Distributed Randomness from Approximate Agreement

Luciano Freitas ✉
LTCI, Télécom Paris, Institut Polytechnique de Paris

Petr Kuznetsov ✉
LTCI, Télécom Paris, Institut Polytechnique de Paris

Andrei Tonkikh ✉
LTCI, Télécom Paris, Institut Polytechnique de Paris

Abstract

Randomisation is a critical tool in designing distributed systems. The *common coin* primitive, enabling the system members to agree on an unpredictable random number, has proven to be particularly useful. We observe, however, that it is impossible to implement a truly random common coin protocol in a fault-prone asynchronous system.

To circumvent this impossibility, we introduce two relaxations of the perfect common coin: (1) *approximate common coin* generating random numbers that are *close* to each other; and (2) *Monte Carlo common coin* generating a common random number with an arbitrarily small, but non-zero, probability of failure. Building atop the *approximate agreement* primitive, we obtain efficient asynchronous implementations of the two abstractions, tolerating up to one third of Byzantine processes. Our protocols do not assume trusted setup or public key infrastructure and converge to the perfect coin exponentially fast in the protocol running time.

By plugging one of our protocols for *Monte Carlo common coin* in a well-known consensus algorithm, we manage to get a *binary Byzantine agreement* protocol with $O(n^3 \log n)$ communication complexity, resilient against an adaptive adversary, and tolerating the optimal number $f < n/3$ of failures without trusted setup or PKI. To the best of our knowledge, the best communication complexity for binary Byzantine agreement achieved so far in this setting is $O(n^4)$. We also show how the *approximate common coin*, combined with a variant of Gray code, can be used to solve an interesting problem of Intersecting Random Subsets, which we introduce in this paper.

2012 ACM Subject Classification Theory of computation → Design and analysis of algorithms

Keywords and phrases Asynchronous, approximate agreement, weak common coin, consensus, Byzantine agreement

1 Introduction

Generating randomness in distributed systems is an essential part of many protocols, such as Byzantine Agreement [4], Distributed Key Generation [20] or Leader Election [34]. Any application that needs an unpredictable or unbiased result will most likely rely on randomness. Although sometimes local sources of randomness are enough for some protocols [33], having access to a common random number can guarantee faster termination [3]. Producing a common unpredictable random number has been extensively studied in the literature on cryptography and distributed systems under the names of *random beacon*, distributed (multi-party) *random number generation* or *common coin* (even if the result is not binary). In essence, these protocols ensure:

Termination: every correct process eventually outputs some value;
Agreement: no two correct processes output different values;
Randomness: the value output by a correct process must be uniformly distributed over some domain $\mathcal{D}$, $|\mathcal{D}| \geq 2$.

We call a protocol that ensures the three properties (*Termination*, *Agreement*, and *Randomness*) a **perfect common coin**. There are many message-passing protocols without trusted setup that implement a perfect common coin in the presence of Byzantine adversary [40, 11, 24, 9, 7, 28, 18]. These protocols are either *synchronous*, meaning that every message sent by a correct process is delivered within a certain (known *a priori*) bound of time, or *partially-synchronous*, meaning that such a bound exists but is unknown.

In contrast, one can also consider an *asynchronous* system, where no bounds on communication delays can be assumed. In a seminal work of Fischer, Lynch, and Paterson, it has been shown that the problem of *consensus* has no asynchronous fault-tolerant solutions [17]. As we show in Appendix A, this impossibility also holds for perfect common coins: no algorithm can implement a perfect common coin in a message-passing asynchronous system where at least one process might crash. Note that this statement cannot be proven by a simple black-box reduction from consensus to a perfect common coin and a reference to FLP [17]. Indeed, if such a reduction existed, the resulting protocol would have to always terminate in a bounded number of steps, even with unfavourable outputs of the black-box common coin. Hence, if we were to replace the common coin protocol by a protocol that always returns 0, it would still provide termination as well as all other properties of consensus, violating [17].

Note that this impossibility applies even to systems with *trusted setup*, such as the one assumed in [9]. Such protocols typically do not satisfy the Randomness property of a perfect common coin. The outputs of these protocols follow deterministically from the information received by the processes during the setup.

In light of the impossibility of a perfect coin, one might look for *relaxed* versions of the common-coin problem that allow asynchronous fault-tolerant solutions. For example, one can relax the Agreement property by only requiring the output to be common with some constant probability, which results in an abstraction sometimes called *weak common coin*[1] or *δ-matching common coin*. In this paper, we call this abstraction a **probabilistic common coin**, in order to avoid confusion with other relaxations we introduce. More precisely, probabilistic common coins replace the Agreement property above with *probabilistic δ-consistency*.

Probabilistic δ-Consistency: with probability at least δ, no two correct processes output different values.

We also introduce the concept of a *Monte Carlo common coin* which is a probabilistic common coin whose success rate δ can be parameterized as follows: the more rounds of the protocol are executed, the more reliable the outcome is. In our case δ starts at $\frac{2}{3}$ in the first round of the protocol and converges to 1 at an exponential rate in the number of rounds. In most probabilistic common coins, δ could be increased by decreasing the resilience level (the allowed fraction of Byzantine processes). However, to the best of our knowledge, this paper is the first to present an implementation of a Monte Carlo common coin that can achieve arbitrarily small (but non-zero) δ by increasing the running time of the protocol (Sections 5 and 6).

We also propose a novel, alternative relaxation of Agreement: instead of ensuring that the *same* output is produced (with some probability), we may require that the produced outputs are *close* to each other according to some metric. For this variant, we have to also slightly relax randomness so that only one correct process is guaranteed to obtain a truly random value. More precisely, assume a discrete range of possible outputs $[0..D-1]$, and let

[1] Many weak common coin protocols such as the one in [10] also relax Randomness.

$d_q(x, y)$ denote the distance between x and y in the algebraic ring $\mathbb{Z}_q$[2]. The **Approximate common coin** abstraction then satisfies Termination and the following two properties:

Approximate ϵ-Consistency: if one correct process outputs value x and another correct process outputs y, then $d_D(x, y) \leq \lceil \epsilon D \rceil$, for a given parameter $\epsilon \in (0, 1]$;

One Process Randomness: the value output by *at least one* correct process must be uniformly distributed over the domain $[0..D-1]$.[3]

Our implementations of Monte Carlo and Approximate common coins build upon the abstraction of *Approximate Agreement* [15]. It appears that the abstraction perfectly matches the requirements exposed by our relaxed common-coin definitions: it naturally grasps the notion of outputs being close where the precision can be related to the execution time. Building atop existing asynchronous Byzantine fault-tolerant implementations [15, 1], we introduce and discuss an efficient implementation of the *bundled* version of this abstraction which is, intuitively, equivalent to n parallel instances of Approximate Agreement, but is much more efficient.

We discuss two applications of our protocols. First, we observe that our Monte Carlo common coin can be plugged into many existing Byzantine agreement protocols [6, 10, 13, 32]. This helps us to obtain a *binary Byzantine agreement* protocol with $O(n^3 \lambda \log n)$ communication complexity, where λ is the security parameter. The protocol exhibits optimal resilience of $f < n/3$, tolerates adaptive adversary, and assumes no trusted setup or PKI. In this setting, the best prior protocols for binary Byzantine agreement we are aware of have communication complexity of $O(n^4 \lambda)$ [27, 2].

We also introduce *Intersecting Random Subsets*, a new problem that can be used to asynchronously choose random committees with large intersections. Using elements of coding theory, namely Gray Codes [22, 36], we show how our Approximate common coin can be used to solve this problem without additional communication overhead.

We present our model definitions in Section 2 and describe the building blocks used in our constructions in Section 3. We describe our protocols in Sections 4 to 6, including implementation details and complexity analysis. In Section 7, we describe applications of our abstractions: binary Byzantine agreement and Intersecting Random Subsets. In Section 8, we overview the related work and in Section 9, we conclude the paper.

For completeness, we delegate all necessary complementary material to the appendix. In Appendix A, we prove the impossibility of an asynchronous perfect common coin. In Appendix B and Appendix C, we prove correctness of our approximate and Monte Carlo common coins, respectively. Appendix D presents two implementations of Random Secret Draw, one of the major building blocks of our common coins. Appendix E gives a modular formulation of a common coin proposed by Canetti and Rabin in 1993 [10]. Finally, we show how to build the codewords necessary for our Intersecting Random Subsets application in Appendix F.

2 System Model

We consider a system of n processes able to communicate using reliable communication channels. Among the participants, at most $f < \frac{n}{3}$ are Byzantine and might display arbitrary behaviour.

---

[2] I.e., $d(x, y) = \min\{|x - y|, q - |x - y|\}$.

[3] With a small modification to our protocol, we can easily achieve $(f+1)$-Process Randomness instead on One Process Randomness. However, we do not know if it possible to guarantee that all outputs of correct processes are random without relaxing other properties.

We assume the *adaptive* adversarial model: up to f Byzantine processes are chosen by the adversary depending on the execution. A non-Byzantine process is called *correct*. The communication complexity of our baseline protocols can be improved by a factor of n using Aggregatable Publicly Verifiable Secret Sharing (APVSS) [23]. However, as we are not aware of APVSS implementations that are secure against the adaptive adversary, the improved protocols can only be proved correct in the presence of the static one.

The adversary can control the time the messages sent by correct processes take to arrive, as well as reorder them. However, it cannot drop a message sent by a correct process unless it corrupts this process before the message has arrived.

We assume that each process has access to a local random number generator that can be accessed as follows:

RandomInt(D): produces a uniformly distributed random integer number in the range $[0..D-1]$.

The proposed protocols as well as some of the building blocks rely on the use of cryptographic hash functions. The hash of an arbitrary string s is denoted $H(s)$ and has length λ that we call the *security parameter*. It is computationally infeasible to find two strings $s \neq s'$ such that $H(s) = H(s')$, as well as inverting a hash without knowing which input was used a priori.

We assume a computationally bounded adversary, so that it is incapable of breaking cryptographic primitives with all but negligible probability. However, since such a probability exists, we allow the properties of all our protocols as well as all building blocks to be violated with a negligible in λ probability.

3 Building Blocks

Our protocols make use of a wide range of building blocks. None of them are completely new, but some of them are modified according to our needs. In particular, we introduce the *Random Secret Draw* abstraction inspired by the ideas from [10] and [2] (Section 3.3 and appendix D). We also provide a *bundled* version of the *Approximate Agreement* [15, 1] abstraction (Section 3.5). In addition, we use *Byzantine Reliable Broadcast* [6, 14] (Section 3.1), *Asynchronous Verifiable Secret Sharing* [8, 14] (Section 3.2), *Aggregatable Publicly Verifiable Secret Sharing* [23] (Appendix D.2.1), and *Gather* [10, 39] (Section 3.4).

3.1 Byzantine Reliable Broadcast

Byzantine Reliable Broadcast (BRB) [6] allows a designated leader to communicate a single message to all processes in such a way that, if any correct process delivers a message, then every other correct process eventually delivers exactly the same message (even if the leader is Byzantine). More precisely, a BRB protocol must satisfy the following properties:

Validity: if the leader is correct and it broadcasts message m, then every correct process will eventually deliver m;

Consistency: if two correct processes j and k deliver messages m_j and m_k, then $m_j = m_k$.

Totality: if a correct process j delivers some message m, then eventually all correct processes will deliver m.

The performance of reliable broadcast is of crucial importance to our protocols. We believe that the BRB implementation recently proposed by Das, Xiang, and Ren [14] will be the most suitable option. It has total communication complexity of just $O(n|M| + n^2\lambda)$,

where $|M|$ is the size of the message and λ is the security parameter, total message complexity of $O(n^2)$, and the latency of 3 message delays in case of a correct leader and 4 message delays in case of a Byzantine leader.

In this paper, we always use BRB in groups of n instances, with each process being the leader of one. We use the following notation:

BRB_i.Broadcast(m): allows process i to broadcast a message in an instance of BRB where i is the leader;

BRB_i.Deliver(m): an event indicating that message m from process i has been delivered.

3.2 Asynchronous Verifiable Secret Sharing

Asynchronous Verifiable Secret Sharing (AVSS) [8] allows a process to securely share information with other participants and to keep its contents secret until the moment a threshold of participants agree to open it.

In our protocols, AVSS is used with the following interface:

AVSS_i.ShareSecret(x): allows process i to share a secret x among the participants;

AVSS_i.SharingComplete(): an event issued when a secret is correctly shared by process i;

AVSS_i.EnableRetrieve(): enables responses to retrieval requests;

AVSS_i.Retrieve(): returns x if it was previously shared and all correct processes invoked AVSS_i.EnableRetrieve().

