SeF: A Secure Fountain Blockchain Architecture Suited for Resource-Constrained Machines

Swanand Kadhe, Jichan Chung, and Kannan Ramchandran

Dept. of Electrical Engineering and Computer Sciences,
University of California, Berkeley
{swanand.kadhe, jichan3751, kannanr}berkeley.edu

Full nodes, which synchronize the full blockchain history and independently validate all the blocks, form the backbone of any blockchain network by playing a vital role in ensuring security properties. On the other hand, a user running a full node needs to pay a heavy price in terms of storage costs. In particular, blockchain storage requirements are growing near-exponentially, easily outpacing Moore's law for storage devices. For instance, the Bitcoin blockchain size has grown over 201GB, in spite of its low throughput. The ledger size for a high throughput blockchain Ripple has already reached 8.4TB, and it is growing at an astonishing rate of 12GB per day!

In this paper, we propose an architecture based on fountain codes that allows full nodes to cut down their storage costs by at least three orders of magnitude without affecting the security properties of the blockchain. We demonstrate that the peeling decoder admitted by fountain codes turns out to be crucial for ensuring security against malicious nodes. Specifically, it enables us to introduce error-resiliency by leveraging the hash-chain structure of the blockchain. Further, the rateless property of fountain codes helps in achieving high decentralization and scalability. We evaluate the performance of the SeF architecture by performing experiments on the Bitcoin blockchain.

1 Introduction

Blockchains have played an instrumental role as the foundational technology for cryptocurrencies such as Bitcoin and Ethereum. Moreover, they have the potential to impact diverse fields such as the Internet-of-Things [1], medicine [2], healthcare [3], and supply-chains [4] among others. This great potential of blockchains comes from their key differentiating properties of decentralization, security, trustlesssness, and scalability. (For simplicity, we refer to these properties as security properties.)

Full nodes, which store all the blocks and validate transactions and blocks, form the backbone of any blockchain network, as they play a vital role in ensuring the security properties. More specifically, by independently verifying transactions and blocks without relying on any other node, full nodes contribute to the health of the network by safeguarding its security and trustlessness, and by helping to bootstrap new nodes joining the network, they ensure decentralization and scalability of the network.

On the other hand, a user running a full node needs to pay a heavy price in terms of storage and computation costs. In particular, blockchain storage requirements are growing near-exponentially, easily outpacing Moore's law for storage devices. To get a glimpse of the heavy costs required for storing the blockchain's historical data, consider the case of Bitcoin. In spite of its low throughput of just 4-7 transactions per second, the Bitcoin blockchain size has grown over $215GB$ as of April 2019 [5] (See Fig. 1). In fact, storage costs are going to be a pressing concern in the near future for high throughput blockchains like Ripple. For instance, the Ripple (XRP) ledger size has already reached $8.4TB$, and it is growing at an astonishing rate of $12GB$ per day! (See [6].)

Research Report for Ripple University Blockchain Research Initiative.

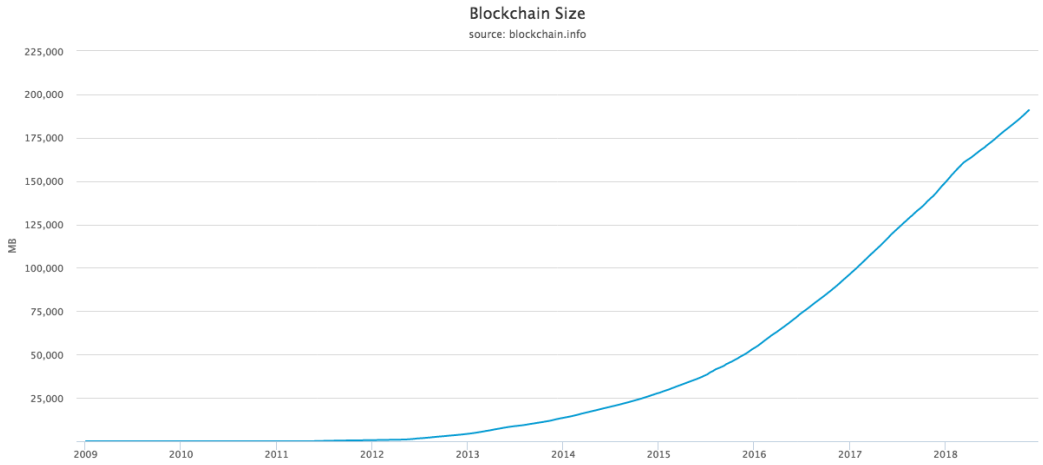

Figure 1: The total size of all block headers and transactions for the Bitcoin blockchain, not including database indexes (source: [5]).

In current practice, there are two solutions for saving costs: (i) run a light client, also known as simplified payment verification (SPV) client [7], or (ii) enable *block pruning* [8]. Running a light client is the most economical way of saving costs. Light clients store only block headers, and do not validate transactions. However, light clients are known to be vulnerable to several security and privacy attacks (see, *e.g.*, [9, Chapter 6]). A pruned node stores only a budgeted number of most recent blocks, and deletes old blocks after they are validated. Though, unlike light clients, pruned nodes have strong security properties, they cannot contribute to scaling up the network in a secure and decentralized manner as they are unable to assist new full nodes. Indeed, if a large number of full nodes enable pruning, then new nodes will need to rely on a small number of *archival nodes* ([10]) that store the entire blockchain in order to bootstrap, greatly reducing decentralization (see Fig. 2 (a)).

