times M \cup \{\text{verify}\} \times M \times \{0,1\}^* \times \Pi$$

$$O_{\text{signatures}} := \{\text{sign}\} \times \{0,1\}^* \cup \{\text{verify}\} \times \{\text{true}, \text{false}\}$$

$A_{\text{signatures}}(T)$ is defined as follows: Upon input (sign, m) from node p, immediately output (sign, σ) to p, where σ may be any value in $\{0,1\}^*$. Upon input $(\text{verify}, m, \sigma, q)$ from node p, immediately output (verify, b) to p, where $b = \text{true}$ iff node q previously made the input (sign, m) to the module and received (sign, σ).

Random oracle Contrarily to the ideal signatures where even an unbounded adversary cannot forge a signature, the random oracle may still require to be used with a polynomial adversary, *e.g.*, when using the collision resistance property.

Definition 33 (Random Oracle)**.** The module "RO" is parameterized with a message space M and has $I_{\text{RO}} := M$, $O_{\text{RO}} := [\![0, 2^\lambda - 1]\!]$, and $A_{\text{RO}}(T)$ is defined as follows: Upon input x from node p, if x has never been input to the module, then immediately output a probabilistic event with the uniform distribution on O_{RO}. Otherwise, immediately output the outcome from the first probabilistic output that followed x input.

Verifiable Random Functions Since this cryptographic primitive is a component of several schemes presented (including StakeCube), we give a definition here. This definition follows the same paradigm than with ideal signatures.

Definition 34 (Verifiable Random Function)**.** The module "VRF" is parameterized by a countable message space M.

$$I_{\text{VRF}} := \{\text{eval}\} \times M \cup \{\text{verify}\} \times M \times \{0,1\}^* \times [\![0, 2^\lambda - 1]\!] \times \Pi$$

$$O_{\text{VRF}} := \{\text{eval}\} \times \{0,1\}^* \times [\![0, 2^\lambda - 1]\!] \cup \{\text{verify}\} \times \{\text{true}, \text{false}\}$$

$A_{\text{VRF}}(T)$ is defined as followss: Upon input (eval, m) from node p, immediately output a probabilistic event (eval, σ, r) to p where r is from the uniform distribution on $[\![0, 2^\lambda - 1]\!]$ and $\sigma \in \times\{0,1\}^*$. Upon input $(\text{verify}, m, \sigma, r, q)$ from node p, immediately output (verify, b) to p, where $b = \text{true}$ iff node p previously input (eval, m) to the module and obtained (sign, σ, r).

PoW oracle We now give a module for the PoW oracle. Obviously the random oracle alone is not sufficient to express PoW, since it has no notion of time. Hence we define a PoW oracle tightly linked to the clock module. In short, it lets nodes configure it with a message m and probability d, and periodically, depending on the node hashrate, it will output a value which is a valid PoW string with probability d.

Definition 35 (Proof-of-Work oracle)**.** The module "PoW" is parametrized by a hashrate $h_p \in \mathbb{N}$ for each node $p \in \Pi$ and a countable message space M. It relies on the existence of the clock module.

$$I_{\text{PoW}} := \{\text{config}\} \times M \times \mathbb{N} \cup \{\text{verify}\} \times \{0,1\}^* \times M \times \mathbb{N}$$

$$O_{\text{PoW}} := \{\text{mine}\} \times \{\text{success}, \text{fail}\} \times \{0,1\}^* \cup \{\text{verify}\} \times \{\text{true}, \text{false}\}$$

$A_{\mathrm{PoW}}(T)$ is informally explained after its definition.

$$
\begin{aligned}
A_{\mathrm{PoW}}(T) := \quad & \\
& \forall p \in \Pi, \forall i, j \in E^{-1}(\mathfrak{m}, \mathfrak{o}, \mathrm{PoW}, p, (\mathrm{mine}, \_, \_), \_), \\
& \#\{k \in E^{-1}(\mathfrak{m}, \mathfrak{o}, \mathrm{PoW}, \mathrm{clock}, p, \mathrm{tick}) \mid i < k < j\} = h_p \qquad (3.5) \\
\wedge & \\
& \forall p \in \Pi, \\
& \min E^{-1}(\mathfrak{m}, \mathfrak{o}, \mathrm{clock}, \mathrm{PoW}, p, \mathrm{tick}) = \min E^{-1}(\mathfrak{m}, \mathfrak{i}, \mathrm{PoW}, p, (\mathrm{config}, \_, \_)) + 1 \qquad (3.6) \\
\wedge & \\
& \forall p \in \Pi, \forall \sigma \in \{0,1\}^*, \forall \mu \in \mathcal{D}, \forall i \in E^{-1}(\mathfrak{m}, \mathfrak{o}, \mathrm{PoW}, p, (\mathrm{mine}, \_, \sigma), \mu), \\
& \Omega_\mu = \{(\mathrm{mine}, r, \sigma) \mid r \in \{\mathrm{success}, \mathrm{fail}\}\} \wedge \mathbb{P}_\mu(\{(\mathrm{mine}, \mathrm{success}, \sigma)\}) = d \qquad (3.7) \\
\wedge & \\
& \forall p \in \Pi, \forall (\mathrm{verify}, \sigma, m, d) \in \{\mathrm{verify}\} \times \{0,1\}^* \times M \times \mathbb{N}, \\
& \forall i \in E^{-1}(\mathfrak{m}, \mathfrak{i}, \mathrm{PoW}, p, (\mathrm{verify}, \sigma, m, d)), \\
& i+1 \in E^{-1}(\mathfrak{m}, \mathfrak{o}, \mathrm{PoW}, p, (\mathrm{verify}, b)), \text{ where,} \\
& b = \mathrm{true} \Leftrightarrow \exists q \in \Pi, \exists j \in E^{-1}(\mathfrak{m}, \mathfrak{o}, \mathrm{PoW}, q, (\mathrm{mine}, \mathrm{success}, \sigma), \_),\ j < i \wedge \\
& \exists d' \geq d,\ E_k.v = (\mathrm{config}, m, d') \text{ where } k := \max\{k' \in E^{-1}(\mathfrak{m}, \mathfrak{i}, \mathrm{PoW}, p, \_) \mid k' < i\} \qquad (3.8)
\end{aligned}
$$

Predicate 3.5 requires that, for any node p, for any two outputs E_i and E_j to p, there are exactly h_p "tick" outputs from the clock module to p. Predicate 3.5 requires that, for any node p, the first output to p is immediately after the first input (config, m, d) from p. Predicate 3.7 requires that, for any node p, the outputs of the form $(\mathrm{mine}, r, \sigma)$ are probabilistic with $\mathbb{P}[r = \mathrm{success}] = 1 - \mathbb{P}[r = \mathrm{fail}] = d$ and $\sigma \in \{0,1\}^*$. Predicate 3.8 requires that, upon input $(\mathrm{verify}, \sigma, m, d)$ from p, immediately output (verify, b) to p, where $b = \mathrm{true}$ iff there is a node q such that q previously received the output E_i with value $(\mathrm{mine}, \mathrm{success}, \sigma)$ and q's last input before E_i was (config, m, d') with $d' \geq d$.

First and foremost, we made the choice to capture a slighter higher level abstraction than a hash function with a bounded number of queries. Specifically, we encapsulate the mechanism that compares the hash output with a target, and discards it if the target is not met. This formulation makes the intended function of Proof-of-Work mining more evident.

As previously observed by Garay*et al.* [GKP20], the PoW mining primitive is analogous to digital signatures, with the difference that instead of authenticating a value with respect to a specific node, it authenticates a value with respect to the ownership of a specific amount of hashrate. Intuitively, both primitives are used in a similar way, to let the protocol know what share of the network is vouching for a message. In both cases, the share of the node is deduced (implicitly or not) from the knowledge of the size of the whole network. The fundamental difference from digital signatures is the introduction of time, which makes equivocation impossible: Given a valid PoW, we are guaranteed (probabilistically) that some node has spent the required time to "sign" the associated message, *and no other*.

Notably, the PoW oracle does not directly give the value of the nodes hashpower to each other, instead, it has to be estimated from the received PoWs. In turn, if ones wishes to estimate node hashpower only from PoWs, then this is only possible within a strongly synchronous networks. The intuition is that delaying the transmission of a PoW decreases the apparent hashpower of the mining node. To compute a bound on the miner hashpower, the estimate on the mining time has to include the network delay, which therefore must be known. To put this argument more formally:

Theorem 1. *Let* $P \in \mathcal{P}[\text{PoW}, \text{weak\_net}]$ *such that* $\mathcal{M}_P^{(\text{unbounded}, \mathcal{C}, (E,i,p) \mapsto \top)} \models^{\lambda} B$ *where* $\mathcal{C} \neq \emptyset$ *and* B *is a module that outputs a bound on the nodes hashpower, i.e.,*

$$O_B := \{0,1\}^* \times \mathbb{N} \; ; \; I_B := \emptyset$$
$$A_B(T) := \forall E \in T, \exists (\sigma, b) \in O_B, \; \#E^{-1}(\mathfrak{m}, \mathfrak{o}, l_B, \_, (\sigma, b)) \neq \emptyset \; \wedge \; b \geq h_{p_2}$$

Then there exists $P' \in \mathcal{P}[\text{PoW}, \text{weak\_net}]$ *whose output is a bound on the network delay* δ.

Proof Sketch. First let us observe that a valid PoW only contains information on the number of attempts that have been made to obtain it. Thus, for a fixed security parameter λ, a received PoW only gives a bound on ζh_p^{-1} where ζ is the time spent mining on that PoW

and p is the mining node. Thus computing a bound on h_p implies a known bound $\zeta' \geq \zeta$. At one extreme, corrupt nodes are able to send a PoW instantly after mining, therefore we must assume that mining ended at the latest time, *i.e.*, at the reception of the PoW string. At the other extreme, the adversary can force any PoW sent over the network to be delayed by δ. As a result, regardless of how ζ' is computed, we have $\zeta' > \zeta + \delta$ which implies that ζ' is actually a known bound on the network delay. Thus P' is simply constructed by outputting ζ' after the first P output. □

One may be surprised that the oracle apparently does not give information on the total hashpower to nodes. However if honest nodes are required to transmit their PoW and if they manage to agree on a common set of these PoWs, they can deduce a common upper bound on the hashpower of honest nodes and in turn of the whole network. In essence, this is what the difficulty adjustment mechanism does for Bitcoin: it uses the set of common PoW stored in the blockchain to deduce a bound on the overall hashrate change and update the protocol parameters accordingly. With that in mind, it becomes apparent that at any time during execution, the evaluation of the hashpower share associated to a PoW string is done with respect to *last estimated* total hashpower. Hence the protocol must assume that the hashpower of newly joining miners has a bound which can be accounted for [GKL20b, CEM+20], which goes against the folklore idea that joining nodes in PoW-based protocols do not need to register. In that sense, Bitcoin ignores the participation of new nodes, until they have participated for long enough to be "noticed" and have their hashpower registered as part of the difficulty adjustment.

3.2.3 Discussion

To synthesize, we have a framework where a protocol P uses a module list M and securely satisfy a specification module S against an adversary $\mathcal{A}$. Statements of perfect and statistical security are $\mathcal{M}_M^{\mathcal{A}} \models S$ and $\mathcal{M}_M^{\mathcal{A}} \models^{\lambda} S$, respectively. The constraints on the overall execution are defined by choosing an adversary $\mathcal{A}$. We provide definition for an adversary that may be computationally bounded or not, limited by a given corruption structure, and static or dynamic. Furthermore, we have defined the following modules:

- Three versions of a clock plus network module. The clock is always the same and represents the loose clock synchrony assumption, *i.e.*, a bounded offset. The three versions are for asynchronous, partially and strongly synchronous networks.
- A setup module that lets nodes obtain public protocol parameters, including the security parameter λ in the case of statistical security.
- Four oracle modules, one for ideal signatures, one for ideal Verifiable Random Functions, one Proof-of-Work oracle, and one Random Oracle.
- One module to modelize local randomness generator (coin toss).

Limitations There are two important points that limit the usefulness of our framework as currently stated. First, we cannot express dynamic participation, where the node set Π may change during execution. This does not preclude us from analyzing protocols that support this model (*e.g.*, reconfigurable protocols) since strictly speaking we capture a subset of the possible executions. Secondly, we only capture an omniscient adversary that has all information about the system. This is technically not an issue within this thesis due to our idealized treatment of *e.g.*, signatures that would otherwise require secret information.

Multiple sessions The presented framework defines the executions of a single session of a protocol. One may represent multiple sessions of a module through a single module that can start an arbitrary number of concurrent sessions. In general, this is implemented by assigning unique session identifiers.

On the notion of time The notion of time backing the PoW oracle is taken from the nodes clocks. An alternative could have been to use a newly defined absolute, global time and require its consistency with local clocks, such as in [LHSW20]. However we argue that this second approach is more cumbersome, and is at best equivalent because in any case nodes only get time information from their clocks. Furthermore, our formulation clearly highlights that the oracle is as useful as the clocks (and network) are synchronized.

Infinite runs The possibility of infinite executions instead of say, arbitrarily long execution, seems unrealistic and more complicated than necessary. However, infinite runs are required to modelize admissible non-terminating runs, *e.g.*, in reliable broadcast. Even more, we could describe an ever-going protocol specification that has no finite admissible run, because the protocol is not intended to be stopped.

But more importantly, a protocol requiring a computationally bounded adversary cannot have infinite execution. Indeed, a polynomial Turing Machine is obviously unable to output an infinite sequence of event. Protocols that implement a specification requiring infinite runs do so by inductive reasoning : Assuming that the protocol has been correct up until time t, they show how to extend its execution to time $t+1$[5]. In the case of computational security, the step from t to $t+1$ has a negligible chance to fail, hence why such security proofs are conditional to a polynomial execution length. It becomes apparent that this condition is redundant if the formal framework for the security proof already requires a polynomial adversary. With regard to practice, this remark translates to the fact that with a concrete parametrization, the tolerated probability of failure directly implies a limit in the execution length.

3.3 Problems

In the following section we give specifications for BFT protocols, in terms of properties to be satisfied. The modules described in this section can be used as an abstraction of the problem solved by a protocol, thus with the models from Section 3.2 we are completing the library of component we aimed to build.

Each relevant property is expressed both as English statement from the literature and its formal interpretation as a predicate on executions[6]. Each module m is defined by having its admissibility predicate A_m be the conjunction of the stated properties. Nonetheless, we allow ourselves to change the formulation of the properties from their original statement as long as it stays equivalent, for the sake of coherence and precision. Only if the original statement is not correct strictly speaking we will make non-equivalent changes.

[5] A common term for such steps is *epoch*.

[6] All specifications in this section are deterministic, otherwise the properties would be predicates on execution trees.

Remark We expressed all interactions between the node and the protocol implementation in terms of input/output. More specifically, nodes instantiate and start the protocol by making inputs. This has the advantage of making explicit how each specification expects the protocol to be started, *e.g.* by assuming that all nodes eventually makes an input.

Reductions The main way to understand the relations between different definitions is to examine the *reductions* between models. But in order to state the definition, we first introduce two notations. To ensure that a reductions holds for any adversary, we use the strongest possible adversary $\mathcal{A}^S$.

$$\mathcal{A}^S := (\text{unbounded}, \mathcal{P}(\Pi), (E, i, p) \mapsto \top)$$

Additionally, for a module A, we note A^* the set of modules with an arbitrary but finite number of copies of A, each copy being distinguished by a unique label.

Definition 36 (Modules Reduction)**.** For modules A and B, a reduction from A to B is a protocol $P \in \mathcal{P}[B]$ such that $\mathcal{M}_P^{\mathcal{A}^S} \models A$.

Reductions induce a preordering on the modules, where $A \geq B$ iff there is a reduction from B to A. We say that A is weaker, stronger, or equivalent to B to mean $A \leq B$, $A \geq B$ and $A \geq B \wedge A \leq B$, respectively[7].

This is the natural definition to order models by solvability, which is mostly useful to characterise the strength relation between different problems. However in many cases it may seem overly restrictive. For instance, strictly speaking, consensus cannot readily be transformed into reliable broadcast although it is considered to be a stronger problem. This is because, although there is a reduction that solves reliable broadcast given an implementation of consensus, it relies on the broadcast sender to initially transmit its input value, *e.g.* through the network. However the consensus specification module by itself does not include a network. More intuitively, this strict strength relation should be interpreted as a direct implication between the modules properties.

This is why we put forth a less restrictive order relation, where the protocol may use an asynchronous network as well as other instances of the initial modules.

[7]Note that weaker or stronger are not in the sense of "strictly weaker" or "strictly stronger" here.

Definition 37 (Loose Modules Reduction)**.** For modules A and B, a reduction from A to B is a protocol $P \in \mathcal{P}[B \cup M]$ such that $\mathcal{M}_P^{A^S} \models A$, where $M \subset \{\text{async\_net}, \text{signatures}, A^*\}$.

The loose reductions also induce preordering on the modules, where $A \geq_{\text{loose}} B$ iff there is a loose reduction from B to A. We also say that they are loosely weaker, loosely stronger, or loosely equivalent with respect to that order.

The goal of this definition is to still faithfully capture a "solvability" order, because we do not assume specific corruption structure nor synchronicity, while not guaranteeing that the transformation is efficient in practice. The downside is that

Note that we do not consider reductions that requires assumptions on the network synchronicity or the adversarial structure, because such assumptions can be used to build protocols from scratch. They can be useful to build practical protocols, but they are unfit to characterize strength relationship between definitions.

The relationship between the models we have given so far can be simply deduced from the choices of the modules and the adversary.

One-shot and continuous protocols We propose to distinguish two types of protocols: one-shot and continuous. Intuitively, One shot are protocols such that for all executions, either : there is a time t at which we can consider the protocol as terminated, or, the protocol did not terminate on that run. One-shot protocols may be simpler to analyse because executions don't need to be infinite, only finite and arbitrarily long. This dichotomy is typically made informally, however we can give it a precise meaning with our framework.

Definition 38 (One shot and Continuous protocols)**.** A module m is said to be one-shot iff for all its admissible executions its set of outputs is finite, *i.e.*,

$$\forall E \in \{E' \in T \mid T \in \mathcal{T}[\text{Events}(m)] \wedge A_m(T)\},\ \#E^{-1}(\mathfrak{m}, \mathfrak{o}, \_, \_, \_) \in \mathbb{N}$$

Otherwise, m is continuous.

Note that this definition relies on the assumption that Byzantine nodes do not make protocol outputs or inputs, and we will continue to do so in the remaining of this thesis. In particular this mean that, in context, for most of the following uses, the statement "for all honest nodes" will be equivalent to "for all nodes". We chose to keep the "honest" qualifier only for statements made in plain English, to keep the wording closer to the literature.

3.3.1 Consensus and variants

In this subsection we define five variants of consensus : multivalued, binary, strong, weak and vector consensus. They all are one-shot protocols, they all are parametrised by a countable input set V, *i.e.*, $I_{consensus} := V$, they all require that honest nodes eventually make an input with a value in V, and they all include the two following properties, agreement and liveness.

Property (Agreement/Consistency)**.** All honest nodes' outputs are equal.

$$\exists v_{\text{out}} \in V, \forall i \in E^{-1}(\mathfrak{m}, \mathfrak{o}, \_, \_, v'),\ v' = v_{\text{out}}$$

This property defines a unique value v_{out} which is the protocol output. We reuse such defined v_{out} for other problems where the same agreement property is also required.

Property (Liveness/Termination)**.** All honest nodes eventually output a value (exactly once).

$$\forall p \in H,\ \#E^{-1}(\mathfrak{m}, \mathfrak{o}, \_, p, \_) = 1$$

The definition of each consensus variant is completed by a different Validity property. They are listed below.

Binary and multivalued consensus In the binary version of consensus (BC), the input domain V has only two values. Binary consensus is of particular interest, because its simplicity means that there are no variations in the definition and as a result is a useful base case to make impossibility proofs, bounds, and reductions.

Property (Binary Validity)**.** $\#V = 2$, and if all honest nodes input the same value v, then the output will be v.

$$\exists v \in V,\ \forall i \in E^{-1}(\mathfrak{m}, \mathfrak{i}, \_, \_, v'), v' = v \Rightarrow v = v_{\text{out}}$$

Because there is only two values to choose from, we can see that the output will always be the input of some honest node, even if honest nodes propose different values. In practical implementations, this version of consensus is mostly useful when used as primitive

for other protocols, *i.e.* when the parent protocol may ensure that all honest nodes have the same input.

In the multivalued version the input domain V is arbitrary. The validity condition is less obvious than in the binary case, which led to multiple (non equivalent) definitions for consensus (C), strong consensus (SC) and weak consensus (WC), respectively.

Property (Validity [DLS88, MR97, Fit03])**.** If all honest nodes input the same value v, then the output will be v.

$$\exists v \in V,\ \forall i \in E^{-1}(\mathfrak{m}, \mathfrak{i}, \_, \_, v'),\ v' = v \Rightarrow v = v_{\text{out}}$$

Property (Strong Validity [FG03, Nei94])**.** There is some honest node such that its input is the protocol output.

$$\exists p \in H,\ E^{-1}(\mathfrak{m}, \mathfrak{i}, \_, p, v_{\text{out}}) \neq \emptyset$$

Property (Weak Validity [Fis83])**.** If there are no failure, then there is some (honest) node such that its input is the protocol output.

$$C = \emptyset \Rightarrow E^{-1}(\mathfrak{m}, \mathfrak{i}, \_, \_, v_{\text{out}}) \neq \emptyset$$

The first two are directly inspired from the binary case. In fact, if $\#V = 2$ then they are both equivalent to binary validity.

A relaxed version of Strong Validity has been considered, where the output may be the input of *any* node, corrupt or not [BHRT00, CVL10, SvR08]. However this definition is flawed in the context of Byzantine faults, since Byzantine nodes do not have an input. Indeed, this relaxed version of Strong Validity has been primarily intended for the model with crash faults. More generally, in the context of Byzantine faults, statements about state and/or computation of nodes can only qualify over honest nodes.

One may think Validity as an ill-suited generalization of the binary consensus. Indeed, if we think of an application where all nodes make input directly from the environment (*i.e.* there is no top level protocol determining inputs), then it would seem that in normal operation there is always two honest parties with different inputs and therefore the output can be anything. However we will see in the following that this is not as restricting as it seems.

On the contrary, Strong Validity may intuitively be seen as a "correct" reformulation of Binary Validity in the multivalued case. However this leads to a much stronger definition of consensus (see Section 3.5).

External validity consensus (EC) External Consensus is parametrized by a validity predicate $P : V \to \{\text{true}, \text{false}\}$, which determines whether an output value is *valid* or not. This definition, introduced by Cachin *et al.* [CKPS01], has been put forward in the context of blockchains, with the justification that to iteratively output new blocks through consensus, we only need the block to be valid with respect to the application semantics, and we don't mind to output messages (*i.e.* transactions) made by Byzantine nodes [CV17, Gra20].

Property (External Validity [CKPS01])**.** The output value of honest nodes satisfy P.

$$P(v_{\text{out}}) = \text{true}$$

This definition however calls for some clarifications: we choose to interpret P as an algorithm given as a public parameter to the protocol. The other case would be where P is essentially a placeholder predicate that parametrise the problem definition itself, such as "$P(x) := x$ is the input of an honest node". This second option seems problematic because it is essentially equivalent to having an arbitrary validity property, which would be impossible to solve in general. For this reason we put it aside.

It is worth noticing that this issue on clarification of the nature of the P parameter can also be raised for the nature of the input set V. Indeed, like P, little is said about what the protocol knows about V, for instance: Is it computable, is there a test membership/sampling procedure (*i.e.*"$\in$"), does the protocol have representation of V ? At the very least, a representation and an (efficient) test membership procedure must be given to the protocol, otherwise it is not possible to ensure that the output is in V. This leads us to see that the requirement that the output being in V was implicit until now, although it must be present for any multivalued consensus. More formally:

$$O_{consensus} := V$$

where "consensus" is any of the consensus we have defined so far: binary consensus, strong consensus, consensus, external consensus. This observation leads to the conclusion that there is no point to distinguish *externally valid* values from the values in V, since in both cases their only purpose is to ensure that the output belongs to some arbitrary set. Indeed, external consensus may be equivalently stated as a multivalued consensus without any further Validity property, simply by setting $P(v) := v \in V$, and conversely $V := \{v \mid P(v)\}$.

This may come as a surprise, because if EC removes the validity requirement with respect to consensus, then it would seem that there is no requirement at all on the output value. However, it still has to be related to the nodes inputs, because the protocol implementation only has the P procedure at its disposal, and no way to sample or find an element in V otherwise. This could seen as a technical quirk, but we argue that this is a fundamental property of external consensus. In practice, the set V *is* generally difficult to sample from, *e.g.* because it requires a valid signature. Conversely, say if $0 \in V$, then an external validity consensus can trivially always output 0. In the context of blockchain, if the empty block is considered valid, then there is nothing *in the specification* preventing the adversary to always force the output of the empty block, *e.g.*, in [CGLR18]. Furthermore, even if the external consensus protocol may include the honest node's input in a normal operation (*e.g.* without faults), the adversary could force it to always output the empty bloc.

As a small additional note, there is another implicit parameter to multivalued consensus: A procedure for deciding equality in V, otherwise different inputs cannot distinguished.

Vector consensus (VC) Closely related to interactive consistency [PSL80] in the context of crash faults. It let all nodes agree on a vector of values where each element is a node input. This mean that for vector consensus we have $O_{VC} := \Pi \to V \cup \{\boxtimes\}$ where $\boxtimes$ represents "no value". This formulation has been initially proposed by Doudou [DS98] as an alternative to strong consensus, motivated by the observation that quantifying over all nodes is meaningless with Byzantine nodes. The vector validity is stated as follow [CNV06, DS98]: If node p is honest, then the element of the output at the index p must be equal to p's input or have no value at all. Additionally, at least $t+1$ non-empty values must be from honest nodes, where t is the maximum count of tolerated faults.

This definition is not entirely satisfactory. In [BKR94], Ben-Or*et al.* defined a close cousin of vector consensus, the Asynchronous Common Subset [MXC+16]. In their definition, instead of requiring (I) strictly more than t non-empty values from honest nodes, they require (II) less than or equal to t empty values. Indeed, the adversary can always make all corrupt nodes mute, ensuring t empty values. Then requirement (II) essentially states that there are no more empty values than the ones that may be from corrupt nodes. It is easy to see that (II) implies (I) if and only if $n > 3t$, and (I) does not imply (II). But

more importantly, atomic broadcast (*i.e.* SMR) is only stronger than vector consensus with (II), and therefore is not stronger than vector consensus with (I) if $n \leq 3t$. Similarly, the loose reduction to binary consensus is with (I). Therefore, we prefer to choose definition (II).

To the extent of our knowledge, there is no definition of vector consensus for general corruption structures. In any case, the generalization is straightforward, the key observation being that "$\leq t$" translates to "$\in \mathcal{C}$".

Property (Vector validity - II, generalized)**.** If node p is honest, then the element of the output at the index p must be equal to p's input or have no value at all. Additionally, the set of nodes with empty values must be a member of $\mathcal{C}$.

$$\begin{aligned}&\text{Let } v_{\text{in}} : \Pi \to V \text{ such that } \forall p \in \Pi,\ E^{-1}(\mathfrak{m}, \mathfrak{i}, \_, p, v_{\text{in}}(p)) \neq \emptyset \\ &\text{then } \forall p \in H,\ v_{\text{out}}(p) = v_{\text{in}}(p) \wedge \\ &\{p \in \Pi \mid v_{\text{out}}(p) = \boxtimes\} \in \mathcal{C}\end{aligned}$$

Interestingly, this is the only problem that is dependent on the existence of a threshold (or a corruption structure), although the definition itself does not make any assumption on its value. In the following we will implicitly refer to this version of vector consensus.

Reductions

Lemma 1. *Strong consensus is stronger than consensus and weak consensus.*

Proof. If all nodes submit the same input, then a strong consensus protocol guarantees that it will be the output. Also Weak validity is exactly Strong Validity conditioned to the absence of corrupt nodes. □

Lemma 2. *Consensus is stronger than binary consensus and external consensus.*

Proof. Simply instantiate Consensus with $V := \{0,1\}$ for the binary case. Also external consensus is just consensus with Validity removed. □

Lemma 3. *Weak consensus is stronger than external consensus.*

Proof. Trivially, external consensus is also consensus with Validity removed. □

Lemma 4 ([BKR94])**.** *Binary consensus is loosely stronger than vector consensus.*

This reduction has been first proposed by Ben-Or*et al.* [BKR94]. It relies on reliable broadcast, which itself is loosely weaker than weak consensus (Lemma 8).

Proof. Every node start a reliable broadcast instance as a sender, with its value as input. Then they wait until the set senders from the reliable broadcast that did *not* terminate is a member of $\mathcal{C}$. For each node p, spawn an associated binary consensus instance and input 1 if the RBC instance with sender p delivered a value, and 0 otherwise. Wait for all binary consensus to terminate, and for all RBC instances where the associated binary consensus returned 1. The vector output is formed by those RBC values and $\perp$ for the nodes where binary consensus returned 0. □

Lemma 5. *External consensus is loosely stronger than vector consensus.*

This reduction is very similar to lemma 4 but replaces the binary consensus instances by one external consensus, it starts in the same manner and also requires reliable broadcast.