An AVSS implementation must satisfy the following properties:

Validity: if a correct process i invokes AVSS_i.ShareSecret(x), then every correct process eventually receives the AVSS_i.SharingComplete() event and no value other than x can be returned from the AVSS_i.Retrieve() operation invoked by a correct process;

Notification Totality: if one correct process receives the AVSS_i.SharingComplete() event, then every correct process eventually receives it;

Retrieve Termination: if all correct processes invoke AVSS_i.EnableRetrieve() and any correct process invokes AVSS_i.Retrieve(), then this operation will eventually terminate and the process will obtain the shared secret;

Binding: if some correct process receives the AVSS_i.SharingComplete() notification, then there exists a fixed secret x such that no value other than x can be returned from the AVSS_i.Retrieve() operation invoked by a correct process;

Secrecy: if process i is correct and no correct process invoked AVSS_i.EnableRetrieve(), then the adversary has no information about the secret shared by i.

Das, Xiang, and Ren [14] proposed an AVSS protocol with quadratic communication complexity, constant latency, and without assuming trusted setup. Notice that in order to secretly share a long string s, it is better to follow the method proposed in [29]: encrypt s using a much shorter secret key *sym*, reliably broadcast the encrypted value $\{s\}_{sym}$ and then perform secret sharing of the key *sym*. Thus, we shall assume that the total communication complexity of secret sharing of string s is $O(n|s| + n^2\lambda)$.

3.3 Random Secret Draw

One of the key ideas of the weak common coin protocol of Canetti and Rabin [10] is to *assign* each process a random number in a given domain $[0..D-1]$ in such a way that:

Assignment Termination: if a correct process i participates, then it is eventually assigned a value. Moreover, everyone will eventually receive a notification that i has been assigned a random value;

Notification Totality: if process i receives a notification that some process j has been assigned a value, then every correct process will eventually receive such a notification;

Randomness: the assigned numbers are independent and distributed uniformly over the domain $[0..D-1]$. The distribution of the value assigned to process j cannot be affected by the adversary even if j itself is Byzantine;

Unpredictability: until at least one correct process agrees to reveal the assigned values, the value assigned to each process j remains secret, even to process j itself;

Reveal Termination: if all correct processes want to reveal the assigned values, then they will eventually succeed.

Although this idea has been widely used as part of the implementation of asynchronous consensus protocols, to the best of our knowledge, it was never considered a separate primitive and assigned a name. Hence, we shall call it *Random Secret Draw (RSD)*.

This abstraction resembles a well known concept of a Verifiable Random Function (VRF) [31]. However, the important difference is that process j itself cannot know the value it is assigned until the reveal phase. Hence, a Byzantine process cannot choose whether it wants to participate or not based on the random value it is assigned. Moreover, unlike Random Secret Draw, VRF schemes typically require a seed chosen at random *after* the process chose the public key for its pseudo-random function. In fact, a variant of RSD has been recently used to generate such seeds [19].

We use the following interface for the RSD abstraction:

RSD.Start(): allows a process to start participating in RSD and, eventually, to be assigned a random number. We assume that this function is non-blocking, i.e., that an invocation of this function terminates after 0 message delays;

RSD.EnableRetrieve() : used by the processes to start participating in the process of reconstructing the assigned values;

RSD.RetrieveValues(S): returns a map from the ids of processes in the set S to the assigned values if all correct processes invoked RSD.EnableRetrieve() and all processes in S have been assigned some values.

The original RSD implementation by Canetti and Rabin [10] used n^2 instances of AVSS. To the best of our knowledge, to this day, there is no known AVSS protocol that would allow to do it with less than $\Omega(n^4)$ bits of communication in total. We, therefore, give two possible implementations. The first one is secure against an adaptive adversary and does not rely on PKI, while the second one uses the implementation from [2] that relies of *Aggregatable Publicly Verifiable Secret Sharing* (described in Appendix D.2.1) instead of AVSS. While saving a linear factor in communication complexity, this solution lacks security against adaptive adversary and requires PKI. Since both [10] and [2] did not considered RSD as a separate abstraction and did not provide separate pseudocode for it, we present both RSD implementations in Appendix D.

3.4 Gather

Yet another important contribution made by Canetti and Rabin in their weak common coin construction [10] is a multi-broadcast protocol that has been recently given the name *Gather* [39, 2]. In this protocol, every process starts by broadcasting a single message through Byzantine Reliable Broadcast. The processes then do a few more rounds of message exchanges and, in the end, each participant i outputs a set of process ids $\mathcal{S}_i$ such that for all $j \in \mathcal{S}_i$:

i has received the message of j through reliable broadcast.[4] Moreover, the sets output by correct processes satisfy a strong intersection property:

Binding Common Core: There exists a set $\mathcal{S}^*$ of process ids of size at least $n - f$, called the *common core*, such that for every correct process i: $\mathcal{S}^* \subseteq \mathcal{S}_i$. Moreover, once the first correct process outputs, $\mathcal{S}^*$ is fixed and the adversary cannot manipulate it anymore.

The fact that the adversary cannot affect the common core once a single correct process outputs will be important in our protocols. The adversary should not be able to choose the common core based on the generated random numbers after some of the correct processes invoked EnableRetrieve.

We slightly generalize the interface of Gather by using it in conjunction with BRB, but also with other similar primitives (in particular, AVSS and RSD) and their combinations. When a process invokes Gather, it passes to it an arbitrary callable function GatherAccept that takes a process id j and returns *true* if the message from this process is considered to be delivered (not necessarily through BRB). We assume that Gather exports the following interface:

Gather.Start(GatherAccept): allows a process to start participating in the Gather protocol;
Gather.DeliverSet(S): provides the output of the Gather protocol.

In order for the protocol to terminate, the GatherAccept function has to satisfy properties similar to those of reliable broadcast.

Accept Validity: if a correct process i invoked Gather.Start, then for every correct process j, GatherAccept(i) invoked by process j must eventually return *true*. Moreover, for all i, once GatherAccept(i) returned to *true* to some correct process, it must never switch back to *false*;

Accept Totality: if GatherAccept(i) invoked by one correct process returned *true*, then eventually it must return *true* to all correct processes.

Thanks to the properties of AVSS and Random Secret Draw (in particular, to the Notification Totality property), in our protocols, this assumption is trivially satisfied. For Gather, we use the original protocol of [10] (to the best of our knowledge, it was first described as a separate primitive in [39]).

3.5 Bundled Approximate Agreement

The last building block that we shall need is *Approximate Agreement (AA)* [16]. In a (one-dimensional) AA instance, the processes propose inputs and produce outputs (real values) so that the following properties are satisfied:

Validity: the outputs of correct processes must be in the range of inputs of correct processes.

Approximate $\tilde{\epsilon}$-consistency: the values decided by non-faulty processes must be at most a distance $\tilde{\epsilon}$ apart from each other.[5]

Termination: every non-faulty process eventually decides.

[4] In [39] and [2], Gather returns a set of pairs $(id, value)$. However, for our purposes, working with sets of ids is more convenient. The values will be delivered through normal BRB.Deliver event.

[5] We use $\tilde{\epsilon}$ to distinguish it from ϵ used in the definition of Approximate Common Coin.

In our algorithms, Approximate Agreement is always executed in *bundles* of n parallel instances. For the sake of efficiency, one can treat it as a bundled problem with an input vector of size n, corresponding to the different instances and then, for every message, send information about all instances at the same time, but treat them separately as before. We call this abstraction *Bundled Approximate Agreement* (BAA). BAA **should not be confused** with *Multidimensional Approximate Agreement* [30], which is a stronger abstraction than the one we rely upon.

Assuming binary inputs, the processes access BAA via the following interface:

BAA.Run($[x_1, x_2, \ldots, x_N]$)**:** Launches n instances of Approximate Agreement protocol, where the input for the i-th instance is x_i. For a given parameter $\tilde{\epsilon}$, the protocol is executed until $\tilde{\epsilon}$-approximation is satisfied in every instance and then returns a vector of outputs $[y_1, y_2, \ldots, y_N]$.

For implementing BAA, we suggest using the Approximate Agreement protocol proposed in [1] with resilience $f < \frac{n}{3}$. Since in our protocols, the inputs are either 0 or 1, we do not need the termination detection techniques described in [1] neither do we need the "init" phase of the protocol. With the aforementioned *BRB* and some trivial changes[6], this implementation will give us the communication complexity of $O(n^3\lambda)$ and latency $4 \cdot \log_2(1/\tilde{\epsilon})$.

4 Approximate Common Coin

Algorithm 1 Approximate common coin

Instance parameters: domain D, precision ϵ

Distributed objects:
- $\forall j \in [n]$: AVSS_j – instance of Asynchronous Verifiable Secret Sharing with leader j
- Gather – instance of Gather
- BAA – instance of Bundled Approximate Agreement with precision $\tilde{\epsilon} = \epsilon / f$

function GatherAccept(j)
- **return** *true* if received AVSS_j.SharingComplete()

operation Toss() **returns** integer
- $x = \text{RandomInt}(D)$
- AVSS_i.ShareSecret(x)
- Gather.Start(GatherAccept)
- **wait for** Gather.DeliverSet(S)
- $\forall j \in [n]$: let $w_j = \begin{cases} 1, & j \in S, \\ 0, & otherwise \end{cases}$
- $[w'_1, \ldots, w'_n] = \text{BAA.Run}([w_1, \ldots, w_N])$
- $\forall j \in [n]$: AVSS_j.EnableRetrieve() // only after BAA completes
- $\forall j \in [n]$: let $x_j = \begin{cases} \text{AVSS}_j\text{.Retrieve}(), & if\ w'_j \neq 0 \\ 0, & otherwise \end{cases}$
- // Compute and return the final random number
- **return** $\left(\left\lceil \sum_{j \in [n]} x_j \cdot w'_j \right\rceil \bmod D \right)$

The main idea of this protocol is to aggregate numbers locally generated by enough different processes so that at least one of them is correct and the number it generated is truly

[6] In the 'report' messages, hashes of the values should be sent instead of the values themselves.

random and uniformly distributed. With a good aggregation function, the resulting value will also be uniformly distributed. An example of such an aggregation function is addition modulo the size of the domain. Indeed, it is easy to see that, if x is uniformly distributed over $[0..q-1]$ and y is any number chosen independently of x, then $(x+y) \bmod q$ will also be uniformly distributed over $[0..q-1]$. Another example of an aggregation operation that satisfies a similar property is bit-wise xor (as long as the domain size is a power of 2).

However, without being able to solve consensus, we cannot just elect $f+1$ or $n-f$ processes from whom we shall take these values. Thankfully, unlike xor, addition has one more useful property: it is continuous. If we take two numbers, x and y, such that $d_q(x,y) \leq \alpha$, then for any number z, $d_q(z+x, z+y) \leq \alpha$.[7] Hence, a natural idea is to *approximately* elect the set of processes to provide the random inputs.

More precisely, in order to produce an *approximate common coin* in the range $[0..D-1]$, each process locally generates and secret-shares a random number in this range. Then each process gathers a set of ids of processesthat have completed the sharing (line 12).

The next step is to create a binary vector with n positions, where each position j is set to 1 if and only if j is present in the gathered set (line 13). This vector is then used as an input for the BAA protocol (line 14), which outputs a vector W of *weights* such that for each position j, the weights received by different processes are at most $\tilde{\epsilon}$ apart.

The value of $\tilde{\epsilon} = \epsilon/f$ is chosen such that the final outputs of the coin are at most ϵ apart from each other. For the details on how this particular value was computed, see Appendix B.3.

Recall that BAA ensures that the output values lie within the range of of inputs of correct processes. Moreover, by the properties of Gather and AVSS, if at least one correct process has j in its gathered set, then j has correctly shared its value and it can be later retrieved by the correct processes. Therefore if the j-th value is irretrievable, the j-th component will always be assigned weight 0. On the other hand, due to the common core property of Gather, at least $n-f$ values will have weight 1, which, as demonstrated in Appendix B.2, guarantees that the result is uniformly distributed in the desired range.

Finally, the processes reveal the secrets (lines 15 and 16) and compute the resulting random number.

▶ **Theorem 1.** *Algorithm 1 implements an approximate common coin.*

Proof. Appendix B shows that the algorithm guarantees *Termination*, *One process randomness* and *Approximate ϵ-consistency.* ◀

Complexity analysis. The communication complexity of our *approximate common coin* can be broken down in:

1. n instances of AVSS.ShareSecret in parallel $\Rightarrow O(n^3\lambda)$;
2. One instance of Gather $\Rightarrow O(n^3\lambda)$;
3. One instance of BAA with $\tilde{\epsilon} = \epsilon/f \Rightarrow O(n^3\lambda(\log f + \log \frac{1}{\epsilon}))$;
4. n instances of AVSS.Retrieve in parallel $\Rightarrow O(n^3\lambda)$;

Hence, the total communication complexity is $O(n^3\lambda(\log f + \log \frac{1}{\epsilon}))$ with Bundled Approximate Agreement being the bottleneck.