Compelled by the essential role that full nodes play in ensuring the security properties and the heavy costs they incur, we ask the following question: *is it possible to design protocols that enable storage-constrained machines to act as full nodes without affecting the security properties of the blockchain?*

Any such protocol faces the following challenges. First, the protocol must ensure that the blockchain network can scale up in a *secure* manner even if a subset of full nodes are adversarial. Specifically, any new node should be able to recover the blockchain even if a subset of full nodes act adversarially and provide maliciously formed data to the new node. Moreover, the computational cost associated with recovering the blockchain must be small. Next, the protocol must be *decentralized*: every full node should be able to perform computations to reduce its storage space independent of any other full node. Finally, the protocol must have limited *bootstrap cost* in terms of the number of full nodes that a new node needs to contact in order to recover the blockchain.

In fact, there is a fundamental trade-off between the storage savings and the bootstrap cost. To understand this, consider a scenario in which every full node restricts its storage space to $1GB$. Then, a new node in the Bitcoin network will need to contact at least 215 (honest) nodes to obtain the $215GB$ Bitcoin blockchain. Whereas, a new node in the Ripple network will need to contact at least 84000 (honest) nodes to obtain the $8.4TB$ Ripple (XRP) ledger. This is simply because the total amount of data downloaded by a new node must at least be the size of the blockchain. As a result, the larger the storage savings per full node, the higher the bootstrap cost for a new node.

In a centralized system, it is easy to keep the bootstrap cost to its minimum. For instance, let us assume for the moment that some 10000 full nodes in a blockchain network can (magically) coordinate with each other to decide which node should store the first 10000 blocks, which one should store the next 10000 blocks, and so on. In this case, each of these full nodes attains a storage savings of $10000\times$, but a new node will need to contact exactly 10000 nodes to obtain the entire blockchain. In a decentralized system where nodes cannot coordinate, there will be overlaps between stored data, resulting in higher bootstrap cost.

Indeed, using naïve approaches to achieve decentralization can result in prohibitively high bootstrap cost. As an example, consider the following simple protocol for a full node. For every k blocks (say, $k = 10000$), a node stores a randomly selected block, independent of other nodes. Each node thus achieves k-fold storage savings. However, it is not hard to show that, to obtain the entire blockchain, a new node will need to contact $O(k\ln(k))$ honest nodes on an average.[1] Our experiments demonstrate that for $k = 1000$, the bootstrap cost is 7499, while for $k = 10000$, it is 98530.

This paper presents Secure Fountain codes (*SeF codes*) which address the aforementioned challenges. The core of SeF codes is built up on a class of erasure codes called fountain codes [11, 12] (see also [13, 14]). The *encoder* of a fountain code is a metaphorical fountain that takes as an input a set of blocks of fixed size and produces a potentially endless supply of *water drops* (*i.e.*, *coded blocks*). Anyone who wishes to recover the original blocks holds a *bucket* under the fountain and collects drops until the number of drops in the bucket is slightly larger than the number of original blocks. They can then *decode* the original blocks from the collected drops.

Fountain codes are *rateless* in the sense that it is possible to produce a potentially limitless number of drops (coded blocks) from a fixed number of blocks. The rateless property allows each node to produce coded blocks independent of the other nodes, which achieves decentralization by making *every node* useful during during a bootstrap.

A key technical innovation in SeF codes is to make fountain codes secure against adversarial nodes (hence, the name secure fountain codes).[2] Fountain codes admit a computationally efficient decoding process, called a *peeling decoder*. SeF codes introduce error-resiliency in the peeling process by enabling the decoder to identify maliciously formed drops (encoded blocks) by leveraging Merkle roots stored in block-headers and the hash-chain structure of the block-headers. In essence, the idea is to use the Merkle root to check whether a coded block is maliciously formed while it is getting *peeled*. Indeed, the peeling decoder turns out to be crucial in identifying maliciously formed droplets, and thus, achieving high security.

SeF codes achieve a favorable bootstrap cost for a target storage savings. In particular, SeF codes allow the network to tune the storage savings as a parameter, depending upon how much bootstrap cost new nodes can tolerate. When using SeF codes tuned to achieve k-fold storage savings, a new node needs to contact $k + O(\sqrt{k}\ln^2(k))$ honest nodes on an average to recover the blockchain. In fact, our experiments show much better results: for $k = 1000$, the bootstrap cost is 1128, while for $k = 10000$, it is 10473.

[1] Obtaining the blockchain in this scheme is, in fact, identical to the well-known coupon collector problem. The classical analysis of the coupon collector problem yields the result. See Sec. 4.2 for details.

[2] Fountain codes have originally been designed to cater random erasures, and cannot be directly used to correct adversarial errors. See Sec. 1.2 for details.