Proof. Every node starts a reliable broadcast instance as a sender, with its value as input. Then they wait until the set senders from the reliable broadcast that did *not* terminate is a member of $\mathcal{C}$. Input the vector of values from RBCs that terminated to external consensus, with V requiring a vector of signed values with the set of empty values in $\mathcal{C}$. Wait from the external consensus output and return its value. □

Lemma 6. *Vector consensus is stronger than weak consensus.*

Proof. The consensus input are directly transmitted to vector consensus instantiated with $O_{VC} = \Pi \to V \cup \{\perp\}$. When the vector is output, return the value from an arbitrarily fixed index. If there are no faults, this value can only be from one of the honest processes. □

3.3.2 Broadcasts protocols

Terminating reliable broadcast (TRBC)

TRBC is the broadcast variant of consensus. In the Byzantine case, termed as Byzantine generals problem, or Byzantine agreement [LSP82]. It is a one shot protocol parametrized by a specific sender node $s \in \Pi$. It assumes that eventually the sender makes an input, and has the same Agreement and Liveness properties as consensus.

Property (Validity)**.** If the sender is honest, then the output of all honest nodes is the sender's input.

$$s \in H \Rightarrow E^{-1}(\mathfrak{m}, \mathfrak{i}, \_, s, v_{\text{out}}) \neq \emptyset$$

Implicitly the output can be anything if the sender is corrupt. Because a corrupt sender may not send any message, honest nodes need a default value that they can output, such that $O_{TRBC} = \{0,1\}^l \cup \{\boxtimes\}$. For the same reason, broadcast agreement can only be solved with synchronous networks (at least in all the models we formalise), as the nodes need a way to detect a silent sender in finite time to make an output.

A weaker variation is possible by allowing honest nodes to output $\perp$ even if some others did not. Contrary to the consensus problem, there is no issue here about the validity condition. Furthermore, with this validity condition, we can see that nodes don't need any knowledge about the input set, merely a representation for storage and a procedure for testing equality in V. Hence we decide to choose a canonical one, *i.e.* binary strings, and drop the need for an arbitrary set V as a parameter.

Reliable broadcast (RBC)

RBC [LSP82] is a one shot protocol parametrized by a specific sender node $S \in \Pi$. It assumes that eventually the sender makes an input.

Property (Agreement/Consistency)**.** All honest nodes outputs are equal.

$$\exists v_{\text{out}} \in V, \forall i \in E^{-1}(\mathfrak{m}, \mathfrak{o}, \_, \_v'), v' = v_{\text{out}}$$

Property (Liveness/Termination)**.** If an honest node output a value, then all honest players eventually output a value (exactly once).

$$\exists p \in H,\ E^{-1}(\mathfrak{m}, \mathfrak{o}, \_, p, \_) \neq \emptyset \Rightarrow \forall p' \in H,\ \#E^{-1}(\mathfrak{m}, \mathfrak{o}, \_, p', \_) = 1$$

Property (Validity)**.** If the sender is honest, then some honest node eventually output the sender's input.

$$S \in H \Rightarrow \exists p \in H,\ E^{-1}(\mathfrak{m}, \mathfrak{o}, \_, p, v_{\text{in}}(S)) \neq \emptyset$$

The Liveness property for RBC is significantly different and much weaker than with consensus and broadcast: the protocol is allowed to not terminate if the sender is malicious and no honest node terminates. In that sense RBC can be seen as a tool that gives safety properties but is fundamentally asynchronous.

Reductions

Lemma 7. *Terminating reliable broadcast is stronger than reliable broadcast.*

Proof. Trivially, because RBC is TRBC with a weakened liveness property. □

Lemma 8. *External consensus is loosely stronger than reliable broadcast.*

Proof. The sender sends its message to all participants. Nodes wait for the sender message and then use it as input value for an external consensus instance, with V requiring a signed message from the sender. If the sender is honest, all honest nodes will receive and submit the same value to consensus. Because only the sender only signed its input, it's the only value in V that can be output by external consensus. □

Lemma 9. *Binary consensus is loosely stronger than reliable broadcast.*

Proof. First, let us see that Lemma 8 also works with consensus, because if the sender is honest all nodes submit the same message. Now, we can replace consensus with binary consensus, by having in parallel one instance of binary consensus for each bit of the sender message. □

Atomic broadcast (ABC)

Atomic broadcast is a continuous protocol that makes no assumption on inputs.Informally, it lets any node spawn RBC instances as sender, with the requirement that the order of *all* RBC outputs must be the same for all honest nodes [Lam78, CASD95]. For practical applications, protocols often use a formalism base on the State Machine Replication (SMR) paradigm [Sch90b] rather than consensus, because this abstraction is indeed closer to how a real-life application might interact with the protocol. There are many equivalent ways to formulate specifications for SMR protocols, in this thesis we selected ABC as a simple and common abstraction. This approach is also relevant for blockchain, which can also be seen as a growing common log of transactions, and therefore we view the ABC abstraction as being suitable to define blockchain protocols. Like consensus, the ABC module is parametrized by the set of values V. We also have $I_{\text{ABC}} := V$ and $O_{\text{ABC}} := V$.

Property (Agreement/Consistency)**.** $\forall k$, all honest nodes' k-th output are equal.

Let $v_{\text{out}} : \Pi \times \mathbb{N} \to E^I$ such that
$\forall p \in \Pi, \forall k \in \mathbb{N},\ \exists i \in E^{-1}(\mathfrak{m}, \mathfrak{o}, \_, p, v_{\text{out}}(p,k)),\ \#\{j \in E^{-1}(\mathfrak{m}, \mathfrak{o}, \_, p, \_) \mid j < i\} = k$
then, $\forall k \in \mathbb{N}, \forall p_1, p_2 \in H,\ v_{\text{out}}(p_1,k) = v_{\text{out}}(p_2,k)$

Property (Validity and Liveness)**.** All honest players inputs are eventually output by all honest nodes.

$$\exists R : E^{-1}(\mathfrak{m}, \mathfrak{i}, \text{ABC}, \_, \_) \leftrightarrow \mathbb{N}, \text{ such that,}$$
$$\forall q, p \in H, \forall v \in V, v_o ut(q, R(\mathfrak{m}, \mathfrak{i}, \text{ABC}, p, v)) = v$$

The Validity and Liveness properties have been merged to make the formulation simpler, *i.e.*, we require the existence of a bijection between the input of a value v and the output index at which v is delivered.

Is the SMR/ABC abstraction appropriate for blockchain? One may argue that Bitcoin is not a SMR protocol but something weaker, because it lacks *finality* [Vuk15]. Bitcoin transactions have a probability of being reverted that tend to 0 over time but never reaches it, and they are deemed *final* once they reach some predefined threshold. By comparison, protocols based on signatures and votes can label a transaction as final instantly after gathering sufficiently many votes. First, we emphasize that the outputs of atomic broadcast are only the final transactions, thus for a Nakamoto-style blockchain, the transactions whose revert probability is not under the threshold that susceptible to chain reorganisation are all internals of the protocol that are unknown to the ABC abstraction. Secondly, we can observe that lack of finality may be seen as a nonzero probability of violating safety [GPS18], *i.e.*,

$$\text{Finality}(P) := \forall T \in \mathcal{M}_P^{\mathcal{A}}, \mathbb{P}_T[\text{Agreement}(P)] = 1$$

It is easy to see that finality in that sense is not satisfied by most protocols of interest because it implies perfect security[8] and the usage of most of cryptography implies a nonzero

[8] Although for Agreement only.

probability of protocol failure. Hence non-final protocols are not weaker than those as satisfying regular ABC probabilistically, *e.g.*, in the case of a computationally bounded adversary. This is explicitly visible if we consider that signatures-based protocols can have safety violated with some (negligible) probability due to, *e.g.*, a forged signature, the same way that PoW protocols can have safety violated with some (negligible) probability due to, *e.g.*, a few PoW mined way faster than expected. For instance, Lemma 10 shows that this intuitive notion of finality does not apply to Tendermint [Buc16], a protocol designed with a per-block agreement, which goes against common belief.

Lemma 10 (Tendermint [Buc16] is not final)**.**

$$\neg\text{Finality}(\text{Tendermint})$$

Proof. Let $(\text{KeyGen}, \text{Sign}, \text{Verify})$ be the signature scheme of Tendermint and let $\mathbb{P}_{\text{KeyGen}}$ be the probability mass function of KeyGen. Let $T \in \mathcal{M}^{\mathcal{A}}_{\text{Tendermint}}$ be the following execution tree. After setup, the adversary generates $\#\Pi$ key pairs (sk_i, pk_i) using KeyGen. The adversary then impersonates all nodes by assuming that sk_i is the private key of node i, and breaks Agreement, *e.g.*, by equivocating on behalf of honest nodes. If for all honest nodes the generated (sk_i, pk_i) pair is equal to node i own key pair, then the attack succeeds. This event happens with probability $(\sum_{(sk,pk)} \mathbb{P}_{\text{KeyGen}}(sk, pk)]^2)^{\#\Pi} > 0$, therefore T is such that $\mathbb{P}_T[\text{Agreement}(\mathit{Tendermint})] < 1$. □

Reductions

Lemma 11. *Terminating reliable broadcast is loosely stronger than atomic broadcast.*

Proof. Nodes simply execute a TRBC instance as a sender in a round robin manner. They can reliably wait for the previous instance to terminate, and each honest node can periodically contribute some output to ABC. □

Lemma 12. *Strong consensus is loosely stronger than ABC.*

Proof. This is done by buffering inputs and sending them to all nodes, thus creating something similar to an "input pool" by analogy with Bitcoin's transaction pool. The outputs are obtained by repeatedly executing strong consensus. That is, the first instance of strong

consensus is started right at the beginning of the protocol, and honest nodes provide input to the i-th instance right when all the instances from 0 to $i-1$ terminated. For each strong consensus instance, the input provided is the current contents of the input pool, and the output of each consensus constitutes a batch of outputs for ABC. □

Note that using consensus does not work in this reduction because ABC requires that (honest) inputs are eventually committed, and the adversary could force one honest node to always have an empty value. This may not be an issue *e.g.* in the case of a blockchain because having merely valid outputs is sufficient.

Lemma 13. *Atomic broadcast is stronger than vector consensus.*

Proof. All nodes A-broadcast their input once, and wait for A-delivery of other values until the set of nodes that didn't deliver is a member of $\mathcal{C}$. □

We now give two reductions that are conditioned by the corruption structures $\neg 2$-cover and $\neg 3$-cover, analogous to the $1/3$ and $1/2$ corruption thresholds. We chose to still include these reductions because they motivate the justification that "SMR" is equivalent to "consensus", but also highlights the preconditions to this equivalence. Recall that k-cover $:= \exists \pi_1, \ldots, \pi_k \in \mathcal{C}, \bigcup \pi_i = \Pi$.

Lemma 14. *Assuming* $\neg 3$-cover*, atomic broadcast is stronger than consensus.*

Proof. The reduction starts exactly as in Lemma 13. Given the vector output, let S be the set of honest sender nodes with non-empty values S. We have that $S \notin \mathcal{C}$, because otherwise S together with the set of senders of empty values and the set of remaining nodes would all be in $\mathcal{C}$ and cover Π. If all honest nodes had the same input v then out of the vector output we can compute a set of senders not in $\mathcal{C}$ such that they all have the same value v'. Because at least one of them is honest, $v = v'$, which can be the output for consensus. □

Lemma 15. *Assuming* $\neg 2$-cover*, vector consensus is loosely stronger than atomic broadcast.*

Proof. This is done in the same iterated manner than from strong consensus (lemma 12), except that the output batch is obtained by taking the union of the vector values. The set of non-empty values from the vector cannot be in $\mathcal{C}$, otherwise this set together withe set of empty values would form a 2-cover, per vector validity. Therefore, there is at least one honest value in each vector output, which ensures that all honest input will eventually appear in the output. □

Equivalence graph

The given reductions can be naturally represented by a graph with specifications as vertices and reductions as edges, shown in Figure 3.1. We draw the transitive reduction of this graph based on the reductions given. For readability we do not distinguish the two types of reductions.

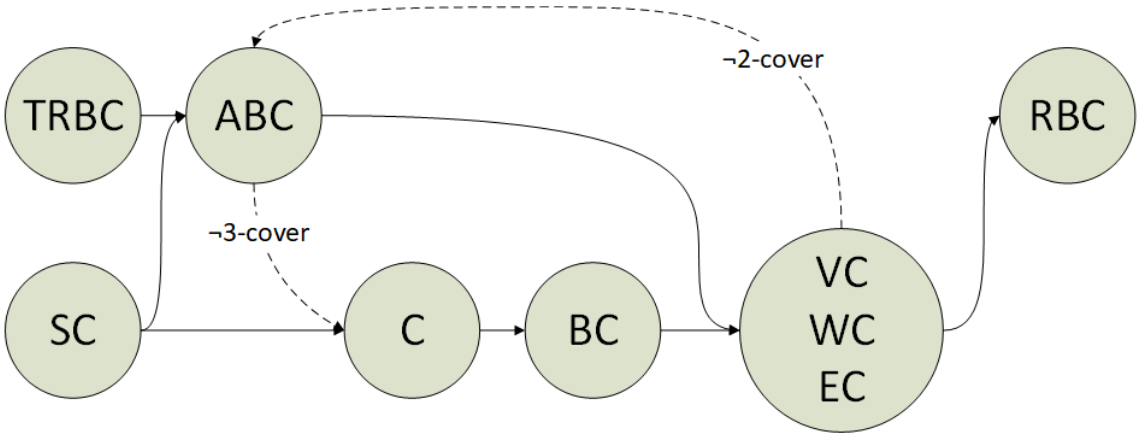

FIGURE 3.1: Problems relationships. Arrows mean "is loosely stronger than". The dashed arrows are conditional.

3.4 Analyzing performance

One of the motivations of this chapter is to compare existing blockchain protocols. But beyond model and specifications, metrics such as communication cost and latency are a key point of comparison. In this section we precisely define the evaluation of these metrics as a function of a few dimensions, namely node count, network delay (actual and bound), output size and security parameter. These definitions are solely based on the network module and ABC module described in sections 3.2 and 3.3, which result in standardized performance analysis of BFT protocols

There are two objectives in mind. First, the definitions must assume as little as possible on the underlying model, in order to be applicable to any ABC protocol. In particular this

means that definitions must stay correct even within infinite executions. Secondly, we generalize complexities to capture multiple dimensions, even if their value changes during execution. In particular this will let us capture the actual network delay and the protocol payload size as dimensions in the complexity analysis, leading to fine-grained protocol comparisons. As a result the stated definitions are slightly involved, however in simple cases they collapse to the usual evaluation of communication complexity and latency.

Our approach is structured as follows. Each metric is defined by a function that takes a point in time of an execution (*i.e.*, an event) and returns the metric value aggregated *up until that point*. For instance, for communication cost it will return the number of bits sent by honest nodes so far. Then we consider a tuple $\mathcal{D}$ that represents all the dimensions in which the metrics are expressed, such as Δ and λ. For each dimension dim, $\mathcal{D}_{dim}$ is a function that also takes a point in time and returns the dimension value at that point. For instance, for Δ it will be the maximum delay of the messages sent so far. Note that all dimensions are defined in this manner, although it is not necessary for the ones that are constant during all executions.

The general idea is to define a metric as a function that maps the dimension values to the metric value, by taking a point in execution where the dimensions evaluation matches the input. More precisely, the metrics value *for one execution* is the function that takes a tuple of dimension values D and returns the metric value at the latest point in the execution i such that at that point each dimension dim has a value $\mathcal{D}_{dim}(i)$ have a value smaller than in the tuple D_{dim}.

Definition 39 (Metrics and dimensions)**.** Let P be a protocol, $T \in \mathcal{T}[\text{Events}(P)]$ such that $\mathcal{M}_P^A(T)$ holds, and let $E \in T$. A metric is a function $M : E^I \to \mathbb{N}$. A dimension is a monotonically increasing function $D : E^I \to \mathbb{N}$.

Definition 40 (Metric evaluation)**.** Let P be a protocol, M a metric and $\mathcal{D}$ a named tuple of dimensions of length l. The metric value of one execution is $M(E) : \mathbb{N}^l \to \mathbb{N}$.

$$M(E) := D \mapsto M(\max\{i \in E^I \mid \forall D_{dim} \in D,\ \mathcal{D}_{dim}(i) \leq D_{dim}\})$$

The overall metric value for the protocol P is the worst case value over the averages of the admissible execution trees of P.

$$M(P) := D \mapsto \max_{\substack{T \in \mathcal{T}[\text{Events}(P)] \\ \mathcal{M}_P^A(T)}} \mathbb{E}_T[M(E)(D)]$$

For simplicity we have defined dimensions and metrics with domain $\mathbb{N}$, but it can easily be generalized to arbitrary ones.

Our approach may seem slightly counter-intuitive. Indeed, a natural approach could be to model a metric directly as a single value $M(E)$ associated to each execution E. But then, to evaluate the asymptotic complexity of $\max_T M(E)$ as a function of the dimensions, they must also have a single value $D(E)$ for each execution. This breaks down when considering infinite executions, for which some metrics and dimensions cannot have a finite value, *e.g.* communication cost. Furthermore, associating a single dimension value to each execution can raise difficulties, for instance for network delay in the asynchronous models.

Instead, our intuition is to see the metric for one execution as a function from the dimensions values to the metric value, which is illustrated in our definition of $M(E)$. As a result, our approach is much more appropriate to make complexity statements with dimensions that changes value during execution.

Taking the metrics average may be questionable at first glance. However because our framework appropriately distinguishes probabilistic and non-probabilistic events, we can naturally take the metrics average *while* taking the worst case value for non-probabilistic choices, which is manifested by taking the maximum over execution trees.

In our case, we are mostly interested on the performance guarantees, which are represented by taking a worst case value (since our metrics evaluate a cost to be minimized). One possible criticism of taking the worst case value is that it is not representative of real execution. Indeed the existence of arbitrary behavior means that the worst case performance is essentially the resilience to Denial-of-Service attacks, although this is only really relevant for optimistic protocols.

3.4.1 Metrics

Latency The latency metric is noted $\mathcal{L}$. In the context of cryptocurrencies, latency translates directly to the time an end user has to wait before his transaction is effective. We first define the latency of one input, which is the number of ticks that an honest process sees between the input broadcast and its delivery. Then, the execution latency is the maximum latency of all the inputs.

Definition 41 (Latency)**.**

$$\mathcal{L} := i \mapsto \max_{\substack{j \in E^{-1}(\mathfrak{p},\mathfrak{i},\_,\_) \\ j<i}} INPUT\_LATENCY(j)$$

$$\text{where } INPUT\_LATENCY(i) := \text{ Let } (\mathfrak{p},\mathfrak{i},p,v) = E_i \text{ in}$$

$$\#\{j \in E^{-1}(\mathfrak{m},\mathfrak{o},\text{clock},p,\text{tick}) \mid i < j < E^{-1}(\mathfrak{p},\mathfrak{o},p,v)\}$$

This notion of latency is specific to ABC protocols, however, a similar approach can easily be applied to define the latency of all one-shot protocols, by taking the number of clock tick between the first input and the last output (for terminating execution).

Communication cost The communication cost metric is noted $\mathcal{CC}$. It is the number of bits sent by honest players.

Definition 42 (Communication cost)**.**

$$\mathcal{CC} := i \mapsto \sum_{\substack{j \in E^{-1}(\mathfrak{m},\mathfrak{i},\text{net},\_,\_) \\ j<i \,\wedge\, E_j.p \in H_j}} length(E_j.v)$$

We will use the term confirmation time as a synonym to latency, and bit complexity as a synonym to communication complexity.

3.4.2 dimensions

Node count The first dimension is noted n and has value $\mathcal{D}_n := i \mapsto \#\Pi$, which does not change during execution. Notably, the dependence of communication cost on n is what characterises a protocol's *scalability*.

Security parameter The other dimension that does not change during execution is λ with value $\mathcal{D}_\lambda := i \mapsto \lambda$.

Total payload Noted d, this dimension quantifies the minimal amount of data that was output by an honest node.

$$\mathcal{D}_d := i \mapsto \min_{p \in H_i} \sum_{\substack{j \in E^{-1}(\mathfrak{p},\mathfrak{o},\_,\_) \\ j \leq i}} length(E_j.v)$$

As a side effect, and especially in the context of ABC protocols, the use of d allows to distinguish a "constant overhead" communication cost, *i.e.*, that occurs independently of the data to broadcast, from an "efficiency" cost which reflects the overhead that scales with the amount of data transmitted. We also remark that, depending on the algorithm, the bound on network delays Δ may also cover messages of size d. Thus, in practice, increasing d could in turn increase Δ.

Network delay Noted δ, this dimension is the actual network delay.

$$\mathcal{D}_\delta := i \mapsto \max_{\substack{j \in E^{-1}(\mathfrak{m},\mathfrak{o},\text{net},p,m) \\ j \leq i}} Delay(j)$$

where $Delay(j) := \#\{k \in E^{-1}(\mathfrak{m},\mathfrak{o},\text{clock},p,\text{tick}) \mid \text{sent}(j) \leq k \leq j\}$

Network delay bound For protocols that use a strongly or partially synchronous network, we additionally have the network bound dimension $\mathcal{D}_\Delta := i \mapsto \Delta$.

3.4.3 Evaluation

The metrics as currently presented are not intended to be used with their precise value. In general, only their *asymptotic complexity* are evaluated and discussed. This is the case first and foremost with impossibility proofs: we will give asymptotical bounds on communication cost when considered as a function of the number of nodes, and on latency when considered as a function of the message delay. To this end, we use the standard multivariate big-O notations $\Omega(\cdot)$, $\Theta(\cdot)$ and $O(\cdot)$ for asymptotical upper, exact and lower bounds.

- $f(x) = O(g(x))$ iff $\limsup_{||x|| \to \infty} \frac{|f(x)|}{g(x)} < \infty$

Notation	Meaning
Δ	Network delay, known upper bound
δ	Network delay, (actual) upper bound
n	number of nodes
b	protocol output size
$\text{poly}(\lambda)$	any polynomial in λ
$\text{poly}_v(\lambda)$	any polynomial in λ and arbitrary in v

TABLE 3.2: Summary of metrics

- $f(x) = \Theta(g(x))$ iff $f(x) = O(g(x)) \wedge f(x) = \Omega(g(x))$
- $f(x) = \Omega(g(x))$ iff $\liminf_{||x|| \to \infty} \frac{|f(x)|}{g(x)} > 0$

where x is the tuple of arguments to functions f and g, and $||\cdot||$ is the infinite norm, *i.e.*, $||x|| = \max_i(x_i)$. Specifically, the arguments considered in the big-O notations are the dimensions defined above.

Handling the security parameter It is common to have the performance metrics depend on some intricate variable which is shown to be polynomial in λ, but its precise dependencies to the other dimensions are difficult to track, or are even unknown. As a result, we choose to use $\text{poly}(\lambda)$ to denote any function polynomial in the security parameter, but we will add in subscript the additional dimensions it takes (with an arbitrary complexity). For instance, $\text{poly}_\delta(\lambda)$ is a function that is polynomial in λ and arbitrary in δ. Importantly, although the length of digital signatures and hashes are technically $\text{poly}(\lambda)$ we will treat them as constants, notably because in practice these values are evaluated once and kept the same for all further use of the primitive (in any application), and thus they are in effect independent from the protocol security parameter.

Finally, we summarize the introduced dimensions and notations in Table 3.2.

3.5 Lower Bounds

An impossibility theorem, or lower bound, is a theorem of the form $\forall P \in \mathcal{P}[A],\ \neg \mathcal{M}_P^{\mathcal{A}} \models B$, for modules A, B and adversary $\mathcal{A}$. Typically they are made by contradiction, that is, we assume the existence of such protocol P, then show the existence of an execution (that is, an attack on P) that implies a contradiction with B.

Impossibility results are transitive, therefore we state impossibility results in the strongest model possible for the weakest problem applicable.

Claim. *Given modules A, B, B' and A', if B' is loosely stronger than B and A' is loosely stronger than A, and if $\neg \mathcal{M}^{\mathcal{A}}[A] \models B$, then $\neg \mathcal{M}^{\mathcal{A}}[A'] \models B'$.*

3.5.1 On the corruption structure

The following proofs give conditions for solving problems that can be satisfied through appropriate corruption structures. Except for the celebrated Fischer Lynch Paterson impossibility (FLP) impossibility [FLP85], they all rely on the impossibility for honest nodes to distinguish multiple contradictory scenarios. To create these scenarios, the adversary needs a corruption structure that form a cover of the nodes, thus materializing the trust assumptions required for our protocols.

We recall the k-cover predicate from definition 22 :

$$k\text{-cover} := \exists \pi_1, \ldots, \pi_k \in \mathcal{C},\ \bigcup \pi_i = \Pi$$

In the following proofs the k-cover assumption implicitly defines the sets $\pi_1, \ldots, \pi_k$ such that $\bigcup \pi_i = \Pi$ and $\forall i, j,\ \pi_i \cap \pi_j = \emptyset$, which is an implication of the monotonic corruption structure (see Section 3.2.1).

Asynchronous deterministic consensus impossibility The celebrated FLP impossibilty [FLP85] shows that no protocol can deterministically terminate in asynchronous networks. Because network asynchrony and determinism are the only options in our models that may be weakened, this theorem establishes that there are no one-size-fits-all solution for consensus.

Theorem 2 (FLP)**.** *For any protocol P in the model with crash faults (and therefore with byzantine faults too), deterministic algorithm, asynchronous network, static adversary and no signatures, if* $\mathcal{C} \neq \emptyset$ *then P cannot be a Binary Consensus protocol.*

$$\forall \mathcal{C} \neq \emptyset, \forall P \in \mathcal{P}[\mathit{async\_net}],\ \neg \mathcal{M}_P^{(\mathit{unbounded},\mathcal{C},\mathit{static})} \models BC$$

We do not give the full proof here, as it does not benefit from being reformulated in our model, and many high quality proofs already exist [Abr19, Rob08].

Split-brain scenarios The two following proofs are variations on the idea that, by splitting all nodes into three sets π_1, π_2, π_3, the malicious nodes, *e.g.* π_1, can act with respect to each set *as if* the other one was corrupt. Without any mean to cross-check their information, an honest set of nodes, *e.g.* π_2, cannot know whether π_1 or π_3 is actually corrupt and therefore must pick a fixed set to agree with, breaking consistency in case where it is corrupt.

The first theorem is essentially an adaptation from the original consensus impossibility [PSL80, FLM85] to probabilistic, computational broadcast. Interestingly, the proof for this specific case was only made rigorous in 2003 by Fitzi [Fit03].

Theorem 3 (Unauthenticated RBC [Fit03])**.** *For any protocol P in the model with Byzantine faults, probabilistic algorithm, synchronous network, static adversary (and no signatures), if* 3-cover *holds, then P cannot be a RBC protocol.*

$$\forall P \in \mathcal{P}[\text{rand}, \text{sync\_net}], \forall \mathcal{C} \subseteq \mathcal{P}(\Pi),\ 3\text{-cover} \Rightarrow \neg \mathcal{M}_P^{(\text{computational},\mathcal{C},\text{static})} \models^{\lambda} RBC$$

Proof. First, we consider a protocol P' in the same model than P, in which each node emulates some execution of P. In particular P' acts as an intermediary between the simulated execution of P and the network, and in the following the view of P' nodes implicitly refers to the simulated P view. The nodes Π' in P' are partitioned into six sets π'_i, $i \in \mathbb{Z}/6\mathbb{Z}$, such that each node in π'_i executes a session of P among $\pi'_{i-1} \cup \pi'_i \cup \pi'_{i+1}$, and additionally there is an bijection f between π'_i and π'_{i+3} such that p and $f(p)$ are executing the same instance[9] of P. In effect, this means that the π'_i are organized in an hexagon where each set

[9] By instance, we mean same parameters and identities, but not necessarily the same inputs.

only communicates to its neighbor, each pair of opposing sets are copies of each other, and all nodes are behaving as if they were executing P within a smaller network partitioned into three sets.

Let R' be an execution tree of P' with no corruption event, such that the two sender nodes in R' belong to π'_s and π'_{s+3}, and they have input v_0 and v_1, respectively, with $v_0 \neq v_1$.

We now show that for every adjacent pair π'_i, π'_{i+1}, there is an execution tree $R_{i,i+1}$ of P where π_j is corrupt and the joint view of π_{j+1} and π_{j-1} is the same as the simulated execution of P from π'_i and π'_{i+1} in R'. Indeed, the adversary can map π'_{i+1} to π_{j+1} and π'_i to π_{j-1} ; make π_j act with respect to π_{j+1} and π_{j-1} as π'_{i+2} with π'_{i+1} and π'_{i-1} with π'_i, respectively. In simpler words, the adversary in P can make π_j simulate four adjacent sets of nodes in R', such that the two honest sets in P interact with themselves and the two different copies of π_j in R'.

