Aleph: Efficient Atomic Broadcast in Asynchronous Networks with Byzantine Nodes

Adam Gągol
Aleph Zero Foundation
Jagiellonian University

Damian Leśniak
Aleph Zero Foundation
Jagiellonian University

Damian Straszak
Aleph Zero Foundation

Michał Świętek
Aleph Zero Foundation
Jagiellonian University

ABSTRACT

The spectacular success of Bitcoin and Blockchain Technology in recent years has provided enough evidence that a widespread adoption of a common cryptocurrency system is not merely a distant vision, but a scenario that might come true in the near future. However, the presence of Bitcoin's obvious shortcomings such as excessive electricity consumption, unsatisfying transaction throughput, and large validation time (latency) makes it clear that a new, more efficient system is needed.

We propose a protocol in which a set of nodes maintains and updates a linear ordering of transactions that are being submitted by users. Virtually every cryptocurrency system has such a protocol at its core, and it is the efficiency of this protocol that determines the overall throughput and latency of the system. We develop our protocol on the grounds of the well-established field of Asynchronous Byzantine Fault Tolerant (ABFT) systems. This allows us to formally reason about correctness, efficiency, and security in the strictest possible model, and thus convincingly prove the overall robustness of our solution.

Our protocol improves upon the state-of-the-art HoneyBadgerBFT by Miller *et al.* by reducing the asymptotic latency while matching the optimal communication complexity. Furthermore, in contrast to the above, our protocol does not require a trusted dealer thanks to a novel implementation of a trustless ABFT Randomness Beacon.

CCS CONCEPTS

• Security and privacy → Cryptography; **Distributed systems security**; **• Theory of computation → Distributed algorithms**.

KEYWORDS

Byzantine Fault Tolerance, Asynchrony, DAG, Atomic Broadcast, Randomness Beacon, Consensus, Cryptography

ACM Reference Format:
Adam Gągol, Damian Leśniak, Damian Straszak, and Michał Świętek. 2019. Aleph: Efficient Atomic Broadcast in Asynchronous Networks with Byzantine Nodes. In *AFT '19: Conference on Advances in Financial Technologies, October 21–23, 2019, Zurich, Switzerland*. ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3318041.3355467

1 INTRODUCTION

The introduction of Bitcoin and the Blockchain in the seminal paper of Satoshi Nakamoto [37] is already considered a pivotal point in the history of Finanancial Technologies. While the rise of Bitcoin's popularity clearly shows that there is significant interest in a globally distributed currency system, the scalability issues have become a significant hurdle to achieve it. Indeed, Bitcoin's latency of 30 to 60 minutes, the throughput of 7 transactions per second, and the excessive power usage of the proof of work consensus protocol have motivated the search for alternatives.

At the core of virtually every cryptocurrency system lies a mechanism that collects transactions from users and constructs a total ordering of them, i.e., either explicitly or implicitly forming a blockchain of transactions. This total ordering is then used to determine which transaction came first in case of double-spending attempts and thus to decide which transactions should be validated. The protocol that guides the maintenance and growth of this total ordering is the heart of the whole system. In Bitcoin, the protocol is Proof of Work, but there are also systems based on Proof of Stake [11, 28] and modifications of these two basic paradigms [30, 41]. Aside from efficiency, the primary concern when designing such protocols is their security. While Bitcoin's security certainly has passed the test of time, numerous newly proposed designs claim security but fall short of providing convincing arguments. In many such cases, serious vulnerabilities have been discovered, see [2, 16].

Given these examples, one may agree that for a new system to be trusted, strong mathematical foundations should guarantee its security. What becomes important then are the assumptions under which the security claim is pursued – in order to best imitate the highly adversarial execution environment of a typical permissionless blockchain system, one should work in the strictest possible model. Such a model – the Asynchronous Network model with Byzantine Faults – has spawned a large volume of research within the field of Distributed Systems for the past four decades. Protocols that are designed to work in this model are called Asynchronous

The full version of this paper is available at https://arxiv.org/abs/1908.05156.

Byzantine Fault Tolerant (ABFT) – and are resistant to harsh network conditions: arbitrarily long delays on messages, node crashes, or even multiple nodes colluding in order to break the system. Interestingly, even though these protocols seem to perfectly meet the robustness requirements for these kinds of applications, they have still not gained much recognition in the crypto-community. This is perhaps because the ABFT model is often considered purely theoretical, and in fact, the literature might be hard to penetrate by an inexperienced reader due to heavy mathematical formalism. Indeed, several of the most important results in this area [8, 9, 17, 22] have been developed in the '80s and '90s and were likely not meant for deployment at that time but rather to obtain best asymptotic guarantees. Now, 30 years in the future, perhaps surprisingly, the ABFT model has become more practically relevant than ever, since the presence of bad actors in modern distributed ledger systems is inevitable, and their power ranges from blocking or taking over several nodes to even entirely shutting down large parts of the network.

In recent important work [33], Miller *et al.* presented the HoneyBadgerBFT (HBBFT) protocol, taking the first step towards practical ABFT systems. HBBFT achieves optimal, constant communication overhead, and its validation time scales logarithmically with the number of nodes. Moreover, importantly, HBBFT is rather simple to understand, and its efficiency has been also confirmed by running large-scale experiments. Still, an unpleasant drawback of HBBFT, especially in the context of trustless applications, is that it requires a trusted dealer to initialize.

In this paper, we present a completely new ABFT protocol that keeps all of the good properties of HBBFT and improves upon it in two important aspects: tightening the complexity bounds on latency from logarithmic to constant and eliminating the need for a trusted dealer. Furthermore even though being developed for the asynchronous setting, it matches the optimistic-case performance of 3-round validation time of state-of-the-art synchronous protocols [19]. We believe that our protocol is simple to understand, due to its transparent structure that clearly separates the network layer from the protocol logic. We also present a contribution that might be of independent interest: an efficient, completely trustless ABFT Randomness Beacon that generates common, unpredictable randomness. Since such a mechanism is necessary in many blockchain-based systems, we believe it might see future applications. Finally, we believe that this paper, while offering a valuable theoretical improvement, also contributes to bridging the gap between the theory and practice of ABFT systems.

2 OUR RESULTS

The main goal of this paper[1] is to design a distributed system that runs in a trustless network environment and whose purpose is to build a collective total ordering of messages submitted by users. Apart from blockchain, such systems have several other applications, for example implementing state machine replication, where messages can be arbitrary operations to be executed by a state machine, and the purpose of the system is to keep the states of several copies of the machine consistent by executing the commands in the same order.

A major decision to make when designing systems of this kind is how to realistically model a network environment where such a system would run. In the subsequent paragraphs, we introduce the model we work in and argue why we find it the most suitable for applications in distributed financial systems.

Nodes and Messages. The system consists of N parties $\mathcal{P} = \{\mathcal{P}_1, \mathcal{P}_2, \ldots, \mathcal{P}_N\}$ that are called *nodes*. Each node $\mathcal{P}_i$ identifies itself through its public key pk_i for which it holds a private key sk_i that allows it to sign messages if necessary. Messages in the system are point-to-point, i.e., a node $\mathcal{P}_i$ can send a message m to another node $\mathcal{P}_j$; the node $\mathcal{P}_j$ is then convinced that the message came from $\mathcal{P}_i$ because of the signature. We assume signatures are unforgeable, so a node also cannot deny sending a particular message, since it was signed with its private key.

Network Model. The crucial part of the model are the assumptions about the message delivery and delays. These assumptions are typically formulated by defining the abilities of an adversary, i.e., a powerful entity that watches the system and performs actions to slow it down or cause its malfunction. The first assumption that is somewhat necessary is that the adversary cannot destroy messages that were sent, i.e., when a node sends a message, then it eventually reaches the recipient. In practice, this assumption can be enforced by sending the same message multiple times if necessary. Note also that an adversary with the ability to destroy messages would easily block every possible system, by just destroying all messages. Given that, the most powerful ability an adversary could possibly have is to delay messages for an *arbitrary* amount of time. This is what we assume and what is known in the literature as the *Asynchronous Network Model.* That means the adversary can watch messages being sent and schedule their delivery in an arbitrary order.

In contrast, another popular model is the *Synchronous Network Model*[2], where a global-bound Δ exists such that whenever a message is sent, it is delivered after time at most Δ. As one can imagine, this assumption certainly makes protocol design easier; however, the crucial question to address is: which of these models – asynchronous or synchronous – better fits the typical execution environment of a cryptocurrency system, i.e., the Internet.

While the asynchronous model may seem overly conservative, since no real-life adversary has full control over the network delays, there are mechanisms that may grant him partial control, such as timed DoS attacks. Additionally, synchrony assumptions may be violated due to factors such as transient network partitions or a massive CPU load on several nodes preventing them from sending messages timely.

Finally, the key quality of any protocol meant for finance-related applications is its overall robustness, i.e., a very general notion of resilience against changing network conditions and against other unforeseeable factors. The archetypical partially synchronous algorithm PBFT [19] (and its numerous variants [1, 10, 29]) works in two modes: optimistic and pessimistic. The often-claimed simplicity of PBFT indeed manifests itself in the optimistic mode, but the

[1]In order to make this text accessible also for readers with no background in Secure Distributed Systems, the narration of the paper focuses on providing intuitions and explaining the core ideas, at the cost of occasionally being slightly informal. At the same time, we stay mathematically rigorous when it comes to theorems and proofs.

[2]The Synchronous Model comes in several variants depending on whether the global bound Δ is known to the algorithm or not and whether there is an initial, finite period of asynchrony.

pessimistic mode (that could as well become the default one under unfavorable network conditions) is in fact a completely separate algorithm that is objectively complex and thus prone to implementation errors. Notably, Miller *et al.* in [33] demonstrate an attack scenario on protocols from the PBFT family that completely blocks their operation. In the same paper [33], it is also reasoned that asynchronous protocols are substantially more robust, as the model somewhat forces a single, homogeneous operation mode of the algorithm. As such, we believe that the asynchronous model, even though it enforces stricter conditions, is the best way to describe the environment in which our system is going to operate.

Node Faults. For the kind of system we are trying to design, one cannot assume that all the nodes always proceed according to the protocol. A node could simply crash, or go offline, and thus stop sending messages. Alternatively, a node or a set of nodes could act maliciously (be controlled by the adversary) and send arbitrary messages in order to confuse the remaining nodes and simply break the protocol. These latter kinds of nodes are typically referred to in the literature as *dishonest*, *malicious*, *faulty*, or *Byzantine* nodes, and a protocol that solves a given problem in the presence of dishonest nodes is called Byzantine Fault Tolerant (BFT). In cryptocurrency systems the presence of dishonest nodes is more than guaranteed as there will always be parties trying to take advantage of design flaws in order to gain financial benefits. It is known that no asynchronous system can function properly (reach consensus) in the presence of $N/3$ or more dishonest nodes [9]; thus, we make the standard assumption that the total number of nodes is $N = 3f + 1$, and f of them are dishonest.

2.1 Our Contribution

Before presenting our contributions, let us formalize the problem of building a total ordering of transactions, which in the literature is known under the name of *Atomic Broadcast*.

Definition 2.1 (Atomic Broadcast). *Atomic Broadcast is a problem in which a set of nodes commonly constructs a total ordering of a set of transactions, where the transactions arrive at nodes in an on-line fashion, i.e., might not be given all at once. In a protocol that is meant to solve Atomic Broadcast, the following primitives are defined for every node:*

(1) $\mathsf{Input}(tx)$ *is called whenever a new transaction tx is received by the node,*
(2) $\mathsf{Output}(pos, tx)$ *is called when the node decides to place the transaction tx at the position* $pos \in \mathbb{N}$.