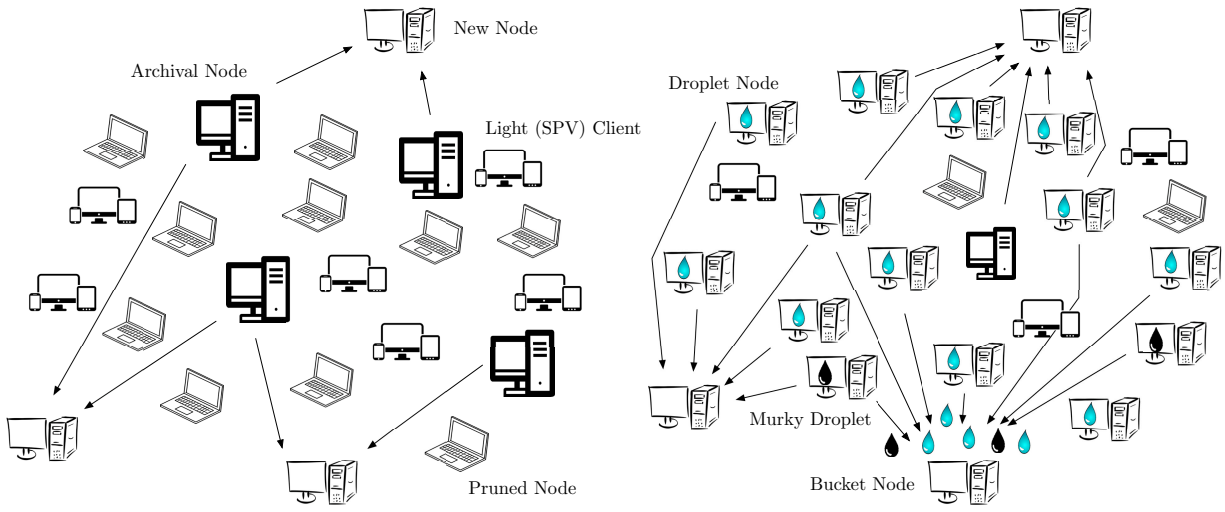

Figure 2: (a) *Current architecture for a blockchain network consists of archival nodes, pruned nodes, and light clients, out of which only archival nodes can help in bootstrapping a new node joining the network.* (b) *We envision a blockchain network mainly consisting of the proposed* **droplet nodes** *that require low storage and computation resources. During bootstrap, a new node, called a* **bucket node,** *collects sufficiently many droplets and recovers the blockchain even when some droplet nodes are adversarial, providing* **murky (malicious) droplets**. *After validating the blockchain, a bucket node will perform encoding to turn itself into a droplet node. In this way, droplet nodes will slowly replace archival nodes.*

1.1 Our Contributions

Our main focus is on decreasing the cost of storing the blockchain's historical data, which is often much larger than that of storing its *state*. (*e.g.*, The state, UTXO set, of Bitcoin is around 3GB, as compared to its overall size of 190GB [5]). The key challenge in reducing the cost of storing blockchain's historical data is that it is required to bootstrap new nodes that join the network, and bootstrpping plays a key role in scaling up the security and decentralization of the network.

We present *SeF codes* to create a blockchain network consisting of nodes with low storage resources, referred to as *droplet nodes* (see Fig. 2(b)). Every droplet node independently *encodes* validated blocks into small-sized *droplets*, thereby requiring significantly less storage space. To recover the blockchain during bootstrap, a new node acts like a *bucket*, and collects sufficiently many droplets by contacting any arbitrary subset of droplet nodes. (Hence, the terms droplets and droplet nodes, as any droplet is as useful as the other!) Even if a fraction of droplet nodes act adversarially and provide maliciously formed droplets (called *murky droplets*), our proposed decoding can identify such murky droplets and delete them. Finally, the new (bucket) node turns itself into a droplet node by validating blocks and encoding the blockchain into droplets, and the process continues.

Our objective is to build a secure digital fountain architecture that can achieve the following performance metrics. (i) Every droplet node pays only a thousandth of the storage cost, as compared to an archival full node. In other words, a droplet node on a Bitcoin network will encode $\sim 190GB$ of the blockchain to little over $195MB$. (Reducing the storage cost by a thousandth is just an example. In fact,

the storage savings can be tuned as a parameter, and it is possible to achieve any trade-off between storage savings and bootstrap cost.) (ii) Decentralization of the network enhances with every droplet node. This is achieved by ensuring that every droplet node can contribute in bootstrapping a new node. (iii) Security guarantees of the network get stronger with every droplet node. For instance, our initial analysis shows that, in a network with more than 5000 droplet nodes, even if 49% of the nodes are adversarial, bootstrapping will be successful with high probability.

1.2 Related Work

Bitcoin allows full nodes to reduce their storage costs by enabling block pruning [8]. However, pruned nodes cannot help new nodes joining the network and do not contribute in preserving the historical blockchain data. Ethereum uses state tree pruning [15] to reduce storage overhead, however, full nodes typically store the entire blockchain. A recent proposal [16] for pruning the Ethereum blockchain discusses several ways of scaling storage requirements, such as offloading the historical blockchain data to decentralized archives such as IPFS, Swarm, or BitTorrent. On the other hand, SeF codes enable full nodes to reduce their storage costs in such a way that they can still contribute in bootstrapping new nodes and preserving the blockchain history.