Because of consistency in $R_{i,i+1}$, π'_i and π'_{i+1} must output with overwhelming probability the same value. Since this holds for all $i \in \mathbb{Z}/6\mathbb{Z}$, all nodes in R' also output the same value v. Additionally, because of liveness and validity in $R_{s,s+1}$, π'_s must output $v = v_0$, and because of validity in $R_{s+3,s+4}$, π'_{s+1} must output $v = v_1$, yielding a contradiction with overwhelming probability.

□

In this proof, the synchronous network allows the two set of nodes (*i.e.*, the two π_{j+1} and π_{j-1} when they are not senders) to communicate, but they cannot cross-check their views because without signatures there is no way to verify whether π_i is really acting according to what it received from the other set.

Alternatively, a very similar proof can be made if the network doesn't give the opportunity to the two $\pi_{j\pm1}$ to communicate at all. As a result, the same theorem applies in the case of a partially synchronous network, even assuming signatures.

Theorem 4 (Authenticated RBC)**.** *For any protocol P in the model with Byzantine faults, probabilistic algorithm, partially synchronous network, static adversary, signatures, if* 3-cover *holds, then P cannot be a RBC protocol.*

$$\forall P \in \mathcal{P}[\text{rand}, \text{weak\_net}, \text{signatures}], \forall \mathcal{C} \subseteq \mathcal{P}(\Pi),\ 3\text{-cover} \Rightarrow \neg \mathcal{M}_P^{(\text{computational},\mathcal{C},\text{static})} \models^{\lambda} \mathit{RBC}$$

In this case the adversary only has to simulate the absence of communication from one set of nodes, instead of the six-partitioned simulation. As a result, although the proof idea is very similar to theorem 3, the proof itself is significantly simpler.

Proof. Without loss of generality, assume that the sender node is in π_1. Let v_2 and v_3 be two different sender inputs. Let j and k in $\{2,3\}$ and $j \neq k$.

Let R_j be an execution tree where π_j is corrupt and do not send any messages, and the sender has input v_j. Liveness and Validity imply that, with overwhelming probability, all nodes in π_k must output v_j within a finite time T_j. Let R'_j be an execution tree where all nodes are honest, all communications from/to π_k are delayed by Δ, and the sender had input v_j. Let R be an execution tree where π_1 is corrupt, and acts exactly as in R'_j towards π_k and as in R_k towards π_j.

Because π_1 acts the same towards π_k in R and in R_j, both executions trees are indistinguishable by π_k up until time Δ when π_k starts receiving messages from π_j only in R. However, if $\Delta > \max(T_j, T_k)$, then in R π_k and π_j have already terminated by that time, since they did so in R_j and R_k, respectively. In particular, their output value are v_j and v_k respectively, and therefore consistency is violated with overwhelming probability.

□

Because RBC is the weakest protocol we presented, these results also apply to all of them. Indeed these two theorems together make up the baseline requirements for agreement protocols, such as one third of nodes being honest.

Byzantine distinguishers The previous theorems lets open the possibility to tolerate an arbitrary number of faults in synchronous networks. The next one closes this possibility for Consensus. Roughly, the reason is that if an honest node can get their input accepted, then Byzantine nodes can act indistinguishably from honest nodes and get their "input" accepted. The proof by Neiger [Nei94] is for deterministic algorithm ; we extend it to the probabilistic case.

Theorem 5 (synchronous Consensus)**.** *Let V be a finite set, for any protocol P in the model with Byzantine faults, probabilistic algorithm, strongly synchronous network, static adversary, signatures, if $\#V$ -*cover *holds, then P cannot be a strong consensus protocol*

with input set parameter V.

$$\forall V, \forall P \in \mathcal{P}[\text{rand}, \text{weak\_net}, \text{signatures}], \forall \mathcal{C} \subseteq \mathcal{P}(\Pi),$$
$$\#V\text{-cover} \Rightarrow \neg \mathcal{M}_P^{(\text{computational}, \mathcal{C}, \text{static})} \models^{\lambda} SC(V)$$

Proof. Let i and j in $\{1,2,3\}$ and $i \neq j$. First, let A be an execution tree where no nodes are corrupt, all nodes in π_j have the same input v_j, and $v_i \neq v_j$. By liveness and validity, honest nodes in π_j will all output a value v_j, with overwhelming probability.

Then, let execution tree B be exactly the same as A, with the only exception that the partition π_j is corrupted. Because corrupting a node only *removes* an admissibility condition, the π_j nodes in B act exactly in the same way, and all other events are the same, thus B is also an admissible execution tree. For the same reason, the output in B is v_j with overwhelming probability. Given that v_j was the input only of π_j in A, and that all nodes in π_j are corrupt in B, v_j is not the input of any honest node. Therefore there is an admissible execution tree (namely, B) where strong validity does not hold with overwhelming probability. □

Note that this proof in fact applies to Binary Consensus. Indeed, if $\#V = 2$ then Strong consensus is equivalent to consensus and binary consensus.

3.5.2 On performance

This subsection gives proofs on the minimum communication complexity and latency to solve problems, for varying synchrony and randomization assumptions. These theorems are weaker however, because they are not transitive under loose reduction. Indeed, loose reductions are allowed to use the network and therefore they do not preserve communication and latency metrics.

As a further consequence, those results are much more prone to be invalidated by assuming the existence of some cryptographic primitive. For instance, a common way to improve communication complexity is by using threshold signature that enable to compress the votes of n nodes into a constant sized string.

Theorem 6 (Quadratic deterministic RBC communication cost)**.** *For any protocol P in the model with a deterministic algorithm, synchronous network, static adversary and signatures, if P is an RBC protocol, then the communication cost of P is $\Omega(n \max_{C \in \mathcal{C}} \#C)$.*

$$\forall P \in \mathcal{P}[\mathit{sync\_net}, \mathit{signatures}], \forall \mathcal{C} \in \mathcal{P}(\Pi),$$
$$\mathcal{M}_P^{(\text{computational},\mathcal{C},\text{static})} \models \mathit{RBC} \Rightarrow \mathcal{CC}(P) = \Omega(n \max_{C \in \mathcal{C}} \#C)$$

The proof idea is from [DR82] and [HH93, Section 6], which bounds the minimum number of *signatures* exchanged during the protocol.

Proof. First, define the communication graph $G(E)$ of an execution E to be the undirected graph where nodes are vertices and there is an edge between p_1 and p_2 iff either p_1 or p_2 exchange at least one signature from the other in E (possibly indirectly). Additionally, for graphs G, G' that share the same vertices, $G \cup G'$ is the graph where the set of edges is the union of the edges of G and G'.

Let's assume the following: There are two admissible executions, A and B, such that no nodes are corrupt, the sender is the same in both executions, its input value is v_a and v_B in A and B respectively, with $v_a \neq v_b$. Let $U := G(A) \cup G(B)$, and furthermore assume that there is $C \in \mathcal{C}$ such that, removing the all the nodes in C from U makes it disconnected.

We now show that this assumption leads to a contradiction, in a similar manner to Theorem 4 but only in the deterministic case: let R be the execution where the set of corrupt nodes C makes U disconnected, and U_A and U_B the two remaining disconnected subgraphs of U. The adversary makes nodes in C behave towards nodes in U_A and U_B exactly as in A and B, respectively. They are able to do so because no signatures are exchanged between nodes in A and B, hence all other messages are unverifiable. Because R is indistinguishable from A and B in the view of nodes in A and B, respectively, they must each output v_a and v_b respectively in R. Therefore consistency is violated.

The negation of our assumption holds: For each such A, B pair, for any $C \in \mathcal{C}$, removing C from $U = G(A) \cup G(B)$ does not make it disconnected. Therefore U has a connectivity of at least $\max_{C \in \mathcal{C}} \#C$, which implies at least $\lceil (n \max_{C \in \mathcal{C}} \#C)/2 \rceil$ edges. So $G(A)$ and $G(B)$ have $\Omega(\max_{C \in \mathcal{C}} \#C)$ edges, and so does the number of sent signatures and the number of sent bits. □

The next theorem is a result by Garay, Katz, Koo and Ostrovsky. Although they technically state their proof in the model with a rushing, dynamic adversary, we observe that the same proof holds with a static adversary, because all corruptions events take place before the protocol starts.

Theorem 7 (Linear Latency [GKKO07])**.**

$$\forall P \in \mathcal{P}[\text{rand}, \text{sync\_net}, \text{signatures}], \forall \mathcal{C} \subseteq \mathcal{P}(\Pi),$$
$$\exists \pi_1, \ldots, \pi_k \subset \Pi, \bigcup \pi_i = \Pi \ \wedge \ \forall i, \Pi \setminus (\pi_i \cup \pi_{i+1}) \in \mathcal{C} \ \wedge$$
$$\mathcal{M}_P^{(\text{computational}, \mathcal{C}, \text{static})} \models^{\lambda} RBC \Rightarrow \mathcal{L}(P) = \Omega(\delta k)$$

Proof. Let $\pi_1, \ldots, \pi_k \subset \Pi$ such that $\bigcup \pi_i = \Pi$, $\forall i, \Pi \setminus (\pi_i \cup \pi_{i+1}) \in \mathcal{C}$, and without loss of generality $\forall i \neq j,\ \pi_i \cap \pi_j = \emptyset$. For convenience, we also define $\pi_0 := \pi_{k+1} := \emptyset$. Let $v_0 \neq v_1$ and S_i^b be the execution trees such that: The sender is in π_1 with input v_b, all communications are delayed by Δ, the nodes in $\Pi \setminus (\pi_i \cup \pi_{i+1})$ are corrupt, and for all $j \in [\![1, k]\!] \setminus \{i, i+1\}$, each corrupt node in π_j execute P except that it ignores and doesn't send any message to nodes in $\pi_{j-1} \cup \pi_j \cup \pi_{j+1}$.

For any $i \geq 2$, in S_{i-1}^b, nodes from π_{i-1} and π_i must have the same output, nodes in π_i cannot distinguish S_{i-1}^b from S_i^b. Given that in S_1^b the sender is honest, by induction on i, in S_i^b all nodes in π_i must output v_b with overwhelming probability. However we also have that for nodes in π_i, the execution trees S_i^0 and S_i^1 are indistinguishable until $\Delta(i-1)$ time has elapsed, because any information on b must be sent through $\pi_1, \pi_2, \ldots, \pi_{i-1}$, taking Δ ticks each time. If with overwhelming probability π_i outputs b before $\Delta(i-1)$ in S_i^b, then it must do the same in $S_i^{b+1 \mod 2}$, thus violating Validity. The conclusion is reached by taking $i = k$. □

In particular if $\mathcal{C} = \mathcal{P}(\Pi)$ then the latency is $\Omega(\delta n)$

An updated look on the problem space The results above are mapped into the representation of the problems relationships in Figure 3.2, by drawing the separations we have just proven. This figure illustrates that, although choosing definition of the BFT variant of consensus is broadly seen as a technical detail, the choice is far from inconsequential. For instance, if the objective is to implement ABC, then consensus is needlessly strong, despite being the most popular choice. Also notably, this figure shows that being thoughtful of problem reductions led us to cover most cases using only four theorems.

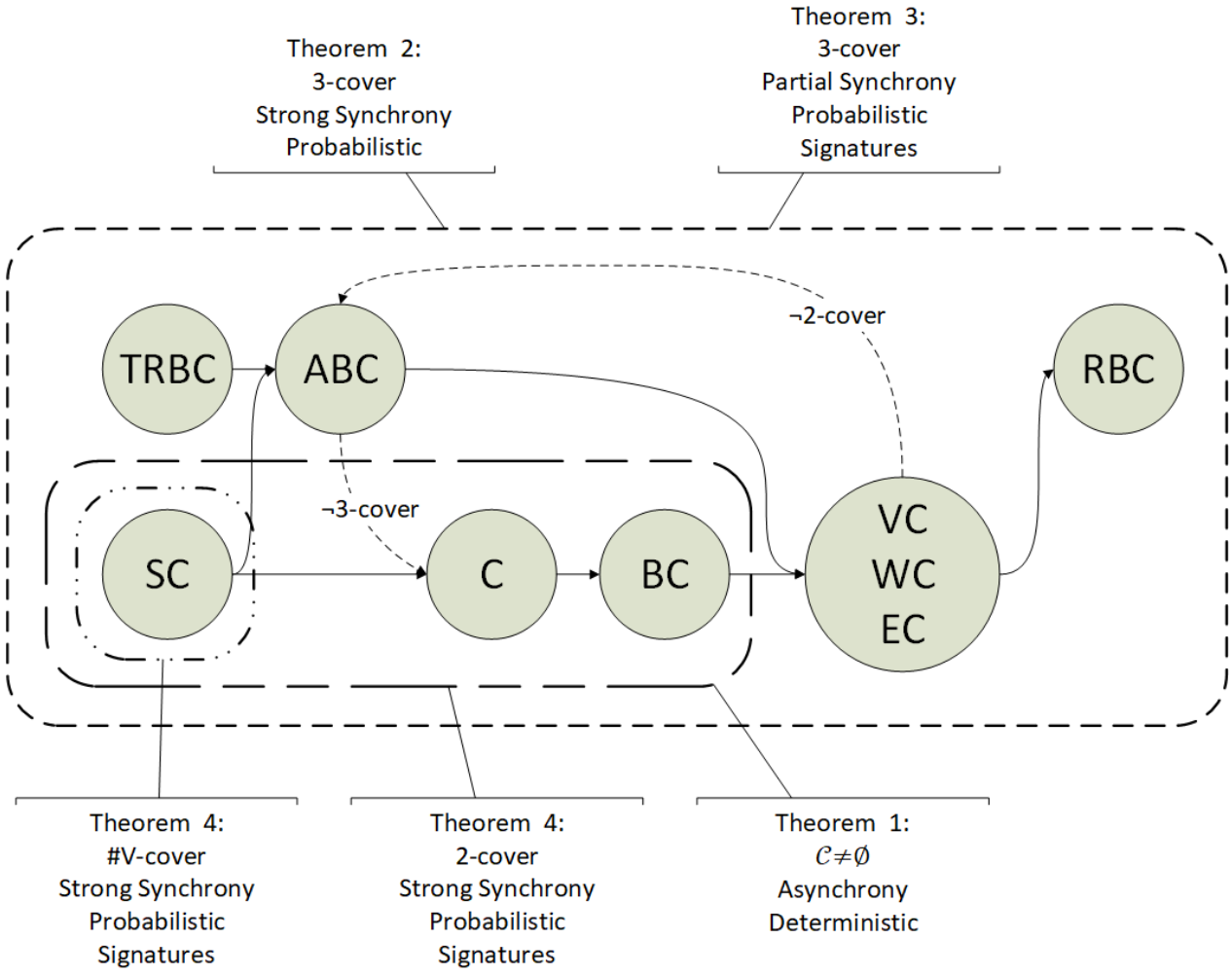

FIGURE 3.2: Problems relationships, with impossibility results. Each dotted box regroups all the problems that cannot be solved under some models, and they are annotated with the precise impossibility theorem that apply as well as the models that are excluded.

3.6 Analyzing protocols

A preliminary version of this section has been published online [DHLM19].

The main objective of this section is to show how different blockchains can be understood and compared using our framework. To do so, we selected a few prominent blockchain protocols which we interpret them within our framework. We have chosen protocols to be representative of the different techniques and working principles of blockchains, and we only considered those that are sufficiently formalised to enable a fair assessment of their model and properties. Phantom [SZ18] to have a second PoW-based protocol to compare with the Nakamoto consensus [GKL20a], Ouroboros Praos [BGKR18] is a non PoW-based synchronous protocol, Algorand [Mic17] is somewhat similar to Ouroboros but in a partially synchronous setting, Tendermint [Kwo14] operates similarly to classical SMR protocols [CL02], HoneyBadgerBFT [MXC+16] is asynchronous and StakeCube [DAL19] is our contribution.

In concrete terms, we state the model under which they operate, and we give complexities for latency and communication cost. In turn, those results are also useful by themselves to make sensible comparisons between the different protocols. Note that even if we provide a high-level reasoning to motivate the metrics results, we entirely rely on the proofs provided in the reference papers, and we only complete them when necessary. In particular, we do not attempt to make the results tighter.

All the protocols presented satisfy Atomic Broadcast, assume Byzantine faults, and are probabilistically secure against a computational adversary.Let us recall however that the computational security implies that the protocol cannot run for an unbounded length of time.

3.6.1 Using the Bitcoin backbone protocol

For the Nakamoto consensus[10] and Ouroboros Praos, we will rely on the backbone protocol formalism from Garay *et al.* [GKL15, GKL20a, BGKR18]. For both protocols the

[10]We use the term "Nakamoto consensus" or "Nakamoto" for short to refer to Bitcoin's underlying *ABC* protocol.

authors adopt the notion of protocol execution from the Universal Composability framework [Can01], thus we fall in the category of computationally bounded adversaries. However, they don't use the UC notion of security (UC-realization). Instead, they state the specifications through some ad-hoc properties on the view of all participants, *i.e.* the execution. As a result they follow a similar definitional approach to us, with the difference that the model of protocol execution comes from the UC framework. Both protocols satisfy the specification of a *Transaction Ledger* as detailed in Section 2.5, which is decomposed into three parametrized properties (Common Prefix, Chain Growth and Chain Quality) that we reuse to give the asymptotical complexities.

For Nakamoto [GKL20a] Garay *et al.* present two analyses, one with a synchronous network and one extended to a partially synchronous network. For Ouroboros Praos [BGKR18] Bernardo *et al.* use a partially synchronous network similar to the one from the Nakamoto analysis. In all three cases, the network only allows to send a message to all nodes, *i.e.* a multicast interface. They describe a network functionality (analogous of our modules) that offers a global round clock to all nodes. Sent messages are delivered either in the same round in the case of strong synchrony and after up to Δ rounds in the case of partial synchrony, meaning that such rounds in the partially synchronous case do not have a prescribed duration and act more as "real time step" or "time slot".

However, we believe there is a slight issue in the formulation used for the partially synchronous networks. First of all, let us remark that both protocols are stated to be secure assuming that less than *half* of the stake/hashpower is owned by malicious nodes, *i.e.*, $\neg 2$-cover holds. This comes as surprising, since we have shown that Binary Consensus (and by extension, Atomic Broadcast) cannot be solved in partially synchronous networks when the adversary can form a partition with three set of corrupt nodes. This is clearly the case here, for instance by corrupting sets of nodes owning a third of the stake/hashpower.

On the other hand, we have seen in the previous section that in the case where we assume the negation of 2-cover, a strongly synchronous network is required to solve consensus. Furthermore, recall that the only difference between partial and strong synchrony is that the Δ bound is made available to nodes in the strongly synchronous case.

Indeed, with careful examination we observe that both Nakamoto and Ouroboros Praos actually *needs* the value Δ to execute, and as a result are better stated to be strongly synchronous protocols. In essence, for both cases, even if the protocols can be proven secure assuming they are correctly parametrized, to be able to output transactions with a known

value for the probability of failure, the nodes must have a bound on Δ. To make the claim more concrete, we trace the usage of Δ for both protocols.

For Ouroboros Praos, the protocol takes a parameter f that tunes the probability for a node to be eligible to multicast the current block. To know whether a given node is eligible to multicast a block, a procedure that takes f as a parameter is executed *within the protocol*. Then to prove the security of the protocol, the authors require that f satisfies an inequality [BGKR18, Theorem 9 equation 12], which encodes the "Majority of Honest Stake" assumption but also depends on Δ. In particular, if f and all the other protocol parameters are known to the nodes, then assuming that the inequality holds means that nodes can solve it to obtain (an upper bound on) Δ.

For Nakamoto the value of Δ is required in two places. First, with a reasoning similar to Ouroboros, we have an inequality [GKL20a, Honest Majority Assumption (Bounded Delay)] that is stated depending on Δ and another parameter f. In turn, f is explicitly used in the protocol, to set the target difficulty for Proof-of-Work mining, and this means again that Δ must be known. But in a more straightforward manner, they prove the Persistence property of the Transaction Ledger with parameter $k = O(\Delta)$. Given that the knowledge of k is necessary to output *any* transaction (otherwise it is not possible to know which ones are confirmed), this is another reason mandating a known Δ.

Formally, these remark can be expressed with the following lemma.

Lemma 16 (Nakamoto and Praos are strongly synchronous)**.**

$$\exists P \in \mathcal{P}[M_{\text{Praos}}],\ \mathcal{M}_P^{\mathcal{A}_{\text{Praos}}} \models \text{strong\_net}$$

$$\exists P' \in \mathcal{P}[M_{\text{Nakamoto}}],\ \mathcal{M}_{P'}^{\mathcal{A}_{\text{Nakamoto}}} \models \text{strong\_net}$$

Where $(M_{\text{Nakamoto}}, \mathcal{A}_{\text{Nakamoto}})$ *and* $(M_{\text{Praos}}, \mathcal{A}_{\text{Praos}})$ *are the models of Nakamoto consensus and Ouroboros Praos, respectively.*

Proof Sketch. We have that weak_net $\in M_{\text{Nakamoto}}$ and weak_net $\in M_{\text{Praos}}$. Let the sets PP_{Nakamoto} and PP_{Praos} be the set of public parameters of both protocols, *i.e.*, $\text{params}(PP_{\text{Nakamoto}}) \in M_{\text{Nakamoto}}$ and $\text{params}(PP_{\text{Praos}}) \in M_{\text{Praos}}$. We have shown above that, knowing an element from PP_{Nakamoto} or PP_{Praos}, it is possible to compute a bound on the delay Δ of the weak_net module. Recall that strong_net requires a setup event that outputs the module's Δ to all nodes. Both P and P' work in the same manner; after receiving the protocol's

parameters from the params module, they compute Δ and output it to all nodes. Finally, inputs and outputs to the protocol are forwarded to the weak_net module, to implement the sending and receiving of the strongly synchronous network. □

As an additional remark, the original Ouroboros [KRDO16] protocol is clearly stated to be strongly synchronous. It is possible that the change of network terminology between Ouroboros and Ouroboros Praos is due to the fact that the network in Ouroboros is expressed in a round-based formalism, and not in Ouroboros Praos where any message can independently take up to Δ time to be diffused. Similarly, the analysis of Nakamoto presented as partially synchronous may still be appealing over the first, strongly synchronous one, because in effect rounds are translated into "real time steps".

3.6.2 Nakamoto consensus

Model We have already shown that the model from Garay *et al.* [GKL20a] is strongly synchronous, with a computational adversary. Their model is completed by the Random Oracle ideal functionality, which also integrate the Proof-of-Work mining primitive. That is, the Random Oracle is given the ability to answer "mining" queries from node, with a limit of q queries per node per network round (or 1 in the "partially" synchronous analysis). Then an additional interface is added to be able to verify the result of a query without having to make the same query again and be limited by q.

The equivalence with our PoW module is as follows: the RO input plays the role of the input value m, and the query bound q is Δh_p^{-1}. In both cases, the verification interface checks whether a value is indeed the output of the RO to some node. The difficulty parameter d as well as the probability of mining a valid PoW is encoded within the Nakamoto protocol itself, instead of being part of the model in our framework with the PoW module. Contrarily to the model from Garay *et al.*, our PoW module does not have an interface to compute hashes, therefore we also have to add the standard Random Oracle module to the model.

Regarding the corruption structure, they state an "Honest Majority Assumption" such that the proportion of hashpower (expressed in RO queries per round) available to the malicious nodes is lower than $\frac{1-d(\lambda,\Delta)}{2-d(\lambda,\Delta)}$ with d a function bounded between 0 and 1. This "Honest Majority Assumption" is easily modeled as a weighted corruption structure with

each node being weighted by its hashpower. As expected, d is an increasing function, meaning that the amount of tolerated malicious hashpower gets further from the optimal value $(1/2)$ down to 0 as the the security parameter is increased.

Corruption takes effect immediately, therefore the adversary is dynamic ($D = (E, i, p) \mapsto$ *true*). This does not come as a surprise since honest miners multicast their PoW string as soon as they find it, after which there is no advantage in corrupting that node anymore.

Note that Bitcoin Backbone analysis does not require digital signatures to prove the security of the transaction ledger, and indeed we do not include them in our model. To clarify, although the actual Bitcoin does use them, they are part of the application-layer transactions, *i.e.* the messages sent through ABC.

Our inability to express dynamic participation in the case of Nakamoto consensus means that we do not consider changes in the computing power of the participants, nor the addition of new miners. The original analysis from Garay *et al.* does not capture this aspect either, but it has been addressed in subsequent works [GKL20b, CEM+20].

Claim.

$$\begin{aligned}
&\text{BITCOIN} \in \text{P}[\text{params}, \text{RO}, \text{async_net}, \text{PoW}],\\
&\mathcal{M}_{\text{BITCOIN}}^{(\text{computational}, \mathcal{C}_{(W,1/2-\varepsilon(\lambda,\Delta))}, \text{dynamic})} \models^{\lambda} \mathit{ABC}\\
&\mathit{with}\ W(p) := h_p(\sum_{q \in \Pi} h_q)^{-1}\ \mathit{and}\ \varepsilon : \mathbb{N}^2 \to [0, \frac{1}{2}]
\end{aligned}$$

Metrics In the Bitcoin Backbone analysis, the authors articulate their proofs on the assumption of a *typical execution*, that roughly states that parties produce blocks at a rate close enough to their expected value (*i.e.*, hashpower). Then they show that any execution of k rounds is typical with probability $1 - e^{-\Theta_\varepsilon(k)}$, where ε is a variable that quantifies how close the block production rate is to its expected value. In turn, the proofs will rely on ε being appropriately bounded, an assumption that is integrated in their version "honest majority assumption". In particular this assumption gives a bound on ε that depends on Δ, hence, an execution longer than $\text{poly}_\Delta(\lambda)$ rounds is typical with overwhelming probability.

CP, CQ and CG are proven assuming a typical execution, with $k_p = k_q \geq 2kf$ and $\tau = (1-\varepsilon)f$ for some variable f, hence we have u-Liveness with $u \geq \frac{4k}{1-\varepsilon} = \text{poly}_\Delta(\lambda)$.

The analysis regarding communication complexity is simpler: All b bits from the output are blocks that have been multicast once, hence the communication cost is at least bn. Furthermore, this cost is only increased if the adversary forces orphaned blocks. Since all messages on the network are blocks with a valid PoW, it is clear that the overall number of blocks received by honest nodes is a constant factor of the number of blocks that will end up in the blockchain. Hence the communication cost of such protocols is $O(bn)$.

Claim. $\mathcal{L}(\text{BITCOIN}) = \text{poly}_\Delta(\lambda)$ *and* $\mathcal{CC}(\text{BITCOIN}) = \Theta(bn)$.

3.6.3 Ouroboros Praos

Model We have already shown that the model from Bernardo *et al.* [BGKR18, BGK+18] is strongly synchronous, with Byzantine faults and a computational adversary. Like with Bitcoin, we now complete the model.

The ideal functionalities additionally included are the Random Oracle and digital signatures, for which we already have the corresponding modules defined. However to deal with the dynamic adversary, they must prevent the adversary from reusing the keys of corrupt nodes to simulate alternative executions from the protocol past phases. To do so they use *Forward Secure* signatures, which allows the signer to update its key after each signature, deleting the previous one. This additional functionality however is not captured by our signature module.

Node corruptions take effect immediately and the adversary is fully dynamic. This is possible due to the usage of VRFs, on a principle similar to how Bitcoin miner can be corrupted just after sending a block without issue. The Ouroboros nodes uses the VRF to learn locally whether they are randomly selected to multicast a block, and once the block is sent they don't have a privileged role anymore. The verifiability of the VRF output ensures that nodes cannot cheat their eligibility.

Regarding the corruption structure, the authors assume that the proportion of stake owned by honest nodes is higher than $\frac{1}{2}d(\lambda,\Delta)$, where d is an (increasing) function lower bounded by 1. This assumption is modeled by a weighted corruption structure using stake as weights.

Claim.

$$\begin{aligned}&\text{PRAOS} \in \mathcal{P}[\text{params}, \text{RO}, \text{sync\_net}, \text{VRF}, \text{forward\_signatures}],\\&\mathcal{M}_{\text{PRAOS}}^{(\text{computational}, \mathcal{C}_{(W,1/2-\varepsilon(\lambda,\Delta))}, \text{dynamic})} \models^{\lambda} ABC\\&\textit{with } W(p) := S_p(\sum_{q\in\Pi} S_q)^{-1} \textit{ and } \varepsilon : \mathbb{N}^2 \to [0, \frac{1}{2}]\end{aligned}$$

where S_p is node p's stake, as defined by the protocol.

Metrics In Ouroboros Praos, for every time slot, each node can be independently elected to multicast a new block, with a probability that only change depending on the amount of stake owned. The generation of the randomness to seed the election of block leaders is implemented by having each leader include a VRF output in their block, and periodically concatenating the VRF output to form a random seed for the future VRF evaluation.

Given this method of block production, Common Prefix, Chain Growth and Chain Quality can be proven using only combinatorial arguments on the distribution of honest and malicious leaders in the block tree. This is done through the analysis of "characteristics strings", which are an encoding of the schedule of honest and malicious leaders. The authors show that a string of length of k time slots is "forkable" with probability $\text{negl}(k)$, for an appropriate notion of "forkable" that is latter used to prove CP, CQ and CG. In particular, the distribution of characteristics strings only depends on the (stakewise) proportion of malicious leaders, and therefore k is only a function of the security parameter, *i.e.* $k = \text{poly}(\lambda)$ and $u = \Theta(k)$

Claim. $\mathcal{L}(\text{PRAOS}) = \Theta(\Delta\text{poly}(\lambda))$ *and* $\mathcal{CC}(\text{PRAOS}) = \Theta(bn)$.