The time complexity of the protocol is $O(\log f + \log \frac{1}{\epsilon})$.

[7] Recall that $d_q(x,y)$ is the distance between x and y in the ring $\mathbb{Z}_q$, i.e., $d_q(x,y) = \min\{|x-y|, q-|x-y|\}$.

5 Monte Carlo Common Coin from Approximate Common Coin

Algorithm 2 Monte Carlo Common Coin from Approximate Common Coin, code for process i

19: **Instance parameters:** domain size D, success probability δ

20: let $k = \lfloor \frac{2}{1-\delta} \rfloor$

21: **Distributed objects:**
22: AC – instance of approximate common coin with domain size kD and precision $\epsilon = \frac{1}{kD}$

23: **operation** Toss() **returns** integer
24: **return** $\left\lfloor \frac{\text{AC.Toss()}}{k} \right\rfloor$

In this section, we present a simple reduction from an approximate common coin to a Monte Carlo common coin. The very short pseudocode is in Algorithm 2.

The transformation requires first to generate an approximate common coin of domain kD where k is an integer number $\lfloor \frac{2}{1-\delta} \rfloor$ and $\epsilon = \frac{1}{kD}$. This implies that different processes shall get values at most $\left\lceil \frac{1}{kD} \cdot kD \right\rceil = 1$ apart.

The domain of the approximate coin is k times larger than the domain of the targeted Monte Carlo common coin. By dividing the result by k, we get the desired range of values and success probability δ, where k values of the *approximate common coin* are mapped to one value of *Monte Carlo common coin*.

▶ **Theorem 2.** *Algorithm 2 implements a Monte Carlo common coin with domain D and success probability δ.*

Proof. Termination and Unpredictability follow from the properties of *approximate common coin*, while Randomness follows from One Process Randomness since exactly k values from the larger domain (kD) are mapped to each value in the smaller domain (D). Let x' be the resulting toss of the first correct process that completes BAA in its approximate common coin toss. Then, as established, every other process will be at a distance at most 1 from it. Hence, if the remainder of the division of x' by k is neither 0 nor $k-1$, every correct process decides the same value:

$$1 - \delta \geq \frac{2}{k} = \frac{2}{\lfloor \frac{2}{1-\delta} \rfloor} \geq 1 - \delta$$

◀

Complexity analysis. This protocol runs a single instance of an Approximate Common Coin, with precision $\epsilon = \frac{1}{D\lfloor \frac{2}{1-\delta} \rfloor}$. If used with our algorithm from Section 4, it will take $O(\log f + \log \frac{1}{\epsilon}) = O(\log f + \log D + \log \frac{1}{1-\delta})$ rounds of approximate agreement.

Hence, the overall time complexity of the protocol is $O(\log f + \log D + \log \frac{1}{1-\delta})$ and the communication complexity is $O(n^3 \lambda (\log f + \log D + \log \frac{1}{1-\delta}))$.

6 Direct Implementation of Monte Carlo Common Coin

Overview. The main idea of this protocol is to assign to each process a random value and a random ticket. Then, using approximate agreement, the protocol is able to select the

Algorithm 3 Monte Carlo Common Coin, code for process i

25: **Instance parameters:** domain size D, success probability δ, security parameter λ

26: **Functions:**
27: CALIBRATE(w) – returns the weight to apply to a ticket given that BAA returned w

28: **Distributed objects:**
29: TicketDraw – instance of Random Secret Draw with domain size 2^λ
30: ValueDraw – instance of Random Secret Draw with domain size D
31: Gather – instance of Gather
32: BAA – instance of Bundled Approximate Agreement with precision $\tilde{\epsilon}$,
33: where $\tilde{\epsilon}$ depends on the calibration function (see "Weight calibration" below)

34: **function** GATHERACCEPT(j) **returns** boolean
35: **return** *true* iff received both TicketDraw.VALUEASSIGNED(j) and ValueDraw.VALUEASSIGNED(j)

36: **operation** TOSS() **returns** integer
37: TicketDraw.START()
38: ValueDraw.START()
39: Gather.START(GATHERACCEPT)

40: **wait for** event Gather.DELIVERSET(S)
41: $\forall j \in [n]$: let $w_j = \begin{cases} 1, & j \in S, \\ 0, & otherwise \end{cases}$
42: $[w'_1, \ldots, w'_n] =$ BAA.RUN($[w_1, \ldots, w_N]$)
43: $candidates = \{j \in [n] \mid w'_j > 0\}$

44: TicketDraw.ENABLERETRIEVE()
45: ValueDraw.ENABLERETRIEVE()
46: $tickets =$ TicketDraw.RETRIEVEVALUES($candidates$)
47: $values =$ ValueDraw.RETRIEVEVALUES($candidates$)

48: $winner = \underset{j \in candidates}{\arg\max}$ CALIBRATE(w'_j) $\cdot$ $tickets[j]$
49: **return** $values[winner]$

process with maximum ticket with adjustable probability of success and adopt the value corresponding to this ticket as the coin value.

Tickets and values. The protocol first assigns random values and random tickets to each participant, maintaining both secret until later (lines 37 and 38). Processes then gather a list of participants who have both drawn a ticket and a value, guaranteeing that all processes will hold sets that all intersect in at least $n - f$ participants (line 39).

Approximate Agreement. In a similar manner as in the previous protocol, each process runs Bundled Approximate Agreement inputting 1 in the dimensions corresponding to the processes it has received from Gather, and 0 in the other dimensions (line 42). Similar to the previous protocol, if a process has not made a valid draw it will always be assigned weight zero, whereas if the weight is different than zero then it is possible to recover the secretly drawn number.

Opening the secrets. Prior to the first decision of a correct process in BAA, no secrets are leaked, as the underlying Random Secret Draw abstraction requires at least one correct process to invoke the ENABLERETRIEVE operation before any information about the generated numbers is revealed. After this first decision of a correct process, the secrets can be opened (line 45), but at this point the adversary can only induce other processes deciding values

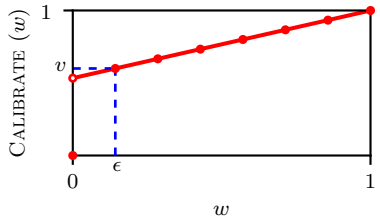

Figure 1 The weight calibration function.

which are at most $\tilde{\epsilon}$ apart from the first decision, which does not undermine the safety of the protocol.

Decision. With the tickets and values now openly available, the processes calibrate the tickets by multiplying the plain ticket by a *calibration function* applied to the weights. The simplest calibration function is an identify function, the calibrated ticket of a process i is simply the product of the output i-th output of BAA and the original ticket t_i. In their final steps, processes decide the value corresponding to the highest calibrated ticket (lines 48 and 49).

Weight calibration. As shown in Appendix C.3, the protocol without weight calibration (i.e., with $\textsc{Calibrate}(w) = w$) requires $3 + \lceil \log_2(n) + \log_2\left(\frac{1}{1-\delta}\right)\rceil$ rounds of approximate agreement in order to achieve the success probability δ. A similar performance is achieved by the protocol in Section 5.

In order to get rid of the $\log_2(n)$ part in the time complexity, we use a calibration function that is linear on $(0, 1]$ with a discontinuity point at 0, as illustrated in Figure 1. If $w_1 > 0$ and $w_2 > 0$ and $|w_1 - w_2| = \tilde{\epsilon}$, then, after calibration, $|\textsc{Calibrate}(w_1) - \textsc{Calibrate}(w_2)| \approx \tilde{\epsilon} \cdot (1 - v)$. This is similar in effect to running extra $\log_2 \frac{1}{1-v}$ rounds of approximate agreement, but at no extra latency cost. We then balance the value of the parameter v in such a way that, intuitively, the discontinuity at 0 is very unlikely to cause disagreement. An example of a good value for v that achieves this goal is $1 - \frac{\ln(2/(1-\delta))}{2n/3}$. A detailed proof of the solution with weight calibration is presented in Appendix C.4. In order to achieve success probability δ, $5 + \left\lceil \log_2\left(\frac{1}{1-\delta}\right) + \log_2\left(\log_2\left(\frac{1}{1-\delta}\right)\right)\right\rceil$ rounds of approximate agreement are required.

▶ **Theorem 3.** *Algorithm 3 implements a Monte Carlo common coin.*

Proof. Appendix C shows that the algorithm guarantees *Termination*, *Uniform distribution*, and *Probabilistic δ-consistency.* ◀

Complexity analysis. The communication complexity of our *Monte Carlo common coin* can be broken down in:

1. 2 instances of Random Secret Draw $\Rightarrow O(n^3\lambda)$;
2. 1 instance of Gather $\Rightarrow O(n^3\lambda)$
3. One instance of BAA with $\tilde{\epsilon} = O(1/(1-\delta)) \Rightarrow O(n^3\lambda \log \frac{1}{1-\delta})$;
4. $2n$ secret retrievals in parallel $\Rightarrow O(n^3\lambda)$;

Hence, the total communication complexity is $O(n^3\lambda \log \frac{1}{1-\delta})$ with Bundled Approximate Agreement being the bottleneck. The time complexity of the protocol is $O(\log \frac{1}{1-\delta})$.

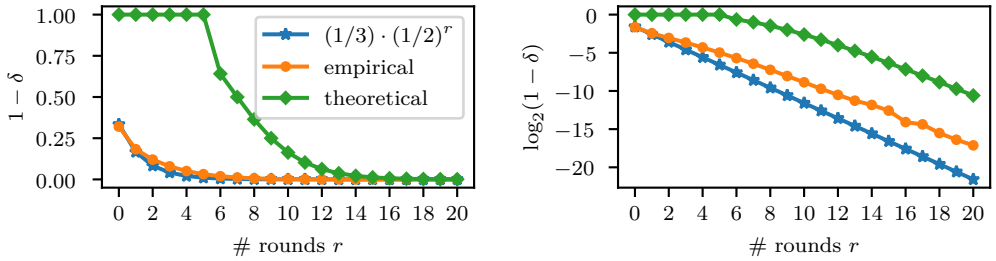

Figure 2 Empirical estimate of the failure probability of the Monte Carlo common coin protocol with weight calibration, for $n = 50$, compared to the theoretical estimate.

Empirical performance analysis via simulation. In Appendix C.4, we proved that our Monte Carlo common coin with a calibration function achieves time complexity of $O(\log \frac{1}{1-\delta})$. However, the derived upper bound on the exact number of rounds of approximate agreement $\left(5 + \left\lceil \log_2\left(\frac{1}{1-\delta}\right) + \log_2\left(\log_2\left(\frac{1}{1-\delta}\right)\right)\right\rceil\right)$ is clearly pessimistic. To get a better understanding of the actual performance of the protocol, we performed simulations.

Figure 2 presents the empirical estimate of failure probability of our protocol compared to the theoretical value and the "ideal" value of $(1/3)\cdot(1/2)^r$, where r is the number of rounds of approximate agreement. Each point on the "empirical" curve is based on 3 experiments and each experiment consists of 1 million simulated executions. In each execution, $n = 50$ processes are assigned random tickets and the adversary wins if it manages to get processes in disagreement on which process has the maximum ticket. The result of each experiment is the ratio between the number of executions where the adversary wins to the total number of executions (1 million). Then we average over 3 such experiments to get the value for each point on the curve.

The estimate for the optimal value of the parameter v in the calibration function for each number of rounds was also computed based on simulations (independent from the ones used to estimate the final failure probabilities). Note that it may only increase the estimated empirical latency compared to the (unknown) optimal value for v.

7 Applications

7.1 Binary Byzantine Agreement

Monte Carlo common coin can be plugged into any Byzantine Agreement protocol that makes a call to a probabilistic common coin, such as [6, 10, 13, 32].

Using our Monte Carlo common coin obtained via the transformation from approximate common coin (Section 5), we get a protocol that is secure against an adaptive adversary, assumes no trusted setup or PKI, and exhibits communication complexity $O(n^3\lambda \log n)$ at the expense of extra $O(\log n)$ factor in time complexity. As far as we know, the best existing setup-free solutions that are resilient against adaptive adversary and tolerate up to $f < n/3$ failures exhibit communication complexity of $O(n^4\lambda)$ [27, 2].