Ripple uses a *random sampling* scheme, referred to as *history sharding*, for enabling servers to reduce their storage in such a way that the ledger history is still preserved by the network [17]. In particular, the transaction history of the XRP Ledger is partitioned into segments, called shards. A server that has enabled history sharding acquires and stores randomly selected shards. As we discuss in Sec. 4.2, random sampling results in huge bootstrap cost.

It is worth noting that, in a conventional blockchain network, every full node stores the full history of the blockchain. From the perspective of storage, such a network can be viewed as a distributed storage system with replication. As erasure codes are known to be successful in reducing storage costs in distributed storage systems without reducing reliability [18, 19, 20], it is natural to consider erasure codes to reduce storage costs in blockchains. This idea is considered in [21, 22, 23, 24].

References [21, 22] propose *low-storage nodes* which split every block into small, fixed-sized fragments, and only store *coded fragments*. These coded fragments are obtained by linearly combining the block fragments with random coefficients. The main limitation of these works is that they only consider the the case when nodes can leave the network or can be unreachable; they do not consider adversarial nodes that can provide maliciously formed coded fragments.

In [23], the authors consider the problem of storing a blockchain with confidentiality and reduced storage. They propose to first dynamically partition the network into zones. Then each block is encrypted with a key specific to a zone and the encrypted block is distributed across the nodes in a zone using a distributed storage code, such as [18, 20].

In [24], the authors considered a sharded blockchain, and proposed to compute a coded shard by linearly combining uncoded shards. In particular, Reed-Solomon codes (see, *e.g.*, [25]) are used to generate the coded shard. With Reed-Solomon codes, it is possible to recover the original data in the presence of (a limited number of) adversarial nodes providing malicious data [25].

All these coding schemes – random linear codes, distributed storage codes, and Reed-Solomon codes – need to operate over a sufficiently large finite field, and incur high computational complexity for decoding. On the other hand, SeF codes are based on fountain codes, especially LT codes, which are substantially better in terms of computational cost (see 4.2).

It is important to note that LT codes (and, in general, fountain codes) have been designed to handle (random) erasures. While it is possible to decode from random errors (see, *e.g.*, [26, 27, 28]), adversarial

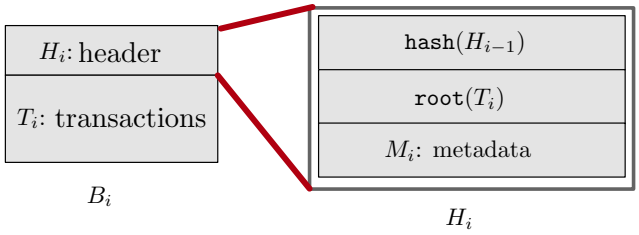

Figure 3: *Structure of a block and its header.*

errors can be difficult to deal with.[3] In general, iterative decoding algorithm for LT codes will readily propagate (and amplify) any error in the received data into the recovered data. This is because LT codes do not provide any mechanism for checking the integrity of the decoded data. The key observation of this paper is that the Merkle root of a block together with the header-chain structure of a blockchain enables one to check the integrity of the decoded blocks.

2 System Overview

2.1 Blockchain Model

A blockchain is simply a hash-based chain of blocks. Each block contains a list of transactions and a header. In particular, we consider the following generalized structure of a block (see Fig. 3).

- Let $\texttt{hash}(\cdot)$ denote a cryptographic hash function (such as SHA-256).
- Let $\texttt{root}(T)$ denote the Merkle (tree) root of a list of items T.
- The i-th block B_i in the blockchain is denoted as $B_i = \{H_i, T_i\}$, where the payload T_i is a list of transactions, and the header $H_i = \{\texttt{root}(T_i), \texttt{hash}(H_{i-1}), M_i\}$, where M_i denotes metadata such as timestamp and consensus related information such as nonce (the exact contents of the metadeta are not relevant here). We set $\texttt{hash}(H_{-1}) = 0$ as a convention.

For simplicity, we assume that each block is of size L bits.[4] For simplicity, we assume that the first L_h $(< L)$ bits of the block correspond to its header, whereas the remaining $L - L_h$ bits correspond its payload.

A blockchain network uses a consensus algorithm to determine which chain should be selected in case there is a fork. For the clarity of exposition, we focus our attention to the proof-of-work based Nakamoto consensus [7] in the paper. In the Nakamoto consensus, the chain with the most accumulated work (referred to as the longest chain) is selected in the event of a fork. In addition, there are protocol rules to determine the validity of transactions and blocks.

A typical node in a blockchain network, referred to as a full node, stores a copy of the entire blockchain, and validates new blocks as well as transactions. Whenever a new full node joins the network, it first needs to synchronize to the current state by downloading and validating the blockchain until that time.[5] A typical full node stores the entire blockchain to help bootstrap new nodes, and for preserving the history.

[3] Techniques proposed to handle adversarial errors such as [29] require shared secret between the encoder and the decoder. This is not possible in a blockchain network since nodes are supposed to encode the blockchain in a decentralized manner.

[4] We discuss how one can handle variable block sizes in Sec. 5.1.

[5] This is typically referred to as *full synchronization*. A blockchain may offer other faster ways of synchronization (*e.g.*, fast synchronization in Ethereum). However, the full synchronization is the most secure way to join a blockchain network [30].