3.6.4 Tendermint

Tendermint had a first complete description in [Kwo14, Buc16], than was subsequently analysed by Amoussou-Guenou *et al.* [APPT19, APPT18], which we use as a basis.

Model Tendermint follows more classical approaches to State Machine Replication [CL02] and as such fits very nicely into one of our models.

They assume a partially synchronous network with the GST formalism. All nodes sign their messages with a digital signature algorithm. Tendermint uses a hash function that we modelize with the Random Oracle.

The corruption threshold is $\lfloor n/3 \rfloor$ nodes, which is simply represented as a corruption structure with equal weights for all nodes. In practice however the protocol members are intended to be chosen through an external mechanism, and the selection may assign weights to nodes, *e.g.* in a Proof-of-Stake fashion.

The adversarial adaptivity is not explicitly specified, however it is straightforward to see that it can be dynamic: the leader is not expected to be honest, and the schedule of leaders may be well known when the adversary choose corruptions. However, in a model with transient failures, this argument does not hold anymore.

Claim.

$$\begin{aligned}&\text{TENDERMINT} \in \mathcal{P}[\text{params}, \text{RO}, \text{weak\_net}, \text{signatures}],\\ &\mathcal{M}_{\text{TENDERMINT}}^{(\text{computational}, \mathcal{C}_{(W,1/3)}, \text{dynamic}} \models^{\lambda} ABC\\ &\textit{with } W(p) := \#\Pi^{-1}\end{aligned}$$

Metrics Its normal case operation is reminiscent of PBFT [CL02], except that leaders are always changed after each broadcast, successful or not. For each block, there is a leader that will execute RBC implemented through two all-to-all voting rounds. To optimize bandwidth, only the hash of the block is included in the voting messages. This requires the b bytes of the block content to be sent to all n nodes, and the additional cost of n^2 bytes for the Reliable Broadcast, taking three of communication steps. That is, a single leader attempting to append a block takes $\Theta(bn+n^2)$ bit complexity and $\Theta(1)$ latency. Malicious leaders can ensure that their attempt does not succeed, and because the leader order is arbitrary they may be all malicious first, thus increasing latency by a factor $\max_{C \in \mathcal{C}} \#C = \Theta(n)$. This does not impact the overall communication cost however, because all honest leaders will also append a block and only a constant fraction of nodes are malicious.

Tendermint is optimistic in the sense that in failure free executions, *i.e.* if $C = \emptyset$, the Latency becomes $\Theta(\delta)$.

Claim. $\mathcal{L}(\text{TENDERMINT}) = \Theta(\delta n)$ *and* $\mathcal{CC}(\text{TENDERMINT}) = \Theta(bn+n^2)$.

Interestingly, the use of hash functions in Tendermint is only needed to optimize communication if the value to be agreed on is large enough. Without hash functions, Tendermint is deterministic and benefits from Information Theoretic security (obviously assuming ideal signatures), but because all votes must then include the full values its communication cost raises to $\Theta(bn^2)$.

3.6.5 HoneyBadgerBFT

Model In HoneyBadger BFT [MXC+16] the network is asynchronous, up to $1/3$ of the nodes may be corrupt, and the adversary is static. The protocol makes use of a hash function and digital signatures. The authors explicitly assume a "Purely asynchronous network", "Static Byzantine faults" and "Trusted setup", and they aim to solve Atomic Broadcast. The setup only serves to implement a PKI for the digital signatures and the common coin, both of which we directly modeled with modules.

The protocol is based on an asynchronous implementation of Binary Consensus, which by the FLP impossibility must be probabilistic. The Binary Consensus used is a protocol from Mostefaoui *et al.* [MHR14] which relies on a *Common coin* protocol, implemented using a threshold signature scheme and a trusted setup. To complete HoneyBadger BFT's model, we give the module for the common coin:

Definition 43 (Common Coin specification module)**.** The "common_coin" module is a one shot protocol with $I_{\text{common\_coin}} := \emptyset$ and $O_{\text{common\_coin}} := \{0,1\}^{\lambda}$. It has the same Liveness and Consistency properties from consensus, as well as Unpredictability. Recall that per the Agreement property, all honest nodes outputs the same value v_{out}.

Property (Unpredictability)**.** The honest nodes output follows a uniform random law on $O_{\text{common\_coin}}$.

$$\forall v \in O_{\text{common\_coin}},\ \mathbb{P}_T[v_{\text{out}} = v] = 2^{-\lambda}$$

Claim.

$$\begin{aligned}
&\text{HBBFT} \in \mathcal{P}[\text{params}, \text{RO}, \text{async_net}, \text{signatures}, \text{common_coin}],\\
&\mathcal{M}_{\text{HBBFT}}^{(\text{computational}, \mathcal{C}_{(W,1/3)}, \text{static})} \models^{\lambda} ABC\\
&\textit{with } W(p) := \#\Pi^{-1}
\end{aligned}$$

Metrics This protocol is built on the Asynchronous Common Subset (ACS) primitive. ACS is a slight variation of vector consensus where the information of which node is associated to each value is erased, although the requirements themselves do not change [MXC+16]. HoneyBadgerBFT (HBBFT) uses the reduction to Reliable Broadcast and binary consensus from Ben-Or *et al.* [BKR94] to implement ACS. This reductions is essentially identical to the reduction of vector consensus in Lemma 4 from Section 3.3.

The RBC terminates in three rounds and has $\Theta(bn + n^2 \log(n) \operatorname{poly}(\lambda))$ bit complexity. The binary consensus is an asynchronous probabilistic protocol with $\Theta(n^2 \operatorname{poly}(\lambda))$ bit complexity. At each round it has $1/2$ probability to terminate. HBBFT has to wait for all BA instances of them to terminate, the time for this to happen is $\Theta(\log(n))$ rounds on average.

However one of HBBFT achievements is that at the end of this procedure, it commits data from all nodes inputs. That is, if all nodes have an input of size B, then the batch committed will be of size $\Theta(nB)$ bits. Hence, for b bits committed, there are $\Theta(bn + n^3 \log(n) \operatorname{poly}(\lambda))$ bits received by honest players. Note that the authors do provide an analysis, but they state their results in terms of *overhead*, *i.e.*, the total cost divided by b. Furthermore, by specifying minimum input size (batching policy) of $\Omega(n^2 \log(n) \operatorname{poly}(\lambda)) = O(b)$, they obtain the $O(n)$ figure which is a constant per-node overhead. This emphasis on the low overhead is less visible in our results, although it is translated by the fact that HBBFT complexity on the b factor is bn instead of bn^2 for other BFT-style protocols, *i.e.* a reduction factor of n consistent with the authors' analysis.

Claim. $\mathcal{L}(\text{HBBFT}) = \Theta(\delta log(n))$ *and* $\mathcal{CC}(\text{HBBFT}) = \Theta(bn + n^3 log(n) \operatorname{poly}(\lambda))$.

3.6.6 Phantom

Model Phantom [SZ18] is a Proof-of-Work protocol whose mining operation is very similar to Bitcoin, except that blocks may have more than one parent. As a result, the model is essentially the same as for Bitcoin, however the authors use their own formulation based on an earlier proposal [SLZ16]. Formally there is only a statement of safety (Proposition 7) which is compatible with the ABC formalism, moreover the authors explicitly aims to be a generalized version of Bitcoin, thus we are confident in choosing ABC to faithfully capture Phantom's properties.

The network is strongly synchronous, the authors state "if an honest node $v \in \mathcal{N}$ sends a message of size b MB at time t, it arrives at all honest nodes by time $t+D$ the latest.", where a bound one D is known to the protocol. Up to $\frac{1}{2}(1-d(\lambda,\Delta))$ of the hashpower may be corrupt, with d an (increasing) function lower bounded by 0. Although not mentioned explicitly, the adversary is dynamic for the same reason than with Bitcoin.

The Proof-of-Work mining is modelled by a Poisson process. This is well in accordance with our module which modelizes mining by a Binomial distribution, and the latter converges towards the Poisson distribution whenever the number of tries goes to infinity. Like with Bitcoin, we also require the Random Oracle to answer hash queries unrelated to mining.

Claim.

$$\begin{aligned}
&\text{PHANTOM} \in \mathcal{P}[\text{params}, \text{RO}, \text{sync_net}, \text{PoW}],\\
&\mathcal{M}_{\text{PHANTOM}}^{(\text{computational}, \mathcal{C}_{(W,1/2-\varepsilon(\lambda,\Delta))}, \text{dynamic})} \models^{\lambda} ABC\\
&\textit{with } W(p) := h_p(\sum_{q\in\Pi} h_q)^{-1} \textit{ and } \varepsilon : \mathbb{N}\times\mathbb{N} \to [0, \frac{1}{2}]
\end{aligned}$$

Metrics In Phantom the block structure is a Directed Acyclic Graph (DAG), which allows blocks to be linked to any number of previous blocks instead of one. A key observation in Phantom is that the PoW merely serves as a network-level synchronization primitive, thus decoupling the mining hardness from the security of the protocol. And indeed the mining rate is independent from λ and only required to be high enough so that blocks are not mined faster than they can be transmitted through the network. As such, there is little restriction on the mining process. More precisely, the entirety of the online protocol consists in mining on top of all the blocks with no successor in the block graph.

The online protocol maintains the DAG that grows over time, and another sub-protocol independently determines which values from the DAG will be the ABC output. Roughly speaking, transactions are extracted from the blocks graph, and for each of them the protocol computes whether their probability to be undone is greater than a given ε. Confirming a transaction still requires $\text{poly}(\lambda)$ blocks to be mined on top of it, but since block creation is more flexible, we only need to wait a $O(\Delta)$ additive delay, to ensure that all honest nodes received the blocks needed to confirm the transaction.

More precisely, Phantom implements a procedure *Risk* that gives a bound on the probability that safety will not hold for this transaction. The authors then show that the bound returned is smaller than a given ε after $O(\log(\frac{1}{\varepsilon}))$ honest blocks are created. Hence, with $\varepsilon = O(e^{-\lambda})$ and since this procedure requires the upper bound Δ, we have that the number of honest blocks required is $\text{poly}_\Delta(\lambda)$. The time taken for this to happen is obtained by multiplying it by the rate of honest block production, which only depends on Δ[11]. Then for all nodes to be aware of these blocks an additional Δ overhead is added, resulting in $O(\Delta + \text{poly}_\Delta(\lambda))$ latency.

Communication complexity on the other hand is simpler. In fact, the same analysis as the others blockchain style algorithms is applicable. Blocks all require a valid PoW to be sent through the network, and the PoW creation rate is bounded according to the corruption structure.

Claim. $\mathcal{L}(\text{PHANTOM}) = \Theta(\Delta + \text{poly}_\Delta(\lambda))$ *and* $\mathcal{CC}(\text{PHANTOM}) = \Theta(bn)$.

3.6.7 Algorand

Model Algorand [Mic17, GHM+17] uses a partially synchronous network, the Random Oracle, forward secure signatures, and Verifiable Random Functions.

As expected in a partially synchronous network they assume that a proportion $h > \frac{2}{3}$ of the stake is owned by honest nodes. However since Algorand uses an election mechanism, the probability of failure of the protocol increases as h goes to $\frac{2}{3}$.

Like with Ouroboros Praos, their usage of VRFs gives the possibility to tolerate dynamic adversaries. In fact, the Algorand protocol tolerates after-the-fact removal[12], which we could modelize by allowing the adversary to corrupt nodes right after they receive input from a module (*i.e.* before the events returned by the transition function are included in the execution).

[11] Indirectly, through the mining hardness.

[12] See Section 2.1.3.

Claim.

$$\begin{aligned}
&\text{ALGORAND} \in \mathcal{P}[\text{params}, \text{RO}, \text{weak_net}, \text{VRF}, \text{forward_signatures}],\\
&\mathcal{M}_{\text{ALGORAND}}^{(\text{computational}, \mathcal{C}_{(W,1/3-\varepsilon(\lambda))}, \text{dynamic})} \models^{\lambda} ABC\\
&\textit{with } W(p) := S_p(\sum_{q\in\Pi} S_q)^{-1} \textit{ and } \varepsilon : \mathbb{N} \to [0, \frac{1}{3}]
\end{aligned}$$

where S_p is node p's stake, as defined by the protocol.

Metrics Like Ouroboros and Snow White [DPS16], Algorand relies on a public randomness computed in previous blocks. It is used to elect a committee (instead of a leader) *at each round* that will have sufficiently many honest nodes, with overwhelming probability. These committees run a consensus protocol which does not require private state from the nodes (except from their private key), since committees would not be able to pass it on to the next committee properly. This property is called *player replaceability* by the authors. Except from this property, common techniques from BFT algorithms can be used to reach agreement in a constant number of rounds. In particular, since it has been ensured that committees have a constant fraction of honest nodes, standard quorum-based arguments are still valid.

More precisely, at each round, each user has a fixed probability p to be part of the committee. To bound the number of malicious nodes in a committee, the authors leverage the fact that $\forall t' < t$, the probability of having at most t' malicious nodes in a uniformly sampled committee of size k is $1 - \text{negl}(k)$. Thus, the committees expected size is $\text{poly}(\lambda)$.

As a result, each round that would be equivalent to a n^2 all-to-all communication in a traditional BFT algorithm is now a "committee-to-all", $\Theta(n\,\text{poly}(\lambda))$ communication.

To optimize bandwidth, Algorand does some kind of leader election, though a block proposing step. The average committee size for the block proposers is the smallest such that there is at least one proposer with overwhelming probability, which is still $\text{poly}(\lambda)$ asymptotically.

Claim. $\mathcal{L}(\text{ALGORAND}) = \Theta(\delta)$ *and* $\mathcal{CC}(\text{ALGORAND}) = \Theta(bn\,\text{poly}(\lambda))$.

Discussion

In Table 3.3 we summarize the above results as well as from Chapter 4. Concretely, For each protocol P in Table 3.3, we have made the following claim.

$$P \in \mathcal{P}[\text{params}, \text{RO}, M],\ \mathcal{M}_P^{(\text{computational}, \mathcal{C}_{(W,\alpha)}, (E,i,p) \mapsto \top)} \models^{\lambda} ABC$$

where: $W(p) := h_p(\sum_{q \in \Pi} h_q)^{-1}$ if the protocol uses the PoW module, or, if the protocol has a *stake* public parameter, $W(p)$ is p's relative stake. α is the contents of the "Adversary" column, M is the contents of the "Modules" column. We also claimed that $\mathcal{L}(P)$ is equal to the contents of the "Latency" column and that $\mathcal{CC}(P)$ is equal to the contents of the "Communication Cost" column.

This table clearly outlines the relation between the protocol's models and their performance metrics. The major drawback of Bitcoin is visible in its latency: the dependence on Δ is arbitrary, and could very well be exponential for instance. The same concern also applies to Phantom, and this is due to the fact that the confidence in a transaction depends on the time it take for PoW string to be transmitted. Moreover, we must recall that Δ must hold for all messages, of any length; thus increasing block size has a theoretically unknown impact the protocol latency. Theses considerations points towards the idea that PoW-based protocols should analyse how their latency depends on Δ to be correctly assessed. Interestingly, we can see that all three synchronous protocols have their corruption threshold dependent on Δ, and for the same reason for each of them: the "honest majority" assumption is stated as an inequality that depends on Δ. As expected, the $1/2$ corruption threshold is synonymous with strong synchrony, and the PoW oracle is able to replace digital signatures. The $1/x - \varepsilon$ is indicative of protocols making use of the election mechanism, which also imply the presence of $\text{poly}(\lambda)$ in either latency or communication cost. On the other hand Tendermint and HoneyBadgerBFT doesn't use the election mechanism and bear a communication cost at least quadratic in n. StakeCube and Algorand seem very similar, and indeed they share a common approach, their differences are found in more subtle details: Algorand tolerates a rushing adversary, while the constants in the communication cost are lower for StakeCube.

Algorithm	Adversary	Modules	Latency	Communication Cost
Nakamoto [GKL20a]	$1/2-\varepsilon(\lambda,\Delta)$	sync_net*, PoW	$\text{poly}_\Delta(\lambda)$	$\Theta(bn)$
Phantom [SZ18]	$1/2-\varepsilon(\lambda,\Delta)$	sync_net, PoW	$\Theta(\Delta+\text{poly}_\Delta(\lambda))$	$\Theta(bn)$
Ouroboros Praos [BGKR18]	$1/2-\varepsilon(\lambda,\Delta)$	sync_net*, signatures†, VRF	$\Theta(\Delta\,\text{poly}(\lambda))$	$\Theta(bn)$
Algorand [Mic17]	$1/3-\varepsilon(\lambda)$	weak_net, signatures†, VRF	$\Theta(\delta)$	$\Theta(bn\,\text{poly}(\lambda))$
StakeCube [DAL19, DHMA20]	$1/3-\varepsilon(\lambda)$	weak_net, signatures, VRF	$\Theta(\delta)$	$\Theta(bn\,\text{poly}(\lambda))$
Tendermint [Buc16]	$1/3$	weak_net, signatures	$\Theta(\delta n)$, f.f. $\Theta(\delta)$	$\Theta(bn+n^2)$
HoneyBadgerBFT [MXC+16]	$1/3$	async_net, signatures	$\Theta(\delta\log(n))$	$\Theta(bn+n^3\log(n)\,\text{poly}(\lambda))$

TABLE 3.3: Comparison Summary. The minus ε indicates that the fractional threshold is lowered as a function of the security parameter and in some cases of the network bound. f.f. is the value for failure free executions. * : The respective papers claims partial synchrony, see Section 3.6.1 for clarification. † : The primitive required is *forward-secure* signatures, which we do not have modelised.

Chapter 4

StakeCube: Leveraging sharding to scale Proof-of-Stake protocols

This chapter is based on previous work [DAL19] and [DHMA20] that has been improved and extended.

4.1 Introduction

FIGURE 4.1: StakeCube logo. Courtesy of Atos.

Permissionless blockchains aim at achieving the impressive result of being a persistent, distributed, consistent and continuously growing log of transactions, publicly auditable and writable by anyone. Despite the openness of the environment and thus the inescapable presence of malicious behaviours, security and consistency of permissionless blockchains do not demand the presence of a trusted third party.

Unfortunately, resilience of PoW-based solutions fundamentally relies on the massive use of computational resources, which is a real issue today. Lot of investigations have

been devoted to finding a secure alternative to PoW, but most of them either rely on the intensive use of a large quantity of physical resources (*e.g.*, proof-of-space [ABFG14], proof-of-space/time [MO16]) or make compromises in their trust assumptions (*e.g.* proof-of-elapsed-time [Int19], delegated proof-of-stake [EOS19]). In contrast, solutions based on proof-of-stake (PoS) seem to be a quite promising way to build secure and permissionless blockchains. Indeed, proof-of-stake relies on a limited but abstract resource, the crypto-currency, in such a way that each node weight in the protocol is proportional to the fraction of currency it owns. It is an elegant alternative in the sense that all the information needed to verify the legitimacy of a stakeholder to create a block (i.e., crypto-currency possession) is already stored in the blockchain. Finally, by being a sustainable alternative (creating a block requires just a few number of operations), throughput scalability concerns exhibited by PoW-based solutions should be a priori more tractable.

In this chapter we present a new blockchain protocol called StakeCube which aims at improving scalability of the block-wise Byzantine agreement approach by combining sharding techniques, users presence and stake transfer to operate in a PoS setting. The key idea of StakeCube is to organize nodes (*i.e.* stakeholders) into shards and within each shard, to randomly choose a constant size committee in charge of executing the distributed algorithms that contribute to the creation of blocks. Each block at height h in the blockchain is by design unique (no fork is possible), and once a block is accepted in the blockchain, the next one is created by a sub-committee of shards whose selection is random with a distribution that depends on the content of the last accepted block.

In a typical Proof-of-Stake fashion, StakeCube relies on a sufficiently high fraction of stake being owned by honest nodes. The stake distributed is recorded and maintained in the blockchain itself, allowing participating nodes to update it by making input to the protocol, *i.e.* submitting transactions. We might expect that solely relying on stakeholders (the owners of the coins of the crypto-currency system) to the secure construction of the blockchain makes sense due to their incentive to be fully involved in the blockchain governance, rather than delegating it to powerful miners.

StakeCube relies on an election mechanism as described in Chapter 1 to be efficient in terms of communication cost. The mechanism in question is *sharding*. Sharding refers to a distributed architecture design that partitions the processing load among disjoint *shards*, such that the overall system can become scalable. The concept of sharding has been identified as a promising development path to help solve blockchain scalability issue, and we

demonstrate with StakeCube the practical viability of such approach.The sharding structure in StakeCube is provided by a distributed hash table, PeerCube [ALRB08]. PeerCube is Byzantine fault tolerant and well suited to an environment with high churn rates, which is typical in a permissionless context. Each node is assigned to a unique shard, and within each shard a committee of nodes – the core set – will be tasked to realize the communication heavy tasks. In turn, to prevent the adversary from adaptively corrupting committees, we guarantee that the shard assignment is unpredictable, and that the sojourn time of users in their shard is limited. Doing so is an effective way to protect the system against eclipse attacks [ASLT11, AS07].

Given this sharding structure, we let nodes issue requests to join the protocol to their respective shards. Each shard locally updates its view to take into account the newcomers and departures. Views are updated, signed and installed once, and this occurs right before the creation of a new block. The creation of blocks is efficiently handled by an agreement among a verifiable sub-committee of shards. Indeed, to fine tune the shard size, StakeCube is able to tolerate a set number of corrupt shards, *i.e.* shards that contain too many malicious nodes to be able to correctly execute the protocols they are tasked with. This implies that the block creation procedure cannot be done by a single shard, but requires an *inter-shard agreement* protocol.

On a high level, the inter-shard agreement protocol works similarly to a classical consensus algorithm, except that it is tailored to work in a setting where a shard's vote may come from different nodes at each step. In a round-robin manner, it lets each participating shard internally generate a new block to be proposed. If the block goes through the all the voting steps, it will be the next block in the blockchain. The procedure for a single shard to generate a block is based on vector consensus and Verifiable Random Functions, both to make the block content (transactions) and to generate the common random value that seeds the randomness required in subsequent executions.

The overall scalability and efficiency of StakeCube comes from the fact that the core shards size is independent from the total number of nodes, and that shards only execute protocols that require a constant number of communication rounds.

4.1.1 Related work

Omniledger [KKJG+18] is the closest work to ours. It is a PoS-compatible, sharded, distributed ledger, resilient against a weakly dynamic adversary that corrupts up to $\frac{1}{4}$ of participants. In contrast to our approach, Omniledger assumes a strongly synchronous setting, and each shard maintains its own ledger and, global synchronization of transactions is achieved through an atomic commit protocol tailored to their usage. Ouroboros [KRDO16], representative of the leader-based approach, is a synchronous PoS protocol resilient against a weakly dynamic adversary that owns $1/2 - \varepsilon$ of stake. Moreover, Ouroboros has been recently improved to work in the partially synchronous setting against a dynamic adversary [BGKR18, BGK+18], while keeping the same design principles as the original one. In Ouroboros, a unique leader is elected at each round to broadcast its block which contrasts with our sharded approach where the block creation process is distributed. Snow White [DPS16] is a synchronous PoS protocol resilient against a weakly dynamic adversary that owns $1/2$ of the *active* stake. This protocol also relies on a leader election. Algorand [Mic17] is a representative of the blockwise Byzantine agreement approach. It provides a distributed ledger against an strongly adaptive adversary without assuming strong synchrony assumptions. However, by its design, agreement for each block of the blockchain is achieved by involving a very large number of stakeholders so that each one needs to effectively participate only for one exchange of messages.

Other blockchain experimentations In Section 4.5.1 we describe the experiments we made for StakeCube. On that matter, existing large scale blockchain experiments have also been made for Algorand [Mic17], Elastico [LNZ+16] and the Red Belly Blockchain [CGLR17b]. Runchao Han *et al.* evaluated several blockchains specifically with IoT in mind [HSGX20], namely Hyperledger Fabric v0.6 with PBFT, Fabric v1.0 with BFT-SMaRt, Ripple with BFT Ripple consensus, Tendermint with hybrid PBFT and Casper, R3 Corda with BFT-SMaRt. Their work showed that these algorithms do not scale well passed the tens of devices. There are several works that evaluate Hyperledger Fabric [TNV18, NQTN18, SWTR18] with up to 32 nodes. Some of these works make use of the Caliper [Fun] tool, which also supports most of the Hyperledger projects as well as Ethereum. Blockbench [DWC+17] is a framework supporting Ethereum, Parity and Fabrice, although it focuses more on resources utilised by smart contracts rather than network usage.

4.2 Model

On a high level, our solution is structured as a reduction to a few existing problems as building blocks, which could technically serve as an (abstract) model by themselves. However this view in terms of reductions is mostly useful for pedagogical purposes. The full StakeCube protocol includes specific implementations of these building blocks that all use the same base model. Namely :

- Computational security.
- Byzantine faults, with the assumption that at any point in time the fraction of stake owned by byzantine nodes is at most α.
- Weakly adaptive adversary. Specifically corruption must wait T blocks before being effective, where T is a protocol parameter. Each block being produced in a constant number of round, a weaker (but not as tight) requirement is to wait $O(\Delta T)$ time.
- Any set of nodes owning a fraction of $\frac{1}{3} - \varepsilon(\lambda)$ of the total stake can be corrupted.
- Probabilistic. This is due to the use of verifiable random functions.
- Partially synchronous network (including the loosely synchronized clocks).
- As setup, we assume an initial uniformly random value is known to all participants. Otherwise, we require the initial knowledge of other nodes public keys, their stake, and the protocol parameters. All this initial information forms the *genesis block*.
- Random oracle, digital signatures, Verifiable random functions.

Users own some minimal amount of stake (*i.e.* money), which gives them the right to participate to StakeCube. We adopt (a simplified version of) what is commonly known as the Bitcoin Unspent Transaction Output (UTXO) model. An UTXO can be roughly seen as a user's account credited by some stake. An UTXO is uniquely characterized by a public key pk_i and its associated amount of stake s_i. Each public key is related to the digital signature scheme, which allows stakeholders to use the public keys (or a hash thereof) of their UTXOs as a reference to them, as demonstrated in the "Public Keys as Identities principle" of Chaum [Cha88]. At any time, a user can own multiple UTXOs.

UTXOs can be debited only once, and once debited, an UTXO does not exist anymore. To simplify discussion, we make transactions outputs contain pk_i instead of its hash.

StakeCube is also able to accommodate dynamic participation. Since nodes are identified with a UTXO key, executing a transaction implies the removal of the node with a spent UTXO, and the addition of the node with the newly created UTXO. As already mentioned, our model from Chapter 4 does not support dynamic participation, thus we describe this aspect in a more classical manner. In any case, StakeCube's guarantees are given to the honest nodes *currently participating*. The trust assumption requires $\frac{2}{3} + \varepsilon(\lambda)$ stake to be owned by honest participating nodes, *at any point in time*. In turn, the stake distribution considered in this assumption is the one agreed on by all nodes, *i.e.*, the result of applying all accepted transactions to the initial distribution. Similarly to the fact that the PKI assumption is necessary to authenticate nodes, we must also assume that newly joining nodes are able to correctly transmit their public key (or its hash) when the transaction is submitted. For the same reason, they must also be able to obtain the public keys of the other currently participating nodes.

StakeCube's parameters

- The security parameter $\lambda \in \mathbb{N}$. We tolerate an adversary with polynomial runtime in λ, and all StakeCube's properties are guaranteed with probability $1 - negl(\lambda)$.
- The epoch length $T \in \mathbb{N}$, in blocks. In a nutshell, increasing this parameter makes the protocol more efficient, but also weakens the tolerated adversary, due to the weak adaptivity assumption.
- The maximum share of stake owned by malicious nodes $\alpha \in [0, 1/2]$
- The maximum number of corrupted shards $F_{\text{shard}} \in \mathbb{N}$.
- The number of core members in a shard $s_{\min} \in \mathbb{N}$.
- The maximum number of nodes in a shard $s_{\max} \in \mathbb{N}$.
- The initial stake distribution $S^0 : \Pi \to \mathbb{N}$.

Increasing either F_{shard} and $s_{\min}$ reduces the adversary probability of defeating the protocol, but increases the communication cost. In the security analysis Section 4.4, we show that StakeCube is secure and scalable for any F_{shard} if $s_{\min} = \text{poly}(\lambda)$ and $s_{\max} = \sqrt{(n)}$. However this analysis does not yield practical values for these parameters. In the implementation, we run experiments to heuristically estimate safe values for these parameters.

We assume the presence of Byzantine (*i.e.* malicious) users which controls up to $\alpha \leq 1/3 - \varepsilon(\lambda)$ of the total amount of stake currently available in the system. Here, $\varepsilon(\lambda)$ quantifies the gain in the effective adversarial power, related to the security parameter.