7.2 Intersecting Random Subsets

A direct application of an approximate common coin is a problem we call *intersecting random subsets*. This problem consists of given a globally known set S of cardinality n, to chose a subset $s \subseteq S$ of cardinality $m \leq n$.

The following variation of Gray Codes [22, 36] will be instrumental:

▶ **Definition 4.** *Code $C_{n,m}$ is a list of $\binom{n}{m}-1$ binary strings (called* code words*) satisfying the following conditions:*

- *for all i, $C_{n,m}[i]$ is a string of m ones and $n-m$ zeros;*
- *every string of m ones and $n-m$ zeros is present in $C_{n,m}$ exactly once;*
- *$\forall i \in \left\{1,\ldots,\binom{n}{m}-1\right\} : C_{n,m}[i]$ and $C_{n,m}[i-1]$ differ in two bits;*
- *$C_{n,m}[\binom{n}{m}-1]$ and $C_{n,m}[0]$ differ in two bits.*

In Appendix F, we provide a concrete code that satisfies these properties and an algorithm (in the form of a recursive formula) that allows to efficiently generate a code word by its index.

Intuitively, this code is composed of binary strings of length n and it can be read in the following manner: if the i-th position of the string is 1, then i-th element is selected, otherwise it is considered to be left outside. Moreover, this code has the property that consecutive numbers differ only by swapping exactly one position set to 1 with a position marked with a 0. Therefore all consecutive subsets have the same fixed size and include $s-1$ common elements and differ by only one.Hence, by generating an approximate common random coin over the domain $\left\{0..\binom{n}{m}-1\right\}$ with parameter $\epsilon \leq k \cdot \binom{n}{m}^{-1}$, processes can select subsets of size m differing by at most k elements.

This could be interesting for selecting a committee among n users in scenarios subject to a mobile Byzantine adversary, i.e. on systems where the set of processes who display malicious behaviour changes, provided the time to corrupt a majority of processes in any given committee is higher than an asynchronous round. Note that this solution provides an interesting alternative to committee elections in protocols such as Algorand [21]. It not only is completely asynchronous, but it also guarantees a fixed committee size and provides a way to control the intersection of quorums obtained by different users. Recall that in the case of Algorand, with non-zero probability, it might happen that quorums do not intersect at all.

8 Related Work

Ben-Or [4] proposed the first randomized consensus algorithm based on the "independent choice" common coin. In this algorithm, every participant tosses a local random coin and with probability 2^{-n}, the values picked by n participants are identical. Bracha [6] extended this algorithm to the Byzantine fault model with $f < n/3$ faulty participants.

Rabin [37] proposed an implementation of a perfect common coin based on Shamir's secret sharing [38], assuming trusted setup (a trusted dealer distributes *a priori* a large number of secrets). Today, most protocols that allow trusted setup use the solution proposed by Cachin et al. [9] who have described a practical perfect common coin based on threshold pseudorandom functions (tPRF), assuming that a trusted dealer distributes a short tPRF key. These protocols use the pre-distributed randomness in a clever way to obtain multiple numbers that are computationally indistinguishable from random (much like a PRNG [5]). In contrast, our algorithms do not assume trusted setup.

In the setup-free context, Canetti and Rabin [10] proposed a weak common coin algorithm based on asynchronous verifiable secret sharing (AVSS), which resulted in an efficient randomized *binary* consensus. In designing our protocols, we make use of multiple ideas from this work, taking into account recent improvements, such as the use of Aggregatable Publicly Verifiable Secret Sharing [23], suggested in a similar context by Abraham et al. [2]. We give a modern version of their coin using our building blocks in Appendix E.

Using standard PKI cryptography, Cohen et al. [12] built two common coins relying on VRFs. The first one with resilience $f < (1/3 - \epsilon)$ achieves success rate $\delta = \frac{18\epsilon^2+24\epsilon-1}{6(1+6\epsilon)}$. The second coin involves sampling committees among the processes and also guarantees a constant success rate that depends on the system's resilience.

Kogias et al. [27] proposed a relaxed abstraction called *eventually perfect common coin*. They first build a weak distributed key generator (wDKG) which is a protocol that never terminates: each party outputs a sequence of candidate keys to be used for encryption and decryption with the property that they will eventually agree on a set of keys. This mechanism can replace the trusted setup in [9]. Moreover, it can be shown that the participants may disagree on the set of keys at most $f + 1$ times. The resulting coin eventually terminates with a perfect result, hence its name. In contrast, the challenge of our work was to devise (one-shot) unbiased common coins with provable termination.

Gao et al. [19] combined VRFs and AVSS to produce the first random coin which has $O(n^3\lambda)$ communication complexity. With the advent of new broadcast and APVSS implementations, the classic protocol of [10] gains the same complexity (see Appendix E). They created a weak form of *Gather* called *core-set selection* in which $f+1$ correct participants share at least $n - f$ VRFs coming from different processes. They then use AVSS to build the seeds for VRFs and whenever the highest VRF is in the common core, the nodes successfully agree in the coin outcome. Their protocol assumes the static adversary, but it can be made adaptive with a *relatively weak* form of trusted setup: a single common random number must be published after the public keys of the participants are fixed. In contrast, our protocols do not assume any form of trusted setup. Assuming static adversary, our protocols can achieve the same communication cost while additionally enabling parameterized success rate.

Our Monte Carlo coin from Section 6 was inspired by the Proposal Election protocol recently introduced by Abraham et al. [2]. Technically, it is not a common coin *per se* but it uses elements of it. In this protocol, every party inputs some externally valid value, and with a constant probability, all parties output the same value that was proposed by a non-faulty party chosen at random. Intuitively, the main contribution of the protocol in Section 6 is the use of approximate agreement to *amplify* the success probability.

In the *full information* model, without using cryptography, King and Saia [26] observed that the strength of the Byzantine adversary is in its anonymity, but it cannot bias the coin indefinitely without being detected. Even though their Byzantine agreement algorithm with polynomial expected time does allow the adversary to bias the coin, but amended this with statistical tests aiming at detecting this kind of deviation and evicting misbehaving participants. The resilience level of this algorithm is, however, orders of magnitude lower than n ($f < n/400$ in the best case). Huang et al. [25] recently extended this work to achieve the resilience of $f < n/4$. In contrast, our algorithms use cryptographic tools to produce unbiased (approximate) outputs and maintain optimal resilience $f < n/3$.

Monte Carlo protocols are randomized algorithms that have a fixed number of rounds and yield results that are correct with a given probability, while Las Vegas protocols always give the correct results but do not have a fixed number of rounds. Notice that Las Vegas algorithms must have a fixed probability of terminating every round and thus can be converted into Monte Carlo by stopping after a fixed number of rounds and deciding a random value if termination is not attained.

Such a transformation could be applied to [27], but since their latency is a function of $O(f)$, the resulting Monte Carlo common coin would also have a latency which is a function of f, while our solution is independent of this parameter. Another option would be to create a set of keys using [2] which could be run a fixed number of rounds and then to use [9]. Since

their expected number of rounds is not a function of f, this transformation would have the same asymptotic complexity as ours, but it would include many unnecessary message delays from the consensus protocol, from the verifiable gather and other parts of their ADKG that are not present in a direct implementation such as ours.

9 Conclusion

In this paper, we suggest 2 new relations of the common coin primitive implementable in a fully asynchronous environment. We provide efficient implementations based on a range of novel techniques. Our protocols are the first use of approximate agreement to generate random numbers, which is used to keep decided values close in the approximate common coin protocol and to increase the probability of agreement in the direct implementation of the Monte Carlo common coin. Moreover, we also introduced elements of coding theory that were not previously applied to the distributed computing realm in the solution of what we called intersecting random subsets.

Further studies are necessary to explore the full potential of using these new abstractions in the design of distributed protocols and to understand the theoretical limits of their performance.

Tight performance analysis for probably common coin. In Appendices C.3 and C.4, we proved that in order to achieve success probability δ, our probably common coin protocol requires $3 + \lceil \log_2(n) + \log_2\left(\frac{1}{1-\delta}\right)\rceil$ or $5 + \left\lceil \log_2\left(\frac{1}{1-\delta}\right) + \log_2\left(\log_2\left(\frac{1}{1-\delta}\right)\right)\right\rceil$ rounds of approximate agreement, depending on whether weight calibration is used. While it seems to correctly reflect the asymptotic behaviour of the actual distribution, these bounds seem to be rather pessimistic when only a few rounds of approximate agreement are run.

For example, according to these bounds, with weight calibration, we will need 8 rounds in order to achieve $\delta = \frac{2}{3}$. However, in practice, $\delta = \frac{2}{3}$ is achieved with 0 rounds of approximate agreement as with probability at least $2/3$ a process in the common core will have the largest ticket. As we estimated empirically through simulations, the actual δ that we obtain after running 8 rounds of approximate agreement is around 0.993 instead of $2/3$.

Non-linear calibration function for Monte Carlo common coin.. Weight calibration is necessary to achieve the latency of $O(\log \frac{1}{1-\delta})$ rounds in our Monte Carlo common coin protocol. We chose a concrete linear function because it was relatively simple to analyze (as we could do a reduction to the case without the calibration). However, this function is unlikely to be optimal. The extra $\log_2(\log_2 \frac{1}{1-\delta})$ part in the resulting estimate on the number of rounds of approximate agreement is likely to be due to sub-optimal choice of the weight calibration function.

Approximate common coin without extra $\log_2(f)$ rounds. Using the magic of weight calibration, for Monte Carlo common coin, we managed to achieve $O(\log(1/\epsilon))$ time complexity, which is likely to be optimal. However, our approximate common coin protocol requires $\log_2 f + \log_2(1/\epsilon)$ rounds of approximate agreement and, hence, its time complexity depends on two variables: f and ϵ. In some applications, ϵ may be constant and $\log_2(f)$ can become the bottleneck.

Creating a protocol without this extra delay or proving a $\Omega(\log(f))$ lower bound would be an interesting contribution to the understanding of the approximate common coin abstraction.

It would also mean that the transformation from approximate common to Monte Carlo common would result in more efficient coins of the latter type.

References

1 Ittai Abraham, Yonatan Amit, and Danny Dolev. Optimal resilience asynchronous approximate agreement. OPODIS'04, page 229–239, Berlin, Heidelberg, 2004. Springer-Verlag. doi:10.1007/11516798_17.

2 Ittai Abraham, Philipp Jovanovic, Mary Maller, Sarah Meiklejohn, Gilad Stern, and Alin Tomescu. Reaching consensus for asynchronous distributed key generation. In *Proceedings of the 2021 ACM Symposium on Principles of Distributed Computing*, pages 363–373, 2021.

3 Marcos K Aguilera and Sam Toueg. The correctness proof of ben-or's randomized consensus algorithm. *Distributed Computing*, 25(5):371–381, 2012.

4 Michael Ben-Or. Another advantage of free choice: Completely asynchronous agreement protocols (extended abstract). In *PODC '83: Proceedings of the annual ACM symposium on Principles of distributed computing*, pages 27–30, 1983.

5 Manuel Blum and Silvio Micali. How to generate cryptographically strong sequences of pseudorandom bits. *SIAM journal on Computing*, 13(4):850–864, 1984.

6 Gabriel Bracha. Asynchronous byzantine agreement protocols. *Information and Computation*, 75(2):130–143, 1987.

7 Benedikt Bünz, Steven Goldfeder, and Joseph Bonneau. Proofs-of-delay and randomness beacons in ethereum. *IEEE Security and Privacy on the blockchain (IEEE S&B)*, 2017.

8 Christian Cachin, Klaus Kursawe, Anna Lysyanskaya, and Reto Strobl. Asynchronous verifiable secret sharing and proactive cryptosystems. In *Proceedings of the 9th ACM Conference on Computer and Communications Security*, pages 88–97, 2002.

9 Christian Cachin, Klaus Kursawe, and Victor Shoup. Random oracles in constantinople: Practical asynchronous byzantine agreement using cryptography. *Journal of Cryptology*, 18(3):219–246, 2005.

10 Ran Canetti and Tal Rabin. Fast asynchronous byzantine agreement with optimal resilience. In *Proceedings of the twenty-fifth annual ACM symposium on Theory of computing*, pages 42–51, 1993.

11 Ignacio Cascudo and Bernardo David. Scrape: Scalable randomness attested by public entities. In *International Conference on Applied Cryptography and Network Security*, pages 537–556. Springer, 2017.