2.2 Threat Model and Problem Formulation

We are interested in designing protocols that significantly reduce the storage costs at full nodes. There are two key components associated with blockchain storage costs: (a) The cost of storing the current state that is necessary for validating the content getting added. For example, the state can be all currently spendable transactions (*e.g.*, Bitcoin) or all current account balances (*e.g.*, Ethereum). This essentially is the information necessary for full nodes to perform transaction validation. (b) The cost of storing the blockchain's historical data. This is necessary to bootstrap new nodes that join the network, and is often much larger than the state. For example, the size of the Bitcoin state (the so-called UTXO set), is less than 3GB.

In this work, we focus our attention to reducing storage costs associated with storing the entire blockchain. Our goal is to design a protocol that enables a full node to reduce its storage space in such a way that the node is still able to help in bootstrapping a new node. We refer to a node with reduced storage space as a *droplet node*, and a new node joining the system as a *bucket node*.

Threat Model: We consider a Byzantine adversary that can control an arbitrary subset of droplet nodes. These malicious droplet nodes may collude with each other and can deviate from the protocol in any arbitrary manner, *e.g.*, by storing/sending arbitrary data to a bucket node, or remaining silent. The remaining nodes are honest and faithfully follow the protocol. We assume that the adversary is oblivious, *i.e.*, it does not observe the storage contents of droplet nodes before choosing which nodes to control. *Our goal is to design protocols that allow a bucket node to reconstruct the blockchain as long as a small number of droplet nodes are honest.* We measure the security performance of a coding scheme by the minimum number of honest droplet nodes that are sufficient to recover the blockchain with overwhelming probability (see Sec. 2.3).

Our proposed scheme assumes that a bucket node can first obtain the correct header chain. Towards this end, we assume that the majority of the consensus (*i.e.*, block producing nodes or miners) is honest. Further, we assume that the adversary is computationally bounded, and cannot construct a longer chain than the one constructed by the honest consensus.

Problem Formulation: Let t denote the current height of the (longest) blockchain, and let $B = \{B_1, B_2, \ldots, B_t\}$. For an arbitrary subset of blocks $B' \subseteq B$, let $\texttt{size}(B')$ denote the size of B' in bits. Our goal is to design a pair of encoding and decoding schemes $(\texttt{Enc}, \texttt{Dec})$, referred to as a coding scheme, that satisfy the following properties:

1. $\texttt{Enc}$ is a (randomized) *encoding scheme* that enables a full node to significantly reduce its storage space. In particular, node j computes and stores $C_j = \texttt{Enc}(B, j)$, which satisfies $\texttt{size}(C_j) \ll \texttt{size}(B)$. We refer to the *coded blocks* C_j as *droplets*, and any node storing droplets as a *droplet node*.

   As an example, using the proposed SeF codes, a droplet node can encode $191.48GB$ of the Bitcoin blockchain into $195.6MB$ droplets.

2. $\texttt{Dec}$ is a *decoding scheme* that allows a *bucket node* – a new node joining the network – to recover the blockchain B from an arbitrary set of droplet nodes that contains a sufficient number of honest droplet nodes. Specifically, there exist positive integers K, n $(\geq K)$ such that, for an arbitrary set of droplet nodes $\{j_1, j_2, \cdots, j_n\}$ that contains at least K honest ones, $\texttt{Dec}(C_{j_1}, C_{j_2}, \cdots, C_{j_n}) = B$ with overwhelming probability. Our goal is to design coding schemes that work for small K. As we will see in the next section, K depends on the amount of storage savings.

   As an example, in our proposed SeF scheme targeted at achieving $1000\times$ storage savings, a bucket node can recover the blockchain with high probability from $K \approx 1100$ honest droplet nodes.

2.3 Design Objectives

We measure the performance of a coding scheme using the following metrics:

Storage Savings of a node is the ratio of the total blockchain size to the size of the droplets it stores. Specifically, the storage savings of a droplet node j is $\frac{\texttt{size}(B)}{\texttt{size}(\texttt{Enc}(B,j))}$. For a given parameter γ, our goal is to design coding schemes that achieve the average storage savings of γ. For instance, our proposed scheme can achieve the storage savings of $1000\times$.

Bootstrap Cost: Consider a coding scheme with a storage savings of γ. For a given $0 < \delta < 1$, the bootstrap cost of a coding scheme is measured by the minimum number of honest droplet nodes $K(\gamma,\delta)$ that a bucket node needs to contact in order to ensure that the blockchain can be recovered with probability at least $1-\delta$. Note that the bootstrap cost of a coding scheme reflects the *security performance* of the system. In particular, $K(\gamma,\delta)$ can be considered as the minimum number of honest droplet nodes that the system must contain to guarantee that the historical blockchain data is available with high probability. The smaller the bootstrap cost of a coding scheme, the better the security performance of the system using the scheme. For instance, out proposed SeF codes demonstrate that, at $1000\times$ storage savings, it is possible to recover the blockchain with 99% probability when the network contains at least 1100 honest droplet nodes.