We say that such a protocol implements Atomic Broadcast if it meets all the requirements listed below:

(1) **Total Order.** *Every node outputs a sequence of transactions in an incremental manner, i.e., before outputting a transaction at position* $pos \in \mathbb{N}$ *it must have before output transactions at all positions* $< pos$, *and only one transaction can be output at a given position.*
(2) **Agreement.** *Whenever an honest node outputs a transaction tx at position pos, every other honest node eventually outputs the same transaction at this position.*
(3) **Censorship Resilience.** *Every transaction tx that is input at some honest node is eventually output by all honest nodes.*

The above definition formally describes the setting in which all nodes listen for transactions and commonly construct an ordering of them. While the Total Order and Agreement properties ensure that all the honest nodes always produce the same ordering, the Censorship Resilience property is to guarantee that no transaction is lost due to censorship (especially important in financial applications) but also guarantees that the system makes progress and does not become stuck as long as new transactions are being received.

Let us briefly discuss how the performance of such an Atomic Broadcast protocol is measured. The notion of time is not so straightforward when talking about asynchronous protocols, as the adversary has the power to arbitrarily delay messages between nodes. For this reason, the running time of such a protocol is usually measured in the number of asynchronous rounds it takes to achieve a specific goal [17]. Roughly speaking, the protocol advances to round number r whenever all messages sent in round $r - 2$ have been delivered; for a detailed description of asynchronous rounds, we refer the reader to the full version of the paper [3].

The second performance measure that is typically used when evaluating such protocols is *communication complexity*, i.e., how much data is being sent between the nodes (on average, by a single honest node). To reduce the number of parameters required to state this result, we assume that a transaction, a digital signature, an index of a node, etc., all incur a constant communication overhead when sent; in other words, the communication complexity is measured in machine words, which are assumed to fit all the above objects. Our first contribution is the Aleph protocol for solving Atomic Brodcast whose properties are described in the following

Theorem 2.1 (Atomic Broadcast). *The* Aleph *protocol implements Atomic Broadcast over* $N = 3f + 1$ *nodes in asynchronous network, of which* f *are dishonest and has the following properties:*

(1) **Latency.** *For every transaction tx that is input to at least k honest nodes, the expected number of asynchronous rounds until it is output by every honest node is* $O(N/k)$.
(2) **Communication Complexity.** *The total communication complexity of the protocol in R rounds*[4] *is* $O(T + R \cdot N^2 \log N)$ *per node, where T is the total number of transactions input to honest nodes during R rounds.*

We believe that the most important parameter of such a protocol and also the one that is hardest to optimize in practice is the transaction latency. This is why we mainly focus on achieving the optimal $O(1)$ latency[5]. In the Honey Badger BFT [33] the latency is $\Omega(\log N)$ in the optimistic case, while it becomes $\Omega(\beta \log N)$ when there are roughly $\beta N^2 \log N$ unordered transactions in the system at the time when tx is input. In contrast, the latency of our protocol is $O(1)$ independently from the load.

Similarly to [33] we need to make somewhat troublesome assumptions about the rate at which transactions arrive in the system to reason about the communication complexity, see Section 2.2 for comparison. Still, in the regime of [33] where a steady inflow of transactions is being assumed (i.e., roughly N^2 per round), we match the optimal $O(1)$ communication complexity per transaction

[3]The full version of this paper is available at https://arxiv.org/abs/1908.05156.
[4]Here by a round we formally mean a DAG-rounds, as formally defined in Section 3.1.
[5]The $O(1)$ latency is achieved for transactions that are input to at least $k = \Omega(N)$ honest nodes. This is the same regime as in [33] where it is assumed that $k \geqslant 2/3 N$.

of Honey Badger BFT [33]. As a practical note, we also mention that a variant of our protocol (based on random gossip, see full version of the paper) achieves the optimal 3-round validation latency in the "optimistic case" akin to partially synchronous protocols from the PBFT family [10, 19]. On top of that, our protocol satisfies the so-called *Responsiveness* property [38] which intuitively means that it makes progress at speed proportional to the instantaneous network throughput and is not slowed down by predefined timeouts.

Importantly, we believe that aside from achieving optimal latency and communication complexity, Aleph is simple, clean, and easy to understand, which makes it well fit for a practical implementation. This is a consequence of its modular structure, which separates entirely the network communication layer from the protocol logic. More specifically, the only role of the network layer is to maintain a common (among nodes) data structure called a Communication History DAG, and the protocol logic is then stated entirely through combinatorial properties of this structure. We introduce our protocol in Section 3, a formal analysis and proofs appear in the full version of the paper.

Another important property of our protocol is that unlike [15, 33], it does not require a trusted dealer. The role of a trusted dealer is typically to distribute certain cryptographic keys among nodes before the protocol starts. Clearly, in blockchain systems, no trusted entities can be assumed to exist, and thus a trusted setup is tricky if not impossible to achieve in real-world applications.

Our second contribution is an important, stand-alone component of our protocol that allows us to remove the trusted dealer assumption. More specifically, it can be seen as a protocol for generating unpredictable, common randomness, or in other words, it implements an ABFT Randomness Beacon. Such a source of randomness is indispensable in any ABFT protocol for Atomic Broadcast, since by the FLP-impossibility result [22], it is not possible to reach consensus in this model using a deterministic protocol. Below we give a formalization of what it means for a protocol to implement such a randomness source. The number λ that appears below is the so-called security parameter (i.e., the length of a hash, digital signature, etc.).

Definition 2.2 (Randomness Beacon). *We say that a pair of protocols* (Setup, Toss(m)) *implements a Randomness Beacon if after running* Setup *once, each execution of* Toss(m) *(for any nonce $m \in \{0,1\}^{\star}$) results in λ fresh random bits. More formally, we require*

- **Termination.** *All honest nodes correctly terminate after running either* Setup *or* Toss(m),
- **Correctness.** *For a nonce $m \in \{0,1\}^{\star}$,* Toss(m) *results in all honest nodes outputting a common bitstring $\sigma_m \in \{0,1\}^{\lambda}$,*
- **Unpredictability.** *No computationally bounded adversary can predict the outcome of* Toss(m) *with non-negligible probability.*

In the literature such a source of randomness (especially the variant that outputs just a single bit) is often called a *Common Coin* [14, 15, 33] or a *Global Coin* [17]. We note that once the Toss(m) protocol terminates, the value of σ_m is known, and after revealing it, another execution of Toss(m) will not provide new random bits; thus, the Unpredictability property is meant to be satisfied only before Toss(m) is initiated. As our second main contribution, in Section 4 we introduce the ABFT-Beacon protocol, and in the full version of the paper we prove the following

Theorem 2.2 (ABFT-Beacon). *The* ABFT-Beacon *protocol implements a Randomness Beacon* (Setup, Toss(m)) *such that:*

- *the* Setup *phase takes $O(1)$ asynchronous rounds to complete and has $O(N^2 \log N)$ communication complexity per node,*
- *each subsequent call to* Toss(m) *takes* 1 *asynchronous round and has $O(N)$ communication complexity per node.*

We also remark that the ABFT-Beacon is relatively light when it comes to computational complexity, as the setup requires roughly $O(N^3)$ time per node, and each subsequent toss takes typically $O(N)$ time (under a slightly relaxed adversarial setting); see Section 4.

As an addition to our positive results, in the full version of the paper we introduce the *Fork Bomb* – a spam attack scenario that affects most known DAG-based protocols. In this attack, malicious nodes force honest nodes to download exponential amounts of data and thus likely crash their machines. This attack when attempted to prevent at the implementation layer by banning "suspect nodes" is likely to harm honest nodes as well. Thus, we strongly believe that without a mechanism preventing this kind of attacks already at the protocol layer, liveness is not guaranteed. The basic version of Aleph is resistant against this attack through the use of reliable broadcast to disseminate information among nodes. In the full version of the paper we also show a mechanism to defend against this attack for a gossip-based variant of Aleph.

2.2 Related Work

Atomic Broadcast. For an excellent introduction to the field of Distributed Computing and overview of Atomic Broadcast and Consensus protocols we refer the reader to the book [12]. A more recent work of [16] surveys existing consensus protocols in the context of cryptocurrency systems.

The line of work on synchronous BFT protocols was initiated in [19] with the introduction of PBFT. The PBFT protocol and its numerous variants [1, 10, 29] tolerate byzantine faults, yet their efficiency relies on the good behavior of the network, and drops significantly when entering the (in some cases implicit) "pessimistic mode". As thoroughly reasoned in [33], synchronous algorithms might not be well suited for blockchain-related applications, because of their lack of robustness and vulnerability to certain types of attacks.

In the realm of asynchronous BFT protocols, a large part of the literature focuses on the more classical Consensus problem, i.e., reaching binary agreement by all the honest nodes. As one can imagine, ordering transactions can be reduced to a sequence of binary decisions, and indeed there are known generic reductions that solve Atomic Broadcast by running Consensus multiple times [5, 14, 20, 34]. However, all these reductions either increase the number of rounds by a super-constant factor or introduce a significant communication overhead. Thus, even though Consensus protocols with optimal number of $O(1)$ rounds [17] and optimal communication complexity [15, 36] were known early on, only in the recent work of [33] the Honey Badger BFT (HBBFT) protocol with optimal $O(1)$ communication complexity per transaction was proposed.

The comparison of our protocol to HBBFT is not straightforward since the models of transaction arrivals differ. Roughly, HBBFT assumes that at every round, every node has $\Omega(N^2)$ transactions in its buffer. Under this assumption the communication complexity of HBBFT per epoch, per node is roughly $O(N^2)$, and also $\Omega(N^2)$ transactions are ordered in one epoch, hence the optimal $O(1)$ per transaction is achieved. However, the design of HBBFT that is optimized towards low communication complexity has the negative effect that the latency might be large under high load. More precisely, if βN^2 transactions are pending in the system[6] when tx is being input, the latency of tx is $\approx \beta$ epochs, thus $\approx \beta \log(N)$ rounds. Our algorithm, when adjusted to this model would achieve $\approx \beta$ rounds of latency (thus $\log(N)$-factor improvement), while retaining the same, optimal communication complexity.

In this paper we propose a different assumption on the transaction buffers that allows us to better demonstrate the capabilities of our protocol when it comes to latency. We assume that at every round the ratio between lengths of transaction buffers of any two honest nodes is at most a fixed constant. In this model, our protocol achieves $O(1)$ latency, while a natural adaptation of HBBFT would achieve latency $O(\log(N))$, thus again a factor-$\log(N)$ improvement. A qualitative improvement over HBBFT that we achieve in this paper is that we completely get rid of the trusted dealer assumption. We also note that the definitions of Atomic Broadcast slightly differ between this paper and [33]: we achieve Censorship Resilience assuming that it was input to even a single honest node, while in [33] it has to be input to $\Omega(N)$ nodes.

The recent work[7] of Abraham et al. [3] studies a closely related Validated Asynchronous Byzantine Agreement (VABA) problem, which is, roughly speaking, the problem of picking one value out of N proposed by the nodes. The protocol in [3] achieves $O(1)$ latency and has optimal communication complexity of $O(N)$ per node. We believe that combining it with the ideas present in HoneyBadgerBFT can yield an algorithm with the same communication complexity as HoneyBadgerBFT but with latency improved by a factor of $\log N$. However, when compared to ours, such a protocol still requires a trusted dealer and achieves weaker censorship resilience.

Finally, we remark that our algorithm is based on maintaining a DAG data structure representing the communication history of the protocol. This can be seen as a realization of Lamport's "happened-before" relation [31] or the so-called Causal Order [25]. To the best of our knowledge, the first instance of using DAGs to design asynchronous protocols is the work of [35]. More recently DAGs gained more attention in the blockchain space [4, 21, 41].

Common Randomness. For a thorough overview of previous work on generating randomness in distributed systems and a discussion on the novelty of our solution we refer to Section 4.1.