12 Shir Cohen, Idit Keidar, and Alexander Spiegelman. Not a COINcidence: Sub-Quadratic Asynchronous Byzantine Agreement WHP. In Hagit Attiya, editor, *34th International Symposium on Distributed Computing (DISC 2020)*, volume 179 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 25:1–25:17, Dagstuhl, Germany, 2020. Schloss Dagstuhl–Leibniz-Zentrum für Informatik. URL: https://drops.dagstuhl.de/opus/volltexte/2020/13103, doi:10.4230/LIPIcs.DISC.2020.25.

13 Tyler Crain. Two more algorithms for randomized signature-free asynchronous binary byzantine consensus with $t < n/3$ and $o(n^2)$ messages and $o(1)$ round expected termination, 2020. URL: https://arxiv.org/abs/2002.08765, doi:10.48550/ARXIV.2002.08765.

14 Sourav Das, Zhuolun Xiang, and Ling Ren. Asynchronous data dissemination and its applications. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*, pages 2705–2721, 2021.

15 D. Dolev, N. A. Lynch, S. S. Pinter, E. W. Stark, and W. E. Weihl. Reaching approximate agreement in the presence of faults. *Journal of the ACM*, 33(3):499–516, July 1986.

16 A.D. Fekete. Asynchronous approximate agreement. *Information and Computation*, 115(1):95–124, 1994. URL: https://www.sciencedirect.com/science/article/pii/S0890540184710947, doi:https://doi.org/10.1006/inco.1994.1094.

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

18 Luciano Freitas de Souza, Andrei Tonkikh, Sara Tucci-Piergiovanni, Renaud Sirdey, Oana Stan, Nicolas Quero, and Petr Kuznetsov. Randsolomon: Optimally resilient random number generator with deterministic termination. In *25th International Conference on Principles of Distributed Systems (OPODIS 2021)*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2022.

19 Yingzi Gao, Yuan Lu, Zhenliang Lu, Qiang Tang, Jing Xu, and Zhenfeng Zhang. Efficient asynchronous byzantine agreement without private setups. *arXiv preprint arXiv:2106.07831*, 2021.

20 Rosario Gennaro, Stanisław Jarecki, Hugo Krawczyk, and Tal Rabin. Secure distributed key generation for discrete-log based cryptosystems. In *International Conference on the Theory and Applications of Cryptographic Techniques*, pages 295–310. Springer, 1999.

21 Yossi Gilad, Rotem Hemo, Silvio Micali, Georgios Vlachos, and Nickolai Zeldovich. Algorand: Scaling byzantine agreements for cryptocurrencies. In *Proceedings of the 26th Symposium on Operating Systems Principles*, SOSP '17, page 51–68, New York, NY, USA, 2017. Association for Computing Machinery. doi:10.1145/3132747.3132757.

22 Frank Gray. Pulse code communication. *United States Patent Number 2632058*, 1953.

23 Kobi Gurkan, Philipp Jovanovic, Mary Maller, Sarah Meiklejohn, Gilad Stern, and Alin Tomescu. Aggregatable distributed key generation. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pages 147–176. Springer, 2021.

24 Timo Hanke, Mahnush Movahedi, and Dominic Williams. DFINITY technology overview series, consensus system. *CoRR*, abs/1805.04548, 2018. URL: http://arxiv.org/abs/1805.04548, arXiv:1805.04548.

25 Shang-En Huang, Seth Pettie, and Leqi Zhu. Byzantine agreement in polynomial time with near-optimal resilience. *arXiv preprint arXiv:2202.13452*, 2022.

26 Valerie King and Jared Saia. Byzantine agreement in expected polynomial time. *J. ACM*, 63(2), mar 2016. doi:10.1145/2837019.

27 Eleftherios Kokoris Kogias, Dahlia Malkhi, and Alexander Spiegelman. *Asynchronous Distributed Key Generation for Computationally-Secure Randomness, Consensus, and Threshold Signatures.*, page 1751–1767. Association for Computing Machinery, New York, NY, USA, 2020. URL: https://doi.org/10.1145/3372297.3423364.

28 Mikhail Krasnoselskii, Grigorii Melnikov, and Yury Yanovich. No-dealer: Byzantine fault-tolerant random number generator. In *IEEE INFOCOM 2020-IEEE Conference on Computer Communications Workshops (INFOCOM WKSHPS)*, pages 568–573. IEEE, 2020.

29 Hugo Krawczyk. Secret sharing made short. In *Annual international cryptology conference*, pages 136–146. Springer, 1993.

30 Hammurabi Mendes and Maurice Herlihy. Multidimensional approximate agreement in byzantine asynchronous systems. In *Proceedings of the Forty-Fifth Annual ACM Symposium on Theory of Computing*, STOC '13, page 391–400, New York, NY, USA, 2013. Association for Computing Machinery. doi:10.1145/2488608.2488657.

31 Silvio Micali, Michael Rabin, and Salil Vadhan. Verifiable random functions. In *40th annual symposium on foundations of computer science (cat. No. 99CB37039)*, pages 120–130. IEEE, 1999.

32 Achour Mostéfaoui, Hamouma Moumen, and Michel Raynal. Signature-free asynchronous binary byzantine consensus with t < n/3, o(n2) messages, and o(1) expected time. *J. ACM*, 62(4), sep 2015. doi:10.1145/2785953.

33 Achour Mostefaoui, Matthieu Perrin, and Michel Raynal. A new insight into local coin-based randomized consensus. In *2019 IEEE 24th Pacific Rim International Symposium on Dependable Computing (PRDC)*, pages 207–20709. IEEE, 2019.

34 Achour Mostefaoui and Michel Raynal. Leader-based consensus. *Parallel Processing Letters*, 11(01):95–107, 2001.

35 Rajeev Motwani and Prabhakar Raghavan. *Randomized algorithms*. Cambridge university press, 1995.

36 Andre Neubauer, Jurgen Freudenberger, and Volker Kuhn. *Coding theory: algorithms, architectures and applications*. John Wiley & Sons, 2007.

37 Michael Rabin. Randomized Byzantine generals. In *Proceedings of the 24th Symposium on Foundations of Computer Science*, pages 403–409. IEEE Computer Society Press, November 1983.

38 Adi Shamir. How to share a secret. *Communications of the ACM*, 22(11):612–613, 1979.

39 Gilad Stern and Ittai Abraham. Living with asynchrony: the gather protocol. `https://decentralizedthoughts.github.io/2021-03-26-living-with-asynchrony-the-gather-protocol`, 2021. Accessed: 2022-02-12.

40 Ewa Syta, Philipp Jovanovic, Eleftherios Kokoris Kogias, Nicolas Gailly, Linus Gasser, Ismail Khoffi, Michael J Fischer, and Bryan Ford. Scalable bias-resistant distributed randomness. In *2017 IEEE Symposium on Security and Privacy (SP)*, pages 444–460. Ieee, 2017.

A Impossibility of an Asynchronous Perfect Common Coin

A perfect common coin protocol makes sure that the correct processes agree on a random number taken uniformly over a specific domain.

By reusing the classical arguments of the impossibility of asynchronous consensus [17], we are going to show that no asynchronous perfect common coin protocol may exist, if a single process is allowed to fail by crashing. Recall that we consider protocols in which processes may non-deterministically choose its actions based on local coin tosses.

Formally, a protocol provides each process with an automaton that, given the process' state and an input (a received message and a result of a local random coin toss), produces an output (a finite set of messages to send and/or the application output). We assume that the automaton itself is deterministic, i.e., all non-determinism is delegated to the outcomes of local random coins. A step of process p is therefore a tuple (p, m, r), where m is the message p receives (can be $\perp$ if no message is received in this step) and r is the outcome of its local random coin.

A *configuration* of the protocol assigns a local state to each process' automaton and a set of messages in transit (we call it the *message buffer*). The *initial configuration* C_{init} assigns the initial local state to each process and assumes that there are no messages in transit. A step $s = (p, m, r)$ is *applicable to a configuration* C if m is $\perp$ or m is in the message buffer of C. The result of s applied to C is a new configuration $C.s$, where, based on the automaton of p, the local state of p is modified and finitely many messages are added to the message buffer.

A sequence of steps $E = s_1, s_2, \ldots$ is applicable to a configuration C_0 if each s_i, $i = 1, 2, \ldots$ is applicable to C_{i-1}, where, for all $i \geq 1$, $C_i = C_{i-1}.s_i$. We use $C_0.E$ to denote the resulting configuration; we also say that $C = C_0.E$ is an *extension of* C_0. By convention, C is a trivial extension of C.

An *execution* of the protocol is a sequence of steps E applicable to C_{init}. A finite execution E results in a *reachable* configuration C_E. Somewhat redundantly, we call a sequence of steps applicable to a reachable configuration C_E an *extension of* E. It immediately follows that steps of disjoint sets of processes commute:

▶ **Lemma 5.** *Let C be a reachable configuration, and E and E' be sequences of steps of disjoint sets of processes. If both E and E' are applicable to C, then $C.E.E'$ and $C.E'.E$ are identical reachable configurations.*

In an infinite execution, a process is *correct* if it executes infinitely many steps. We assume that every message m that is sent to a correct process p is eventually received, i.e., the execution will eventually contain a step $(p, m, -)$. As we assume that at most one process is allowed to fail by crashing, we only consider infinite executions in which at least $n - 1$ out of n processes are correct.

Without loss of generality, we assume that the implemented coin is binary: the correct processes either all output 0 or all output 1.

A configuration C is called *bivalent* if it has an extension $C.E_0$ in which some process outputs 0 and an extension $C.E_1$ in which some process outputs 1. Notice that any configuration preceding a bivalent configuration must be bivalent. Also, no process can produce a random-coin output in a bivalent configuration: otherwise, we get an execution in which two processes disagree on the output.

A protocol configuration that is not bivalent is called *univalent*: 0-valent if it has no extension in which 1 is decided or 1-valent otherwise.

▶ **Lemma 6.** *The initial configuration C_{init} is bivalent.*

Proof. The algorithm must output each of the two values with a positive probability. Thus, for each $v \in \{0, 1\}$, there exists an assignment of local coin outcomes and a message schedule that result in an execution with outcome v. ◀

▶ **Lemma 7.** *Let C be a reachable configuration, and E and E' be sequences of steps of a process p applicable to C. If $C.E$ and $C.E'$ are univalent, then they have the same valencies.*

Proof. The difference between $C.E$ and $C.E'$ consists in the local state of p and the message buffer. Since the algorithm is required to tolerate a single crash fault, there must exist a sequence of steps E'' that does not include any steps of p such that some process q outputs a value $v \in \{0, 1\}$ in $C.E''$. Moreover, as E'' contains no steps of p, E'' is also applicable to both $C.E$ and $C.E'$. But as the two configurations have opposite valences, and q decides the same value in $C.E.E''$ and $C.E'.E''$. Thus, if $C.E$ and $C.E'$ are univalent, then they must have the same valencies. ◀

Following the steps of [17], we show that the protocol must have a *critical* configuration D for which:

- there exist steps s_p and s_q of processes p and q such that both s_p and $s_q.s_p$ are applicable to D
- $D.s_p$ is 0-valent;
- $D.s_q.s_p$ is 1-valent.

▶ **Lemma 8.** *There must exist a critical configuration.*

Proof. Starting with the initial (bivalent by Lemma 6) configuration $C := C_{init}$, we pick process $p := p_1$ and check if there exists C', an extension of C, and s_p, a step of p applicable to C' in which the oldest message to p in the message buffer is consumed (if any), such that $C'.s_p$ is bivalent. If this is the case, we set C to $C'.s_p$, pick up the next process $p := p_2$. Again, if there exists C', an extension of C, and s_p, a step of p in which the oldest message to p in the message buffer is consumed, such that $C'.s_p$ is bivalent, then we set C to $C'.s_p$. We repeat the procedure, each time picking the next process (in the round-robin order, i.e., after p_n we go to p_1, etc.), as long as it is possible.

The first observation is that the procedure cannot be repeated indefinitely. Indeed, otherwise, we obtain an infinite sequence of steps in which every process takes infinitely many steps and receives every message sent to it (and which is, thus, an execution of the protocol) that goes through bivalent configurations only. Hence, this execution cannot produce an output—a contradiction with the Termination property.

Thus, there must exist a bivalent configuration C and a process p, such that for each C', an extension of C and each $s_p = (p, m, -)$, a step of p applicable to C', $C'.s_p$ is univalent, where m is the oldest message addressed to p in C.

Let $s_p = (p, m, r)$ be a step of p applicable to C. Without loss of generality, let $C.s_p$ be 0-valent.

As C is bivalent, it must have an extension $E = e_1, \ldots, e_k$ such that $C.E$ is 1-valent.