Bandwidth Overhead is the percentage overhead in terms of the amount of data that a bucket node needs to download for recovering the blockchain with high probability. Note that, in any protocol, malicious nodes can cause heavy bootstrap overheads by *surrounding* a bucket node, say by hijacking its connections, and by providing *garbage* data. To estimate the average overhead, we make the following simplifying assumption. For some $0 \leq \sigma < 1$, if a σ fraction of droplet nodes are malicious, then a droplet node contacted by a bucket node turns out to be malicious with probability σ. Here, we implicitly assume that a bucket node can contact a random subset of droplet nodes. For a coding scheme with storage savings of γ, wherein a σ fraction of droplet nodes are malicious, we denote the bandwidth overhead by $\lambda(\gamma,\sigma)$. As an example, in our proposed SeF scheme, a bucket node can recover the blockchain with an average bandwidth overhead of $\beta(1000,0.1) \sim 20\%$.

Computation Cost: We measure the computation cost of a coding scheme in terms of the number of arithmetic operations associated with the encoder `Enc` and the decoder `Dec`. In particular, the encoding cost is the average number of arithmetic operations sufficient for generating droplets, divided by the number of original blocks. Similarly, the decoding cost is the average number of arithmetic operations sufficient to recover the blockchain, divided by the number of original blocks.

In addition, we are interested in designing encoding schemes that are decentralized, as described in the following.

Decentralization: A droplet node should be able to generate its droplets without knowing what any other node in the system is storing. For instance, in proposed SeF codes, the output of $\texttt{Enc}(\cdot)$ does not depend on the identity j of a node.

3 Secure Fountain Architecture

3.1 Generic Framework

We begin with a generic framework for a coding scheme, which enables a node to *code* across blocks and save its storage space by storing only a small number of *coded blocks*. We refer to the nodes storing coded blocks as *droplet nodes*, and the coded blocks as *droplets*.

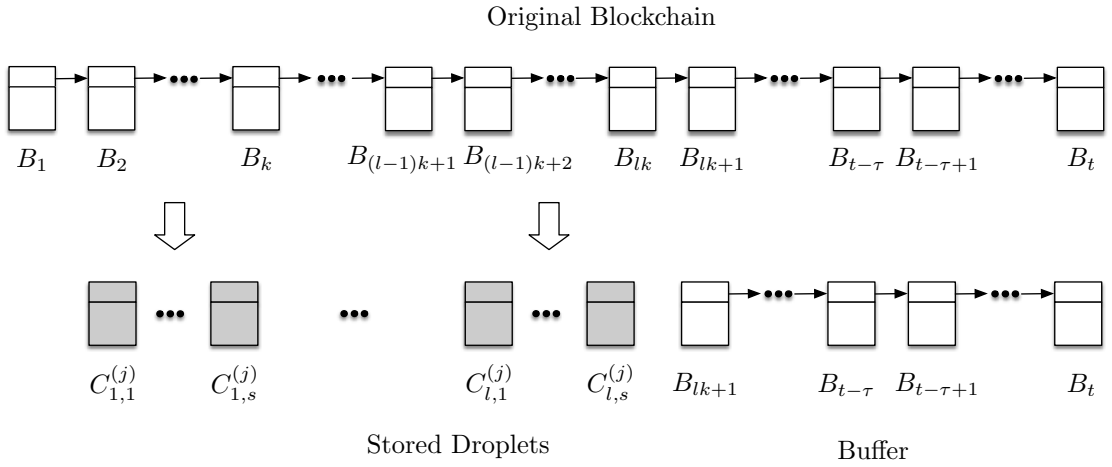

Figure 4: *Encoding happens in* **epochs**. *An epoch is defined as the time required for the blockchain to grow by k blocks. In the current epoch, when the blockchain grows by k blocks, the sub-chain of length k is encoded into s droplets. Then, the encoding process continues to the next epoch. For example, for* $k = 10000$ *and* $s = 10$*, each droplet node* **reduces its storage cost by a factor of 1/1000**. *This means a node can encode the Bitcoin blockchain of size 190GB into little over 190 MB.*

(a) Encoding: We propose to compute droplets in epochs, where an epoch is defined as the time required for the blockchain to grow by k blocks (*e.g.*, $k = 10000$). In the current epoch, when the blockchain grows by k blocks, the sub-chain of length k is *encoded* into a s of *coded blocks* (*e.g.*, $s = 10$). Then, the encoding process continues to the next epoch. To handle blockchain reorganizations due to potential forks, the most recent τ blocks are excluded from encoding and are stored in an uncoded format (*e.g.*, $\tau = 550$). In addition, each node stores the header-chain for the original blockchain.

More specifically, the first epoch starts from the $(\tau+1)$-th block. When the blockchain grows up to block $B_{k+\tau}$, a node encoded the blocks $B_1, B_2, \ldots, B_k$ into s droplets. The node then deletes the k original blocks, and only stores the s droplets for the l-the epoch. The process then continues into the next epoch. Let us denote the s droplets stored by node j in epoch l as $C_{l,1}^{(j)}, C_{l,2}^{(j)}, \ldots, C_{l,s}^{(j)}$. See Fig. 4 for a schematic representation.

(b) Decoding: Consider a new node, referred to as *a bucket node*, joining the system when the height of the blockchain is t. Let $e = \lfloor (t-\tau)/k \rfloor$. The bucket node first contacts an arbitrary subset of n droplet nodes (of sufficient size), and collects (downloads) their droplets for epochs $1 \leq l \leq e$. The bucket node also downloads the uncoded blocks (from B_{ek+1} onward) from the droplet nodes.