3 ATOMIC BROADCAST

This section is devoted to an outline and discussion of the Aleph protocol. We start by sketching the overall idea of the algorithm and explaining how it works from a high-level perspective. In this process we present the basic variant of the Aleph protocol, which already contains all the crucial ideas and gives a provably correct implementation of Atomic Broadcast. We refer to the full version of the paper for proofs of correctness and efficiency of Aleph.

3.1 Asynchronous communication as a DAG

The concept of a "communication round" as explained in the preliminaries, is rather natural in the synchronous model, but might be hard to grasp when talking about asynchronous settings. This is one of the reasons why asynchronous models are, generally speaking, harder to work with than their (partially) synchronous counterparts, especially when it comes to proving properties of such protocols.

Units and DAGs. To overcome the above issue, we present a general framework for constructing and analyzing asynchronous protocols that is based on maintaining (by all the nodes) the common "communication history" in the form of an append-only structure: a DAG (Directed Acyclic Graph).

The DAGs that we consider in this paper originate from the following idea: we would like to divide the execution of the algorithm into virtual rounds so that in every round r every node emits exactly one *Unit* that should be thought of as a message broadcast to all the other nodes. Moreover, every such unit should have "pointers" to a large enough number of units from the previous round, emitted by other nodes. Such pointers can be realized by including hashes of the corresponding units, which, assuming that our hash function is collision-free, allows to uniquely determine the "parent units". Formally, every unit has the following fields:

- **Creator.** Index and a signature of unit's creator.
- **Parents.** A list of units' hashes.
- **Data.** Additional data to be included in the unit.

The fact that a unit U has another unit V included as its parent signifies that the information piece carried by V was known to U's creator at the time of constructing U, i.e., V causally preceeds U. All the nodes are expected to generate such units in accordance to some initially agreed upon rules (defined by the protocol) and maintain their local copies of the common DAG, to which new units are continuously being added.

Communication History DAGs. To define the basic rules of creating units note first that this DAG structure induces a partial ordering on the set of units. To emphasize this fact, we often write $V \leqslant U$ if either V is a parent of U, or more generally (transitive closure) that V can be reached from U by taking the "parent pointer" several times. This also gives rise to the notion of DAG-*round* of a unit U that is defined to be the maximum length of a downward chain starting at U. In other words, recursively, a unit with no parents has DAG-round 0 and otherwise a unit has DAG-round equal to the maximum of DAG-rounds of its parents plus one. We denote the DAG-round of a unit U by $\mathrm{R}(U)$. Usually we just write "round" instead of DAG-round except for parts where the distinction between DAG-round and async-round is relevant (we refer to the full version of the paper for a formal introduction of asynchronous rounds). We now proceed to define the notion of a ch-*DAG* (communication history DAG) that serves as a backbone of the Aleph protocol.

[6]For the sake of this comparison we only consider transactions that have been input to $\Omega(N)$ honest nodes.
[7]We would like to thank the anonymous reviewer for bringing this work to our attention.

Definition 3.1 (ch-DAG). *We say that a set of units $\mathcal{D}$ created by $N = 3f + 1$ nodes forms a* ch*-DAG if the parents of every unit in $\mathcal{D}$ also belong to $\mathcal{D}$ and additionally the following conditions hold true*

(1) **Chains.** *For each honest node $\mathcal{P}_i \in \mathcal{P}$, the set of units in $\mathcal{D}$ created by $\mathcal{P}_i$ forms a chain.*
(2) **Dissemination.** *Every round-r unit in $\mathcal{D}$ has at least $2f + 1$ parents of round $r - 1$.*
(3) **Diversity** *Every unit in $\mathcal{D}$ has parents created by pairwise distinct nodes.*

What the above definition tries to achieve is that, roughly, every node should create one unit in every round and it should do so after learning a large enough portion (i.e. at least $2f + 1$) of units created in the previous round. The purpose of the **Chains** rule is to forbid *forking*, i.e., a situation where a node creates more than one unit in a single round. The second rule, **Dissemination**, guarantees that a node creating a unit in round r learned as much as possible from the previous round – note that as there are only $N - f = 2f + 1$ honest nodes in the system, we cannot require that a node receives more than $2f + 1$ units, as byzantine nodes might not have created them. The unit may have additional parents, but the **Diversity** rule ensures that they are created by different nodes - otherwise units could become progressively bigger as the ch-DAG grows, by linking to all the units in existence, hence increasing the communication complexity of the protocol.

Building DAGs. The pseudocode DAG−Grow($\mathcal{D}$) provides a basic implementation of a node that takes part in maintaining a common DAG. Such a node initializes first $\mathcal{D}$ to an empty DAG and then runs two procedures CreateUnit and ReceiveUnits in parallel. To create a unit at round r, we simply wait until $2f + 1$ units of round $r - 1$ are available and then, for every node, we pick its unit of highest round (i.e., of round at most $r - 1$) and include all these N (unless some nodes created no units at all, in which case $< N$) units as parents of the unit. In other words, we wait just enough until we can advance to the next round, and attach to our round-r unit everything we knew at that point in time.

```
DAG-Grow($\mathcal{D}$):
1  CreateUnit(data):
2      for $r = 0, 1, 2, \ldots$ do
3          if $r>0$ then
4              wait until $|\{U \in \mathcal{D} : \mathrm{R}(U) = r - 1\}| \geqslant 2f + 1$
5          $P \leftarrow \{$maximal $\mathcal{P}_i$'s unit of round $< r$ in $\mathcal{D} : \mathcal{P}_i \in \mathcal{P}\}$
6          create a new unit $U$ with $P$ as parents
7          include data in $U$
8          add $U$ to $\mathcal{D}$
9          RBC($U$)
10 ReceiveUnits:
11     loop forever
12         upon receiving a unit $U$ via RBC do
13             add $U$ to $\mathcal{D}$
```

Both CreateUnit and ReceiveUnits make use of a primitive called RBC that stands for *Reliable Broadcast.* This is an asynchronous protocol that guarantees that every unit broadcast by an honest node is eventually received by all honest nodes. More specifically we use the validated RBC protocol which also ensures that incorrect units (with incorrect signatures or incorrect number of parents, etc.) are never broadcast successfully. Furthermore, our version of RBC forces every node to broadcast exactly one unit per round, thus effectively banning forks. We refer to the full version of the paper for a thorough discussion on Reliable Broadcast.

The validated RBC algorithm internally checks whether a certain Valid(U) predicate is satisfied when receiving a unit U. This predicate makes sure that the requirements in Definition 3.1 are satisfied, as well as verifies that certain data, required by the protocol is included in U. Consequently, only valid units are added to the local ch-DAGs maintained by nodes. Furthermore, as alluded above, there can be only a single copy of a unit created by a particular node in a particular round. This guarantees that the local copies of ch-DAGs maintained by different nodes always stay consistent.

Let us now describe more formally the desired properties of a protocol used to grow and maintain a common ch-DAG. For this it is useful to introduce the following convention: we denote the local copy of the ch-DAG maintaned by the ith node by $\mathcal{D}_i$.

Definition 3.2. *We distinguish the following properties of a protocol for constructing a common* ch*-DAG*

(1) **Reliable:** *for every unit U added to a local copy $\mathcal{D}_i$ of an honest node $\mathcal{P}_i$, U is eventually added to the local copy $\mathcal{D}_j$ of every honest node $\mathcal{P}_j$.*
(2) **Ever-expanding:** *for every honest node $\mathcal{P}_i$ the local copy $\mathcal{D}_i$ grows indefinitely, i.e., $\mathrm{R}(\mathcal{D}_i) := \max\{r(U) \mid U \in \mathcal{D}_i\}$ is unbounded.*
(3) **Fork-free:** *whenever two honest nodes i_1, i_2 hold units $U_1 \in \mathcal{D}_{i_1}$ and $U_2 \in \mathcal{D}_{i_2}$ such that both U_1, U_2 have the same creator and the same round number, then $U_1 = U_2$.*

Having these properties defined, we are ready to state the main theorem describing the process of constructing ch-DAG by the DAG-Grow protocol.

Theorem 3.1. *The* DAG-Grow *protocol is reliable, ever-expanding, and fork-free Additionally, during asynchronous round r each honest node holds a local copy of $\mathcal{D}$ of round at least $\Omega(r)$.*

For a proof we refer the reader to the full version of the paper.

Benefits of Using DAGs. After formally introducing the idea of a ch-DAG and explaining how it is constructed we are finally ready to discuss the significance of this concept. First of all, ch-DAGs allow for a clean and conceptually simple separation between the communication layer (sending and receiving messages between nodes) and the protocol logic (mainly deciding on relative order of transactions). Specifically, the network layer is simply realized by running Reliable Broadcast in the background, and the protocol logic is implemented as running off-line computations on the local copy of the ch-DAG. One can think of the local copy of the ch-DAG as the *state* of the corresponding node; all decisions of a node are based solely on its state. One important consequence of this separation is that the network layer, being independent from the logic, can be as well implemented differently, for instance using regular broadcast or random gossip (see full version of the paper).

In the protocol based on ch-DAGs the concept of an adversary and his capabilities are arguably easier to understand. The ability of the adversary to delay a message now translates into a unit being added to some node's local copy of the ch-DAG with a delay. Nonetheless, every unit that has ever been created will still be eventually added to all the ch-DAGs maintained by honest nodes. A consequence of the above is that the adversary can (almost arbitrarily) manipulate the structure of the ch-DAG, or, in other words, he is able to force a given set of round-$(r-1)$ parents for a given round-r unit. But even then, it needs to pick at least $2f+1$ round-$(r-1)$ units, which enforces that more than a half of every unit's parents are created by honest nodes.

3.2 Atomic broadcast via ch-DAG

In this section we show how to build an Atomic Broadcast protocol based on the ch-DAG maintained locally by all the nodes. Recall that nodes receive transactions in an on-line fashion and their goal is to construct a common linear ordering of these transactions. Every node thus gathers transactions in its local buffer and whenever it creates a new unit, all transactions from its buffer are included in the data field of the new unit and removed from the buffer.[8] Thus, to construct a common linear ordering on transactions it suffices to construct a linear ordering of units in the ch-DAG (the transactions within units can be ordered in an arbitrary yet fixed manner, for instance alphabetically). The ordering that we are going to construct also has the nice property that it *extends* the ordering of units induced by the ch-DAG (i.e. the causal order).

Let us remark at this point that all primitives that we describe in this section take a local copy $\mathcal{D}$ of the ch-DAG as one of their parameters and return either

- a result (which might be a single bit, a unit, etc.), or
- $\perp$, signifying that the result is not yet available in $\mathcal{D}$.

The latter means that in order to read off the result, the local copy $\mathcal{D}$ needs to grow further. We occasionally omit the $\mathcal{D}$ argument, when it is clear from the context which local copy should be used.

Ordering Units. The main primitive that is used to order units in the ch-DAG, OrderUnits, takes a local copy $\mathcal{D}$ of the ch-DAG and outputs a list linord that contains a subset of units in $\mathcal{D}$. This list is a prefix of the global linear ordering that is commonly generated by all the nodes. We note that linord will normally not contain all the units in $\mathcal{D}$ but a certain subset of them. More precisely, linord contains all units in $\mathcal{D}$ except those created in the most recent (typically around 3) rounds. While these top units cannot be ordered yet, the structural information about the ch-DAG they carry is used to order the units below them. Intuitively, the algorithm that is run in the ch-DAG at round r makes decisions regarding units that are several rounds deeper, thus the delay.