We denote by H the hash function modelized from the random oracle. We assume that all messages are signed by the sender. Signatures are solely used to authenticate nodes messages, thus by only considering valid messages we never need to refer to the signature scheme directly.

To setup the genesis block, any prior stake distribution can be used, with the only constraint that two thirds of stake must be owned by honest nodes. Then, a common coin tossing protocol modified to take into account the stake weights may be used to generate the initial randomness.

Additionally, StakeCube uses two subprotocol: A vector consensus protocol and an inter-shard agreement protocol, which is roughly similar to a verifiable weak consensus protocol. The inter-shard protocol makes shards coordinate to verify and agree on a single block despite the presence of corrupt *shards*. The vector consensus has two usages : first, to let each shard agree on the set of newcomers to update the view, secondly, to let one leader shard propose a set of transactions for the next block, which is required by our inter-shard protocol.

4.2.1 StakeCube's properties

Formally, StakeCube is a probabilistic Atomic Broadcast protocol with the message space V being valid blocks. Its model and security of StakeCube are stated in claim 4.2.1 using the framework from Chapter 3, although this statement doesn't account for StakeCube's ability to have stake (and therefore participants) change during execution.

Claim (StakeCube is secure)**.**

$$\begin{aligned}
&\text{StakeCube} \in \mathcal{P}[\text{params}, \text{weak_net}, \text{RO}, \text{signatures}, \text{VRF}],\\
&\mathcal{M}_{\text{StakeCube}}^{(\text{computational}, \mathcal{C}_{(W,\alpha)}, D)} \models^{\lambda} ABC\\
\textit{where } &W(p) := S^0(p)\Big(\sum_{q \in \Pi} S^0(q)\Big)^{-1}\\
&D(E,i,p) := \forall q \in H,\ \#\{j \in E^{-1}(\mathfrak{p}, \mathfrak{o}, q, _) \mid j > i\} \geq T
\end{aligned}$$

Internally, it let any honest user i locally maintain a sequence of blocks $B_0^i, B_1^i, \ldots, B_h^i$, where h represents the index (or the height) of the block in the sequence[1]. This sequence of blocks represents i's copy of the distributed ledger, extensions to the ledger are output from the protocol, and input can be made by submitting transactions to be appended.

StakeCube has an inductive structure: we describe a protocol that builds the block B_{h+1}, assuming that it was successful for the blocks up to B_h. The way to obtain the ABC protocol from the block construction protocol is very similar to the reduction to vector consensus (Lemma 15): nodes maintain an input pool from which they take the content that will end up in the blocks. Even more similar to that reduction, the block contents in StakeCube is, too, the union of the output of a vector consensus instance, although it is executed among a very specific set of nodes. The straightforward input pool implementation as described in the reduction would undermine our efficiency results, but simple strategies can mitigate this issue, such as limiting broadcast to neighbouring shards.

In StakeCube, participation is voluntary: Any honest user can join a shard (determined by the protocol), whenever she wishes, with the objective of eventually being involved in the Byzantine resilient protocols executed in this shard. Participation is temporary: The sojourn time of an honest user in a shard is defined by the time it takes for StakeCube to create T blocks. Once she leaves, she can participate again by joining another shard, and does so until she spends her UTXO. As users may own multiple UTXOs, they can simultaneously and verifiably sit in different shards. In the following, a user that issues a join request with its current credential is called an *active* user. StakeCube satisfy Scalability

[1]This is similar to the Transaction Ledger formulation from Garay *et al*.

and Efficiency properties. This is achieved due to the properties of the block creation process. Adding a new block takes two Byzantine fault tolerant protocols to be run in parallel within each shard, one network wide message diffusion by each shard, one inter-shard Byzantine agreement, and finally one broadcast for the block (more details will be given in Section 4.3.3).

Scalability

Theorem 8 (StakeCube is scalable)**.** $\mathcal{CC}(\text{StakeCube}) = \Theta(bn\,\text{poly}(\lambda))$

StakeCube relies on an election mechanism, where the communication heavy protocols are executed by the shards members. For every block, the communication cost is computed as follow: First, $O(bn)$ bits are required to multicast the transactions to the transaction pool of all nodes. Running the view update is $\Theta(s_{\min}^3)$ per shard, which is in total $\Theta(s_{\min}^3 \frac{n}{s_{\max}}) = \Theta(\sqrt{n}\,\text{poly}(\lambda))$. Diffusing views is an all-shards to all-shards communication between core members, taking $\Theta(n\,\text{poly}(\lambda))$ bits. Constructing the next block with the inter-shard agreement requires $\Theta(bs_{\min} + s_{\min}^3)$ for the block proposal and $\Theta(bs_{\min}F_{\text{shard}} + s_{\min}^2 F_{\text{shard}}^2)$ for voting and $\Theta(bn)$ for diffusing the full blocks, which totals to $\Theta(b\,\text{poly}(\lambda) + bn)$.

The communication cost of each individual node varies greatly depending on its current role (*e.g.* core member, shard committee, etc), but because these roles are assigned randomly, the amortized per-node communication cost is sublinear in n.

Efficiency

Theorem 9 (StakeCube is efficient)**.** $\mathcal{L}(\text{StakeCube}) = \Theta(\Delta)$

All Byzantine fault tolerant protocols we rely on use a constant number of rounds. Thus adding a new block also takes a constant number of rounds. Because a transaction, once diffused, will be included in the next block and blocks are permanently attach to the blockchain, it takes at most two blocks to include a newly received transaction.

4.2.2 The PeerCube sharding structure

A Distributed Hash Table (DHT) Distributed hash tables (DHTs) build their topology according to structured graphs, and for most of them, the following principles hold: each

node of the system has an assigned identifier, and the identifier space, *e.g.*, the set of 256-bit strings, is partitioned among all the nodes of the system. Nodes self-organize within the graph according to a distance function based on the identifier space.

Sharded DHT The notion of Sharded DHT is similar to a regular DHT, except that each vertex of the DHT is a set of nodes instead of a single node. That is, nodes gather together into shards, and shards self-organize into a DHT graph topology. Sharded DHTs can be made robust to adversarial strategies as achieved in SChord [FSY05], and PeerCube [ALRB08], and robust to high churn as achieved in PeerCube [ALRB08] by running Byzantine tolerant algorithms within each shard. For these reasons, we rely on PeerCube architecture, while weakening its model by removing the assumption of a global trusted party supplying verifiable random identifiers, and by removing the assumption of a static adversary. For self-containment reasons, we now recall the main design features of PeerCube. Briefly, this is a DHT that conforms to an hypercube. Each vertex (*i.e.* shard) of the hypercube is dynamically formed by gathering nodes that are logically close to each other according to a distance function applied on the identifier space.

Shards are built so that the respective common prefix of their members is never a prefix of one-another. This guarantees that each shard has a unique common prefix, that in turn serves as a shard's *label*. The shard's label characterizes the position of the shard in the overall hypercubic topology, as in a regular DHT. Shards size is upper and lower bounded. Whenever the size of shard $\mathcal{S}$ exceeds a given value $s_{\max}$, $\mathcal{S}$ splits into two shards such that the label of each of these two new shards is prefixed by $\mathcal{S}$ label, and whenever the size of $\mathcal{S}$ falls under a given value $s_{\min}$, $\mathcal{S}$ merges with another shard to give rise to a new shard whose label is a prefix of $\mathcal{S}$ label. Each shard self-organizes into two sets, the core set and the spare set. The core set is a fixed-size random subset of the whole shard. It is responsible for running the Byzantine agreement protocols in order to guarantee that each shard behaves as a single and correct entity (by for example forwarding all the join and lookup requests to their destination) despite malicious participants [ALS12]. Members of the spare set merely keep track of shard state. Joining the core set only happens when some existing core member leaves, in which case the new member of the core set is randomly chosen among the spare set. By doing this, nodes joining the system weakly impact the topology of the hypercube [ALRB08].

To distribute communication load, inter-shard communication is defined to happen between the core sets, which are then responsible for forwarding the message to the spare set. However, if the whole core is corrupted, then shards can now mount eclipse attacks against to prevent any honest spare member from progressing indefinitely. To prevent this situation, we require that there is at least one honest core member in each shard, including corrupted ones. Because StakeCube's security parametrization has two independent parameters, *i.e.* s_{min} and F_{shard}, we can always meet this requirement.

Efficient representation Because in our implementation the entire PeerCube structure is replicated by all nodes, we made a few design choices. It is seen as a binary tree, where a path in the tree represents a label and a leaf is a PeerCube identifier. Hence, shard information is stored at the path addressed by its label, and nodes are stored at the leaves addressed by their identifier. Additionally, each tree node also stores the number of nodes belonging in the subtree. This allows us to efficiently batch update operations: inserting/deleting a node with its identifier is logarithmic; updating shards after several modifications only takes one iteration of the whole tree, with depth limited to the shards nodes.

4.3 Design Principles of StakeCube

The key idea of StakeCube is to have an election mechanism not to elect a single committee but multiple ones in parallel, thus forming a shard system. The randomization of shards members gives a statistical bound on the number of malicious participants sitting at each shard, ensuring the correct execution of the agreement primitives within. More precisely, we compute bounds that may still cause some shards to have too many malicious participants (they become *corrupted shards*), but the overall number of corrupted shards is bounded by the parameter F_{shard}. This technique allows us to fix a small shard size while keeping the ability to make security-efficiency trade-offs.

On a high level, the life cycle of a transaction tx is as follow. First, it gets added in the transaction pool of the nodes. Then, after shards have updated their views, the core members of the current leader shard execute a vector consensus protocol. One of the members may propose tx, and if successful, the output vector will contain tx. From that

vector, a block proposal containing tx is locally computed and sent to the other shards that are members of the shard committee. Following the inter-shard agreement protocol, this block is voted by the shards, and if successful will gather enough votes to form a final block with a certificate, after which it will be sent to the whole network. At that point the block is delivered and tx is confirmed.

To be able to tolerate the presence of a Byzantine adversary, we must guarantee that the adversary cannot predict the shards in which users will sit, and that the sojourn time of users in their shard is limited. To achieve this, we introduce the notion of unpredictable and perishable users' credentials in Section 4.3.1. Then to cope with this induced churn, we show how to update, sign and install the shards' views in Section 4.3.2. This process occurs right before the acceptation of a new block. Finally, as described in Section 4.3.3 the creation of blocks is efficiently handled by an agreement among a verifiable sub-committee of shards.

4.3.1 Unpredictable and Perishable Users' Credentials

As described in Section 4.2.2, PeerCube critically relies on a (global) trusted party supplying verifiable random identifiers to nodes. In this section, we detail how to construct those in our decentralized setting, using the already known public keys and some randomness present in each block. For each unspent public key, *i.e.* for each UTXO, owned by a user, a sequence of unpredictable and perishable credentials are tightly assigned to her. Validity of a credential spans T blocks, with T some positive integer. The credential σ assigned to user i for her UTXO (pk_i, sk_i) is computed as follows. Let B_{h_0} be the block at height h_0 of the blockchain such that pk_i was created in B_{h_0}, *i.e.*, it exists a transaction in B_{h_0} such that pk_i appears in the output list of that transaction. For any blockchain height $h \geq h_0 + T$, such that UTXO (pk_i, sk_i) still exists when B_h is accepted in the blockchain,

$$\sigma_{pk_i}(h) := H(pk_i || B_{h'}.\rho), \quad \text{where} \quad h' := h_0 + \lfloor \frac{h - h_0}{T} \rfloor T, \tag{4.1}$$

with $B_{h'}.\rho$ a random number included in $B_{h'}$ whose computation is detailed in Section 4.3.3. Suppose that i's UTXO (pk_i, sk_i) is created in block B_h. Then by Relation 4.1, i's first credential for UTXO (pk_i, sk_i) is computed based on the content of block B_{h+T} and perishes at block B_{h+2T}. Then, i's second credential for (pk_i, sk_i) is computed based on the content

of block B_{h+2T} and perishes at block B_{h+3T}, and so on until i spends (pk_i, sk_i). User i's credential uniquely characterizes the shard to which user i is allowed to sit, and this shard is the one whose label prefixes i's current credential $\sigma_{pk_i}(h)$. By the non-inclusion property of PeerCube [ALRB08], there does not exist a shard whose label is the prefix of another shard, and thus, there is a unique shard whose label prefixes credential $\sigma_{pk_i}(h)$. When her current credential expires, i leaves the shard she is in, and if she wants to continue to participate to StakeCube, joins a new shard based on her new credential.

There are a couple of details that should be noted.

1. User i does not need to participate in StakeCube for the entire life of her UTXO (pk_i, sk_i). She can join StakeCube (*i.e.* join a shard) at any time h under credential $\sigma_{pk_i}(h)$, however once a user joins her shard, she must stay online (and actively participate if she is a core member) until $\sigma_{pk_i}(h)$ expires. As a result, there does not exist any explicit leave request. A leave simply consists in not issuing a join request upon credential renewal. A consequence of this rule is that, in case user i participates under credential $\sigma_{pk_i}(h)$ and spends her UTXO (pk_i, sk_i) before $\sigma_{pk_i}(h)$ expires, then i continues to participate under $\sigma_{pk_i}(h)$ until $\sigma_{pk_i}(h)$ expires. Note that because a transaction only grants credentials after a delay, this rule does not allow a user to simultaneously own multiple credentials for the same stake. Note also that if i is disconnected for a small amount of time this does not jeopardize the safety of the shard, only its liveness.

2. Recall that the adversary has a bounded fraction α of *stake* in StakeCube. To defend StakeCube against Sybil attacks (*i.e.*, the fact that the adversary creates a considerable number of UTXOs with the objective of overpopulating each shard with malicious owners of those UTXOs), we require that each UTXO cannot be credited with more than M stake, with M some predefined constant. Consequently, by the fact that for any $h > 0$ one credential $\sigma(h)$ represents exactly one UTXO, there is a bound $\alpha_{\text{cred}} > \alpha$ on the fraction of malicious credentials in StakeCube, which is reached when all malicious UTXOs have 1 stake and all honest ones maximize their stake, *i.e.*, each honest UTXO has M stake. Note that UTXOs with M' stake, such that $M' > M$ may be handled by granting them $\lceil M'/M \rceil$ credentials, although we do not treat this case explicitly. Section 4.4 analyzes the distribution of malicious credentials among shards.

Regarding the behaviour of the adversary, there are a couple of remarks to note.

1. At any time, the adversary might spend some selected UTXOs in order to create new ones and thus new credentials with the objective of targeting some shards. However, because of the initial T blocks delay required to obtain the first credential for an UTXO (see Relation 4.1), any newly created UTXO will give rise to a credential only after all existing credentials are renewed as well. Therefore, the adversary has no preferred strategy regarding transactions and forced renewal.

2. Each block B_h contains a random seed, denoted by $B_h.\rho$, which cannot, by construction, be either biased or predictable before the block is created (how such seeds are generated is detailed in Section 4.3.3). Thus by Relation 4.1, the adversary cannot determine nor influence the value of renewed credentials. Consequently, for any blockchain height $h \geq 0$ and for any pk_i, $\sigma_{pk_i}(h+T)$ is unpredictable while for any $0 \leq h' \leq h$, the sequence $(\sigma_{pk_i}(h'+T))_{0 \leq h' \leq h}$ is computable and verifiable from the blockchain.

4.3.2 Shard Membership

As described above, during the period of time that elapses between the creation of an UTXO to its spending, the UTXO owner can participate to the blockchain construction by successively joining a series of shards. In practice this may give rise to a voluminous amount of join requests, which might be highly prejudicial to StakeCube's scalability and efficiency if each joining request led to the insertion of the newcomer in the core running the distributed operations. Rather, by relying on PeerCube design (see Section 4.2.2), a newcomer joins the spare set of the shard and not its core set. This newcomer will be a candidate for being elected as a member of the core set whenever the core set will undergo a membership modification. Management of the view composition, and election in the core set is the subject of the remaining of this section.

View of a Shard The view of a shard $\mathcal{S}$ reflects the composition of both its core and spare sets, denoted respectively by $\mathcal{S}_c$ and $\mathcal{S}_s$. Update of the view is strongly correlated to blockchain events: any block appended to the blockchain is preceded, in each shard, by the update and the installation of the shard view. In the following, the view of shard $\mathcal{S}$

installed right before block B_h is appended to the blockchain is denoted by $view_{\mathcal{S}}(h)$. We have $view_{\mathcal{S}}(h) = (\mathcal{S}_c(h), \mathcal{S}_s(h))$, where $\mathcal{S}_c(h)$ (resp. $\mathcal{S}_s(h)$) represents the composition of $\mathcal{S}$'s core set (resp. spare set) at time h.

Update of the Shard View When a newcomer (*i.e.* a user under a valid credential) issues a request to join her shard $\mathcal{S}$, her request is propagated and broadcast to the members of $\mathcal{S}_c$. Core members i locally store the join request in their buffer b_i of pending requests. Note that expiration of credentials do not need to be locally memorized, prior to being handled by the view update algorithm, since by Relation 4.1, credentials can only expire when a new block is appended to the blockchain. Let $view_{\mathcal{S}}(h-1)$ be the current view of $\mathcal{S}$ when a (honest) core member $i \in \mathcal{S}_c(h-1)$ receives some valid block B_h (Section 4.3.3 details the creation of blocks). The following three steps are successively executed:

1. A Byzantine vector agreement protocol is run among $\mathcal{S}_c(h-1)$ members to decide on the set of newcomers: core members i propose their local buffer b_i, and the outcome of the protocol is a vector $v(h)$ of newcomers such that non-null values for honest core members i are equal to their buffer b_i. Each honest core member i replaces its local buffer b_i with the union of the users of the decided vector. We have $b_i = \cup_{b_j \in v(h), b_j \neq \perp} b_j$.

2. Each user $i \in \mathcal{S}_c(h-1)$ removes from b_i the set $r_{\mathcal{S}}(h)$ of users whose credential expires with B_h. User i initializes a new spare set $S_s(h)$ with $S_s(h) = b_i \cup S_s(h-1) \setminus r_{\mathcal{S}}(h)$, and orders $S_s(h)$.

3. Each user $i \in \mathcal{S}_c(h-1)$ initializes a new core set $S_c(h)$ with $S_c(h) = S_c(h-1) \setminus r_{\mathcal{S}}(h)$. If $\mathcal{S}_c(h-1) \cap r_{\mathcal{S}}(h) \neq \emptyset$, some previous core members $i \in \mathcal{S}_c(h-1)$ have credentials that expire with B_h. As a consequence, an election among the users of $\mathcal{S}_s(h)$ is carried out for i's replacement, so as to keep $|\mathcal{S}_c(h)| = s_{\min}$. The core election works as follows:

   (a) A pseudo-random number generator $PRG(B_h.\rho)$ is initialized with the last block random seed.

   (b) $PRG(B_h.\rho)$ is used to draw a random number $j \in [\![1, |S_s(h)|]\!]$. The j-th member of $\mathcal{S}_s(h)$ is removed from $\mathcal{S}_s(h)$ and added to $\mathcal{S}_c(h)$. This process is repeated until $|\mathcal{S}_c(h)| = s_{\min}$.

Once these steps are completed, each core member j installs her new view $view^j_{\mathcal{S}}(h)$ with the new values of $S_c(h)$ and $S_s(h)$, signs it, and sends it to the spare members. Once a spare receives $\alpha_{\text{core}} s_{\min} + 1$ signatures on the same view, it installs it. In the meantime, each core member j resets its buffer $b_j = \emptyset$. Note that multiple join requests may lead a shard $\mathcal{S}$ to split into two shards, or, on the contrary, may lead two shards $\mathcal{S}'$ and $\mathcal{S}''$ to merge within a single one $\mathcal{S}$. The treatment of such topological changes are detailed in [ASLT11].

To summarize, the shard membership procedure ensures that, for any shard $\mathcal{S}$ of StakeCube, all members of $\mathcal{S}$ install the same view $view_{\mathcal{S}}(h)$ before appending block B_h to their copy of the blockchain.

Diffusing Views Merely installing the new view for each shard is not sufficient. We need the other shards of StakeCube to maintain this knowledge to be able to verify any signed information exchanged during inter-shard communication (*e.g.* during the block proposal procedure, see Section 4.3.3). Therefore, whenever a new view $view_{\mathcal{S}}(h)$ is installed along with its $\alpha_{\text{core}} s_{\min} + 1$ signatures, it is also broadcast to the whole network as a notification of the view update. This broadcast operation can be made efficient by relying on PeerCube's structure, allowing shards to only relay the view to its neighboring shards. Note that shards only store the last view $view_{\mathcal{S}'}(h)$ of any other shard $\mathcal{S}'$ and not the whole history of $\mathcal{S}'$ views. Moreover, a new view $view_{\mathcal{S}'}(h+1)$, can be verified against the last view $view_{\mathcal{S}'}(h)$, so that corrupted shards can only lie on their core members and omit newcomers.

4.3.3 Construction of the Next Block of the Blockchain

We propose a Byzantine resilient cross-shard mechanism to agree on a unique valid block, despite the presence of at most F_{shard} corrupted shards (see Section 4.4 for F_{shard} computation). Indeed, the presence of an adaptive adversary may compromise the safety of some shards by succeeding in having more than a proportion α_{core} of malicious users sitting in their core set. Although the probability of such event can be made arbitrarily low (see the analysis in Section 4.4), we must handle it. The presence of corrupted shards put us in the same situation as in a consensus protocol: given the same initial chain, any shard is able to create the next block, and the decision must be a unique block, despite malicious users

lying or not responding. As will be shortly described, agreeing on a unique valid block is efficiently and robustly achieved by running a verifiable Byzantine agreement among a subset of the shards of StakeCube randomly selected.

Reaching Consensus on the Next Block The process of creating a new block B_h starts right after B_{h-1} has been accepted. A committee of shards, denoted in the sequel by $\mathbb{C}$, is elected among the shards of StakeCube. The election of each of these shards relies on the seed of the previous block B_{h-1}. Once elected, committee $\mathbb{C}$ executes a verifiable Byzantine agreement to decide on the unique block B_h to be appended to the blockchain. The main steps of this process are as follows:

1. All shards $\mathcal{S}$ compute the elected committee $\mathbb{C}$, similarly to the core election procedure (see Section 4.3.2), i.e.,
   (a) Let $\mathbb{L}$ be the set of all the shards' labels (recall from Section 4.3.2 that each shard diffuses its new view $view_{\mathcal{S}}(h)$). $\mathbb{L}$ is then ordered through a canonical order.
   (b) A pseudo-random number generator $PRG(B_{h-1}.\rho)$ is initialized, where $B_{h-1}.\rho$ is the seed of the last block B_{h-1}.
   (c) $PRG(B_{h-1}.\rho)$ is used to draw a random number $j \in [\![1, |\mathbb{L}|]\!]$. The j-th member of $\mathbb{L}$ is removed from $\mathbb{L}$ and added to $\mathbb{C}$ (initially initialized to $\emptyset$). This process is repeated until $\mathbb{C}$ contains $s_{\mathbb{C}}$ shards, with $s_{\mathbb{C}} = (F_{\text{shard}}/\alpha_{corrupted}) + 1$. Recall that F_{shard} is the maximal number of corrupted shards in StakeCube (whose computation is presented in Section 4.4), and $\alpha_{corrupted}$ is the fraction of malicious nodes tolerated by the inter-shard agreement protocol (see Section 4.3.4).
2. Members of committee $\mathbb{C}$ run the inter-shard agreement protocol, with their proposed block B_h as input (the construction of the proposed block is described in the next paragraph). Finally the decision is a block $B_{h'}$ signed by $2F_{\text{shard}} + 1$ shards.
3. Block b_h is broadcast by the committee to all nodes and appended to StakeCube users' copy of the blockchain.

Security remark By definition of $s_{\mathbb{C}}$, committee $\mathbb{C}$ cannot be corrupted, independently of the shards selected by the election. Committee $\mathbb{C}$ is still chosen randomly for two reasons. First, it naturally spreads the load of creating a block across the whole network. Second, it prevents corrupted shards from trying to manipulate the election process to get in the committee and slow it down.

Efficiency remark We rely on a leader-based Byzantine Agreement algorithm to benefit from its optimistic efficiency. Indeed, since F_{shard} can be made arbitrarily small (see Section 4.4), and the members of committee $\mathbb{C}$ are randomly selected, we expect the first leader to almost always be an honest shard.

Construction of the Proposed Block We finally describe how the current leader shard $\mathcal{S}$ of $\mathbb{C}$ constructs its block B_h (see the above case 2). The construction results from an agreement on the content of block B_h among the core members of $\mathcal{S}$ and on the generation of the seed of B_h. Let $view_{\mathcal{S}}(h) = (\mathcal{S}_c(h), \mathcal{S}_s(h))$ be the current view of shard $\mathcal{S}$.

1. Each core member in $\mathcal{S}_c(h)$ proposes *(i)* its list of pending transactions and *(ii)* its VRF value seeded with $B_{h-1}.\rho$ together with the VRF proof, to the Byzantine vector consensus protocol. The decision value is a vector of input values, such that non-null values for honest core members are equal to their list of pending transactions and their VRF value and VRF proof.

2. Construction of block B_h is then realized as follows.

   - The hash of the previous block B_{h-1} is inserted in B_h's header.
   - The union of transactions from the decided vector defines B_h's body.
   - The VRF from the first non-empty value from the decided vector defines the seed $B_h.\rho$ of B_h.
   - The corresponding VRF proof is inserted in B_h's header as a proof of randomness for seed $B_h.\rho$.

Note that the computation of random seed is not entirely unpredictable. Indeed the adversary is able to choose *which* nodes include its VRF, thus giving it $s_{\min}$ choices for the seed. This weakening can be accounted for by increasing the security parameter accordingly.

When a new block is created, it has to be efficiently broadcast to all nodes participating to StakeCube, *e.g.* through a gossip protocol. For ease of implementation, we rely on a simple diffusion protocol based on the sharding structure: Whenever a shard member receives a block for the first time, the node forwards it to all its neighbour shards. Because inter-shard communication is done through core-to-core broadcast, corrupt shards are not an issue regarding this diffusion protocol.

4.3.4 Inter Shard agreement

In [DAL19], an algorithm from Chen & al [GHM$^+$17] is suggested for the inter-shard agreement. However, running an algorithm with shards instead of nodes as participants required some adaptations, as well as some additional optimisation related to our setting. In this section, we present an inter-shard agreement protocol that has been inspired from the suggested algorithm. Critically, we conserve the property of "player-replaceability" which lets different players vote at each step. This property specifically allows us to have different players from the same shard act as one. More precisely, we will say that:

1. A shard S has sent a message m to the shard D whenever at least $2f+1$ core members of S have sent m to all the core members of D.

2. A node p has received a message m from a shard S whenever it has received m from at least $2f+1$ core members of D.

3. A shard D has received a message m from the shard S whenever at least $2f+1$ core members of D have received m from the shard D.

According to these definitions, we can deduce that if shard S sent a message m to shard D, then D will eventually receive m from D, and that eventual synchrony assumptions also apply. That is, inter-shard communication shares the same properties and synchrony assumptions than inter-node communication.

Note that if a honest shard sends a message to a corrupted shard, its honest members will receive the message nonetheless, thus allowing them to continue receiving blocks until the shard becomes honest again.

Moreover, the algorithm from Chen & al. is not entirely optimal for our setting. More precisely, it achieves security against a rushing adversary. In our case, the adversary is

only weakly adaptive and has to wait T blocks before corrupting a node, *i.e.* more than the duration of the agreement. This means that security against static adversaries is sufficient for our inter-shard agreement. Because of the rushing adversary, the original algorithm cannot rely on a known leader and has to determine leadership *after* block proposals are sent. We can define a leader rotation determined from the initial common randomness (*i.e.* the block seed), and all shards will wait for its proposal. Note that if no proposal is received, the timers will eventually timeout and the next period will start.

The pseudo code for the inter-shard agreement is given in Algorithms 1 and 2. This algorithm is presented in an event-driven style, where each "upon" block describes the processing of a specific event. Note that this does not imply any kind of parallelism in the treatment of events. In fact, in our implementation, message processing is exclusively single-threaded to ensure that the node state stays consistent. Let $\mathcal{H}$ be the set of hashes values, $\mathcal{V} := \mathcal{H} \cup \{\perp\}$ the set of votes values, $\mathcal{B}$ the set of blocks and $leader(period)$ the identifier of the leader shards for the current period, determined in a round-robin manner. The **send** primitive sends an inter-shard message to all shards, and **gossip** efficiently diffuses an inter-node message to all shards core members.

Differences with the original algorithm Our algorithm ensures that a vote will always be sent only for one value at each period. Because the leader is known in advance, we are already expecting its proposition to soft vote.

Additionally, when the current period is reached after next voting a value, all other honest nodes should also reach the same period with the same value. Hence we already know that the leader is going to propose it and we can soft vote it in advance. Note that in this case in the original algorithm, the soft voted value also ignores the leader proposal.

Because we can soft vote in advance or just wait for the leader proposal, the "voting" step of the original algorithm becomes empty for us and can be removed. In this algorithm, "value" refer to hashes of block. The block data itself is broadcast when its value is proposed, and we allow soft votes to be sent without knowing the block data.