The encoding should be performed in such a way that the bucket node can recover the blockchain from the collected droplets. In particular, let us denote the n droplet nodes that are contacted as $\{j_1, j_2, \ldots, j_n\}$. Then, for every epoch $1 \leq l \leq e$, the bucket node should be able to *decode* the sub-chain $B_{(l-1)k+1}$, $B_{(l-1)k+2}$, …, B_{lk} from the ns droplets $\{C_{l,p}^{(j_i)} : 1 \leq i \leq n, 1 \leq p \leq s\}$.

3.2 Secure Fountain (SeF) Codes

We propose to perform encoding using a Luby transform (LT) code [12]. At the core of LT codes lies the concept of a *fountain code*. A fountain code takes as an input a vector of k input symbols, and produces

a potentially limitless stream of output symbols.[6] The main property that is required of a fountain code is that it should be possible to recover the k input symbols from any set of K ($\geq k$) output symbols with high probability. The parameter K is desired to be very close to k and ideally equal to k.

LT codes admit a computationally efficient decoding procedure known as *peeling decoder* (also known as *belief-propagation or message-passing decoder*) [31]. However, a key limitation of the peeling decoder is that it cannot handle the maliciously produced output symbols. To overcome this issue, we propose an error-resilient peeling decoder that exploits the chained structure of headers along with Merkle roots. We refer to LT codes with the error-resilient peeling decoder as *Secure Fountain (SeF) codes*.

For simplicity, we focus our attention to the first epoch in the following. The encoding and decoding procedures are the same for all subsequent epochs.

3.3 Encoder of a Luby Transform (LT) Code

Degree Distribution: An important ingredient in encoding is a *degree distribution* $\mu(\cdot)$, which is a discrete probability mass function on integers between 1 and k. We use the *robust soliton distribution* proposed by Luby in [12].[7] We start with the *ideal soliton distribution* defined below [12]:

$$\rho(d) = \begin{cases} \frac{1}{k} & \text{for } d = 1 \\ \frac{1}{d(d-1)} & \text{for } d = 2,\ldots,k. \end{cases} \tag{1}$$

This distribution optimizes the expected number of blocks that are decoded in each iteration. However, it has a low probability of successfully recovering the input symbols (the blockchain in our case), and needs to be modified to add robustness. Towards this, we introduce the following notation. Given $0 < \delta < 1$ and $c > 0$, define

$$R = c\sqrt{k}\ln\left(\frac{k}{\delta}\right). \tag{2}$$

Next, define a function $\theta(\cdot)$ as

$$\theta(d) = \begin{cases} \frac{R}{dk} & \text{for } d = 1,\ldots,k/R-1 \\ \frac{R}{k}\ln\left(\frac{R}{\delta}\right) & \text{for } d = k/R \\ 0 & \text{for } d = k/R+1,\ldots,k. \end{cases} \tag{3}$$

As we will see in Sec. 4, the parameter δ gives a (conservative) bound on the probability that the decoding fails to succeed after a certain number of droplets are downloaded. The parameter c is a free parameter that can be tuned to optimize the number of droplets required to recover the blockchain. Adding the ideal soliton distribution $\rho(\cdot)$ to $\theta(\cdot)$ and normalizing gives the *robust soliton distribution* as [12]:

$$\mu(d) = \frac{\rho(d)+\theta(d)}{\beta}, \quad \text{for } d = 1,\ldots,k, \tag{4}$$

[6] Here, a symbol refers to a sequence of bits, and all symbols are assumed to be of the same size. Note that a block can be considered as a symbol.

[7] While the encoder and the decoder (in Sec. 3.4) are valid for any degree distribution, the probability of successfully decoding the input symbols (the blockchain in our case) from a given number of output symbols (droplets in our case) depends on the choice of the degree distribution. The robust soliton degree distribution is shown to have good probability of success (without any corrupted nodes) in [12].

where

$$\beta = \sum_{d=1}^{k} \rho(d) + \theta(d). \tag{5}$$

Encoding: A droplet node computes its j-th droplet C_j, $1 \leq j \leq s$, independent of the other droplets, as follows. (Recall that each block is of size L bits, and can be considered as a symbol of a finite field $\mathbb{F}_{2^L}$.)

1. Randomly choose the *degree* d of the droplet from the degree distribution $\mu(\cdot)$.
2. Choose, uniformly at random, d distinct blocks from B, and set the droplet C_j as the sum of these d blocks over $\mathbb{F}_{2^L}$, *i.e.*, C_j is a bit-wise XOR of these d blocks. (These d blocks are called *neighbors* of C_j.)

   Denote $C_j = \{H_j, T_j\}$, where H_j are the first L_h bits of C_j, referred as its header, and T_j are the remaining $L - L_h$ bits of C_j, referred as its payload.[8]
3. Store C_j along with a length-k binary vector v_j computed as follows: if the m-th block B_m is among the d blocks chosen to compute C_j then the m-th entry of v_j is 1, else it is 0.

In addition to s droplets, each droplet node stores the header-chain $H = \{H_1, H_2, \ldots, H_k\}$ for the original blockchain. As we will see, vector v_j and header-chain H are required in the decoding process. In particular, v_j will be used to identify which original blocks are combined to generate C_j, while the header-chain will enable the decoder to identify maliciously formed droplets.