Note that different nodes might hold different versions of the ch-DAG at any specific point in time, but what we guarantee in the ch-DAG growing protocol is that all copies of ch-DAG are consistent, i.e., all the honest nodes always see exactly the same version of every unit ever created, and that every unit is eventually received by all honest nodes. The function OrderUnits is designed in such a way that even when called on different versions of the ch-DAG $\mathcal{D}_1$, $\mathcal{D}_2$, as long as they are consistent, the respective outputs linorder_1, linorder_2 also agree, i.e., one of them is a prefix of the other.

```
Aleph:
OrderUnits($\mathcal{D}$):
    linord ← []
    for $r = 0, 1, \ldots, R(\mathcal{D})$ do
        $V_r \leftarrow$ ChooseHead($r, \mathcal{D}$)
        if $V_r = \perp$ then break
        batch ← $\{U \in \mathcal{D} : U \leqslant V_r$ and $U \notin$ linord$\}$
        order batch deterministically
        append batch to linord
    output linord
ChooseHead($r, \mathcal{D}$):
    $\pi_r \leftarrow$ GeneratePermutation($r, \mathcal{D}$)
    if $\pi_r = \perp$ then output $\perp$
    else
        $(U_1, U_2, \ldots, U_k) \leftarrow \pi_r$
        for $i = 1, 2, \ldots, k$ do
            if Decide($U_i, \mathcal{D}$) = 1 then
                output $U_i$
            else if Decide($U_i, \mathcal{D}$) = $\perp$ then
                output $\perp$
        output $\perp$
```

The OrderUnits primitive is rather straightforward. At every round r, one unit V_r from among units of round r is chosen to be a "head" of this round, as implemented in ChooseHead. Next, all the units in $\mathcal{D}$ that are less than V_r, but are not less than any of $V_0, V_1, \ldots, V_{r-1}$ form the rth batch of units. The batches are sorted by their round numbers and units within batches are sorted topologically breaking ties using the units' hashes.

Choosing Heads. Perhaps surprisingly, the only nontrivial part of the protocol is choosing a head unit for each round. It is not hard to see that simple strategies for choosing a head fail in an asynchronous network. For instance, one could try picking always the unit created by the first node to be the head: this does not work because the first node might be byzantine and never create any unit. To get around this issue, one could try another tempting strategy: to choose a head for round r, every node waits till round $r+10$, and declares as the head the unit of round r in its copy of the ch-DAG that has the smallest hash. This strategy is also doomed to fail, as it might cause inconsistent choices of heads between nodes: this can happen when some of the nodes see a unit with a very small hash in their ch-DAG while the remaining ones did not receive it yet, which might have either happened just by accident or was forced by actions of the adversary. Note that under asynchrony, one can never be sure whether missing a unit from some rounds back means that there is a huge delay in receiving it or it was never created (the creator is byzantine). More generally, this also justifies that in any asynchronous BFT protocol it is never correct to wait for one fixed node to send a particular message.

Our strategy for choosing a head in round r is quite simple: pick the first unit (i.e., with lowest creator id) that is visible by every node.

[8]This is the simplest possible strategy for including transactions in the ch-DAG and while it is provably correct it may not be optimal in terms of communication complexity. We show how to fix this in the full version of the paper.

The obvious gap in the above is how do we decide that a particular unit is visible? As observed in the example above, waiting a fixed number of rounds is not sufficient, as seeing a unit locally does not imply that all other nodes see it as well. Instead, we need to solve an instance of *Binary Consensus* (also called *Binary Agreement*). In the pseudocode this is represented by a Decide(U) function that outputs 0 or 1; we discuss it in the subsequent paragraph.

There is another minor adjustment to the above scheme that aims at decreasing the worst case latency, which in the just introduced version is $O(\log N)$ rounds[9]. When using a random permutation (unpredictable by the adversary) instead of the order given by the units creator indices, the latency provably goes down to $O(1)$. Such an unpredictable random permutation is realized by the GeneratePermutation function.

Consensus. For a round-$(r+1)$ unit U, by $\downarrow(U)$ we denote the set of all round-r parents of U. Consider now a unit U_0 in round r; we would like the result of Decide($U_0, \mathcal{D}$) to "answer" the question whether all nodes can see the unit U_0. This is done through voting: starting from round $r+1$ every unit casts a "virtual" vote[10] on U_0. These votes are called virtual because they are never really broadcast to other nodes, but they are computed from the ch-DAG. For instance, at round $r+1$, a unit U is said to vote 1 if $U_0 < U$ and 0 otherwise, which directly corresponds to the intuition that the nodes are trying to figure out whether U_0 is visible or not.

```
Aleph-Consensus($\mathcal{D}$):
1  Vote($U_0, U, \mathcal{D}$):
2      if $R(U) \leqslant R(U_0) + 1$ then output[11] $[U_0 < U]$
3      else
4          $A \leftarrow \{\text{Vote}(U_0, V, \mathcal{D}) : V \in \downarrow(U)\}$
5          if $A = \{\sigma\}$ then output $\sigma$
6          else output CommonVote($U_0, R(U), \mathcal{D}$)
7  UnitDecide($U_0, U, \mathcal{D}$):
8      if $R(U) < R(U_0) + 2$ then output $\bot$
9      $v \leftarrow$ CommonVote($U_0, U$)
10     if $|\{V \in \downarrow(U) : \text{Vote}(U_0, V) = v\}| \geqslant 2f+1$ then output v
11     else output $\bot$
12 Decide($U_0, \mathcal{D}$):
13     if $\exists_{U \in \mathcal{D}}$ UnitDecide($U_0, U, \mathcal{D}$) $= \sigma \in \{0,1\}$ then
14         output $\sigma$
15     else output $\bot$
```

Starting from round $r+2$ every unit can either make a final decision on a unit or simply vote again. This process is guided by the function CommonVote($U_0, r', \mathcal{D}$) that provides a common bit $\in \{0,1\}$ for every round $r' \geqslant r+2$. Suppose now that U is of round $r' \geqslant r+2$ and at least $2f+1$ of its round-$(r'-1)$ parents in the ch-DAG voted 1 for U_0, then if CommonVote($U_0, r', \mathcal{D}$) $= 1$, then unit U is declared to decide U_0 as 1. Otherwise, if either there is no supermajority vote among parents (i.e., at least $2f+1$ matching votes) or the supermajority vote does not agree with the CommonVote for this round, the decision is not made yet. In this case, the unit U revotes using either the vote suggested by its parents (in case it was unanimous) or using the default CommonVote. Whenever any of the units $U \in \mathcal{D}$ decides U_0 then it is considered decided with that particular decision bit.

Crucially, the process is designed in such a way that when some unit U decides $\sigma \in \{0,1\}$ on some unit U_0 then we prove that no unit ever decides $\overline{\sigma}$ (the negation of σ) on U_0 and also that every unit of high enough round decides σ on U_0 as well. At this point it is already not hard to see that if a unit U makes decision σ on U_0 then it follows that **all** the units of round $R(U)$ (and any round higher than that) vote σ on U_0. To prove that observe that if a unit V of round r votes σ' then either:

- all its parents voted σ' and hence $\sigma' = \sigma$, because U and V have at least $f+1$ parents in common, or
- $\sigma' =$ CommonVote($U_0, R(V), \mathcal{D}$), but since U decided σ for U_0 then CommonVote($U_0, R(U), \mathcal{D}$) $= \sigma$ and thus $\sigma = \sigma'$ because $R(U) = R(V)$.

The above gives a sketch of "safety" proof of the protocol, i.e., that there will never be inconsistent decisions regarding a unit U_0.

Another property that is desirable for a consensus protocol is that it always terminates, i.e., eventually outputs a consistent decision. In the view of the FLP Theorem [22] this cannot be achieved in the absence of randomness in the protocol. This is the reason why we need to inject random bits to the protocol and this is done in CommonVote. We show that, roughly, if CommonVote provides a random bit (that cannot be predicted by the adversary well in advance) then at every round the decision process terminates with probability at least 1/2. Thus, the expected number of rounds until termination is $O(1)$.

We provide formal proofs of properties of the above protocol in the full version of the paper. In the next section, we show how the randomness in the protocol is generated to implement CommonVote.

3.3 Common Randomness

As already alluded to in Subsection 3.2, due to FLP-impossibility [22] there is no way to achieve binary agreement in finite number of rounds when no randomness is present in the protocol. We also note here that not any kind of randomness will do and the mere fact that a protocol is randomized does not "protect" it from FLP-impossibility. It is crucial to keep the random bits hidden from the adversary till a certain point, intuitively until the adversary has committed to what decision to pursue for a given unit at a certain round. When the random bit is revealed after this commitment, then there is a probability of 1/2 that the adversary "is caught" and cannot delay the consensus decision further.

That means, in particular, that a source of randomness where the nodes are initialized with a common random seed and use a pseudorandom number generator to extract fresh bits is not sufficient, since such a strategy actually makes the algorithm deterministic. We refer to Section 4.1 for an overview of previous work on common randomness.

In our protocol, randomness is injected via a procedure SecretBits whose properties we formalize in the definition below

[9]The reason is that the adversary could cause the first $\Omega(N)$ units to be decided 0. Since the delay of each such decision is a geometric random variable with expectation $\theta(1)$, the maximum of $\Omega(N)$ of them is $\Omega(\log N)$.

[10]The idea of virtual voting was used for the first time in [35].

[11]In the expression $[U_0 < U]$ we use the Iverson bracket notation, i.e., $[U_0 < U] = 1$ if $U_0 < U$ and it is 0 otherwise.

Definition 3.3 (Secret Bits). *The* SecretBits(i, *revealRound*) *primitive takes an index* $i \in \{1, 2, \ldots, N\}$ *and revealRound* $\in \mathbb{N}$ *as parameters, outputs a* λ*-bits secret* s*, and has the following properties whenever initiated by the* ChooseHead *protocol*

(1) *no computationally bounded adversary can guess* s *with non-negligible probability, as long as no honest node has yet created a unit of round revealRound,*
(2) *the secret* s *can be extracted by every node that holds any unit of round revealRound* $+ 1$.

As one might observe, the first parameter i of the SecretBits function seems redundant. Indeed, given an implementation of SecretBits, we could as well use SecretBits$(1, \cdot)$ in place of SecretBits$(i, \cdot)$ for any $i \in [N]$ (here and in the remaining part of the paper $[N]$ stands for $\{1, \ldots, N\}$) and it would seemingly still satisfy the definition above. The issue here is very subtle and will become clear only in Section 4, where we construct a SecretBits function whose computability is somewhat sensitive to what i it is run with[12]. In fact, we recommend the reader to ignore this auxiliary parameter, as our main implementation of SecretBits(i, *revealRound*) is anyway oblivious to the value of i and thus essentially there is exactly one secret per round in the protocol.

The simplest attempt to implement SecretBits would be perhaps to use an external trusted party that observes the growth of the ch-DAG and emits one secret per round, whenever the time comes. Clearly, the correctness of the above relies crucially on the dealer being honest, which is an assumption we cannot make: indeed if there was such an honest entity, why do not we just let him order transactions instead of designing such a complicated protocol?

Instead, our approach is to utilize a threshold secret scheme (see [42]). In short, at round r every node is instructed to include its *share* of the secret (that is hidden from every node) in the unit it creates. Then, in round $r + 1$, every node collects $f + 1$ different such shares included in the previous round and reconstructs the secret from them. Crucially, any set of f or less shares is not enough to derive the secret, thus the adversary controlling f nodes still needs at least one share from an honest node. While this allows the adversary to extract the secret one round earlier then the honest nodes, this advantage turns out to be irrelevant, as, intuitively, he needed to commit to certain actions several turns in advance (see the full version of the paper for a detailed analysis).

Given the SecretBits primitive, we are ready to implement GeneratePermutation and CommonVote (the pseudocode is given in the table Aleph-CommonRandomness).