However the block data is required to be known to cert vote or output it, which ensure than only valid blocks can be committed. This is an implementation of the "one step" extension of the original algorithm.

Algorithm 1: Inter-shard agreement Pt.1

local variables:
$period \in \mathbb{N}$ the current period, initially 0
$prevNextVote \in \mathcal{V}$ the value that made the node go to the current period, initially $\perp$
$knownBlocks \subset \mathcal{H} \times \mathcal{B}$ the hashset of blocks received from gossip, initially $(\emptyset, \emptyset)$
timer an object that expires after some timeout duration.

procedure *start_period()*
    start $timer$;
    if $prevNextVote \neq \perp \wedge prevNextVote \in knownBlocks.KeySet()$ **then**
        if $leader(period) == self$ **then**
            send $(Propose, period, prevNextVote)$;
            gossip $(period, knownBlocks[prevNextVote])$;
        send $(SoftVote, period, prevNextVote)$;
    else if $leader(period) == self$ **then**
        $newBlock \leftarrow GenBlock()$;
        send $(Propose, period, H(newBlock))$;
        gossip $(period, newBlock)$;
end

upon *receiving* $(Propose, period, v)$ *from* $leader(period)$ **do**
    if $prevNextVote = \perp \vee prevNextVote \notin knownBlocks.KeySet()$ **then**
        send $(SoftVote, period, v)$;
end

upon *receiving* $(SoftVote, period, v)$ *from* $2F_{shard} + 1$ *distinct senders* **do**
    if $v \in knownBlocks.KeySet()$ **then**
        if *timer not expired* **then**
            send $(CertVote, period, v)$;
        else
            send $(NextVote, period, v)$;
end

upon *receiving* $(CertVote, period, v)$ *from* $2F_{shard} + 1$ *distinct senders* **do**
    wait until $v \in knownBlocks.KeySet()$;
    output $knownBlocks[v]$;
end

Algorithm 2: Inter-shard agreement Pt.2

upon *timer expires* **do**
  if $(CertVote, period, v)$ *has been sent* **then**
    send $(NextVote, period, v)$;
  else
    send $(NextVote, period, prevNextVote)$;
end
upon *receiving* $(NextVote, p, v)$ *from* $2F_{shard}+1$ *distinct senders* **do**
  if $p \geq period$ **then**
    $prevNextVote \leftarrow v$;
    increase $timer$ timeout;
    $period \leftarrow p+1$;
    $start\_period()$;
end

4.4 Security Analysis

In this section we first prove the security of the intershard agreement protocol, by showing safety and liveness property. Then we prove security of the sharding mechanism, by showing how to compute StakeCube's parameters to ensure that no shard can be corrupted.

4.4.1 Intershard agreement

Safety

Theorem 10 (Intershard agreement Safety)**.** *No two honest nodes ever decide on different blocks.*

Proof. First, we should note that honest nodes may send a $CertVote$ or $SoftVote$ only once per period, and they may send up to two $NextVote$ per period but with one of them being $v \neq \perp$. Furthermore, because a message needs to be sent by $2f+1$ nodes from the same shard S to be sent by S , the previous remark also applies to shards instead of nodes.

Assume two honest nodes n_1 and n_2 output blocks b_1 and b_2 at periods p_1 and p_2, respectively. W.l.o.g., assume $p_1 \leq p_2$.

First case, if $p_1 = p_2$ Safety is proven through a classical quorum argument: Among the $2F_{\text{shard}} + 1$ *CertVote* received by n_1 and n_2, ar least $F_{\text{shard}} + 1$ of them comes from the same shards. Hence at least one of them is from the same honest shard that sent the same block $b = b_1 = b_2$.

Other cases, if $p_1 < p_2$ First, note that because an honest node received $2F_{\text{shard}} + 1$ *CertVote* for $H(b_1)$, there are at least $F_{\text{shard}} + 1$ honest shards that may only *NextVote* once with value $H(b_1)$. Hence any value $v \neq H(b_1)$ may have at most $2F_{\text{shard}}$ *NextVote* and only $H(b_1)$ may have $2F_{\text{shard}} + 1$ *NextVote* at period p_1.

Therefore, for all honest nodes at period $p_1 + 1$, $prevNextVote = H(b_1)$. For any honest node n, if $prevNextVote \neq \perp$, then n may only $SoftVote$ or *NextVote* their *prevNextVote* value. Hence, only $H(b_1)$ may have $2F_{\text{shard}} + 1$ $SoftVote$, *CertVote* or *NextVote* at period $p_1 + 1$.

Finally, we have that only b_1 may be output at period $p_1 + 1$, and that for all honest nodes at period $p_1 + 2$, $prevNextVote = H(b_1)$. By induction on $p' > p_1$, only b_1 may be output on subsequent periods.

□

Liveness

Theorem 11 (Intershard agreement Liveness)**.** *Eventually, any new block gets accepted*

Proof. We have shown that once a value $v \neq \perp$ receives $2F_{\text{shard}} + 1$ *NextVote* at period p, it will continue to do so for all periods $p' > p$.

For a value $v \neq \perp$ to receive $2F_{\text{shard}} + 1$ *NextVote* at period p, it is sufficient that:

1. All honest nodes are at period p
2. $leader(p)$ is honest
3. They all receive the leader proposal and each other's $SoftVote$ before their timer expire.

As a preliminary remark, we can see that for any period $p' \neq 0$ reached by an honest node, all honest nodes will eventually receive $2F_{\text{shard}} + 1$ $NextVote$ from period $p' - 1$. Moreover, for every period p' where *no* block is output, at least one honest node will reach p'.

Assume that no block has been output yet. We can deduce that for the first item to hold, it is sufficient that all honest nodes receive $2F_{\text{shard}} + 1$ $NextVote$ from period $p - 1$ before their timer expires. Hence, assuming that $leader(p)$ is honest, if the honest nodes' timer duration is greater that the time it takes for these messages to be received, the first and third items will hold. Because the second item holds infinitely often, and due to the eventual synchrony assumption, we can deduce that, eventually, all items will hold.

We now have that there is a period p_s and a value v such that for all period $p' \geq p_s$, all honest nodes have $prevNextVote = v$. For a block to be output a period p', it is sufficient that :

1. All honest nodes are at period p'
2. They all receive each other's $SoftVote$ and $CertVote$ before their timer expire.

Using the eventual synchrony assumption similarly to the previous paragraph, we can show that these conditions will eventually hold, and that a honest node will output a block. □

Verifiability If there are $2F_{\text{shard}} + 1$ $CertVote$ for the value v at the same period, then only a block b s.t. $H(b) = v$ may be output. Therefore the set of $CertVote$ constitutes a certificate for the block b which can convince any node of the protocol output.

This can be used to certify a block for an external party knowing the protocol participants, but we also use it as a mean to guarantee totality: Once a node outputs a block b, broadcasting its certificate will ensure that all honest nodes will eventually output b.

4.4.2 Security of the sharding mechanism

We analyze the probability that some of the shards of StakeCube are corrupted and show that their core set contains more than $\alpha_{\text{core}} s_{\text{min}}$ malicious users, where α_{core} is the proportion of tolerated byzantine nodes in the shards cores, *i.e.*, $\alpha_{\text{core}} = 1/3$ for our instantiation. To conduct such an analysis, we examine a simplified scenario. We approximate the behaviour of StakeCube by taking the amortized execution over one epoch of T blocks. That

is, we study the corruption probability when all the shards are built and the cores are elected over one period. This is equivalent to the scenario in which all credentials are synchronously renewed at the same block. Note that, for a fixed number of active users, the number of credential renewals, core election, and topological changes is statistically the same for every period of length T. In this section, we prove the following.

Theorem 12 (Stakecube's sharding is secure)**.** *In the scenario sketched above, for every shard S in StakeCube, the proportion of corrupt nodes within its core set $\mathcal{S}_c$ is strictly less than $\alpha_{core} s_{\min}$, except with probability negligible in λ.*

Corruption Probability of a Core Set

First, let us analyse the probability of malicious nodes getting from the spare set to the core set. Let s be the size of a shard $\mathcal{S}$, α_{shard} be a bound on the ratio of malicious users within $\mathcal{S}$. We assume that $0 \leq \alpha_{\text{shard}} < \alpha_{\text{core}} \leq \alpha$. We compute an upper bound on the probability that the fraction of malicious users in the core set is higher than α_{core} by the end of the period. As described in Section 4.3.2, the core set is elected by randomly taking $s_{\min}$ credentials from shard $\mathcal{S}$, without replacement. Let Y be the random variable equal to the number of malicious credentials within the core, *i.e.*, Y follows an hypergeometric distribution whose probability measure is given by

$$\forall k \in [\![0, s_{\min}]\!], \mathbb{P}[Y = k] = \binom{\lfloor s\alpha_{\text{shard}} \rfloor}{k} \binom{\lfloor s(1-\alpha_{\text{shard}}) \rfloor}{s_{\min} - k} \binom{s}{s_{\min}}^{-1} \tag{4.2}$$

Applying the Hoeffding bound [Hoe94] on relation (4.2) leads to the following bound

$$\mathbb{P}[Y/s_{\min} \geq \alpha_{\text{core}}] \leq e^{-2(\alpha_{\text{core}} - \alpha_{\text{shard}})^2 s_{\min}}$$

Thus, assuming that the fraction of malicious users in a shard is below α_{shard}[2], we can conclude that

$$\lambda \geq -2(\alpha_{\text{core}} - \alpha_{\text{shard}})^2 s_{\min} \Rightarrow \mathbb{P}[Y/s_{\min} \geq \alpha_{\text{core}}] = \text{negl}(\lambda) \tag{4.3}$$

Where the right-hand side of the implication is exactly the statement of 12 for one shard $\mathcal{S}$.

[2]This assumption is treated in the next subsection.

Distribution of Malicious Credentials among all Shards

The above section assumes that the fraction of malicious users in all the shards is below α_{shard}. In this section we compute an upper bound on the probability that this assumption does not hold for one of the shards. We make simplification assumptions on how the shards are formed. First, we assume that there are K shards of size S, giving rise to *i.e.* $n = SK$ credentials in total. Second, we assume that the shards configuration in StakeCube during the concerned period results from a random credential assignment to all the shards. Recall that α_{cred} is the overall ratio of malicious credentials. Let X_i be the random variable representing the number of malicious credentials in the i-th shard, with $1 < i < K$. And finally, we note $\mathbf{X} = (X_1, \ldots, X_k) \in \{0, S\}^K$ be the vector made of these K random variables. Random variable $\mathbf{X}$ represents the distribution of malicious credentials in StakeCube. It follows a multivariate hypergeometric distribution, *i.e.*, each of the $n = SK$ credentials is assigned to a shard. We analyse the shard assignment of a random sample of size $n\alpha_{\text{cred}}$. Let I be the set of vectors representing StakeCube when $n\alpha_{\text{cred}}$ credentials are malicious. We have

$$\mathrm{I} = \{\mathbf{x} \in [0, S]^K \mid \sum_{i=1}^{K} x_i = n\alpha_{\text{cred}}\}$$

and

$$\forall \mathbf{x} \in \mathrm{I}, \mathbb{P}[\mathbf{X} = \mathbf{x}] = \binom{n}{n\alpha_{\text{cred}}}^{-1} \prod_{i=1}^{K} \binom{S}{x_i}$$

We are interested in computing the probability that a given shard j among the K ones contains more than m malicious credentials, that is, let $\mathrm{I}_{m,j}$ be defined as follows :

$$\mathrm{I}_{m,j} = \{\mathbf{x} \in \mathrm{I} \mid x_j \geq m\}$$

We have:

$$\begin{aligned}
\mathbb{P}[]X_j \geq m] &= \mathbb{P}[\mathbf{X} \in \mathrm{I}_{m,j}] \\
&= \sum_{\mathbf{x} \in \mathrm{I}_{m,j}} \binom{n}{n\alpha_{\text{cred}}}^{-1} \prod_{i=1}^{K} \binom{S}{x_i} \\
&= \sum_{k=m}^{S} \binom{S}{k} \binom{n}{n\alpha_{\text{cred}}}^{-1} \sum_{\substack{x_1,\ldots,x_{K-1} \in [0,S] \\ \sum_{1 \leq i \leq K-1} x_i = n\alpha_{\text{cred}} - k}} \prod_{1 \leq i \leq K, i \neq j} \binom{S}{x_i}
\end{aligned}$$

Knowing that $\sum_{1 \leq i \leq K-1} x_i = n\alpha_{\text{cred}} - k$ and $\sum_{1 \leq i \leq K-1} S = n - S$, we can apply Vandermonde's identity:

$$\forall j, \mathbb{P}[\mathbf{X} \in \mathrm{I}_{m,j}] = \sum_{k=m}^{S} \binom{n}{n\alpha_{\text{cred}}}^{-1} \binom{S}{k} \binom{n-S}{n\alpha_{\text{cred}} - k}$$

We now get our result by applying first the (univariate) Hoeffding bound, and then the union bound.

$$\forall j, \mathbb{P}[\mathbf{X} \in \mathrm{I}_{S\alpha_{\text{shard}},j}] \leq e^{-2(\alpha_{\text{shard}} - \alpha_{\text{cred}})^2 S}$$

Thus the probability that at least one shard of the system contains more than $\alpha_{\text{shard}} S$ malicious credentials is bounded by

$$\mathbb{P}[\mathbf{X} \in \cup_{j=1}^{K} \mathrm{I}_{S\alpha_{\text{shard}},j}] \leq K e^{-2(\alpha_{\text{shard}} - \alpha_{\text{cred}})^2 S}$$

Moreover, due to the union bound, this upper bound also holds if the shards have different sizes and S is the minimum, hence, we can simply use $S := s_{\min}$. As for K, the worst case is reached when there is a maximal number of shards, *i.e.* $K := n/s_{\min}$. Therefore we can conclude that

$$\lambda \geq 2(\alpha_{\text{shard}} - \alpha_{\text{cred}})^2 s_{\min} - \ln \frac{n}{s_{\min}} \Rightarrow \mathbb{P}[\mathbf{X} \in \cup_{j=1}^{K} \mathrm{I}_{S\alpha_{\text{shard}},j}] = \text{negl}(\lambda) \tag{4.4}$$

Where the term $\cup_{j=1}^{K} \mathrm{I}_{S\alpha_{\text{shard}},j}$ is the set of shards assignations to malicious credentials, such that at least one shard has a fraction greater than or equal to α_{shard} of malicious credentials. *i.e.*, equation 4.4 bounds the probability that the assumption required for equation 4.3 does not hold.

Putting it all Together

In the previous subsection we got exponentially decreasing bounds on the probability that at least one shard is corrupted, *i.e.*, proving security when the bound on the number of malicious shards F_{shard} is set to 0. This obviously imply that StakeCube is still secure for other values of F_{shard}.

The adversary has a fraction α of stake. Requiring each credential to be associated to at most M stake gives us the following (worst case) ratio of malicious credentials, which is reached when each malicious UTXOs has 1 stake and each honest one maximizes its stake, *i.e.*, has M stake. We then have:

$$\alpha_{\text{cred}} = \frac{1}{1 + M^{-1}(\alpha^{-1} - 1)} \tag{4.5}$$

Thus M should be as small as possible to decrease the adversary effective stake. However low values of M may require users to participate with a large number of credentials in parallel, increasing the communication cost for individual users.

By combining equations 4.3, 4.4 and 4.5, we obtain the following inequalities that imply theorem 12. Given λ, α and M, they can be solved to obtain α_{shard} and s_{min}.

$$\alpha_{\text{shard}} \leq \frac{1}{1 + M^{-1}(\alpha^{-1} - 1)} + \sqrt{\frac{\lambda - \ln \frac{n}{s_{\text{min}}}}{2 s_{\text{min}}}} \quad \text{and} \quad s_{\text{min}} \geq \frac{\lambda}{2(\alpha_{\text{core}} - \alpha_{\text{shard}})^2} \tag{4.6}$$

4.5 Evaluation and Application

We implemented a prototype version of StakeCube to test its practical performance and verify its behaviour within large-scale applications.

We experimentally measure communication cost as well as transaction confirmation time and throughput under varying network size, security parameters and load. To the best of our knowledge, StakeCube is the first blockchain reaching scalability levels well in the thousands of nodes.

Finally, we demonstrate StakeCube's viability with a large scale IoT application: an energy marketplace [HKMS17a]. In this application energy producers and consumers

sell directly to each other through a blockchain. We implemented this application on StakeCube, and executed it with a large number of nodes, with one Raspberry Pi Zero among them. We measured its resources usage and found it to be largely within capacity. All data related to the experiments available on Github[3].

Modifications To make the implementation more tractable, we had to make a simplification of StakeCube. Specifically, our prototype does not support the join operation. This means that we assume that participating nodes can only leave or join the network by emitting transactions that spend or give them stake, respectively. All honest users that own stake are required to participate in the algorithm and cannot be offline.

As a result, the original shard update phase of StakeCube can be completely removed and the computation of the shard membership can be carried out from the knowledge of the last block. Although this simplification is indeed a major change in the algorithm, we argue that it does not significantly affect the focus of this experiment, which is the evaluation of performance as a function of the total number of users.

4.5.1 Experiments

The implementation contains approximately $10,000$ lines of C++. It runs as a docker image that can be scaled and run on multiple concurrent host machines. Each docker container is allowed only one computing core. We run the experiments across five virtual machines, each with 30GiB of RAM, 6 virtual CPU, and 30 GiB of storage. A trusted setup phase consists for all the nodes in exchanging key material, shared randomness and connection information. Once initialized, the core of the protocol is triggered and all the nodes output their metrics until each node has locally a chain of blocks of a pre-determined size.

Figure 4.2 gives an overview of the software architecture of StakeCube. We have two applicative executables, one for the generic evaluation, and one for the energy marketplace. The latter is configured with a "marketplace" smartcontract protocol.

Parametrization We introduce the notations $f := \alpha_{\text{core}} s_{\min}$ and $F := F_{\text{shard}}$. In StakeCube all nodes store the same instance of the sharding structure provided by PeerCube.

[3] `https://github.com/Maschmalow/StakeCube_experiment_data`

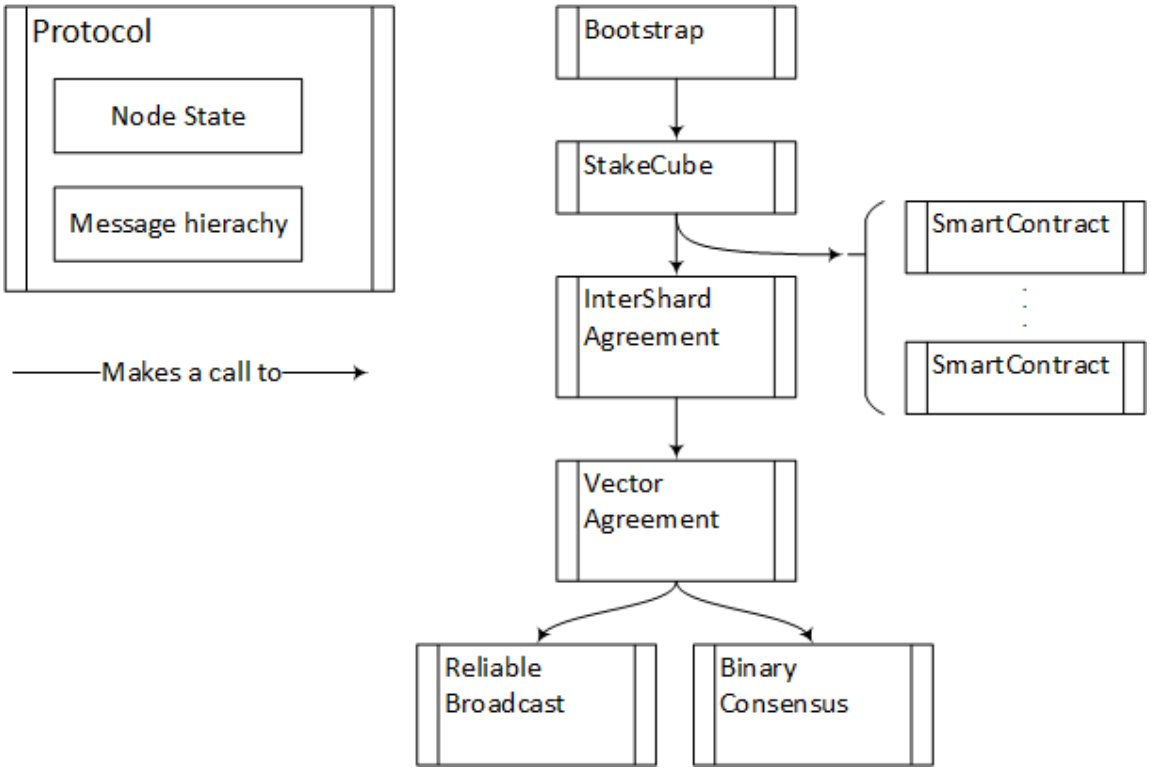

FIGURE 4.2: StakeCube software architecture. The "Protocol" box is a common state machine abstraction implemented by all protocols, where messages are processed to update the state. The bootstrap protocol is responsible for the genesis initialisation (parameters, public keys, stake, network addresses). The smartcontracts are in a static array of protocols used for validating transactions.

This structure represents the shard membership (core and spare) of the whole network. Hence, we can use it to simulate the corruption in StakeCube. To do so, we generate N credentials with αN of them marked as corrupted, compute the shard membership of nodes and then count the number of corrupted shards. For a given α and f, this gives us a simple approximation of the probability distribution of the number of corrupt shards. That is, we can estimate the appropriate F to ensure correctness for varying α and f. Using this method we computed the following security parameters: $f = 5, F = 5, \alpha = 0.1$; $f = 5, F = 10, \alpha = 0.15$; $f = 7, F = 7, \alpha = 0.15$. The full dataset is available in the repository.

Limitations Because the implementation is an early stage prototype, it underperforms in several aspects. Most notably, the networking code had to be kept simple and naive. For instance, each protocol message is sent in a dedicated TCP connexion, which creates a large overhead. We also did not implement a mechanism for nodes to synchronize and update their states when they communicate. For instance, this means that nodes will keep exchanging outdated blocks even when it is not necessary.

In that respect, we expect that a fully optimised implementation to exhibit a large performance gain, and our experimentation can only be a proof of viability. But despite these issues, we were able to run experiments with up to five thousand nodes, and to keep reasonable transaction output even above a thousand nodes.

Metrics For each block, we output the following metrics, aggregated over nodes:

1. The median of the block interval time.
2. The number of bytes sent.

For a given run, the values are averaged over blocks and plotted for varying parameters, namely:

1. Number of nodes. This parameter is set as the 'x' axis to allows us to verify StakeCube's scalability.
2. Block size. The experiments with a small number of transactions per block indicate StakeCube minimal costs. The one with higher number of transaction has been

chosen to optimise transaction throughput, with the intention to be representative of operations under maximum load.

3. Security parameters, as chosen in Section 4.5.1 above.

The error bars represent the standard deviation of the metric for the run. Lines in Figure 4.3 are linear regressions, with the coefficient of determination R^2 in the legend. Note that because the (maximum) number of transactions in a block is fixed, the transaction throughput can be obtained from the block interval time.

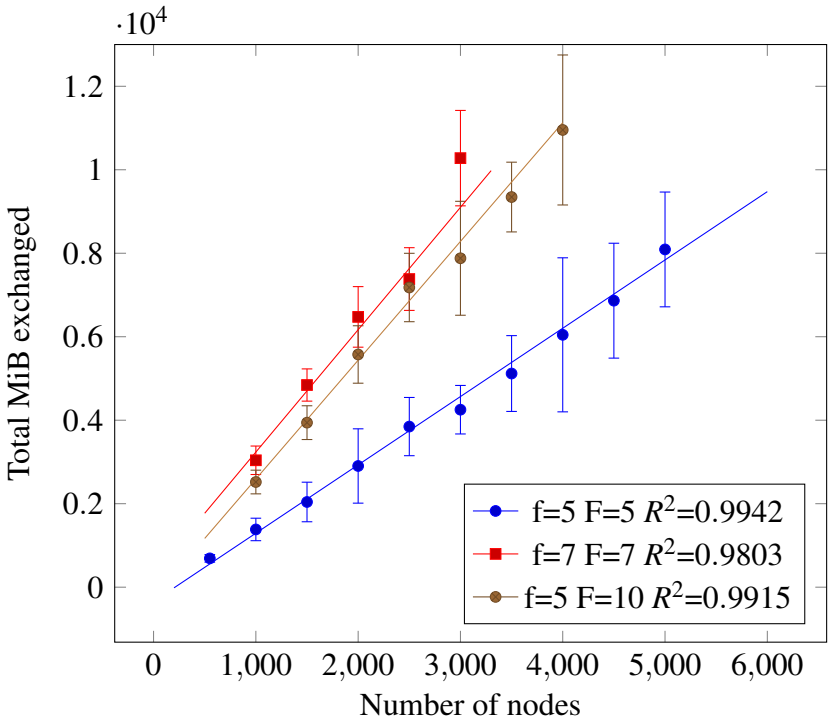

FIGURE 4.3: Communication cost as a function of the number of nodes when the number of transactions per block is 10.

In Figure 4.3 we show the number of bytes exchanged in the network, *i.e.* the communication cost. We can see that the scalability property of StakeCube clearly holds, as

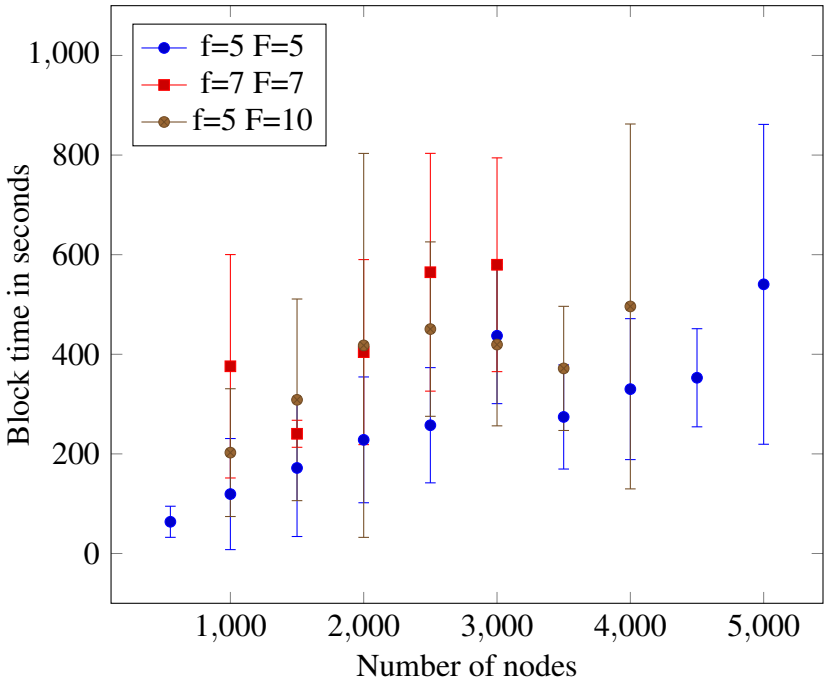

FIGURE 4.4: Block time as a function of the number of nodes when the number of transactions per block is 10

shown by the linear regressions with $R^2 \geq 0.98$. Thus we can conclude that StakeCube is able to tackle the scalability issue of PoS blockchains.

Figure 4.4 shows the block interval time as a function of the number of nodes in the system. Due to the high variance, we cannot conclude that we achieved a constant number of communication rounds per block. Furthermore, even without variance, the block interval time could increase with the number of nodes solely because the network is slowing down. However the fact that experimental variations have much more influence than the number of nodes is a piece of evidence that our algorithm performs correctly.

We observed during the experiments that the variations were mostly due to sporadic communication failures between nodes, which happens when the network and the operating system become overloaded. Unfortunately, we were not capable to address the real causes of such experimental variations.

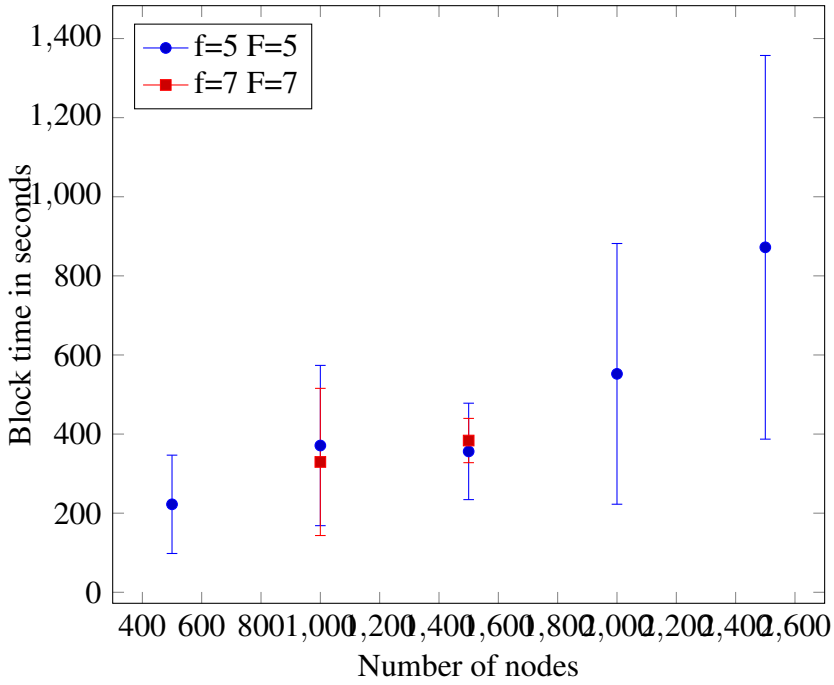

FIGURE 4.5: Block time as a function of the number of nodes when the number of transactions per block is 2,000

In Figure 4.5 we show again the block interval time but with blocks containing 2,000 transactions of 128 bytes each. This Figure shows a transaction output between 1,100 and 300 bytes/s. We did not show the communication cost for 2,000 Tx/block because it is very similar to a scaled version of Figure 4.3 and does not give any additional information.