Let ℓ be the largest index in $\{1, \ldots, k\}$ such that either s_p is not applicable to $C_\ell = C.e_1, \ldots, e_\ell$ or $C_\ell.s_p$ is 1-valent. Such an index exists, as $C.s_p$ is 0-valent $C.E$ is 1-valent and for any C', an extension of C, if s_p is applicable to C', then $C'.s_p$ is bivalent.

Suppose first that s_p is not applicable to C_ℓ. As s_p is applicable to every configuration $C_j = C.e_1, \ldots, e_j$, $j = 1, \ldots, \ell - 1$, e_ℓ must be be of the form (p, m, r'), i.e., e_ℓ must consume

the message received in $s_p = (p, m, r)$. By our assumption $C_\ell = C_{\ell-1}.(p, m, r')$ is univalent. As C_ℓ has a 1-valent descendant C', it must be 1-valent too.

But, by Lemma 7, $C_{\ell-1}.s_p = C_{\ell-1}.(p, m, r)$ and $C_\ell = C_{\ell-1}.(p, m, r')$, must have the same valencies—a contradiction.

Thus, $C_\ell.s_p$ is 1-valent. Hence, we have a bivalent configuration $D = C_{\ell-1}$ and steps s_p and $s_q = e_\ell$ such that both s_p and $s_q.s_p$ are applicable to D. Moreover, $D.s_p$ is 0-valent and $D.s_q.s_p$ is 1-valent. Thus, we get a critical configuration. ◀

Finally, we establish a contradiction by showing that:

▶ **Lemma 9.** *No critical configuration may exist.*

Proof. By contradiction, let a critical configuration D exist, and let s_p and s_q be steps of p and q applicable to D such that $D.s_p$ is 0-valent and $D.s_q.s_p$ be 1-valent.

If s_q is a step of p, then Lemma 7 establishes a contradiction.

Otherwise, consider any infinite execution going through to $D.s_p$ in which all processes but q take infinitely many steps in this execution after $D.s_p$. By the Termination property, there must exist a finite sequence of steps E such that some process outputs a value $v \in \{0, 1\}$ in $D.s_p.E$. As $D.s_p$ is 0-valent, $v = 0$. Moreover, as E contains no steps of q and p has the same state in $D.s_p$ and $D.s_q.s_p$, E is also applicable to $D.s_q.s_p$. Thus, 0 is also decided in $D.s_q.s_p.E$—a contradiction with the assumption that $D.s_q.s_p$ is 1-valent. ◀

Lemmata 8 and 9 imply:

▶ **Theorem 10.** *There does not exist a 1-resilient asynchronous random coin protocol.*

B Proof of Correctness of the Approximate Common Coin Algorithm

B.1 Termination

Consider a correct process i. As every correct process j first performs AVSS$_j$, GatherAccept(j) will eventually evaluate to true at i. Moreover, GatherAccept trivially satisfies the Accept Totality property of from Section 3.4. Thus, i will eventually witness Gather.DeliverSet(S) (line 12). Termination of Bundled Approximate Agreement (BAA) ensures that eventually i receives a weight vector. Recall that, as the output of BAA at any position j lies within the inputs of correct processes, it may carry a positive value only if j indeed completed sharing its value in AVSS$_j$. At least $n-f$ correct processes eventually invoke AVSS$_j$.EnableRetrieve() for each process (line 15). Thus, every such input will be successfully retrieved (line 16) and the coin output will be produced (line 18).

B.2 One Process Randomness

Let i be the first correct process to return from the BAA invocation (line 14). We are going to show that the value i is uniformly distributed over $[0, \ldots, D-1]$.

By the algorithm, the value is coming from a scalar product of the weight vector and the vector of retrieved secrets. The Binding Common Core property of the Gather abstraction guarantees that there is a set $\mathcal{S}^*$ of at least $n-f$ valid proofs delivered at all correct processes. Thus, the properties of BAA, there are at least $n-f$ indices in the weight vector W set to 1, and at least $f+1$ of these indices belong to correct processes. Thus, when process i retrieves the values from AVSS (line 16), at least $f+1$ of them come from correct processes. Each of these values were chosen uniformly at random in the range $[0, \ldots, D-1]$ (line 9).

Therefore, the result is computed from the sum of at least $n - f$ retrieved values (modulo D), and at least $f + 1$ of these values come from the correct processes. The retrieved values coming from the Byzantine processes must have been previously shared in AVSS and, thus, must have been chosen before any correct process reveals its input. Thus, regardless of the values shared by the adversary, the result computed in line 18 is uniformly distributed in the range $[0..D-1]$.

B.3 Approximate ϵ-Consistency

Let w_i^j denote the weight of process i received by process j from the approximate agreement protocol on line 14. The output of the coin for process k, which we will denote as R_k, is computed as follows (line 18): $\left\lceil \sum_{i\in[n]} x_i \cdot w_i^k \right\rceil \bmod D$.

Recall that $d_q(x, y) = \min\{|x - y|, q - |x - y|\}$. We need to show that, for any two processes k and l, $d_D(R_k, R_l) \leq \lceil \epsilon D \rceil$, where ϵ is the parameter of the approximate common coin. For this, we will need the following three inequalities that can be easily verified:

$$\forall q \in \{2, 3, 4, \dots\};\ a, b \in \{1, 2, 3, 4, \dots\} : d_q(a \bmod q, b \bmod q) \leq |a - b| \tag{1}$$

$$\forall a, b \in \mathbb{R}_{\geq 0} : \lceil a \rceil - \lceil b \rceil \leq \lceil a - b \rceil \tag{2}$$

$$\forall a \in \mathbb{R}_{\geq 0} : |\lceil a \rceil| \leq \lceil |a| \rceil \tag{3}$$

Let $\mathcal{S}^*$ denote the set of processes in the common core of the Gather protocol. Due to the Validity property of Bundled Approximate Agreement and the way inputs to BAA are formed (line 13), $\forall i \in \mathcal{S}^* : w_i^k = w_i^l = 1$.

Let us now derive a bound on $d(R_k, R_l)$.

$$\begin{aligned}
d_D(R_k, R_l) &= d_D\left(\left\lceil \sum_{i\in[n]} x_i \cdot w_i^k \right\rceil \bmod D, \left\lceil \sum_{i\in[n]} x_i \cdot w_i^l \right\rceil \bmod D\right) && \text{By definition.}\\
&\leq \left|\left\lceil \sum_{i\in[n]} x_i \cdot w_i^k \right\rceil - \left\lceil \sum_{i\in[n]} x_i \cdot w_i^l \right\rceil\right| && \text{Applying (1)}\\
&\leq \left\lceil \left| \sum_{i\in[n]} x_i \cdot w_i^k - \sum_{i\in[n]} x_i \cdot w_i^l \right| \right\rceil && \text{Applying (2)}\\
&= \left\lceil \left| \sum_{i\in[n]} (w_i^k - w_i^l) x_i \right| \right\rceil && \text{Applying (3)}\\
&= \left\lceil \left| \sum_{i\notin S^*} (w_i^k - w_i^l) x_i \right| \right\rceil && \text{Since } \forall i \in S^* : |w_i^k - w_i^l| = 0\\
&\leq \left\lceil \sum_{i\notin S^*} |w_i^k - w_i^l| D \right\rceil && \text{Since } \forall i : 0 \leq x_i < D\\
&\leq \left\lceil \sum_{i\notin S^*} \tilde{\epsilon} D \right\rceil && \text{By properties of BAA outputs}\\
&\leq \lceil f \tilde{\epsilon} D \rceil
\end{aligned}$$

For any $\epsilon \in (0, 1]$, if we want the output values to be at a distance of at most $\lceil \epsilon D \rceil$ apart, we can set $\tilde{\epsilon} = \epsilon / f$. Hence, in order to reach approximate ϵ-consistency, we need to run the protocol for $\log_2 \frac{1}{\tilde{\epsilon}} = \log_2 f + \log_2(1/\epsilon)$ rounds.

C Proof of Correctness of the Monte Carlo Common Coin Algorithm

C.1 Termination

BAA provides termination and also the guarantee that if all non-faulty processes input 0 in a given dimension, then every process decides 0. For this reason, if a value is irretrievable, then the weight given to it by correct processes is zero and therefore they will not be blocked trying to retrieve them.

C.2 Uniform distribution

Any decided value must be a value random secretly drawn and any such values are composed from $n - f$ values locally generated by different processes. Therefore, at least one of these values will be truly uniformly random, and whenever a uniform value over a domain D is added to any other value in mod D, the result will always be a uniformly random over D.

C.3 Proof of probabilistic δ-consistency without weight calibration

Let T_i denote the ticket of process i. We assume that $\forall i : T_i \sim U(0, 1)$. Note that, in practice, the tickets will have to be binary encoded as fixed precision numbers. However, if the number of bits in the encoding of a ticket is proportional to the security parameter λ, then the estimate that we give with the assumption that the tickets are real numbers holds true with all but negligible probability.

Let w_i^j denote the weight of process i received by process j from the approximate agreement protocol, $w_i^j \in [0, 1]$, and let $[w_1^{1\text{st}}, \ldots, w_n^{1\text{st}}]$ denote the weights received by the correct process that was the first to complete approximate agreement.

Let $X_i^j = w_i^j \cdot T_i$ and $X_i^{1\text{st}} = w_i^{1\text{st}} \cdot T_i$. Due to the properties of RSD, since no correct process invokes RSD.EnableRetrieve until it completes approximate agreement, the tickets are unpredictable to the adversary until $[w_1^{1\text{st}}, \ldots, w_n^{1\text{st}}]$ are fixed. Hence, $X_1^{1\text{st}}, \ldots, X_n^{1\text{st}}$ are independent random variables, and $X_i^{1\text{st}} \sim U(0, w_i^{1\text{st}})$.

Let $w_i^M = \max_j\{w_i^j\}$, $w_i^m = \min_j\{w_i^j\}$, $X_i^M = w_i^M \cdot T_i$, and $X_i^m = w_i^m \cdot T_i$. Unlike the first received weights, the maximum and the minimum weights can be manipulated by the adversary already after it learns the tickets. Hence, $X_1^M, \ldots, X_n^M$, as well as $X_1^m, \ldots, X_n^m$, are not independent and we cannot know their distributions. However, by the properties of approximate agreement, we know that $w_i^M \leq \min\{w_i^{1\text{st}} + \epsilon, 1\}$ and $w_i^m \geq \max\{w_i^{1\text{st}} - \epsilon, 0\}$.

Let $i_M = \arg\max_i X_i^{1\text{st}}$. With a slight abuse of notation, we use subscript M instead of i_M. I.e., $T_M = T_{i_M}$, $X_M^{1\text{st}} = X_{i_M}^{1\text{st}}$, $X_M^M = X_{i_M}^M$, $X_M^m = X_{i_M}^m$, and so on.

▶ **Theorem 11.** *Without weight calibration, running the approximate agreement for* $3 + \lceil \log_2(n) + \log_2(1/Q) \rceil$ *rounds is sufficient to guarantee that the probability of not having consistency is at most* Q*.*