In the pseudocode, by hash we denote a hash function[13] that takes an input and outputs a bistring of length λ. We remark that the CommonVote at round $R(U_0) + 4$ being 0 is to enforce that units that are invisible at this round will be quickly decided negatively. To explain the intuition behind GeneratePermutation, suppose for a second that SecretBits$(i, r + 4)$ outputs the same secret x independently of i (as it is the case for our main algorithm). Then the above pseudocode assigns a pseudorandom priority hash$(x||U)$ (where, for brevity U denotes a serialization of U) to every unit U or round r which results in a random ordering of these units, as required.

```
Aleph-CommonRandomness:
CommonVote($U_0, r, \mathcal{D}$):
    if $r \leqslant \mathrm{R}(U_0) + 3$ then output 1
    if $r = \mathrm{R}(U_0) + 4$ then output 0
    else
        $i \leftarrow$ the creator of $U_0$
        $x \leftarrow$ SecretBits($i, r, \mathcal{D}$)
        if $x = \perp$ then output $\perp$
        output the first bit of hash($x$)
GeneratePermutation($r, \mathcal{D}$):
    for each unit $U$ of round $r$ in $\mathcal{D}$ do
        $i \leftarrow$ the creator of $U$
        $x \leftarrow$ SecretBits($i, r + 4, \mathcal{D}$)
        if $x = \perp$ then output $\perp$
        assign priority($U$) $\leftarrow$ hash($x||U$) $\in \{0, 1\}^\lambda$
    let $(U_1, U_2, \ldots, U_k)$ be the units in $\mathcal{D}$ of round $r$ sorted by
        priority($\cdot$)
    output $(U_1, U_2, \ldots, U_k)$
```

At this point, the only remaining piece of the protocol is the SecretBits procedure. We provide two implementations in the subsequent Section (see Lemma 4.1): one simple, but requiring a trusted dealer, and the second, more involved but completely trustless.

4 RANDOMNESS BEACON

The goal of this section is to construct an ABFT Randomness Beacon, in order to provide an efficient implementation of SecretBits that is required by our Atomic Broadcast protocol. We start by a detailed review of previous works and how do they compare to the result of this paper. Subsequently we describe how to extract randomness from threshold signatures, which is a basic building block in our approach, and after that we proceed with the description of our randomness beacon.

4.1 Comparison to Previous Work on Distributed Randomness Beacons

We distinguish two main approaches for solving the problem of generating randomness in distributed systems in the presence of byzantine nodes: using Verifiable Secret Sharing (VSS) and via Threshold Signatures. We proceed to reviewing these two approaches and subsequently explain what does our work bring to the field. This discussion is succinctly summarized in Table 1.

Verifiable Secret Sharing. The rough idea of VSS can be explained as follows: a dealer first initializes a protocol to distribute shares $(x_1, x_2, \ldots, x_N)$ of a secret x to the nodes $1, 2, \ldots, N$ so that afterwards every set of $(f + 1)$ nodes can extract x by combining their shares, but any subset of f nodes cannot do that[14]. An honest dealer is requested to pick x uniformly at random (and erase x from memory); yet as one can imagine, this alone is not enough to

[12]More precisely, intuitively SecretBits$(i, \cdot)$ is not expected to work properly if for instance node $\mathcal{P}_i$ has produced no unit at all. Also, importantly, the ChooseHead algorithm will never call SecretBits$(i, \cdot)$ in such a case.
[13]We work in the standard Random Oracle Model.

[14]The thresholds f and $(f + 1)$ here are not the only possible, but are most relevant for our setting, see [13] for a more detailed treatment.

build a reliable, distributed randomness source, with the main issue being: how to elect a dealer with the guarantee that he is honest? In the seminal paper [17] Canetti and Rabin introduce the following trick: let all nodes in the network act as dealers and perform VSS with the intention to combine all these secrets into one (say, by xoring them) that would be random and unpredictable. Turning this idea into an actual protocol that works under full asynchrony is especially tricky, yet [17] manage to do that and end up with a protocol that has $O(N^7)$ communication complexity. This $O(N^7)$ essentially comes from executing an $O(N^5)$ communication VSS protocol N^2 times. Later on this has been improved by Cachin et al. [13] to $O(N^4)$ by making the VSS protocol more efficient. The issue with the AVSS approach is that the communication cost of generating one portion of randomness is rather high – it requires to start everything from scratch when new random bits are requested. This issue is not present in the approach based on threshold signatures, where a one-time setup is run and then multiple portions of random bits can be generated cheaply.

Threshold Signatures. For a technical introduction to this approach we refer to Section 4.2. The main idea is that if the nodes hold keys for threshold signatures with threshold $f + 1$, then the threshold signature $\sigma_m \in \{0, 1\}^\lambda$ of a message m is unpredictable for the adversary and provides us with roughly λ bits of entropy.

The idea of using threshold signatures as a source of randomness is not new and has been studied in the literature [15], especially in the context of blockchain [26, 33]. While this technique indeed produces unbiased randomness **assuming** that the tossing keys have been correctly dealt to all nodes, the true challenge becomes: *How to deal secret keys without involving a trusted dealer?* This problem is known in the literature under the name of Distributed Key Generation (DKG). There has been a lot of prior work on DKG [23, 24, 27, 39], however none of the so far proposed protocols has been designed to run under full asynchrony.

Historically, the first implemenation of DKG was proposed in the work of Pedersen [39]. The network model is not formally specified in [39], yet one can implement it in the synchronous BFT model with f out of $N = 3f + 1$ nodes being malicious. Pedersen's protocol, as later shown by Gennaro et al. in [24], does not guarantee a uniform distribution over private keys; in the same work [24] a fix is proposed that closes this gap, but also in a later work [23] by the same set of authors it is shown that Pedersen's protocol is still secure[15]. The DFINITY randomness beacon [26] runs DKG and subsequently uses threshold signatures to generate randomness.

We note that the AVSS scheme by Cachin et al. [13] can be also used as a basis of DKG since it is polynomial-based and has the additional property of every node learning a certain "commitment" to the secret shares obtained by all the nodes, which in the setting of threshold signatures corresponds to *public keys* of the threshold scheme (see Section 4.2). This allows to construct a DKG protocol as follows: each node runs its own instance of AVSS and subsequently the nodes agree on a subset of $\geqslant f+1$ such dealt secrets to build the keys for threshold signatures. This scheme looks fairly simple, yet it is extremely tricky to implement correctly. One issue is security, i.e., making sure that the adversary is not "front-running" and cannot reveal any secret too early, but more fundamentally: this algorithm requires **consensus** to choose a subset of secrets. ABFT consensus requires **randomness** (by FLP theorem [22]), which we are trying to obtain, and thus we fall into an infinite loop. To obtain consensus without the help of a trusted dealer one is forced to use variants of the Canetti-Rabin protocol [17] (recall that we are working here under the hard constraint of having no trusted dealer, which is not satisfied by most protocols in the literature) that unfortunately has very high communication complexity. Nevertheless using this idea, one can obtain an ABFT DKG protocol with $O(N^4)$ communication complexity. The approach of [27] uses the ideas of [13] and roughly follows the above outlined idea, but avoids the problem of consensus by working in a relaxed model: weak synchrony (which lies in between partial synchrony and full asynchrony).

Protocol	Model	Communication	
		Setup	Query
This Work	async. BFT	$O(N^2 \log N)$	$O(N)$
Canetti, Rabin [17]	async. BFT	-	$O(N^7)$
Cachin et al. [13]	async. BFT	-	$O(N^4)$
Kate et al. DKG [27]	w. sync. BFT	$O(N^3)$	-
Pedersen DKG [23, 39]	sync. BFT	$O(N^2)$	-
Gennaro et al. DKG [24]	sync. BFT	$O(N^2)$	-
DFINITY [26]	sync. BFT	$O(N^2)$	$O(N)$
RandShare [43]	sync. BFT	-	$O(N^2)$
SCRAPE [18]	sync. BFT	-	$O(N^2)$

Table 1: Comparison of randomness beacons. The *Model* column specifies what are the necessary assumptions for the protocol to work. The *Communication* columns specify communication complexities (per node) of a one-time setup phase (run at the very beginning), and of one query for fresh random bits. Some of them were designed for DKG and thus we do not specify the complexity of query (yet it can be made $O(N)$ in all cases). Some protocols also do not run any setup phase, in which case the complexity is omitted.

Our Approach. We base our randomness beacon on threshold signatures, yet make a crucial observation that **full DKG is not necessary** for this purpose. Indeed what we run instead as setup (key generation phase) of our randomness beacon is a weaker variant of DKG. Very roughly: the end result of this variant is that the keys are dealt to a subset of at least $2f + 1$ nodes (thus possibly up to f nodes end up without keys). This allows us to construct a protocol with setup that incurs only $\widetilde{O}(N^2)$ communication, whereas a variant with full DKG[16] would require $\widetilde{O}(N^3)$.

To obtain such a setup in an asynchronous setting we need to deal with the problem (that was sketched in the previous paragraph) of reaching consensus without having a common source of randomness in place. Intuitively a "disentanglement" of these two problems seems hard: in one direction this can be made formal via FLP Theorem [22] (consensus requires randomness); in the opposite direction, intuitively, generating a common coin toss requires all the nodes to agree on a particular random bit.

[15]We note that out protocol can be seen as an adaptation of Pedersen's and thus also requires an ad-hoc proof of security (provided in the full version of the paper), as the distribution of secret keys might be skewed.

[16]We do not describe this variant in this paper; it can be obtained by replacing univariate by bivariate polynomials and adjusting the protocol accordingly (see [13]).

Canetti and Rabin present an interesting way to circumvent this problem in [17]: their consensus algorithm requires only a "weak coin", i.e., a source of randomness that is common (among nodes) with constant, non-zero probability (and is allowed to give different results for different nodes otherwise). Then, via a beautiful construction they obtain such a randomness source that (crucially) does not require any consensus.

Our way to deal with this problem is different. Roughly speaking: in the setup each node "constructs" its unbiased source of randomness and reliably broadcasts it to the remaining nodes. Thus after this step, the problem becomes to pick one out of N randomness sources (consensus). To this end we make a binary decision on each of these sources and pick as our randomness beacon the first that obtained a decision of "1"; however each of these binary consensus instances uses its own particular source of randomness (the one it is making a decision about). At the same time we make sure that the nodes that were late with showing their randomness sources to others will be always decided 0. While this is the basic idea behind our approach, there are several nontrivial gaps to be filled in order to obtain $\widetilde{O}(N^2)$ communication complexity and $O(1)$ latency.

Other Designs. More recently, in the work of [43] a randomness source RandShare has been introduced, which runs a certain variant of Pedersen's protocol to extract random bits. The protocol is claimed by the authors to work in the asynchronous model, yet fails to achieve that goal, as any protocol that **waits for** messages from **all the nodes** to proceed, fails to achieve liveness under asynchrony (or even under partial synchrony). In the Table 1 we list it as synchronous BFT, as after some adjustments it can be made to run in this model. In the same work [43] two other randomness beacons are also proposed: RandHound and RandHerd, yet both of them rely on strong, non-standard assumptions about the network and the adversary and thus we do not include them in Table 1. The method used by SCRAPE [18] at a high level resembles Pedersen's protocol but requires access to a blockchain in order to have a total order on messages sent out during the protocol executation.

Finally, we mention an interesting line of work on generating randomness based on VDFs (Verifiable Delay Functions [6, 32, 40, 44]). Very roughly, the idea is to turn a biasable randomness (such as the hash of a Bitcoin block) into unbiasable randomness via a VDF, i.e., a function $f : \{0,1\}^\lambda \to \{0,1\}^\lambda$ that cannot be computed quickly and whose computation cannot be parallelized, yet it is possible to prove that $f(x) = y$ for a given y much faster than actually computing $f(x)$. The security of this approach relies on the assumption that the adversary cannot evaluate f at random inputs much faster than honest participants.