4.5.2 Application

To demonstrate StakeCube's viability in large-scale environment, we used it to implement an IoT application. We ran an energy marketplace backed by StakeCube on a Raspberry Pi Zero, and confirm that it is able to handle the load even over large networks. The energy marketplace is a blockchain-based answer for the demand of decentralized solutions

for the energy infrastructure [HKMS17a]. In this application, energy producers and consumers such as households and electric vehicles are trading electricity on an local energy marketplace. Because participants may be numerous and are typically running in a small system-on-chip, we see this application as particularly fitting for StakeCube.

Indeed, we implemented an energy marketplace pseudo smart contract where each node can sell and buy an electricity token and the orders are matched using a double auction algorithm [BAD+18]. Although StakeCube does not support smart contracts yet, we made the prototype modular enough to easily add any type of transaction with its own validation function. As a result the code is similar to what would be in an actual smart contract, the difference being that it is statically linked with the StakeCube program instead of loaded through a transaction.

We ran this application in a network of 500 nodes including one Raspberry Pi Zero, which has 512MiB of RAM and a 1GHz single-core CPU. During execution, we monitored the usage metrics on the Raspberry using the command "vmstat −t 1"; we saw that the StakeCube executable used at most 15MiB of RAM and that except for short spikes (at most two seconds) when receiving blocks, the CPU was mostly idling. The full monitoring trace is available in the repository.

As a further step, Atos assisted us in developing a user interface for this application. A snippet of this interface is shown in Figure 4.6; it displays the list trade orders made on the blockchain as well as how these orders were matched by the auction algorithm.

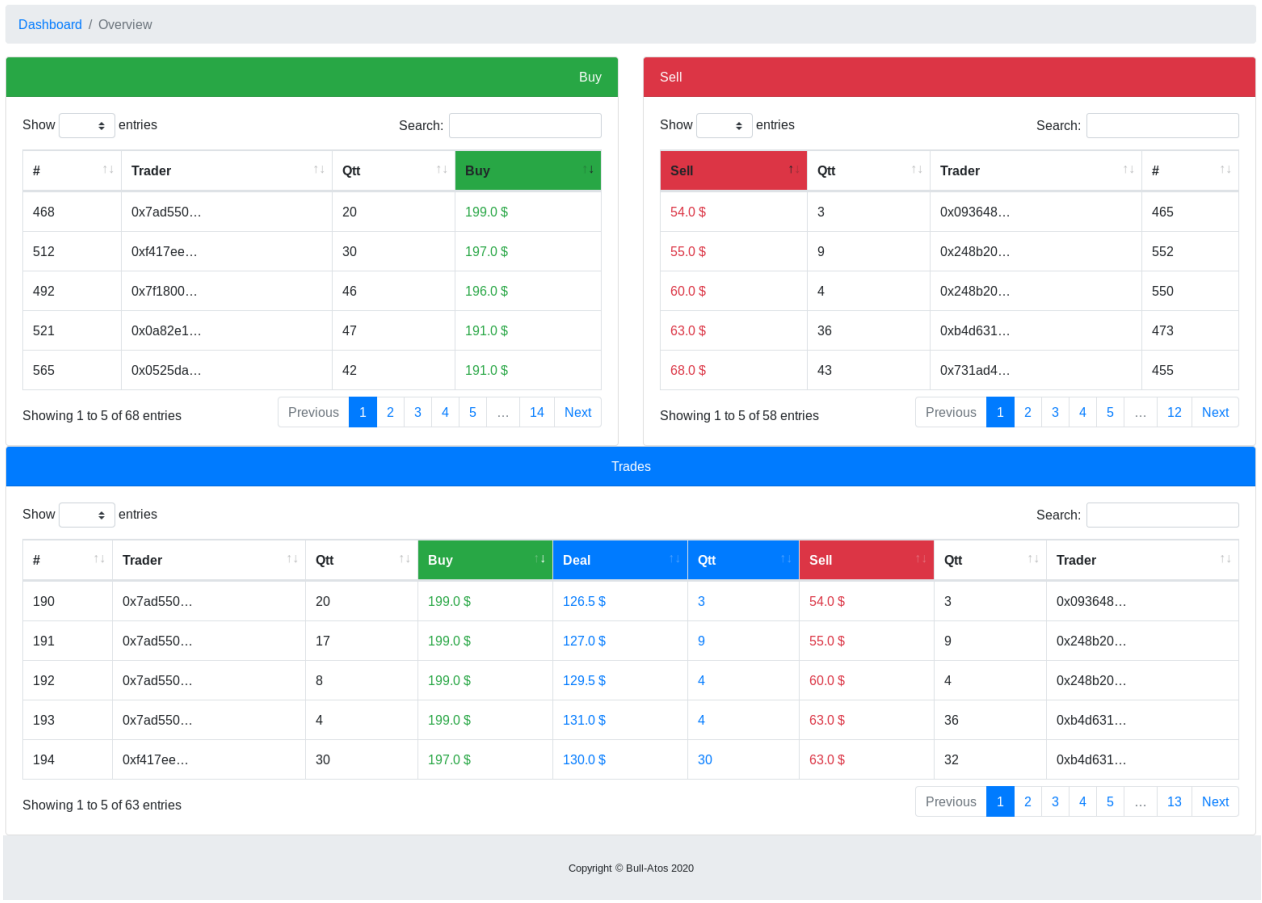

FIGURE 4.6: Energy marketplace Dashboard. Top Tables are the lists of Asks and Bids sent by the traders. Bottom table is the output of the auction algorithm, *i.e.* matching bids and asks.

Chapter 5

Conclusion and Future work

This thesis was initially motivated by the need to understand and solve the issues that hinge the use of blockchain protocols for IoT applications, but our contributions have become much more foundational. At first, we first explored several technologies and undertook to derive a framework to be able to compare them. This framework has been gradually developed until becoming mature enough to tackle the issue of making a taxonomy of the blockchain landscape. In the quest to benefit the most from the blockchain paradigm, we have introduced StakeCube, a new protocol that focuses on the key property of blockchains : scalability. In that sense, we finally answered to the initial motivation of this thesis, with an application to IoT architectures.

As a first foundational step, we have reformulated in a common framework an important part of the existing models for the execution of distributed algorithms, while accounting for the novelties introduced by Proof-of-Work algorithms such as Bitcoin. This framework is a stepping stone towards a comprehensive and consistent formalisation of the model space for Byzantine protocols. We have shown how to understand what is possible and what is not in this generalized model, by re-stating impossibility proofs and problems reductions. Then, we have made a fair comparison of the model and performance metrics of existing blockchain. Our second contribution is the design of StakeCube, a new Proof-of-Stake blockchain that leverage sharding to scale with the number of nodes. StakeCube is based on a Byzantine tolerant distributed hash table, PeerCube [ALRB08], which is taken advantage of to distribute the communication load to multiple shards. We have verified experimentally StakeCube's scalability and its viability for large scale IoT applications.

5.1 Future Work

On the framework

Extensions There are two main aspects that must be captured by our formalism before it could be considered sufficiently complete. First, the ability to modelize dynamic participation and *in fine.* reconfiguration. This is a delicate operation because this means that the set Π now changes with time and is in fact execution dependent. This brings circular dependencies in our model and all the definitions that depends on *one* canonical set of nodes must be rethought.

Secondly, the framework as we laid out in this thesis requires an omniscient adversary which is fed all the input/output between the nodes and modules. Therefore our model could be further generalized by letting modules hide some information from the adversary.

But more importantly, it would be greatly beneficial to replace the basic execution model we gave in Section 3.1 with one of the existing state-of-the-art framework [Can01, CCK$^+$08a, Mau11]. Besides extending the supported models, this would also helps towards the goal of having standardized, reusable, definitions for distributed components.

Exploring the BFT landscape There are a few questions that are prompted by the reductions and impossibilities in Figure 3.2. Completing this picture with other existing problems is a first, straightforward improvement. Taking it one step further, there is an interesting question in finding new advantageous definitions, *e.g.* a weakened atomic broadcast unconditionally equivalent to external consensus.

Surprisingly, we observe that although we have many reductions between different problems, there are few impossibility proof for reductions, *i.e.*, the *non-existence* of reductions. Consequently, it is an open question to re-draw our reduction graph from Figure 3.1 but with only strict relations.

The current picture suggests that weak, external and vector consensus are all solvable exactly when reliable broadcast is, which would be quite surprising. To the extent of our knowledge, there is no complete characterization of the solveability of these problems in the BFT case.

Formal verification Furthermore, we believe it would greatly benefit to the distributed systems community to have our model formally verified using a proof assistant such as Coq or Isabelle/HOL. Such formalisation would readily yield a software framework to implement protocols along with their correctness proof. In fact, distributed systems and blockchain protocols are known to be difficult to implement securely, and formal methods have already been identified as an effective mean to tackle this issue.

On StakeCube

Extending the threat model We plan to extend the fault model to include rational players, wich would open the possibility for make StakeCueb incentive-compatible. We leave to future work to prove StakeCube's security with collision resistant hash functions instead of the random oracle, as well as the investigation of the input pool mechanisms.

Stake-weighted election We plan to improve StakeCube by taking into account the stake associated with each credential as weights into both the core election and the election of the shard in charge of creating the next block. This would remove the adversary gain in power due to UTXOs with different stake being counted the same, thus leading to an important gain in security.

Sharding transactions It is also an open question to leverage StakeCube's sharding structure to improve on blocks storage and processing costs. One possible approach would be to send blocks contents only to a few shard, and then to let the shard committee only agree on *which* shard's input constitute the final block.

Optimisation An interesting path to explore would be to optimize StakeCube in the case where we set the parameter f_{shard} to 0, which is to say that shards are sufficiently big so that none get ever corrupted. This would remove the need for the inter shard agreement and make each shard only need to know the existence of their neighbour. The previous improvement on block storage and processing would be trivial, as each shard could be the only responsible for their assigned transactions. In effect, this would result in what could be described as a network of interconnected and independent systems. Although the shard

size and total number of shards needed to make this construction worth the cost make this question more of theoretical interest.

References

[AAC+05] Amitanand S. Aiyer, Lorenzo Alvisi, Allen Clement, Michael Dahlin, Jean-Philippe Martin, and Carl Porth. BAR fault tolerance for cooperative services. In Andrew Herbert and Kenneth P. Birman, editors, *Proceedings of the 20th ACM Symposium on Operating Systems Principles 2005, SOSP 2005, Brighton, UK, October 23-26, 2005*, pages 45–58. ACM, 2005.

[ABB+18] Elli Androulaki, Artem Barger, Vita Bortnikov, Christian Cachin, Konstantinos Christidis, Angelo De Caro, David Enyeart, Christopher Ferris, Gennady Laventman, Yacov Manevich, Srinivasan Muralidharan, Chet Murthy, Binh Nguyen, Manish Sethi, Gari Singh, Keith Smith, Alessandro Sorniotti, Chrysoula Stathakopoulou, Marko Vukolic, Sharon Weed Cocco, and Jason Yellick. Hyperledger fabric: a distributed operating system for permissioned blockchains. In *Proceedings of the Thirteenth EuroSys Conference, EuroSys 2018, Porto, Portugal, April 23-26, 2018*, pages 30:1–30:15, 2018.

[ABFG14] Giuseppe Ateniese, Ilario Bonacina, Antonio Faonio, and Nicola Galesi. Proofs of space: When space is of the essence. In *Security and Cryptography for Networks - 9th International Conference, SCN 2014, Amalfi, Italy, September 3-5, 2014. Proceedings*, pages 538–557, 2014.

[Abr19] Ittai Abraham. The flp impossibility, asynchronous consensus lower bound via uncommitted configurations, 2019. Online, accessed April 2021.

[AGMS20] Vittorio Astarita, Vincenzo Pasquale Giofrè, Giovanni Mirabelli, and Vittorio Solina. A review of blockchain-based systems in transportation. *Inf.*, 11(1):21, 2020.

[AKGN18] Antonio Fernández Anta, Kishori M. Konwar, Chryssis Georgiou, and Nicolas C. Nicolaou. Formalizing and implementing distributed ledger objects. *SIGACT News*, 49(2):58–76, 2018.

[AL07] Yonatan Aumann and Yehuda Lindell. Security against covert adversaries: Efficient protocols for realistic adversaries. In *Theory of Cryptography, 4th Theory of Cryptography Conference, TCC 2007, Amsterdam, The Netherlands, February 21-24, 2007, Proceedings*, pages 137–156, 2007.

[ALRB08] Emmanuelle Anceaume, Romaric Ludinard, Aina Ravoaja, and Francisco Brasileiro. PeerCube: A Hypercube-Based P2P Overlay Robust against Collusion and Churn. In *IEEE International Conference on Self-Adaptive and Self-Organizing Systems (SASO)*, 2008.

[ALS12] Emmanuelle Anceaume, Romaric Ludinard, and Bruno Sericola. Performance evaluation of large-scale dynamic systems. *ACM SIGMETRICS Performance Evaluation Review*, 39(4), 2012.

[AM17] Ittai Abraham and Dahlia Malkhi. The blockchain consensus layer and BFT. *Bulletin of the European Association for Theoretical Computer Science*, 123, 2017.

[Amo20] Yackolley Amoussou-Guenou. *Governing the commons in blockchains. (Gouvernance des biens communs dans les blockchains)*. PhD thesis, Sorbonne University, France, 2020.

[APB94] James H. Anderson, David Peleg, and Elizabeth Borowsky, editors. *Proceedings of the Thirteenth Annual ACM Symposium on Principles of Distributed Computing, Los Angeles, California, USA, August 14-17, 1994*. ACM, 1994.

[APL+19a] Emmanuelle Anceaume, Antonella Del Pozzo, Romaric Ludinard, Maria Potop-Butucaru, and Sara Tucci Piergiovanni. Blockchain abstract data type. In *The 31st ACM on Symposium on Parallelism in Algorithms and Architectures, SPAA 2019, Phoenix, AZ, USA, June 22-24, 2019*, pages 349–358, 2019.

[APPT18] Yackolley Amoussou-Guenou, Antonella Del Pozzo, Maria Potop-Butucaru, and Sara Tucci Piergiovanni. Correctness of tendermint-core blockchains. In Cao et al. [CERF18], pages 16:1–16:16.

[APPT19] Yackolley Amoussou-Guenou, Antonella Del Pozzo, Maria Potop-Butucaru, and Sara Tucci Piergiovanni. Dissecting tendermint. In Atig and Schwarzmann [AS19], pages 166–182.

[AS07] Baruch Awerbuch and Christian Scheideler. Towards scalable and robust overlay networks. In *6th International workshop on Peer-To-Peer Systems, IPTPS 2007, Bellevue, WA, USA, February 26-27, 2007*, 2007.

[AS19] Mohamed Faouzi Atig and Alexander A. Schwarzmann, editors. *Networked Systems - 7th International Conference, NETYS 2019, Marrakech, Morocco, June 19-21, 2019, Revised Selected Papers*, volume 11704 of *Lecture Notes in Computer Science*. Springer, 2019.

[ASLT11] Emmanuelle Anceaume, Bruno Sericola, Romaric Ludinard, and Frederic Tronel. Modeling and evaluating targeted attacks in large scale dynamic systems. In *Proceedings of the 2011 IEEE/IFIP International Conference on Dependable Systems and Networks, DSN 2011, Hong Kong, China, June 27-30 2011*, pages 347–358, 2011.

[BAD+18] Kei-Leo Brousmiche, Andra Anoaica, Omar Dib, Tesnim Abdellatif, and Gilles Deleuze. Blockchain energy market place evaluation: An agent-based approach. In *2018 IEEE 9th Annual Information Technology, Electronics and Mobile Communication Conference (IEMCON)*, pages 321–327, 2018.

[BBG+17] Aymen Boudguiga, Nabil Bouzerna, Louis Granboulan, Alexis Olivereau, Flavien Quesnel, Anthony Roger, and Renaud Sirdey. Towards better availability and accountability for iot updates by means of a blockchain. In *2017 IEEE European Symposium on Security and Privacy Workshops, EuroS&P Workshops 2017, Paris, France, April 26-28, 2017*, pages 50–58. IEEE, 2017.

[BCG20] Elette Boyle, Ran Cohen, and Aarushi Goel. Breaking the o($\sqrt{n}$)-bit barrier: Byzantine agreement with polylog bits per-party. *IACR Cryptol. ePrint Arch.*, 2020:130, 2020.

[BDH+18] Kei-Léo Brousmiche, Antoine Durand, Thomas Heno, Christian Poulain, Antoine Dalmieres, and Elyes Ben Hamida. Hybrid cryptographic protocol for secure vehicle data sharing over a consortium blockchain. In *IEEE International Conference on Internet of Things (iThings) and IEEE Green Computing and Communications (GreenCom) and IEEE Cyber, Physical and Social Computing (CPSCom) and IEEE Smart Data (SmartData), iThings/GreenCom/CPSCom/SmartData 2018, Halifax, NS, Canada, July 30 - August 3, 2018*, pages 1281–1286, 2018.

[BGK+18] Christian Badertscher, Peter Gaži, Aggelos Kiayias, Alexander Russell, and Vassilis Zikas. Ouroboros genesis: Composable proof-of-stake blockchains with dynamic availability. In *ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2018.

[BGKR18] D. Bernardo, Peter Gaži, Aggelos Kiayias, and Alexander Russell. Ouroboros praos: An adaptively-secure, semi-synchronous proof-of-stake blockchain. In *International Conference on the Theory and Applications of Cryptographic (EUROCRYPT)*, 2018.

[BGM+18] Christian Badertscher, Juan A. Garay, Ueli Maurer, Daniel Tschudi, and Vassilis Zikas. But why does it work? A rational protocol design treatment of bitcoin. In *Advances in Cryptology - EUROCRYPT 2018 - 37th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Tel Aviv, Israel, April 29 - May 3, 2018 Proceedings, Part II*, pages 34–65, 2018.

[BHRT00] Roberto Baldoni, Jean-Michel Hélary, Michel Raynal, and Lénaick Tanguy. Consensus in byzantine asynchronous systems. In Michele Flammini, Enrico Nardelli, Guido Proietti, and Paul G. Spirakis, editors, *SIROCCO*

7, Proceedings of the 7th International Colloquium on Structural Information and Communication Complexity, Laquila, Italy, June 20-22, 2000, pages 1–15. Carleton Scientific, 2000.

[BKR94] Michael Ben-Or, Boaz Kelmer, and Tal Rabin. Asynchronous secure computations with optimal resilience (extended abstract). In Anderson et al. [APB94], pages 183–192.

[BPW07] Michael Backes, Birgit Pfitzmann, and Michael Waidner. The reactive simulatability (RSIM) framework for asynchronous systems. *Inf. Comput.*, 205(12):1685–1720, 2007.

[Buc16] Ethan Buchman. Tendermint: Byzantine fault tolerance in the age of blockchains. 2016.

[Can01] Ran Canetti. Universally composable security: A new paradigm for cryptographic protocols. In *Foundations of Computer Science, 2001. Proceedings. 42nd IEEE Symposium on*, pages 136–145. IEEE, 2001.

[Can04] Ran Canetti. Universally composable signature, certification, and authentication. In *17th IEEE Computer Security Foundations Workshop, (CSFW-17 2004), 28-30 June 2004, Pacific Grove, CA, USA*, page 219. IEEE Computer Society, 2004.

[CASD95] Flaviu Cristian, Houtan Aghili, H. Raymond Strong, and Danny Dolev. Atomic broadcast: From simple message diffusion to byzantine agreement. *Inf. Comput.*, 118(1):158–179, 1995.

[CCK+08a] Ran Canetti, Ling Cheung, Dilsun Kirli Kaynar, Moses D. Liskov, Nancy A. Lynch, Olivier Pereira, and Roberto Segala. Analyzing security protocols using time-bounded task-pioas. *Discret. Event Dyn. Syst.*, 18(1):111–159, 2008.

[CCK+08b] Ran Canetti, Ling Cheung, Dilsun Kirli Kaynar, Nancy A. Lynch, and Olivier Pereira. Modeling computational security in long-lived systems.

In *CONCUR 2008 - Concurrency Theory, 19th International Conference, CONCUR 2008, Toronto, Canada, August 19-22, 2008. Proceedings*, pages 114–130, 2008.

[CDD+01] Ran Canetti, Ivan Damgård, Stefan Dziembowski, Yuval Ishai, and Tal Malkin. On adaptive vs. non-adaptive security of multiparty protocols. In Birgit Pfitzmann, editor, *Advances in Cryptology - EUROCRYPT 2001, International Conference on the Theory and Application of Cryptographic Techniques, Innsbruck, Austria, May 6-10, 2001, Proceeding*, volume 2045 of *Lecture Notes in Computer Science*, pages 262–279. Springer, 2001.

[CDPW07] Ran Canetti, Yevgeniy Dodis, Rafael Pass, and Shabsi Walfish. Universally composable security with global setup. In *Theory of Cryptography, 4th Theory of Cryptography Conference, TCC 2007, Amsterdam, The Netherlands, February 21-24, 2007, Proceedings*, pages 61–85, 2007.

[CEM+20] T.-H. Hubert Chan, Naomi Ephraim, Antonio Marcedone, Andrew Morgan, Rafael Pass, and Elaine Shi. Blockchain with varying number of players. *IACR Cryptol. ePrint Arch.*, 2020:677, 2020.

[CERF18] Jiannong Cao, Faith Ellen, Luis Rodrigues, and Bernardo Ferreira, editors. *22nd International Conference on Principles of Distributed Systems, OPODIS 2018, December 17-19, 2018, Hong Kong, China*, volume 125 of *LIPIcs*. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2018.

[CGK+20] Daniel Collins, Rachid Guerraoui, Jovan Komatovic, Petr Kuznetsov, Matteo Monti, Matej Pavlovic, Yvonne-Anne Pignolet, Dragos-Adrian Seredinschi, Andrei Tonkikh, and Athanasios Xygkis. Online payments by merely broadcasting messages. In *50th Annual IEEE/IFIP International Conference on Dependable Systems and Networks, DSN 2020, Valencia, Spain, June 29 - July 2, 2020*, pages 26–38, 2020.

[CGLR17b] Tyler Crain, Vincent Gramoli, Mikel Larrea, and Michel Raynal. (leader/randomization/signature)-free byzantine consensus for consortium blockchains. *CoRR*, abs/1702.03068, 2017.

[CGLR18] Tyler Crain, Vincent Gramoli, Mikel Larrea, and Michel Raynal. DBFT: efficient leaderless byzantine consensus and its application to blockchains. In *17th IEEE International Symposium on Network Computing and Applications, NCA 2018, Cambridge, MA, USA, November 1-3, 2018*, pages 1–8, 2018.

[Cha88] David Chaum. Untraceable electronic mail, return addresses, and digital pseudonyms. *Communications of the ACM*, 24(2):84–90, 1988.

[Cha93] Soma Chaudhuri. More choices allow more faults: Set consensus problems in totally asynchronous systems. *Inf. Comput.*, 105(1):132–158, 1993.

[CK19] Omar Cheikhrouhou and Anis Koubâa. Blockloc: Secure localization in the internet of things using blockchain. In *15th International Wireless Communications & Mobile Computing Conference, IWCMC 2019, Tangier, Morocco, June 24-28, 2019*, pages 629–634. IEEE, 2019.

[CKPS01] Christian Cachin, Klaus Kursawe, Frank Petzold, and Victor Shoup. Secure and efficient asynchronous broadcast protocols. In Kilian [Kil01], pages 524–541.

[CL02] Miguel Castro and Barbara Liskov. Practical byzantine fault tolerance and proactive recovery. *ACM Trans. Comput. Syst.*, 20(4):398–461, November 2002.

[CNV06] Miguel Correia, Nuno Ferreira Neves, and Paulo Veríssimo. From consensus to atomic broadcast: Time-free Byzantine-resistant protocols without signatures. *The Computer Journal*, 49(1), 2006.

[CS09] Bernadette Charron-Bost and André Schiper. The heard-of model: computing in distributed systems with benign faults. *Distributed Comput.*, 22(1):49–71, 2009.

[CV17] Christian Cachin and Marko Vukolic. Blockchain consensus protocols in the wild (keynote talk). In Richa [Ric17], pages 1:1–1:16.

[CVL10] Miguel Correia, Giuliana Santos Veronese, and Lau Cheuk Lung. Asynchronous byzantine consensus with 2f+1 processes. In Sung Y. Shin, Sascha Ossowski, Michael Schumacher, Mathew J. Palakal, and Chih-Cheng Hung, editors, *Proceedings of the 2010 ACM Symposium on Applied Computing (SAC), Sierre, Switzerland, March 22-26, 2010*, pages 475–480. ACM, 2010.

[CVNV11] Miguel Correia, Giuliana Santos Veronese, Nuno Ferreira Neves, and Paulo Veríssimo. Byzantine consensus in asynchronous message-passing systems: a survey. *Int. J. Crit. Comput. Based Syst.*, 2(2):141–161, 2011.

[DAL19] Antoine Durand, Emmanuelle Anceaume, and Romaric Ludinard. Stakecube: Combining sharding and proof-of-stake to build fork-free secure permissionless distributed ledgers. In Mohamed Faouzi Atig and Alexander A. Schwarzmann, editors, *Networked Systems - 7th International Conference, NETYS 2019, Marrakech, Morocco, June 19-21, 2019, Revised Selected Papers*, volume 11704 of *Lecture Notes in Computer Science*, pages 148–165. Springer, 2019.

[DBD+18] Omar Dib, Kei-Leo Brousmiche, Antoine Durand, Eric Thea, and Elyes Ben Hamida. Consortium blockchains: Overview, applications and challenges. volume 11, 2018.

[DBL11] *25th IEEE International Symposium on Parallel and Distributed Processing, IPDPS 2011, Anchorage, Alaska, USA, 16-20 May, 2011 - Conference Proceedings*. IEEE, 2011.

[DBL17] *Proceedings of the 26th Symposium on Operating Systems Principles, Shanghai, China, October 28-31, 2017*. ACM, 2017.

[DBL19] *26th Annual Network and Distributed System Security Symposium, NDSS 2019, San Diego, California, USA, February 24-27, 2019*. The Internet Society, 2019.

[DHLM19] Antoine Durand, Elyes Ben Hamida, David Leporini, and Gérard Memmi. Asymptotic performance analysis of blockchain protocols. *CoRR*, abs/1902.04363, 2019.

[DHMA20] Antoine Durand, Guillaume Hébert, Gérard Memmi, and Emmanuelle Anceaume. The stakecube blockchain : Instantiation, evaluation & applications. In *2020 Second International Conference on Blockchain Computing and Applications (BCCA)*, pages 9–15, 2020.

[DLP+86] Danny Dolev, Nancy A. Lynch, Shlomit S. Pinter, Eugene W. Stark, and William E. Weihl. Reaching approximate agreement in the presence of faults. *J. ACM*, 33(3):499–516, 1986.

[DLS88] Cynthia Dwork, Nancy A. Lynch, and Larry J. Stockmeyer. Consensus in the presence of partial synchrony. *J. ACM*, 35(2):288–323, 1988.

[Dou02] John R. Douceur. The sybil attack. In Peter Druschel, M. Frans Kaashoek, and Antony I. T. Rowstron, editors, *Peer-to-Peer Systems, First International Workshop, IPTPS 2002, Cambridge, MA, USA, March 7-8, 2002, Revised Papers*, volume 2429 of *Lecture Notes in Computer Science*, pages 251–260. Springer, 2002.

[DP18] Paul Dunphy and Fabien A. P. Petitcolas. A first look at identity management schemes on the blockchain. *IEEE Secur. Priv.*, 16(4):20–29, 2018.

[DPS16] Phil Daian, Rafael Pass, and Elaine Shi. Snow white: Provably secure proofs of stake. Cryptology ePrint Archive, Report 2016/919, 2016. `https://eprint.iacr.org/2016/919`.

[DR82] Danny Dolev and Rüdiger Reischuk. Bounds on information exchange for byzantine agreement. In Probert et al. [PFS82], pages 132–140.

[DS98] Assia Doudou and André Schiper. Muteness detectors for consensus with byzantine processes. In Brian A. Coan and Yehuda Afek, editors, *Proceedings of the Seventeenth Annual ACM Symposium on Principles of Distributed Computing, PODC '98, Puerto Vallarta, Mexico, June 28 - July 2, 1998*, page 315. ACM, 1998.