Proof. A sufficient condition for all processes to output the same value would be if $\forall i \neq i_M : X_i^M < X_M^m$. Indeed, it would mean that, even if process j gets the minimum possible weight for process i_M and maximum possible weights for all other processes $i \neq i_M$, j would still

select the value associated with i_M. Hence, we provide an upper bound on the probability of failure, $P[\text{failure}]$, by estimating the probability there exists i such that $X_i^M \geq X_M^m$. As we demonstrate bellow, in Equation 1, $P[\text{failure}] \leq 8n\epsilon$. Hence, we can conclude that $\epsilon \leq \frac{Q}{8n}$ is sufficient to guarantee that $P[\text{failure}] \leq Q$. Since the approximate agreement protocol we use requires $\lceil \log_2 \frac{1}{\epsilon} \rceil$ rounds, running it for $\lceil \log_2 \frac{8n}{Q} \rceil \leq 3 + \lceil \log_2(n) + \log_2(1/Q) \rceil$ rounds is sufficient to achieve $\epsilon = \frac{Q}{8n}$ and, hence, $P[\text{failure}] \leq Q$. ◀

$$
\begin{aligned}
P[\text{failure}] &\leq P[\exists i \neq i_M : X_i^M \geq X_M^m] \\
&\leq P[\exists i \neq i_M : X_i^{1\text{st}} + \epsilon T_i \geq X_M^{1\text{st}} - \epsilon T_M] \\
&\leq P[\exists i \neq i_M : X_i^{1\text{st}} \geq X_M^{1\text{st}} - 2\epsilon] \\
&= \sum_{k=1}^{n} P[i_M = k] \cdot P[\exists i \neq k : X_i^{1\text{st}} \geq X_k^{1\text{st}} - 2\epsilon \mid i_M = k] \\
&= \sum_{k=1}^{n} \int_0^{w_k^{1\text{st}}} p_{X_k^{1\text{st}}}(x) \cdot P[i_M = k \mid X_k^{1\text{st}} = x] \cdot P[\exists i \neq k : X_i^{1\text{st}} \geq X_k^{1\text{st}} - 2\epsilon \mid i_M = k \wedge X_k^{1\text{st}} = x] \,\mathrm{d}x \\
&= \sum_{k=1}^{n} \int_0^{w_k^{1\text{st}}} p_{X_k^{1\text{st}}}(x) \cdot \left(\prod_{i \neq k} P[X_i^{1\text{st}} \leq x] \right) \cdot P[\exists i \neq k : X_i^{1\text{st}} \geq x - 2\epsilon \mid X_i^{1\text{st}} < x] \,\mathrm{d}x \\
&\quad \text{(applying the union bound)} \\
&\leq \sum_{k=1}^{n} \int_0^{w_k^{1\text{st}}} p_{X_k^{1\text{st}}}(x) \cdot \left(\prod_{i \neq k} P[X_i^{1\text{st}} \leq x] \right) \cdot \left(\sum_{i \neq k} P[X_i^{1\text{st}} \geq x - 2\epsilon \mid X_i^{1\text{st}} \leq x] \right) \cdot \mathrm{d}x \\
&= \sum_{k=1}^{n} \int_0^{w_k^{1\text{st}}} p_{X_k^{1\text{st}}}(x) \cdot \left(\prod_{i \neq k} P[X_i^{1\text{st}} \leq x] \right) \cdot \left(\sum_{i \neq k} \frac{P[X_i^{1\text{st}} \geq x - 2\epsilon \wedge X_i^{1\text{st}} \leq x]}{P[X_i^{1\text{st}} \leq x]} \right) \cdot \mathrm{d}x \\
&= \sum_{k=1}^{n} \int_0^{w_k^{1\text{st}}} p_{X_k^{1\text{st}}}(x) \cdot \sum_{i \neq k} \left(P[X_i^{1\text{st}} \geq x - 2\epsilon \wedge X_i^{1\text{st}} \leq x] \cdot \prod_{j \notin \{k,i\}} P[X_j^{1\text{st}} \leq x] \right) \cdot \mathrm{d}x \\
&\leq \sum_{k=1}^{n} \int_0^{w_k^{1\text{st}}} p_{X_k^{1\text{st}}}(x) \cdot \sum_{i \neq k} \left(\frac{2\epsilon}{x} \cdot \prod_{j \notin \{k,i\}} P[X_j^{1\text{st}} \leq x] \right) \cdot \mathrm{d}x \\
&\quad \text{(since } \forall j \in S : P[X_j^{1\text{st}} \leq x] = P[T_j^{1\text{st}} \leq x] = x \text{ and } |S \setminus \{k,i\}| \geq n - f - 2\text{)} \\
&\leq \sum_{k=1}^{n} \int_0^{w_k^{1\text{st}}} p_{X_k^{1\text{st}}}(x) \cdot \sum_{i \neq k} \left(\frac{2\epsilon}{x} \cdot x^{n-f-2} \right) \cdot \mathrm{d}x = \sum_{k=1}^{n} \int_0^{w_k^{1\text{st}}} p_{X_k^{1\text{st}}}(x) \cdot \sum_{i \neq k} \left(2\epsilon \cdot x^{n-f-3} \right) \cdot \mathrm{d}x \\
&\leq \sum_{k=1}^{n} \int_0^{w_k^{1\text{st}}} p_{X_k^{1\text{st}}}(x) \cdot 2n\epsilon \cdot x^{n-f-3} \cdot \mathrm{d}x = \sum_{k=1}^{n} \frac{2n\epsilon}{w_k^{1\text{st}}} \int_0^{w_k^{1\text{st}}} x^{n-f-3} \,\mathrm{d}x \\
&= \sum_{k=1}^{n} \frac{2n\epsilon}{w_k^{1\text{st}}} \cdot \frac{(w_k^{1\text{st}})^{n-f-2}}{n-f-2} = \sum_{k=1}^{n} \frac{2n\epsilon}{n-f-2} \cdot \frac{(w_k^{1\text{st}})^{n-f-2}}{w_k^{1\text{st}}} \\
&\quad \text{(since } n \geq 3f+1 \text{ and } f \geq 1\text{, it can be easily verified that } n-f-2 \geq n/4 \geq 1\text{)} \\
&\leq \sum_{k=1}^{n} \frac{2n\epsilon}{n/4} \cdot \frac{w_k^{1\text{st}}}{w_f^{1\text{st}}} = \sum_{k=1}^{n} 8\epsilon = 8n\epsilon
\end{aligned}
$$

Equation 1 Estimate on the failure probability of Monte Carlo common coin.

C.4 Proof of probabilistic δ-consistency with weight calibration

Let $Q = 1 - \delta$ be the target error probability. Let r be the number of rounds of approximate agreement. Our goal in this section is to show that, when weight calibration is applied as described in Section 6, $r = 5 + \lceil \log_2(1/Q) + \log_2(\log_2(1/Q)) \rceil$ is sufficient to make sure that the probability of not reaching agreement is below Q. $\epsilon = 2^{-r}$ is the maximum difference between the outputs of Approximate Agreement for two correct processes in the same coordinate.

Recall that we apply this optimization only when $n > \frac{3\ln(2/Q)}{2}$. Otherwise, we just use the protocol without weight calibration. According to the proof from Appendix C.3, in this case, we shall need $3 + \lceil \log_2(n) + \log_2(1/Q) \rceil \le 3 + \lceil \log_2\left(\frac{3\ln(2/Q)}{2}\right) + \log_2(1/Q) \rceil \le 5 + \lceil \log_2(1/Q) + \log_2(\log_2(1/Q)) \rceil$ rounds.

Let $v = 1 - \frac{\ln(2/Q)}{2N/3}$ (the choice of v is dictated by the proof of Theorem 12).

Let the weight calibration function $\textsc{Calibrate}(w)$ be $\begin{cases} 0, & \text{if } w = 0 \\ \frac{w-\epsilon}{1-\epsilon} + \frac{1-w}{1-\epsilon} \cdot v, & \text{otherwise} \end{cases}$

In other words, $\textsc{Calibrate}(0) = 0$ and $\forall w \in (0, 1] : \textsc{Calibrate}(w)$ is a linear function that is equal to v at $w = \epsilon$ and is equal to 1 at $w = 1$ (see Figure 1). The slope of this function for $w > 0$ is $\frac{\partial\, \textsc{Calibrate}(w)}{\partial w} = \frac{1-v}{1-\epsilon} \le \frac{16}{15} \cdot (1 - v)$ for $\epsilon \le \frac{1}{16}$.[8]

Our goal is to reduce the problem to the case without weight calibration, but with smaller ϵ. In other words, by applying weight calibration, we manage to significantly shrink the disagreement of correct processes on the weights "for free", i.e., without running extra rounds of Approximate Agreement. To this end, we define $\epsilon' = \frac{16}{15}\epsilon(1 - v)$, so that $\forall w > 0 : \textsc{Calibrate}(w + \epsilon) - \textsc{Calibrate}(w) \le \epsilon'$.

However, if, for some coordinate, the weight received from BAA by one process is ϵ and the weight received by another process is 0, then the range after the calibration will only grow (i.e., $\textsc{Calibrate}(\epsilon) - \textsc{Calibrate}(0) = v > \epsilon$). Hence, we need to show that, with high probability, this discontinuity does not affect the results of the protocol.

We use the same notation as in Appendix C.3: T_i denotes the ticket of process i; w_i^j denotes the weight of process i received by process j from the approximate agreement protocol; $w_i^M = \max_j\{w_i^j\}$ and $w_i^m = \min_j\{w_i^j\}$; $[w_1^{\text{1st}}, \ldots, w_n^{\text{1st}}]$ denote the weights received by the correct process that was the first to complete approximate agreement; $X_i^j = w_i^j \cdot T_i$, $X_i^{\text{1st}} = w_i^{\text{1st}} \cdot T_i$, $X_i^M = w_i^M \cdot T_i$, and $X_i^m = w_i^m \cdot T_i$. Finally, $i_M = \arg\max_i X_i^{\text{1st}}$ and subscript M is used as a replacement for i_M. I.e., $T_M = T_{i_M}$, $X_M^{\text{1st}} = X_{i_M}^{\text{1st}}$, $X_M^M = X_{i_M}^M$, $X_M^m = X_{i_M}^m$, and so on.

First, let us state two useful lemmas.

▶ **Lemma 12.** *With probability at least* $1 - \frac{Q}{2}$, $X_M^m > v$.

▶ **Lemma 13.** *With the weight calibration applied,* $P[\textit{failure} \mid X_M^m > v] \le P[\exists i \ne i_M : X_i^{\text{1st}} \ge X_M^{\text{1st}} - 2\epsilon']$.

Intuitively, Theorem 12 shows that, with high probability, a certain good event happens (namely, the discontinuity of the calibration function does not affect the outcome), while

[8] The choice of $\frac{1}{16}$ is mostly arbitrary and does not affect the final complexity. However, note that, according to the estimate from Appendix C.3, we always need to run the approximate agreement for at least $3 + \log_2 n$ rounds. Hence, it is safe to assume that $\epsilon \le \frac{1}{2^4} = \frac{1}{16}$.

Theorem 13 shows that, in this good case, the probability of failure is similar to that of non-calibrated algorithm with disagreement on weights equal to ϵ'.

Using these two lemmas, we now can prove the main theorem:

▶ **Theorem 14.** *With weight calibration, running the approximate agreement for* $5 + \lceil \log_2(1/Q) \rceil + \lceil \log_2(\log_2(1/Q)) \rceil$ *rounds is sufficient to guarantee that the probability of not having agreement is at most* Q*.*

Proof. As we show in Theorem 12, with probability at least $1 - \frac{Q}{2}$, X_M^m is greater than v. Then, in Theorem 13, we show that $P[\text{failure} \mid X_M^m > v] \leq P[\exists i \neq i_M : X_i^{1\text{st}} \geq X_M^{1\text{st}} - 2\epsilon']$. By applying the same transformations as in Equation 1, we get $P[\text{failure} \mid X_M^m > v] \leq 8n\epsilon' \leq 8n\left(\frac{16}{15}\epsilon(1-v)\right) \leq \frac{128}{15}n\epsilon(1-v) = \frac{128}{15}n\epsilon\left(\frac{\ln(2/Q)}{2N/3}\right) \leq 9\epsilon\log_2(2/Q)$.

By running Approximate Agreement for $\lceil \log_2(9) + \log_2(2/Q) + \log_2(\log_2(2/Q)) \rceil \leq 5 + \lceil \log_2(1/Q) + \log_2(\log_2(1/Q)) \rceil$ rounds, we can guarantee that $P[\text{failure} \mid X_M^m > v] \leq \frac{Q}{2}$.

Finally, we can compute $P[\text{failure}]$ as $P[X_M^m > v] \cdot P[\text{failure} \mid X_M^m > v] + P[X_M^m \leq v] \cdot P[\text{failure} \mid X_M^m \leq v] \leq 1 \cdot P[\text{failure} \mid X_M^m > v] + P[X_M^m > v] \cdot 1 \leq \frac{Q}{2} + \frac{Q}{2} \leq Q$. ◀

Finally, let us prove the two lemmas.

Proof of Theorem 12. Let $\mathcal{S}^*$ denote the set of processes in the common core. It is sufficient to show that, with probability at least $1 - \frac{Q}{2}$, there is a process $i \in \mathcal{S}^*$ such that $X_i^m = T_i > v$.

Note that, by the *Binding Common Core* property of Verifiable Gather and by the *Unpredictability* property of RSD, the tickets are unpredictable to the adversary until the common core is fixed. Hence $\forall i \in \mathcal{S}^*$: T_i are mutually independent, and $T_i \sim U(0,1)$.

$P[\exists i \in \mathcal{S}^* : T_i > v] = 1 - P[\forall i \in \mathcal{S}^* : T_i \leq v]$

$P[\forall i \in \mathcal{S}^* : T_i \leq v] = v^{|\mathcal{S}^*|} \leq v^{2n/3} = \left(1 - \frac{\ln(2/Q)}{2n/3}\right)^{2n/3} \leq \frac{Q}{2}$

The final inequality is a special case of a more general well-known inequality (see, for example, [35, Appendix B]): $\forall a > 0, x > a : \left(1 - \frac{a}{x}\right)^x < e^{-a}$. To see why it holds, note that $\left(1 - \frac{a}{x}\right)^x \xrightarrow[x\to\infty]{} e^{-a}$ and that $\left(1 - \frac{a}{x}\right)^x$ increases monotonically when $x \geq a$ and $a > 0$. ◀

Proof of Theorem 13. For convenience, we shall prove the equivalent statement that $P[\text{success} \mid X_M^m > v] \geq P[\forall i \neq i_M : X_i^{1\text{st}} < X_M^{1\text{st}} - 2\epsilon')]$.