4.2 Randomness from Threshold Signatures

In this subsection we present the main cryptographic component of our construction: generating randomness from Threshold Signatures. When using a trusted dealer, this component is already enough to implement SecretBits, the bulk of this section though is devoted to proving that a trusted dealer is not necessary.

Randomness through Signatures. The main idea for generating randomness is as follows: suppose that there is a key pair (tk, vk) of private key and public key, such that tk is unknown, while vk is public, and tk allows to sign messages for some public-key cryptosystem that is deterministic[17]. (We refer to tk as to the "tossing key" while vk stands for "verification key"; we use these names to distinguish from the regular private-public key pairs held by the nodes.) Then, for any message m, its digital signature σ_m generated with respect to tk cannot be guessed, but can be verified using vk, thus $\text{hash}(\sigma_m) \in \{0,1\}^\lambda$ provides us with λ random bits! There seems to be a contradiction here though: how can the tossing key be secret and at the same time we are able to sign messages with it? Surprisingly, this is possible using *Threshold Cryptography*: the tossing key tk is "cut into pieces" and distributed among N nodes so that they can jointly sign messages using this key but no node (or a group of dishonest nodes) can learn tk.

More specifically, we use a threshold signature scheme built upon BLS signatures [7]. Such a scheme works over a GDH group G, i.e., a group in which the computational Diffie-Hellman problem is hard (i.e. computing g^{xy} given $g^x, g^y \in G$) but the decisional Diffie-Hellman problem is easy (i.e. verifying that $z = xy$ given $g^x, g^y, g^z \in G$). For more details and constructions of such groups we refer the reader to [7]. We assume from now on that G is a fixed, cyclic, GDH group generated by $g \in G$ and that the order of G is a large prime q. A tossing key tk in BLS is generated as a random element in $\mathbb{Z}_q$ and the public key is $y = g^{tk}$. A signature of a message m is simply $\sigma_m = \widetilde{m}^{tk}$, where $\widetilde{m} \in G$ is the hash (see [7] for a construction of such hash functions) of the message being a random element of G.

Distributing the Tossing Key. To distribute the secret tossing key among all nodes, the Shamir's Secret Sharing Scheme [42] is employed. A trusted dealer generates a random tossing key tk along with a random polynomial A of degree f over $\mathbb{Z}_q$ such that $A(0) = tk$ and privately sends $tk_i = A(i)$ to every node $i = 1, 2, \ldots, N$. The dealer also publishes verification keys, i.e., $VK = (g^{tk_1}, \ldots, g^{tk_N})$. Now, whenever a signature of a message m needs to be generated, the nodes generate *shares* of the signature by signing m with their tossing keys, i.e., each node i multicasts $\widetilde{m}^{A(i)}$. The main observation to make is that now $\widetilde{m}^{A(0)}$ can be computed given only $\widetilde{m}^{A(j)}$ for $f + 1$ different j's (by interpolation) and thus it is enough for a node $\mathcal{P}_j$ to multicast $\widetilde{m}^{A(j)}$ as its share and collect f such shares from different nodes to recover σ_m. On the other hand, any collection of at most f shares is not enough to do that, therefore the adversary cannot sign m all by himself. For details, we refer to the pseudocode of ThresholdSignatures.

SecretBits Through Threshold Signatures. Given the just introduce machinery of threshold signatures, the SecretBits(i, r) primitive is straightforward to implement. Moreover, as promised in Section 3, we give here an implementation that is oblivious to its first argument, i.e., it does only depend on r, but not on i.

First of all, there is a setup phase whose purpose is to deal keys for generating secrets to all nodes. We start by giving a simple version in which an honest dealer is required for the setup. Subsequently, we explain how can this be replaced by a trustless setup, to yield the final version of SecretBits.

In the simpler case, the trusted dealer generates tossing keys and verification keys (TK, VK) for all the nodes using the GenerateKeys

[17] A system that for a given message m there exists only one correct signature for key pair (tk, vk).

```
ThresholdSignatures:
1  GenerateKeys():
2      Let $G = \langle g \rangle$ be a GDH group of prime order $q$
3      generate a random polynomial $A$ of degree $f$ over $\mathbb{Z}_q$
4      let $TK = (tk_1, \dots, tk_N)$ with $tk_i = A(i)$ for $i \in [N]$,
5      let $VK = (vk_1, \dots, vk_N)$ with $vk_i = g^{tk_i}$ for $i \in [N]$
6      output $(TK, VK)$
7  CreateShare($m, tk_i$):
8      $\widetilde{m} \leftarrow$ hash($m$)
9      output $\widetilde{m}^{tk_i}$
10 VerifyShare($m, s, i, VK$):
11     $\widetilde{m} \leftarrow$ hash($m$)
       /* can do the check below since $G$ is GDH */
12     if $\log_g(vk_i) = \log_{\widetilde{m}}(s)$ then output True
13     else output False
14 GenerateSignature($m, S, VK$):
       /* Let $S = \{(s_i, i)\}_{i \in P}$ where $|P| = f + 1$ */
       /* Assume $\forall_j$ VerifyShare($m, s_j, i_j$) = True */
15     interpolate $A(0)$, i.e., find $l_1, l_2, \dots, l_{f+1} \in \mathbb{Z}_q$ s.t.
           $A(0) = \sum_{j=1}^{f+1} l_j A(i_j)$
16     output $\sigma_m \leftarrow \prod_{j=1}^{f+1} s_j^{l_j}$
```

procedure, and then openly broadcasts VK to all nodes and to every node i he secretly sends the tossing key tk_i.

Given such a setup, when SecretBits(j, r) is executed by the protocol, every node i can simply ignore the j argument and generate

$$s_i(m) \leftarrow \text{CreateShare}(m, tk_i)$$

where m is a nonce determined from the round number, say m = "r". Next, after creating its round-($r + 1$) unit U, the P_i collects all the shares S included in U's parents at round r and computes:

$$\sigma_m \leftarrow \text{GenerateSignature}(m, S, VK)$$
$$s(m) \leftarrow \text{hash}(\sigma_m)$$

and $s(m) \in \{0, 1\}^\lambda$ is meant as the output of SecretBits($\cdot, r$).

Finally, to get rid of the trusted dealer, in Subsection 4.3 we describe a trustless protocol that performs the setup instead of a trusted dealer. The crucial difference is that the keys are not generated by one entity, but jointly by all the nodes. Moreover, in the asynchronous version of the setup, every honest node i learns the verification key VK, some key tk_i and a set of "share dealers" $T \subseteq [N]$ of size $2f + 1$, such that every node P_i with $i \in T$ has a correct tossing key tk_i. This, while being slightly weaker than the outcome of the setup of trusted dealer, still allows to implement SecretBits as demonstrated in the below lemma whose proof appears in the full version of the paper.

Lemma 4.1 (Secret Bits). *The above scheme in both versions (with and without trusted dealer) correctly implements* SecretBits.

4.3 Randomness Beacon with Trustless Setup

In Section 4.2 we have discussed how to implement a Randomness Beacon based on a trusted dealer. Here, we devise a version of this protocol that has all the benefits of the previous one and at the same time is completely trustless. For the sake of clarity, we first provide a high level perspective of the new ideas and how do they combine to give a trustless Randomness Beacon, next we provide a short summary of the protocol in the form of a very informal pseudocode, and finally we fill in the gaps by giving details on how the specific parts are implemented and glued together.

Key Boxes. Since no trusted dealer is available, perhaps the most natural idea is to let all the nodes serve as dealers simultaneously. More precisely we let every node emit (via RBC, i.e., by placing it as data in a unit) a tossing *Key Box* that is constructed by a node k (acting as a dealer) as follows:

- sample a random polynomial of degree at most f
$$A_k(x) = \sum_{j=0}^{f} a_{k,j} x^j \in \mathbb{Z}_q[x],$$
- compute a commitment to A_k as
$$C_k = (g^{a_{k,0}}, g^{a_{k,1}}, \dots, g^{a_{k,f}}).$$
- define the tossing keys TK_k and verification keys VK_k
$$tk_{k,i} := A_k(i) \qquad \text{for } i = 1, 2, \dots, N$$
$$vk_{k,i} := g^{tk_i} \qquad \text{for } i = 1, 2, \dots, N.$$

  Note that in particular each verification key $vk_{k,i}$ for $i \in [N]$ can be computed from C_k as $vk_{k,i} = \prod_{j=0}^{f} C_{k,j}^{i^j}$.
- encrypt the tossing keys for every node i using the dedicated public key[18] $pk_{k \to i}$ as
$$e_{k,i} := \text{Enc}_{k \to i}\left(tk_{k,i}\right)$$
  and let $E_k := (e_{k,1}, e_{k,2}, \dots, e_{k,N})$.
- the kth key box is defined as $KB_k = (C_k, E_k)$.

In our protocol, every node $\mathcal{P}_k$ generates its own key box KB_k and places (C_k, E_k) in his unit of round 0. We define the kth key set to be $KS_k = (VK_k, TK_k)$ and note that given the key box $KB_k = (C_k, E_k)$, every node can reconstruct VK_k and moreover, every node i can decrypt his tossing key $tk_{k,i}$ from the encrypted part E_k, but is not able to extract the remaining keys. The encypted tossing keys E_k can be seen as a certain way of emulating "authenticated channels".

Verifying Key Sets. Since at least 2/3 of nodes are honest, we also know that 2/3 of all the key sets are safe to use, because they were produced by honest nodes who properly generated the key sets and the corresponding key boxes an erased the underlying polynomial (and thus all the tossing keys). Unfortunately, it is not possible to figure out which nodes cheated in this process (and kept the tossing keys that were supposed to be erased).

What is even harder to check, is whether a certain publicly known key box KB_k was generated according to the instructions

[18]We assume that as part of PKI setup each node i is given exactly N different key pairs for encryption: $(sk_{k \to i}, pk_{k \to i})$ for $k \in [N]$. The key $pk_{k \to i}$ is meant to be used by the kth node to encrypt a message whose recipient is i (denoted as $\text{Enc}_{k \to i}(\cdot)$). This setup is merely for simplicity of arguments – one could instead have one key per node if using verifiable encryption or double encryption with labels.

above. Indeed, as a node $\mathcal{P}_i$ we have access only to our tossing key $tk_{k,i}$ and while we can verify that this key agrees with the verification key (check if $g^{tk_{k,i}} = vk_{k,i}$), we cannot do that for the remaining keys that are held by other nodes. The only way to perform verification of the key sets is to do that in collaboration with other nodes. Thus, in the protocol, there is a round at which every node "votes" for correctness of all the key sets it has seen so far. As will be explained in detail later, these votes cannot be "faked" in the following sense: if a node $\mathcal{P}_i$ votes on a key set KS_k being incorrect, it needs to provide a proof that its key $tk_{k,i}$ (decrypted from $e_{k,i}$) is invalid, which cannot be done if $\mathcal{P}_k$ is an honest dealer. Thus consequently, dishonest nodes cannot deceive the others that a valid key set is incorrect.

Choosing Trusted Key Sets. At a later round these votes are collected and summarized locally by every node $\mathcal{P}_i$ and a trusted set of key sets $T_i \subseteq [N]$ is determined. Intuitively, T_i is the set of indices k of key sets such that:

- the node $\mathcal{P}_i$ is familiar with KB_k,
- the node $\mathcal{P}_i$ has seen enough votes on KS_k correctness that it is certain that generating secrets from KS_k will be successful,

The second condition is determined based solely on the ch-DAG structure below the ith node unit at a particular round. What will be soon important is that, even though the sets T_i do not necessarily coincide for different i, it is guaranteed that $|T_i| \geqslant f + 1$ for every i, and thus at least one honest key set is included in each of them.