[DSU04a] Xavier Défago, André Schiper, and Péter Urbán. Total order broadcast and multicast algorithms: Taxonomy and survey. *ACM Comput. Surv.*, 36(4):372–421, 2004.

[DWC+17] Tien Tuan Anh Dinh, Ji Wang, Gang Chen, Rui Liu, Beng Chin Ooi, and Kian-Lee Tan. Blockbench: A framework for analyzing private blockchains. In *Proceedings of the 2017 ACM International Conference on Management of Data*, pages 1085–1100. ACM, 2017.

[DY83] Danny Dolev and Andrew Chi-Chih Yao. On the security of public key protocols. *IEEE Trans. Inf. Theory*, 29(2):198–207, 1983.

[EOS19] EOS.IO. Technical white paper v2, 2019. Accessed: 2019-03-10.

[FBGB18] Ben Fisch, Joseph Bonneau, Nicola Greco, and Juan Benet. Scaling proof-of-replication for filecoin mining. *Benet//Technical report, Stanford University*, 2018.

[FG03] Matthias Fitzi and Juan A. Garay. Efficient player-optimal protocols for strong and differential consensus. In Elizabeth Borowsky and Sergio Rajsbaum, editors, *Proceedings of the Twenty-Second ACM Symposium on Principles of Distributed Computing, PODC 2003, Boston, Massachusetts, USA, July 13-16, 2003*, pages 211–220. ACM, 2003.

[FGK11] Felix C. Freiling, Rachid Guerraoui, and Petr Kuznetsov. The failure detector abstraction. *ACM Comput. Surv.*, 43(2):9:1–9:40, 2011.

[FHM99] Matthias Fitzi, Martin Hirt, and Ueli M. Maurer. General adversaries in unconditional multi-party computation. In Kwok-Yan Lam, Eiji Okamoto, and Chaoping Xing, editors, *Advances in Cryptology - ASIACRYPT '99, International Conference on the Theory and Applications of Cryptology and Information Security, Singapore, November 14-18, 1999, Proceedings*, volume 1716 of *Lecture Notes in Computer Science*, pages 232–246. Springer, 1999.

[Fis83] Michael J. Fischer. The consensus problem in unreliable distributed systems (A brief survey). In Marek Karpinski, editor, *Fundamentals of Computation Theory, Proceedings of the 1983 International FCT-Conference, Borgholm, Sweden, August 21-27, 1983*, volume 158 of *Lecture Notes in Computer Science*, pages 127–140. Springer, 1983.

[Fit03] Matthias Fitzi. *Generalized communication and security models in Byzantine agreement*. PhD thesis, ETH Zurich, Zürich, Switzerland, 2003.

[FLM85] Michael J. Fischer, Nancy A. Lynch, and Michael Merritt. Easy impossibility proofs for distributed consensus problems. In Malcolm and Strong [MS85], pages 59–70.

[FLP85] Michael J. Fischer, Nancy A. Lynch, and Mike Paterson. Impossibility of distributed consensus with one faulty process. *J. ACM*, 32(2):374–382, 1985.

[FR03] Faith E. Fich and Eric Ruppert. Hundreds of impossibility results for distributed computing. *Distributed Comput.*, 16(2-3):121–163, 2003.

[FSY05] Amos Fiat, Jared Saia, and Maxwell Young. Making chord robust to byzantine attacks. In *Algorithms - ESA 2005, 13th Annual European Symposium, Palma de Mallorca, Spain, October 3-6, 2005, Proceedings*, pages 803–814, 2005.

[Fun] The Linux Fundation. Hyperledger caliper. accessed February 2010.

[FWW04] Matthias Fitzi, Stefan Wolf, and Jürg Wullschleger. Pseudo-signatures, broadcast, and multi-party computation from correlated randomness. In *Advances in Cryptology - CRYPTO 2004, 24th Annual International CryptologyConference, Santa Barbara, California, USA, August 15-19, 2004, Proceedings*, pages 562–578, 2004.

[GB11a] Vijay K. Garg and John Bridgman. The weighted byzantine agreement problem. In *25th IEEE International Symposium on Parallel and Distributed Processing, IPDPS 2011, Anchorage, Alaska, USA, 16-20 May, 2011 - Conference Proceedings* [DBL11], pages 524–531.

[GB11b] Vijay K. Garg and John Bridgman. The weighted byzantine agreement problem. In *25th IEEE International Symposium on Parallel and Distributed Processing, IPDPS 2011, Anchorage, Alaska, USA, 16-20 May, 2011 - Conference Proceedings* [DBL11], pages 524–531.

[GHM+17] Yossi Gilad, Rotem Hemo, Silvio Micali, Georgios Vlachos, and Nickolai Zeldovich. Algorand: Scaling byzantine agreements for cryptocurrencies. In *Proceedings of the 26th Symposium on Operating Systems Principles (SOSP), Shanghai, China, October 28-31, 2017* [DBL17], pages 51–68.

[GK20] Juan A. Garay and Aggelos Kiayias. Sok: A consensus taxonomy in the blockchain era. In Stanislaw Jarecki, editor, *Topics in Cryptology - CT-RSA 2020 - The Cryptographers' Track at the RSA Conference 2020, San Francisco, CA, USA, February 24-28, 2020, Proceedings*, volume 12006 of *Lecture Notes in Computer Science*, pages 284–318. Springer, 2020.

[GKKO07] Juan A. Garay, Jonathan Katz, Chiu-Yuen Koo, and Rafail Ostrovsky. Round complexity of authenticated broadcast with a dishonest majority. In *48th Annual IEEE Symposium on Foundations of Computer Science (FOCS 2007), October 20-23, 2007, Providence, RI, USA, Proceedings*, pages 658–668. IEEE Computer Society, 2007.

[GKL15] Juan A. Garay, Aggelos Kiayias, and Nikos Leonardos. The bitcoin backbone protocol: Analysis and applications. In Oswald and Fischlin [OF15], pages 281–310. Original version.

[GKL17] Juan A. Garay, Aggelos Kiayias, and Nikos Leonardos. The bitcoin backbone protocol with chains of variable difficulty. In Jonathan Katz and Hovav Shacham, editors, *Advances in Cryptology - CRYPTO 2017 - 37th Annual International Cryptology Conference, Santa Barbara, CA, USA, August 20-24, 2017, Proceedings, Part I*, volume 10401 of *Lecture Notes in Computer Science*, pages 291–323. Springer, 2017.

[GKL20a] Juan A. Garay, Aggelos Kiayias, and Nikos Leonardos. The bitcoin backbone protocol: Analysis and applications. *IACR Cryptol. ePrint Arch.*, 2014:765, 2020. Updated version.

[GKL20b] Juan A. Garay, Aggelos Kiayias, and Nikos Leonardos. Full analysis of nakamoto consensus in bounded-delay networks. *IACR Cryptol. ePrint Arch.*, 2020:277, 2020.

[GKM+19] Rachid Guerraoui, Petr Kuznetsov, Matteo Monti, Matej Pavlovic, and Dragos-Adrian Seredinschi. The consensus number of a cryptocurrency. In *Proceedings of the 2019 ACM Symposium on Principles of Distributed Computing, PODC 2019, Toronto, ON, Canada, July 29 - August 2, 2019*, pages 307–316, 2019.

[GKP20] Juan A. Garay, Aggelos Kiayias, and Giorgos Panagiotakos. Consensus from signatures of work. In Stanislaw Jarecki, editor, *Topics in Cryptology - CT-RSA 2020 - The Cryptographers' Track at the RSA Conference 2020, San Francisco, CA, USA, February 24-28, 2020, Proceedings*, volume 12006 of *Lecture Notes in Computer Science*, pages 319–344. Springer, 2020.

[Gol97] Oded Goldreich. On the foundations of modern cryptography. In *Advances in Cryptology - CRYPTO '97, 17th Annual International Cryptology Conference, Santa Barbara, California, USA, August 17-21, 1997, Proceedings*, pages 46–74, 1997.

[GPS18] Rachid Guerraoui, Matej Pavlovic, and Dragos-Adrian Seredinschi. Blockchain protocols: The adversary is in the details. In *Symposium on Foundations and Applications of Blockchain*, page 24, 2018.

[GPS19] Yue Guo, Rafael Pass, and Elaine Shi. Synchronous, with a chance of partition tolerance. In *Advances in Cryptology - CRYPTO 2019 - 39th Annual International Cryptology Conference, Santa Barbara, CA, USA, August 18-22, 2019, Proceedings, Part I*, pages 499–529, 2019.

[GR06] Rachid Guerraoui and Luís E. T. Rodrigues. *Introduction to reliable distributed programming*. Springer, 2006.

[Gra20] Vincent Gramoli. From blockchain consensus back to byzantine consensus. *Future Gener. Comput. Syst.*, 107:760–769, 2020.

[GS18] Vincent Gramoli and Mark Staples. Blockchain standard: Can we reach consensus? *IEEE Commun. Stand. Mag.*, 2(3):16–21, 2018.

[HD14] Magnús M. Halldórsson and Shlomi Dolev, editors. *ACM Symposium on Principles of Distributed Computing, PODC '14, Paris, France, July 15-18, 2014*. ACM, 2014.

[Her91] Maurice Herlihy. Wait-free synchronization. *ACM Trans. Program. Lang. Syst.*, 13(1):124–149, 1991.

[HH93] Vassos Hadzilacos and Joseph Y. Halpern. Message-optimal protocols for byzantine agreement. *Math. Syst. Theory*, 26(1):41–102, 1993.

[HKL20] Martin Hirt, Ard Kastrati, and Chen-Da Liu-Zhang. Multi-threshold asynchronous reliable broadcast and consensus. In *24th International Conference on Principles of Distributed Systems, OPODIS 2020, December 14-16, 2020, Strasbourg, France (Virtual Conference)*, pages 6:1–6:16, 2020.

[HKMS17a] José Horta, Daniel Kofman, David Menga, and Alonso Silva. Novel market approach for locally balancing renewable energy production and flexible demand. In *2017 IEEE International Conference on Smart Grid Communications, SmartGridComm 2017, Dresden, Germany, October 23-27, 2017*, pages 533–539, 2017.

[HM97] Martin Hirt and Ueli M. Maurer. Complete characterization of adversaries tolerable in secure multi-party computation (extended abstract). In James E. Burns and Hagit Attiya, editors, *Proceedings of the Sixteenth Annual ACM Symposium on Principles of Distributed Computing, Santa Barbara, California, USA, August 21-24, 1997*, pages 25–34. ACM, 1997.

[Hoe94] Wassily Hoeffding. Probability inequalities for sums of bounded random variables. In *The Collected Works of Wassily Hoeffding*. 1994.

[HSGX20] Runchao Han, Gary Shapiro, Vincent Gramoli, and Xiwei Xu. On the performance of distributed ledgers for internet of things. *Internet Things*, 10:100087, 2020.

[IML05] Sergei Izmalkov, Silvio Micali, and Matt Lepinski. Rational secure computation and ideal mechanism design. In *46th Annual IEEE Symposium on Foundations of Computer Science (FOCS 2005), 23-25 October 2005, Pittsburgh, PA, USA, Proceedings*, pages 585–595, 2005.

[Int19] Intel. Hyperledger Sawtooth description, 2019. Accessed: 2019-03-10.

[JS19] Joe Abou Jaoude and Raafat George Saadé. Blockchain applications - usage in different domains. *IEEE Access*, 7:45360–45381, 2019.

[KAD+07] Ramakrishna Kotla, Lorenzo Alvisi, Michael Dahlin, Allen Clement, and Edmund L. Wong. Zyzzyva: speculative byzantine fault tolerance. In *Proceedings of the 21st ACM Symposium on Operating Systems Principles 2007, SOSP 2007, Stevenson, Washington, USA, October 14-17, 2007*, pages 45–58, 2007.

[Kil01] Joe Kilian, editor. *Advances in Cryptology - CRYPTO 2001, 21st Annual International Cryptology Conference, Santa Barbara, California, USA, August 19-23, 2001, Proceedings*, volume 2139 of *Lecture Notes in Computer Science*. Springer, 2001.

[KK20] Dimitris Karakostas and Aggelos Kiayias. Securing proof-of-work ledgers via checkpointing. *IACR Cryptol. ePrint Arch.*, 2020:173, 2020.

[KKJG+18] Eleftherios Kokoris-Kogias, Philipp Jovanovic, Linus Gasser, Nicolas Gailly, Ewa Syta, and Bryan Ford. Omniledger: A secure, scale-out, decentralized ledger via sharding. In *IEEE Symposium on Security and Privacy (SSP)*, 2018.

[KL14] Jonathan Katz and Yehuda Lindell. *Introduction to Modern Cryptography, Second Edition*. CRC Press, 2014.

[KRDO16] Aggelos Kiayias, Alexander Russell, Bernardo David, and Roman Oliynykov. Ouroboros: A provably secure proof-of-stake blockchain protocol. Cryptology ePrint Archive, Report 2016/889, 2016. `https://eprint.iacr.org/2016/889`.

[KT20] Petr Kuznetsov and Andrei Tonkikh. Asynchronous reconfiguration with byzantine failures. In *34th International Symposium on Distributed Computing, DISC 2020, October 12-16, 2020, Virtual Conference*, pages 27:1–27:17, 2020.

[KTR20] Ralf Küsters, Max Tuengerthal, and Daniel Rausch. The IITM model: A simple and expressive model for universal composability. *J. Cryptol.*, 33(4):1461–1584, 2020.

[Kwo14] Jae Kwon. Tendermint: Consensus without mining. 2014.

[KZGJ20] Mahimna Kelkar, Fan Zhang, Steven Goldfeder, and Ari Juels. Order-fairness for byzantine consensus. In *Advances in Cryptology - CRYPTO 2020 - 40th Annual International Cryptology Conference, CRYPTO 2020, Santa Barbara, CA, USA, August 17-21, 2020, Proceedings, Part III*, pages 451–480, 2020.

[KZZ16] Aggelos Kiayias, Hong-Sheng Zhou, and Vassilis Zikas. Fair and robust multi-party computation using a global transaction ledger. In *Advances in Cryptology - EUROCRYPT 2016 - 35th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Vienna, Austria, May 8-12, 2016, Proceedings, Part II*, pages 705–734, 2016.

[Lam78] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, 21(7):558–565, 1978.

[LAQ20] Giuseppe Antonio Di Luna, Emmanuelle Anceaume, and Leonardo Querzoni. Byzantine generalized lattice agreement. In *2020 IEEE International Parallel and Distributed Processing Symposium (IPDPS), New Orleans, LA, USA, May 18-22, 2020*, pages 674–683. IEEE, 2020.

[LH19] Ming Li and George Q. Huang. Blockchain-enabled workflow management system for fine-grained resource sharing in e-commerce logistics. In *15th IEEE International Conference on Automation Science and Engineering, CASE 2019, Vancouver, BC, Canada, August 22-26, 2019*, pages 751–755. IEEE, 2019.

[LHSW20] Hemi Leibowitz, Amir Herzberg, Ewa Syta, and Sara Wrótniak. The modular specifications security framework. *IACR Cryptol. ePrint Arch.*, 2020:1040, 2020.

[LM20] Chen-Da Liu-Zhang and Ueli Maurer. Synchronous constructive cryptography. In *Theory of Cryptography - 18th International Conference, TCC 2020, Durham, NC, USA, November 16-19, 2020, Proceedings, Part II*, pages 439–472, 2020.

[LNZ+16] Loi Luu, Viswesh Narayanan, Chaodong Zheng, Kunal Baweja, Seth Gilbert, and Prateek Saxena. A secure sharding protocol for open blockchains. In *ACM SIGSAC Conference on Computer and Communications Security*, 2016.

[LSGZ19] Derek Leung, Adam Suhl, Yossi Gilad, and Nickolai Zeldovich. Vault: Fast bootstrapping for the algorand cryptocurrency. In *26th Annual Network and Distributed System Security Symposium, NDSS 2019, San Diego, California, USA, February 24-27, 2019* [DBL19].

[LSP82] Leslie Lamport, Robert Shostak, and Marshall Pease. The byzantine generals problem. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 4(3):382–401, 1982.

[LVC+16] Shengyun Liu, Paolo Viotti, Christian Cachin, Vivien Quéma, and Marko Vukolic. XFT: practical fault tolerance beyond crashes. In Kimberly Keeton and Timothy Roscoe, editors, *12th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2016, Savannah, GA, USA, November 2-4, 2016*, pages 485–500. USENIX Association, 2016.

[Lyn96] Nancy A. Lynch. *Distributed Algorithms*. Morgan Kaufmann, 1996.

[Mau11] Ueli Maurer. Constructive cryptography - A new paradigm for security definitions and proofs. In *Theory of Security and Applications - Joint Workshop, TOSCA 2011, Saarbrücken, Germany, March 31 - April 1, 2011, Revised Selected Papers*, pages 33–56, 2011.

[MHR14] Achour Mostéfaoui, Moumen Hamouma, and Michel Raynal. Signature-free asynchronous byzantine consensus with t 2<n/3 and o(n^2) messages. In Halldórsson and Dolev [HD14], pages 2–9.

[MHS11] Zarko Milosevic, Martin Hutle, and André Schiper. On the reduction of atomic broadcast to consensus with byzantine faults. In *30th IEEE Symposium on Reliable Distributed Systems (SRDS 2011), Madrid, Spain, October 4-7, 2011*, pages 235–244, 2011.

[Mic17] Silvio Micali. ALGORAND. *CoRR*, abs/1607.01341, 2017.

[MO16] Tal Moran and Ilan Orlov. Proofs of space-time and rational proofs of storage. In *Cryptology ePrint Archive, Report 2016/035*, 2016.

[MR97] Dahlia Malkhi and Michael K. Reiter. Unreliable intrusion detection in distributed computations. In *10th Computer Security Foundations Workshop (CSFW '97), June 10-12, 1997, Rockport, Massachusetts, USA*, pages 116–125. IEEE Computer Society, 1997.

[MR98] Dahlia Malkhi and Michael K. Reiter. Byzantine quorum systems. *Distributed Comput.*, 11(4):203–213, 1998.

[MS85] Michael A. Malcolm and H. Raymond Strong, editors. *Proceedings of the Fourth Annual ACM Symposium on Principles of Distributed Computing, Minaki, Ontario, Canada, August 5-7, 1985*. ACM, 1985.

[MXC+16] Andrew Miller, Yu Xia, Kyle Croman, Elaine Shi, and Dawn Song. The honey badger of BFT protocols. In Weippl et al. [WKK+16], pages 31–42.

[Nak08a] S. Nakamoto. Bitcoin: A peer-to-peer electronic cash system. `https://bitcoin.org/bitcoin.pdf`, 2008.

[NC17] Arvind Narayanan and Jeremy Clark. Bitcoin's academic pedigree. *Commun. ACM*, 60(12):36–45, 2017.

[Nei94] Gil Neiger. Distributed consensus revisited. *Inf. Process. Lett.*, 49(4):195–201, 1994.

[NQTN18] Qassim Nasir, Ilham A. Qasse, Manar Wasif Abu Talib, and Ali Bou Nassif. Performance analysis of hyperledger fabric platforms. *Security and Communication Networks*, 2018:3976093:1–3976093:14, 2018.

[OF15] Elisabeth Oswald and Marc Fischlin, editors. *Advances in Cryptology - EUROCRYPT 2015 - 34th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Sofia, Bulgaria, April 26-30, 2015, Proceedings, Part II*, volume 9057 of *Lecture Notes in Computer Science*. Springer, 2015.

[PFS82] Robert L. Probert, Michael J. Fischer, and Nicola Santoro, editors. *ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing, Ottawa, CanadaAugust 18-20, 1982*. ACM, 1982.

[Pot20] Maria Potop-Butucaru. Blockchains and the commons. In *Networked Systems - 8th International Conference, NETYS 2020, Marrakech, Morocco, June 3-5, 2020, Proceedings*, pages 28–44, 2020.

[PS17a] Rafael Pass and Elaine Shi. Fruitchains: A fair blockchain. In *Proceedings of the ACM Symposium on Principles of Distributed Computing, PODC 2017, Washington, DC, USA, July 25-27, 2017*, pages 315–324, 2017.

[PS17b] Rafael Pass and Elaine Shi. Hybrid consensus: Efficient consensus in the permissionless model. In Andréa W. Richa, editor, *31st International Symposium on Distributed Computing, DISC 2017, October 16-20, 2017, Vienna, Austria*, volume 91 of *LIPIcs*, pages 39:1–39:16. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2017.

[PS18] Rafael Pass and Elaine Shi. Thunderella: Blockchains with optimistic instant confirmation. In *Advances in Cryptology - EUROCRYPT 2018 - 37th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Tel Aviv, Israel, April 29 - May 3, 2018 Proceedings, Part II*, pages 3–33, 2018.

[PSAG20] Deepa Pavithran, Khaled Shaalan, Jamal N. Al-Karaki, and Amjad Gawanmeh. Towards building a blockchain framework for iot. *Clust. Comput.*, 23(3):2089–2103, 2020.

[PSL80] Marshall C. Pease, Robert E. Shostak, and Leslie Lamport. Reaching agreement in the presence of faults. *J. ACM*, 27(2):228–234, 1980.

[PT86] Kenneth J. Perry and Sam Toueg. Distributed agreement in the presence of processor and communication faults. *IEEE Trans. Software Eng.*, 12(3):477–482, 1986.

[PTM+18] Alfonso Panarello, Nachiket Tapas, Giovanni Merlino, Francesco Longo, and Antonio Puliafito. Blockchain and iot integration: A systematic survey. *Sensors*, 18(8):2575, 2018.

[PW96] Birgit Pfitzmann and Michael Waidner. *Information-theoretic pseudosignatures and byzantine agreement for t≥ n/3*. IBM, 1996.

[QQM+18b] Han Qiu, Meikang Qiu, Gérard Memmi, Zhong Ming, and Meiqin Liu. A dynamic scalable blockchain based communication architecture for iot. In *Smart Blockchain - First International Conference, SmartBlock 2018, Tokyo, Japan, December 10-12, 2018, Proceedings*, pages 159–166, 2018.

[Ray13] Michel Raynal. *Distributed Algorithms for Message-Passing Systems*. Springer, 2013.

[Ric17] Andréa W. Richa, editor. *31st International Symposium on Distributed Computing, DISC 2017, October 16-20, 2017, Vienna, Austria*, volume 91 of *LIPIcs*. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2017.

[Rob08] Henry Robinson. A brief tour of flp impossibility, 2008. Online, accessed April 2021.

[Sch90b] Fred B. Schneider. Implementing fault-tolerant services using the state machine approach: A tutorial. *ACM Comput. Surv.*, 22(4):299–319, 1990.

[Shi20] Elaine Shi. Streamlined blockchains: A simple and elegant approach (A tutorial and survey). *IACR Cryptol. ePrint Arch.*, 2020:87, 2020.

[SLZ16] Yonatan Sompolinsky, Yoad Lewenberg, and Aviv Zohar. SPECTRE: A fast and scalable cryptocurrency protocol. *IACR Cryptol. ePrint Arch.*, 2016:1159, 2016.

[SP18a] Pradip Kumar Sharma and Jong Hyuk Park. Blockchain based hybrid network architecture for the smart city. *Future Gener. Comput. Syst.*, 86:650–655, 2018.

[SP18b] Quinten Stokkink and Johan Pouwelse. Deployment of a blockchain-based self-sovereign identity. In *IEEE International Conference on Internet of Things (iThings) and IEEE Green Computing and Communications (GreenCom) and IEEE Cyber, Physical and Social Computing (CPSCom) and IEEE Smart Data (SmartData), iThings/GreenCom/CPSCom/SmartData 2018, Halifax, NS, Canada, July 30 - August 3, 2018*, pages 1336–1342. IEEE, 2018.

[SvR08] Yee Jiun Song and Robbert van Renesse. Bosco: One-step byzantine asynchronous consensus. In Gadi Taubenfeld, editor, *Distributed Computing, 22nd International Symposium, DISC 2008, Arcachon, France, September 22-24, 2008. Proceedings*, volume 5218 of *Lecture Notes in Computer Science*, pages 438–450. Springer, 2008.

[Swa15] Melanie Swan. *Blockchain: Blueprint for a New Economy*. O'Reilly Media, Inc., 1st edition, 2015.

[SWTR18] Harish Sukhwani, Nan Wang, Kishor S. Trivedi, and Andy Rindos. Performance modeling of hyperledger fabric (permissioned blockchain network). In *17th IEEE International Symposium on Network Computing and Applications, NCA 2018, Cambridge, MA, USA, November 1-3, 2018*, pages 1–8. IEEE, 2018.

[SZ18] Yonatan Sompolinsky and Aviv Zohar. PHANTOM and GHOSTDAG: A scalable generalization of nakamoto consensus. *IACR Cryptol. ePrint Arch.*, 2018:104, 2018.

[TNV18] Parth Thakkar, Senthil Nathan, and Balaji Viswanathan. Performance benchmarking and optimizing hyperledger fabric blockchain platform. In *MASCOTS 2018, Milwaukee, WI, USA, September 25-28*, pages 264–276, 2018.

[Vuk15] Marko Vukolic. The quest for scalable blockchain fabric: Proof-of-work vs. BFT replication. In *Open Problems in Network Security - IFIP WG 11.4 International Workshop, iNetSec 2015, Zurich, Switzerland, October 29, 2015, Revised Selected Papers*, pages 112–125, 2015.

[WKK+16] Edgar R. Weippl, Stefan Katzenbeisser, Christopher Kruegel, Andrew C. Myers, and Shai Halevi, editors. *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, Vienna, Austria, October 24-28, 2016*. ACM, 2016.

ECOLE DOCTORALE

Titre : Consensus Byzantin et blockchain : Modèles unifiés et nouveaux protocoles

Mots clés : blockchain ; consensus ; fondations ; pannes Byzantines ; scalabilité ; sharding

Résumé :
Les applications distribuées utilisent des protocoles de consensus afin de maintenir un état consistent sur plusieurs machines dans un réseau. L'avènement récent de bitcoin et des algorithmes basés sur la blockchain a suscité un regain d'intérêt pour ces protocoles, notamment en ce qui concerne leur capacité à passer à l'échelle et à tolérer les participants malicieux. Cependant, cette attention a été source de confusion, submergeant un sujet déjà vaste et complexe avec des assertions imprécises et une terminologie variable.
Dans cette thèse, nous proposons une unification des fondamentaux de la blockchain, grâce à un formalisme capturant un large éventail de modèles communs pour les protocoles de consensus. Nous utilisons ce formalisme pour décrire les spécifications de divers protocoles de consensus d'intérêt pour la blockchain. Nous recadrons et précisons les théorèmes qui décrivent les conditions dans lesquelles un protocole est possible ou non. Nous utilisons également notre formalisme pour décrire le modèle de plusieurs blockchains de référence malgré leurs différences fondamentales, et nous pouvons également évaluer et comparer finement leurs caractéristiques de performance.
Ensuite, nous faisons une proposition pour une blockchain, StakeCube. La sécurité de StakeCube est basée sur le modèle de preuve d'enjeu (Proof-of-Stake), et sa capacité de passage à l'échelle repose sur un principe du partitionnement (sharding), qui est mis en œuvre par une table de hachage distribuée. Nous avons également implémenté une version restreinte de StakeCube et évalué ses performances, validant ainsi sa propriété de passage à l'échelle. Notamment, comme StakeCube troque la preuve de travail (Proof-of-Work) contre la preuve d'enjeu sans sacrifier le passage à l'échelle, ce protocole est particulièrement bien adapté aux applications IoT. Pour mieux démontrer cet aspect, nous avons implémenté une application IoT de marché d'énergie dans StakeCube et avons pu tester avec succès sa viabilité.

Title : Byzantine consensus and blockchain : Models unification and new protocols

Keywords : blockchain ; consensus ; foundations ; Byzantine faults ; scalability ; sharding

Abstract :
Any distributed application makes use of agreement protocols in order to maintain a consistent state across multiple machines in a network. With the recent advent of Bitcoin and blockchain-based algorithms, there has been a renewed interest around such agreement protocols, especially regarding their ability to scale and tolerate malicious participants. However this attention has been a source of misunderstanding, flooding an already large and complex subject with vague claims and different terminology.
In this thesis, we make a unifying view of the blockchain landscape, by proposing a formulation capturing a wide range of accepted models for agreement protocols. We use this formalism to describe the specifications of various agreement protocols of interest for blockchain. We reframe and make precise the theorems that describe the conditions under which a protocol is possible or not. We also use our framework to describe the model of several prominent blockchains despite their fundamental differences, and we are able to make a fine-grained assessment and comparison of their performance characteristics.
Then, we make a proposal for a scalable blockchain, StakeCube. StakeCube's security is based on the Proof-of-Stake model, and its scalability relies on the sharding paradigm, implemented through a distributed hash table. We also implemented (a restricted version of) StakeCube and evaluated its performance, thus validating its scalability property. Notably, because StakeCube trades Proof-of-Work for Proof-of-Stake without sacrificing scalability, it is particularly well suited for IoT applications. To further demonstrate this aspect, we implemented an energy marketplace IoT application in StakeCube and were able to successfully test its viability.

Institut Polytechnique de Paris
91120 Palaiseau, France