As discussed in Appendix C.3, $P[\text{success}] \leq P[\forall i \neq i_M : X_i^M < X_M^m]$. Hence, $P[\text{success} \mid X_M^m > v] \geq P[\forall i \neq i_M : X_i^M < X_M^m \mid X_M^m > v]$.

It is sufficient to show that, under the condition that $X_M^m > v$, for any $i \neq i_M$: $X_i^{1\text{st}} + \epsilon' < X_M^{1\text{st}} - \epsilon'$ implies $X_i^M < X_M^m$. Indeed, consider two cases:

1. $w_i^{1\text{st}} = 0$: hence, $w_i^M \leq \epsilon$ and $X_i^M = \textsc{Calibrate}(w_i^M) \cdot T_i \leq \textsc{Calibrate}(w_i^M) \leq v < X_M^m$. So the right hand side of the implication is always true and the implication holds trivially;
2. $w_i^{1\text{st}} > 0$: hence, $X_i^M = \textsc{Calibrate}(w_i^M) \cdot T_i \leq \textsc{Calibrate}(w_i^{1\text{st}} + \epsilon) \cdot T_i \leq (\textsc{Calibrate}(w_i^{1\text{st}}) + \epsilon') \cdot T_i \leq \textsc{Calibrate}(w_i^{1\text{st}}) \cdot T_i + \epsilon' = X_i^{1\text{st}} + \epsilon'$. Similarly, $X_M^m \geq X_M^{1\text{st}} - \epsilon'$. Finally, if $X_i^{1\text{st}} + \epsilon' < X_M^{1\text{st}} - \epsilon'$, then we have $X_i^M \leq X_i^{1\text{st}} + \epsilon' < X_M^{1\text{st}} - \epsilon' \leq X_M^m$.

◀

D Random Secret Draw implementations

D.1 Random Secret Draw for adaptive adversary

In RSD every process begins by generating n random numbers that are assigned to each of the processes in the system and then secret sharing these numbers (Lines 60 and 62. Once a process i asserts that a value that was assigned to it was successfully secret share, it can

include the source of this value to a set S (Line 64). Once the number of processes in S is $n - f$, i can reliably broadcast the set it built, making it the choice of secrets to be used to form its value.

When a message containing another process' set of secrets to be used is delivered (Line 67), a ValueAssigned event is triggered. After the underlying application executes EnableRetrieve (Line 71), the correct processes enable the retrieval of the secrets chosen by every process who has already been assigned a value.

Finally, processes compute the values assigned to any user by retrieving all the secrets it has selected and summing then up in mod$D$ (Line 79).

Algorithm 4 Random Secret Draw from AVSS, code for process i

50: **Instance parameters:** domain size D

51: **Distributed objects:**
52: $\forall j, k \in [n]$: AVSS_j^k – instance of AVSS with leader j used to generate secret for process k and domain size D
53: $\forall j \in [n]$: BRB_j – instance of BRB with leader j

54: **State:**
55: *assignedSources* – map from process ids to set of processes
56: *assignedValues* – map from process ids to numbers
57: *retrieveEnabled* – Boolean that allows values to be retrieved

58: **operation** Start()
59: $retrieveEnabled = false$
60: $\forall j \in [n] : x_i^j = \text{RandomInt}(D)$
61: $S = \varnothing$
62: AVSS_i^j.ShareSecret(x_i^j)

63: **upon** AVSS_j.SharingComplete()
64: $S = S \cup j$

65: **upon** $|S| = n - f$
66: BRB_i.Broadcast(S)

67: **upon** BRB_j.Deliver(S_j) $\wedge\ \forall k \in S_j$: AVSS_k.SharingComplete()
68: $assignedSources[j] = S_j$
69: **trigger event** ValueAssigned(j)
70: **if** *retrieveEnabled* **then** $\forall k \in S_j$: APVSS_k^j.EnableRetrieve()

71: **operation** EnableRetrieve()
72: $retrieveEnabled = true$
73: $\forall j \in [n] \wedge \text{ValueAssigned}(j), k \in assignedSources[j]$: APVSS_k^j.EnableRetrieve()

74: **operation** RetrieveValues(S) **returns** a map from process ids to random numbers
75: **wait until** $\forall j \in S$: $\text{ValueAssigned}_j()$
76: // Note that this line may not terminate if for some $j \in S$ secret sharing has not been complete
77: // If APVSS_j.Retrieve() has already been invoked in the past, we assume that it returns a cached value
78: $\forall j \in S : assignedValues[j] = \left(\sum_{k \in assignedSources[j]} \text{AVSS}_k^j.\text{Retrieve}()\right) \bmod D$
79: **return** *assignedValues*

D.2 Cubic Random Secret Draw for static adversary

D.2.1 Aggregatable Publicly Verifiable Secret Sharing

The RSD implementation that we use relies on *Aggregatable Publicly Verifiable Secret Sharing (APVSS)* [23] in order to achieve cubic communication complexity.

Unlike in AVSS, where a process shares a secret of its own choice, in APVSS, the secret is an aggregation of values generated by $n - f$ processes. In the original presentation of [23], these values are shares of a private key that is being generated. In our case, these values are just random numbers and the aggregation is simply a sum modulo the domain size D. Moreover, the process itself cannot know the secret until a threshold of processes are ready to collectively reconstruct it.

For the purposes of this paper, we shall consider a restricted interface of APVSS:

APVSS_i.CreateScriptShare$_j(x)$: can be invoked by process j to generate a *script share* for process i;

APVSS_i.VerifyScriptShare(j, *scriptShare*): returns *true* if *scriptShare* was generated by process j invoking APVSS_i.CreateScriptShare$_j$;

APVSS_i.ShareSecret(*scriptShares*): can be invoked by process i to secret-share an aggregation of the secrets that were used to create *scriptShares*, provided $n - f$ script shares generated by different processes;

APVSS_i.SharingComplete(), APVSS_i.EnableRetrieve(), and APVSS_i.Retrieve(): analogous to AVSS_i.

In practice, the APVSS scheme of [23] provides more general functionality (e.g., it allows combining fewer than $n - f$ script shares). However, for the Random Secret Draw protocol, this simplified interface is sufficient.

APVSS satisfies, Validity, Notification Totality, Retrieve Termination, and Binding properties of AVSS. Compared to AVSS, APVSS provides secrecy regardless of whether the process itself is correct:

Secrecy: if no correct process invoked AVSS_i.EnableRetrieve(), then the adversary has no information about the secret shared by i.

D.2.2 Pseudocode

Similarly to the AVSS version, the APVSS version of RSD begins with every process generating n random numbers to be assigned to the other participants. This time, instead of secret sharing these numbers, a scriptshare is sent which allows the recipient to combine them.

Using APVSS, every process is able to generate a secret which is a sum in modulo D of $n - f$ valid secret shares assigned to them by other processes (lines 91 to 95).

Once the secret is shared, an event ValueAssigned is generated (line 97). This event can be treated by the application that runs RSD so that it enables the retrieval of secrets once enough values are assigned (line 98). The application must then create a set of processes S for which it desires to know the random numbers of and then execute RetrieveValues which returns these values after the secrets for every processes in S are assigned (lines 102 to 107). As before, only after a process gets assigned a value the secret retrieval is enabled.

Algorithm 5 Random Secret Draw from APVSS, code for process i

80: **Instance parameters:** domain size D

81: **Distributed objects:**
82: $\forall j \in [n]$: APVSS$_j$ – instance of APVSS with leader j and domain size D

83: **State:**
84: *apvssScriptShares* – set of pairs $(j, scriptShare)$
85: *assignedValues* – map from process ids to numbers
86: *retrieveEnabled* – Boolean indicating processes may allow for secret retrieval

87: **operation** Start()
88: $retrieveEnabled = false$
89: $\forall j \in [n]$: **send** $\langle$**ApvssShare**, APVSS$_j$.CreateScriptShare$_i$(RandomInt(D))$\rangle$ **to** process j

90: **upon receiving** $\langle$**ApvssShare**, $scriptShare\rangle$ **from** process j
91: **if** not APVSS$_i$.VerifyScriptShare(j, $scriptShare$) **then return** // ignore invalid message
92: $apvssScriptShares = apvssScriptShares \cup (j, scriptShare)$
93: **if** $|apvssScriptShares| = n - f$ **then**
94: // The aggregated secret is a sum modulo D of the $n - f$ original secrets
95: APVSS$_i$.ShareSecret($apvssScriptShares$)

96: **upon** event APVSS$_j$.SharingComplete()
97: **trigger event** ValueAssigned(j)
98: **if** *retrieveEnabled* **then** APVSS$_j$.EnableRetrieve()

99: **operation** EnableRetrieve()
100: $retrieveEnabled = true$
101: $\forall j \in [n] \wedge$ ValueAssigned(j)APVSS$_j$.EnableRetrieve()

102: **operation** RetrieveValues(S) **returns** a map from process ids to random numbers
103: **wait until** $\forall j \in S$: ValueAssigned$_j$()
104: // Note that this line may not terminate if for some $j \in S$ secret sharing has not been complete
105: // If APVSS$_j$.Retrieve() has already been invoked in the past, we assume that it returns a cached value
106: $assignedValues = \text{map}[j \mapsto \text{APVSS}_j.\text{Retrieve}() \mid \forall j \in S]$
107: **return** *assignedValues*

E Pseudocode for Canetti and Rabin [10] common coin

Using our building blocks and presentation, the pseudocode for the coin proposed by Canetti and Rabin becomes very simple. It consists of every process executing random secret draw and then gathering a set of processes who get assigned values. If among the corresponding gathered values there is a 0, then a 0 is decided, else the processes decide 1.

Here, the advantages of a modular approach can be seen: first, the algorithm becomes clearer, and, second, later improvements in building blocks lead to an improvement of the algorithm as a whole.

Algorithm 6 Canetti and Rabin Common Coin, code for process i

```
108: Distributed objects:
109:     RSD – instance of Random Secret Draw with domain size ⌈0.87n⌉
110:     Gather – instance of Gather (verifiability is not necessary for this protocol)

111: function GatherAccept(j) returns boolean
112:     return true iff RSD.ValueAssigned(j)

113: operation Toss() returns integer
114:     RSD.Start()
115:     Gather.Start(GatherAccept)
116:     wait for event Gather.DeliverSet(S)
117:     RSD.EnableRetrieve()
118:     values = ValueDraw.RetrieveValues(candidates)
119:     if 0 ∈ values then
120:         return 0
121:     else
122:         return 1
```

F Concrete Code for Intersecting Random Subsets

The following recursive formula produces a code (i.e., a list of code words) that satisfy the definition in Section 7.2:

$$\forall n \geq 0 \text{ and } m \in [0..n] : C_{n,m} = \begin{cases} [\text{“ ”}] & \text{if } 0 = m = n \\ [\text{“}000\ldots000\text{”}] & \text{if } 0 = m < n \\ [\text{“}111\ldots111\text{”}] & \text{if } 0 < m = n \\ 0\|C_{n-1,m}, \text{reverse}(1\|C_{n,m-1}) & \text{otherwise} \end{cases}$$

For example, here is $C_{5,2}$: [00011, 00110, 00101, 01100, 01010, 01001, 11000, 10100, 10010, 10001].

In order to avoid computing all $\binom{n}{m}$ code words, when n and m are large, we can use the following recursive formula to efficiently (with $O(poly(n))$ operations) find a code word by its index:

$\forall n \geq 0$ and $m \in [0..n], i \in [0..\binom{n}{m}-1]$:

$$C_{n,m}[i] = \begin{cases} \text{“ ”} & \text{if } 0 = m = n \\ \text{“}000\ldots000\text{”} & \text{if } 0 = m < n \\ \text{“}111\ldots111\text{”} & \text{if } 0 < m = n \\ 0 \| C_{n-1,m}[i] & \text{if } 0 < m < n, i < \binom{n-1}{m} \\ 1 \| C_{n-1,m-1}[\binom{n-1}{m-1} - (i - \binom{n-1}{m}) - 1] & \text{if } 0 < m < n, i \geq \binom{n-1}{m} \end{cases}$$