Combining Tosses. We note that once the set T_i is determined for some index $i \in [N]$, this set essentially defines a global common source of randomness that cannot be biased by an adversary. Indeed, suppose we would like to extract the random bits corresponding to nonce m. First, in a given round, say r, every node should include its share for nonce m corresponding to every key set that it voted as being correct. In the next round, it is guaranteed that the shares included in round r are enough to recover the random secret $\sigma_{m,k} = m^{A_k(0)} \in G$ (the threshold signature of m generated using key set KS_k) for every $k \in T_i$. Since up to f out of these secrets might be generated by the adversary, we simply take

$$\tau_{m,i} := \prod_{k \in T_i} \sigma_{m,k} = m^{\sum_{k \in T_i} A_k(0)} \in G$$

to obtain a uniformly random element of G and thus (by hashing it) a random bitstring of length λ corresponding to node $\mathcal{P}_i$, resulting from nonce m. From now on we refer to the ith such source of randomness (i.e., corresponding to $\mathcal{P}_i$) as $\mathsf{MultiCoin}_i$.

Agreeing on Common MultiCoin. So far we have said that every node $\mathcal{P}_i$ defines locally its strong source of randomness $\mathsf{MultiCoin}_i$. Note however that paradoxically, this abundance of randomness sources is actually problematic: which one shall be used by all the nodes to have a "truly common" source of randomness? This is nothing other than a problem of "selecting a head", i.e., choosing one unit from a round – a problem that we already solved in Section 3! At this point however, an attentive reader is likely to object, as the algorithm from Section 3 only works provided a common source of randomness. Therefore, the argument seems to be cyclic as we are trying to construct such a source of randomness from the algorithm from Section 3. Indeed, great care is required here: as explained in Section 3, all what is required for the ChooseHead protocol to work is a primitive SecretBits(i, r) that is supposed to inject a secret of λ bits at the rth round of the protocol. Not going too deep into details, we can appropriately implement such a method by employing the MultiCoins we have at our disposal. This part, along with the construction of MultiCoins, constitutes the technical core of the whole protocol.

Combining Key Sets. Finally, once the head is chosen to be $l \in [N]$, from now on one could use $\mathsf{MultiCoin}_l$ as the common source of randomness. If one does not care about savings in communication and computational complexity, then the construction is over. Otherwise, observe that tossing the $\mathsf{MultiCoin}_l$ at a nonce m requires in the worst case N shares from every single node. This is sub-optimal, and here is a simple way to reduce it to just 1 share per node. Recall that every Key Set KS_k that contributes to $\mathsf{MultiCoin}_l$ is generated from a polynomial $A_k \in \mathbb{Z}_q[x]$ of degree f. By simple algebraic manipulations one can combine the tossing keys and all the verification keys of all key sets KS_k for $k \in T_l$ so that the resulting Key Set corresponds to the sum of these polynomials

$$A(x) := \left(\sum_{k \in T_l} A_k(x) \right) \in \mathbb{Z}_q[x].$$

This gives a source of randomness that requires one share per nonce from every node; note that since the set T_l contains at least one honest node, the polynomial A can be considered random.

Protocol Sketch. We are now ready to provide an informal sketch of the Setup protocol and Toss protocol, see the ABFT – Beacon box below. The content of the box is mostly a succinct summary of Section 4.3. What might be unclear at this point is the condition of the if statement in the Toss function. It states that the tossing key tk_i is supposed to be correct in order to produce a share: indeed it can happen that one of the key boxes that is part of the $\mathsf{MultiCoin}_l$ does not provide a correct tossing key for the ith node, in which case the combined tossing key tk_i cannot be correct either. This however is not a problem, as the protocol still guarantees that at least $2f + 1$ nodes hold correct combined tossing keys, and thus there will be always enough shares at our disposal to run ExtractBits.

4.4 Details of the Setup

We provide some more details regarding several components of the Setup phase of ABFT – Beacon that were treated informally in the previous subsection.

Voting on Key Sets. Just before creating the unit at round 3 the ith node is supposed to inspect all the key boxes that are present in its current copy of the ch-DAG. Suppose KB_k for some $k \in [N]$ is one of such key sets. The only piece of information about KB_k that is known to $\mathcal{P}_i$ but hidden from the remaining nodes is its tossing key. Node $\mathcal{P}_i$ recovers this key by decrypting it using its secret key (dedicated for k) $sk_{k \to i}$

$$tk_{k,i} \leftarrow \mathrm{Dec}_{k \to i}(e_{k,i}).$$

Now, if node $\mathcal{P}_k$ is dishonest, it might have included an incorrect tossing key, to check correctness, the ith node verifies whether

$$g^{tk_{k,i}} \stackrel{?}{=} vk_{k,i},$$

where g is the fixed generator of the group G we are working over. The ith node includes the following piece of information in its

```
ABFT-Beacon:
1  Setup():
       /* Instructions for node $\mathcal{P}_i$.                */
2      Initialize growing the ch-DAG $\mathcal{D}$
3      In the data field of your units include (as specified in
         Section 4.4):
4          At round 0: a key box $KB_i$,
5          At round 3: votes regarding correctness of key boxes
           present in $\mathcal{D}$,
6          At rounds $\geqslant 6$: shares necessary to extract
           randomness from SecretBits.
7      Run ChooseHead to determine the head unit at round 6
       and let $l$ be its creator.
8      Combine the key sets $\{KS_j : j \in T_l\}$ and let $(tk_i, VK)$ be
       the corresponding tossing key and verification keys.
9  Toss($m$):
       /* Code for node $\mathcal{P}_i$.                         */
10     if $tk_i$ is correct then
11         $s_i \leftarrow$ CreateShare$(m, tk_i)$
12         multicast $(s_i, i)$
13     wait until receiving a set of $f + 1$ valid shares $S$
       /* validity is checked using VerifyShare()   */
14     output $\sigma_m :=$ ExtractBits$(m, S, VK)$
```

round-3 unit

$$\mathsf{VerKey}\,(KB_k, i) = \begin{cases} 1 & \text{if } tk_{k,i} \text{ correct} \\ \mathrm{Dec}_{k\to i}(e_{k,i}) & \text{oth.} \end{cases}$$

Note that if a node $\mathcal{P}_i$ votes that KB_k is incorrect (by including the bottom option of VerKey in its unit) it cannot lie, since other nodes can verify whether the plaintext it provided agrees with the ciphertext in KB_k (as the encryption scheme is assumed to be deterministic) and if that is not the case, they treat a unit with such a vote as invalid (and thus not include it in their ch-DAG). Thus, consequently, the only way dishonest nodes could cheat here is by providing positive votes for incorrect key boxes. This can not harm honest nodes, because by positively verifying some key set a node declares that from now on it will be providing shares for this particular key set whenever requested. If in reality the corresponding tossing key is not available to this node, it will not be able to create such shares and hence all its units will be henceforth considered invalid.

One important property this scheme has is that it is safe for an honest recipient $\mathcal{P}_i$ of $e_{k,i}$ to reveal the plaintext decryption of $e_{k,i}$ in case it is not (as expected) the tossing key $tk_{k,i}$ – indeed if $\mathcal{P}_k$ is dishonest then either he actually encrypted some data d in $e_{k,i}$ in which case he learns nothing new (because d is revealed) or otherwise he obtained $e_{k,i}$ by some other means, in which case $\mathrm{Dec}_{k\to i}(e_{k,i})$ is a random string, because no honest node ever creates a ciphertext encrypted with $p_{k\to i}$ and we assume that the encryption scheme is semantically secure. Note also that if instead of having N key pairs per node we used a similar scheme with every node having just one key pair, then the adversary could reveal some tossing keys of honest nodes through the following attack: the adversary copies an honest node's (say jth) ciphertext $e_{j,i}$ and includes it as $e_{k,i}$ in which case $\mathcal{P}_i$ is forced to reveal $\mathrm{Dec}_i(e_{j,i}) = tk_{j,i}$ which should have remained secret! This is the reason why we need dedicated key pairs for every pair of nodes.

Forming MultiCoins. The unit $V := U[i;6]$ created by $\mathcal{P}_i$ in round 6 defines a set $T_i \subseteq [N]$ as follows: $k \in N$ is considered an element of T_i if and only if all the three conditions below are met

(1) $U[k;0] \leqslant V$,
(2) For every $j \in [N]$ such that $U[j;3] \leqslant V$ it holds that

$$\mathsf{VerKey}\,(KB_k, j) = 1.$$

At this point it is important to note that every node that has $U[i;6]$ in its copy of the ch-DAG can compute T_i as all the conditions above can be checked deterministically given only the ch-DAG.

SecretBits via MultiCoins. Recall that to implement CommonVote and GeneratePermutation it suffices to implement a more general primitive SecretBits(i,r) whose purpose is to inject a secret at round r that can be recovered by every node in round $r+1$ but cannot be recovered by the adversary till at least one honest node has created a unit of round r.

The technical subtlety that becomes crucial here is that the SecretBits(i,r) is only called for $r \geqslant 9$ and only by nodes that have $U[i;6]$ in their local ch-DAG. More specifically, this allows us to implement SecretBits$(i,\cdot)$ through MultiCoin$_i$. The rationale behind doing so is that every node that sees $U[i;6]$ can also see all the Key Sets that comprise the ith MultiCoin and thus consequently it "knows" what MultiCoin$_i$ is[19].

Suppose now that we would like to inject a secret at round r for index $i \in [N]$. Define a nonce $m :=$ "i||r" and request all the nodes $\mathcal{P}_k$ such that $U[i;6] \leqslant U[k,r]$ to include in $U[k,r]$ a share for the nonce m for every Key Set KS^j such that $j \in T_i$. In addition, if $\mathcal{P}_k$ voted that KS_j is incorrect in round 3, or $\mathcal{P}_k$ did not vote for KB_j at all (since KB_j was not yet available to him at round 3) then $\mathcal{P}_k$ is not obligated to include a share (note that its unit $U[k,3]$ that is below $U[k,r]$ contains evidence that its key was incorrect).

As we then show, given any unit $U \in \mathcal{D}$ of round $r+1$ one can then extract the value of MultiCoin$_i$ from the shares present in round-r units in $\mathcal{D}$. Thus, consequently, the value of SecretBits(i,r) in a ch-DAG $\mathcal{D}$ is available whenever any unit in $\mathcal{D}$ has round $\geqslant r+1$, hence we arrive at the lemma (for a proof we refer to the full version of the paper)

Lemma 4.2 (Secret Bits from Multicoins). *The above defined scheme based on MultiCoins correctly implements* SecretBits.

5 ACKNOWLEDGEMENTS

First of all, authors would like to thank Matthew Niemerg for introducing us to the topic, constant support, and countless hours spent on valuable discussions. Additionally, we would like to show our gratitude to Michał Handzlik, Tomasz Kisielewki, Maciej Gawron, and Łukasz Lachowski, for reading the paper, proposing changes that improved its consistency and readability.
This research was funded by the Aleph Zero Foundation.

[19]We epmhasize that using a fixed MultiCoin, for instance MultiCoin$_1$ would not be correct here, as there is no guarantee the 1st node has delivered its unit at round 6. More generally, it is crucial that for different units U_0 we allow to use different MultiCoins, otherwise we would have solved Byzantine Consensus without randomness, which is impossible by the FLP Theorem [22].

REFERENCES

[1] Michael Abd-El-Malek, Gregory R. Ganger, Garth R. Goodson, Michael K. Reiter, and Jay J. Wylie. 2005. Fault-scalable Byzantine fault-tolerant services. In *Proceedings of the 20th ACM Symposium on Operating Systems Principles 2005, SOSP 2005, Brighton, UK, October 23-26, 2005.* 59–74. https://doi.org/10.1145/1095810.1095817

[2] Ittai Abraham, Guy Gueta, Dahlia Malkhi, Lorenzo Alvisi, Ramakrishna Kotla, and Jean-Philippe Martin. 2017. Revisiting Fast Practical Byzantine Fault Tolerance. *CoRR* abs/1712.01367 (2017). arXiv:1712.01367 http://arxiv.org/abs/1712.01367

[3] Ittai Abraham, Dahlia Malkhi, and Alexander Spiegelman. 2019. Asymptotically Optimal Validated Asynchronous Byzantine Agreement. In *Proceedings of the 2019 ACM Symposium on Principles of Distributed Computing, PODC 2019, Toronto, ON, Canada, July 29 - August 2, 2019.* 337–346. https://doi.org/10.1145/3293611.3331612

[4] Leemon Baird. 2016. The swirlds hashgraph consensus algorithm: Fair, fast, byzantine fault tolerance. *Swirlds Tech Reports SWIRLDS-TR-2016-01, Tech. Rep.* (2016).

[5] Michael Ben-Or and Ran El-Yaniv. 2003. Resilient-optimal interactive consistency in constant time. *Distributed Computing* 16, 4 (2003), 249–262. https://doi.org/10.1007/s00446-002-0083-3

[6] Dan Boneh, Joseph Bonneau, Benedikt Bünz, and Ben Fisch. 2018. Verifiable Delay Functions. In *Advances in Cryptology - CRYPTO 2018 - 38th Annual International Cryptology Conference, Santa Barbara, CA, USA, August 19-23, 2018, Proceedings, Part I.* 757–788. https://doi.org/10.1007/978-3-319-96884-1_25

[7] Dan Boneh, Ben Lynn, and Hovav Shacham. 2004. Short Signatures from the Weil Pairing. *J. Cryptology* 17, 4 (2004), 297–319. https://doi.org/10.1007/s00145-004-0314-9

[8] Gabriel Bracha. 1987. Asynchronous Byzantine Agreement Protocols. *Inf. Comput.* 75, 2 (1987), 130–143. https://doi.org/10.1016/0890-5401(87)90054-X

[9] Gabriel Bracha and Sam Toueg. 1983. Resilient Consensus Protocols. In *Proceedings of the Second Annual ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing, Montreal, Quebec, Canada, August 17-19, 1983.* 12–26. https://doi.org/10.1145/800221.806706

[10] Ethan Buchman, Jae Kwon, and Zarko Milosevic. 2018. The latest gossip on BFT consensus. *CoRR* abs/1807.04938 (2018). arXiv:1807.04938 http://arxiv.org/abs/1807.04938

[11] Vitalik Buterin and Virgil Griffith. 2017. Casper the Friendly Finality Gadget. *CoRR* abs/1710.09437 (2017). arXiv:1710.09437 http://arxiv.org/abs/1710.09437

[12] Christian Cachin, Rachid Guerraoui, and Luís E. T. Rodrigues. 2011. *Introduction to Reliable and Secure Distributed Programming (2. ed.).* Springer. https://doi.org/10.1007/978-3-642-15260-3

[13] Christian Cachin, Klaus Kursawe, Anna Lysyanskaya, and Reto Strobl. 2002. Asynchronous verifiable secret sharing and proactive cryptosystems. In *Proceedings of the 9th ACM Conference on Computer and Communications Security, CCS 2002, Washington, DC, USA, November 18-22, 2002.* 88–97. https://doi.org/10.1145/586110.586124

[14] Christian Cachin, Klaus Kursawe, Frank Petzold, and Victor Shoup. 2001. Secure and Efficient Asynchronous Broadcast Protocols. In *Advances in Cryptology - CRYPTO 2001, 21st Annual International Cryptology Conference, Santa Barbara, California, USA, August 19-23, 2001, Proceedings.* 524–541. https://doi.org/10.1007/3-540-44647-8_31

[15] Christian Cachin, Klaus Kursawe, and Victor Shoup. 2005. Random Oracles in Constantinople: Practical Asynchronous Byzantine Agreement Using Cryptography. *J. Cryptology* 18, 3 (2005), 219–246. https://doi.org/10.1007/s00145-005-0318-0

[16] Christian Cachin and Marko Vukolic. 2017. Blockchain Consensus Protocols in the Wild. *CoRR* abs/1707.01873 (2017). arXiv:1707.01873 http://arxiv.org/abs/1707.01873

[17] Ran Canetti and Tal Rabin. 1993. Fast asynchronous Byzantine agreement with optimal resilience. In *Proceedings of the Twenty-Fifth Annual ACM Symposium on Theory of Computing, May 16-18, 1993, San Diego, CA, USA.* 42–51. https://doi.org/10.1145/167088.167105

[18] Ignacio Cascudo and Bernardo David. 2017. SCRAPE: Scalable Randomness Attested by Public Entities. In *Applied Cryptography and Network Security - 15th International Conference, ACNS 2017, Kanazawa, Japan, July 10-12, 2017, Proceedings.* 537–556. https://doi.org/10.1007/978-3-319-61204-1_27

[19] Miguel Castro and Barbara Liskov. 1999. Practical Byzantine Fault Tolerance. In *Proceedings of the Third USENIX Symposium on Operating Systems Design and Implementation (OSDI), New Orleans, Louisiana, USA, February 22-25, 1999.* 173–186. https://dl.acm.org/citation.cfm?id=296824

[20] Miguel Correia, Nuno Ferreira Neves, and Paulo Veríssimo. 2006. From Consensus to Atomic Broadcast: Time-Free Byzantine-Resistant Protocols without Signatures. *Comput. J.* 49, 1 (2006), 82–96. https://doi.org/10.1093/comjnl/bxh145

[21] George Danezis and David Hrycyszyn. 2018. Blockmania: from Block DAGs to Consensus. *arXiv preprint arXiv:1809.01620* (2018).

[22] Michael J. Fischer, Nancy A. Lynch, and Mike Paterson. 1985. Impossibility of Distributed Consensus with One Faulty Process. *J. ACM* 32, 2 (1985), 374–382. https://doi.org/10.1145/3149.214121

[23] Rosario Gennaro, Stanislaw Jarecki, Hugo Krawczyk, and Tal Rabin. 2003. Secure Applications of Pedersen's Distributed Key Generation Protocol. In *Topics in Cryptology - CT-RSA 2003, The Cryptographers' Track at the RSA Conference 2003, San Francisco, CA, USA, April 13-17, 2003, Proceedings.* 373–390. https://doi.org/10.1007/3-540-36563-X_26

[24] Rosario Gennaro, Stanislaw Jarecki, Hugo Krawczyk, and Tal Rabin. 2007. Secure Distributed Key Generation for Discrete-Log Based Cryptosystems. *J. Cryptology* 20, 1 (2007), 51–83. https://doi.org/10.1007/s00145-006-0347-3

[25] Vassos Hadzilacos and Sam Toueg. 1994. *A modular approach to fault-tolerant broadcasts and related problems.* Technical Report. Cornell University.

[26] Timo Hanke, Mahnush Movahedi, and Dominic Williams. 2018. Dfinity technology overview series, consensus system. *arXiv preprint arXiv:1805.04548* (2018).

[27] Aniket Kate, Yizhou Huang, and Ian Goldberg. 2012. Distributed Key Generation in the Wild. *IACR Cryptology ePrint Archive* 2012 (2012), 377. http://eprint.iacr.org/2012/377

[28] Aggelos Kiayias, Alexander Russell, Bernardo David, and Roman Oliynykov. 2017. Ouroboros: A Provably Secure Proof-of-Stake Blockchain Protocol. In *Advances in Cryptology - CRYPTO 2017 - 37th Annual International Cryptology Conference, Santa Barbara, CA, USA, August 20-24, 2017, Proceedings, Part I.* 357–388. https://doi.org/10.1007/978-3-319-63688-7_12

[29] Ramakrishna Kotla, Lorenzo Alvisi, Michael Dahlin, Allen Clement, and Edmund L. Wong. 2009. Zyzzyva: Speculative Byzantine fault tolerance. *ACM Trans. Comput. Syst.* 27, 4 (2009), 7:1–7:39. https://doi.org/10.1145/1658357.1658358

[30] Jae Kwon and Ethan Buchman. [n. d.]. A Network of Distributed Ledgers. ([n. d.]). https://cosmos.network/cosmos-whitepaper.pdf

[31] Leslie Lamport. 1978. Time, Clocks, and the Ordering of Events in a Distributed System. *Commun. ACM* 21, 7 (1978), 558–565. https://doi.org/10.1145/359545.359563

[32] Arjen K. Lenstra and Benjamin Wesolowski. 2017. Trustworthy public randomness with sloth, unicorn, and trx. *IJACT* 3, 4 (2017), 330–343. https://doi.org/10.1504/IJACT.2017.10010315

[33] Andrew Miller, Yu Xia, Kyle Croman, Elaine Shi, and Dawn Song. 2016. The Honey Badger of BFT Protocols. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, Vienna, Austria, October 24-28, 2016.* 31–42. https://doi.org/10.1145/2976749.2978399

[34] Zarko Milosevic, Martin Hutle, and André Schiper. 2011. 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.* 235–244. https://doi.org/10.1109/SRDS.2011.36

[35] Louise E. Moser and P. M. Melliar-Smith. 1999. Byzantine-Resistant Total Ordering Algorithms. *Inf. Comput.* 150, 1 (1999), 75–111. https://doi.org/10.1006/inco.1998.2770

[36] Achour Mostéfaoui, Hamouma Moumen, and Michel Raynal. 2015. Signature-Free Asynchronous Binary Byzantine Consensus with t < n/3, O(n2) Messages, and O(1) Expected Time. *J. ACM* 62, 4 (2015), 31:1–31:21. https://doi.org/10.1145/2785953

[37] Satoshi Nakamoto. 2008. Bitcoin: A peer-to-peer electronic cash system. (2008).

[38] Rafael Pass and Elaine Shi. 2017. Hybrid Consensus: Efficient Consensus in the Permissionless Model. In *31st International Symposium on Distributed Computing, DISC 2017, October 16-20, 2017, Vienna, Austria.* 39:1–39:16. https://doi.org/10.4230/LIPIcs.DISC.2017.39

[39] Torben P. Pedersen. 1991. A Threshold Cryptosystem without a Trusted Party (Extended Abstract). In *Advances in Cryptology - EUROCRYPT '91, Workshop on the Theory and Application of of Cryptographic Techniques, Brighton, UK, April 8-11, 1991, Proceedings.* 522–526. https://doi.org/10.1007/3-540-46416-6_47

[40] Krzysztof Pietrzak. 2019. Simple Verifiable Delay Functions. In *10th Innovations in Theoretical Computer Science Conference, ITCS 2019, January 10-12, 2019, San Diego, California, USA.* 60:1–60:15. https://doi.org/10.4230/LIPIcs.ITCS.2019.60

[41] Serguei Popov. 2016. The tangle. *cit. on* (2016), 131.

[42] Adi Shamir. 1979. How to Share a Secret. *Commun. ACM* 22, 11 (1979), 612–613. https://doi.org/10.1145/359168.359176

[43] Ewa Syta, Philipp Jovanovic, Eleftherios Kokoris-Kogias, Nicolas Gailly, Linus Gasser, Ismail Khoffi, Michael J. Fischer, and Bryan Ford. 2017. Scalable Bias-Resistant Distributed Randomness. In *2017 IEEE Symposium on Security and Privacy, SP 2017, San Jose, CA, USA, May 22-26, 2017.* 444–460. https://doi.org/10.1109/SP.2017.45

[44] Benjamin Wesolowski. 2019. Efficient Verifiable Delay Functions. In *Advances in Cryptology - EUROCRYPT 2019 - 38th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Darmstadt, Germany, May 19-23, 2019, Proceedings, Part III.* 379–407. https://doi.org/10.1007/978-3-030-17659-4_13