Remark 1. *There are other, potentially more efficient, ways to convey which original blocks are combined to compute a droplet C_j than storing the length-k binary vector. For instance, it is possible to store a seed using which a pseudo-random generator can produce the binary vector v_j. Since storing v_j takes a small size, we do not consider other methods. We refer the reader to [12, 13] for more details.*

3.4 Error-Resilient Peeling Decoder

Consider a bucket node that is interested in recovering the blockchain B. It contacts an arbitrary subset of n ($n \geq k$) droplet nodes, and downloads the stored data. This includes droplets C_j's, vectors v_j's, and header-chains. Without loss of generality, let us (arbitrarily) label the downloaded droplets as $C_1, C_2, \ldots, C_{ns}$.

We assume that the bucket node has access to the correct header-chain, denoted as $H = \{H_1, H_2, \ldots, H_k\}$. We discuss several ways of obtaining the correct header-chain in Sec. 5.2. Then, the node leverages this header-chain to perform error-resilient peeling decoding for an LT code, described as follows.

The decoding proceeds in iterations. In each iteration the algorithm decodes at most one block until all the blocks are decoded, otherwise the decoder declares failure. The details are as follows.

1. *Initialization:* Form a bipartite graph G with the k original blocks as left vertices and the ns droplets as right vertices. There is an edge connecting a droplet C_j to an original block B_m if B_m is used in computing C_j. (Recall that this can be identified using v_j.)

   Set $\hat{B}_m = \texttt{NULL}$ for $m = 1, 2, \ldots, k$, where NULL denotes null value.

   Set iteration number $i = 1$ and $G^{i-1} = G$.
2. Find a droplet C_l that is connected to *exactly one* block B_m in G^{i-1}. (Such a droplet is called a *singleton*.)

   If there is no singleton, the decoding halts and declares failure.

[8] Note that the header and payload of a coded block may not have any semantic meaning.

3. Let H_l and T_l be the header and payload of C_l, respectively.
   (a) If H_l matches with the header H_m in the header-chain H and if the Merkle root of T_l, $\texttt{root}(T_l)$, matches with the one stored in H_m, then set $\hat{B}_m = C_l$. (In this case, the droplet C_l is said to be *accepted*, and the m-th block is said to be decoded to $\hat{B}_m$.)
   (b) Otherwise, delete C_l together with all its incoming edges from G^{i-1} to obtain G^i. (In this case, the droplet C_l is said to be *rejected*.)
   Increment i by 1.
   Go to Step (2).
4. Set $C_{l'} \leftarrow C_{l'} \oplus \hat{B}_m$ for all droplets $C_{l'}$ that are connected to B_m in G^{i-1}. (Here, $\oplus$ denotes the sum over $\mathbb{F}_{2^L}$, *i.e.*, bit-wise XOR.)
5. Remove all the edges connected to the block B_m from G^{i-1} to obtain G^i.
6. Increment i by 1.
7. If all the original blocks are not yet decoded, go to Step (2).

Note that Step (3) differentiates the error-resilient peeling decoder from the classical peeling decoder for an LT code [12]. More specifically, the classical peeling decoder always *accepts* a singleton, whereas the error-resilient peeling decoder may *reject* a singleton if its header and/or the Merkle root does not match with the one stored in the header-chain. As we will see in the next section, comparing the header and verifying the Merkle root provides error resiliency. We present a toy example for the decoder in Appendix A.

If the decoder declares failure, the bucket node can simply contact additional droplet nodes to collect more droplets until it can find a singleton. In particular, suppose it contacts $\hat{n}$ droplet nodes for some predetermined parameter $\hat{n} \ll n$. Arbitrarily label the downloaded droplets as $C_{ns+1}, C_{ns+2}, \ldots, C_{(n+\hat{n})s}$. Append these droplets as right vertices in G^{i-1}. Add an edge connecting a droplet C_j, $ns+1 \leq j \leq (n+\hat{n})s$, to block B_m if the m-th block is not yet decoded and B_m is used in computing C_j. If there is a singleton amongst the newly downloaded droplet, then proceed to Step (3). Otherwise, it contacts $\hat{n}$ additional droplet nodes. The decoder declares failure when the bucket node is unable to find additional droplet nodes.

After the bucket node decodes the original blockchain, it computes a droplet block following the encoder in Sec. 3.3. At this point, the bucket node turns itself into a droplet node which, in turn, can help a new bucket node. As we will show in the next section, when the bucket node contacts a set of droplet nodes of sufficiently large size, it can successfully decode the original blockchain.

4 Performance Analysis

It is straightforward to demonstrate that there is a trade-off between the storage savings and the bootstrap cost. For simplicity, we focus our attention to coding schemes in which each droplet node achieves the storage savings of γ.[9]

Proposition 1. *For any $0 \leq \delta < 1$, the bootstrap cost of any coding scheme in which each droplet node achieves the storage savings of γ is lower bounded by γ, i.e., $K(\gamma, \delta) \geq \gamma$.*

[9]When this is not the case, using the similar proof as that of 1, it is easy to show that the lower bound on the security performance for a coding scheme is γ_{min}, which is the minimum storage savings achieved by the scheme.

Proof. Suppose that there exist n honest droplet nodes from which it is possible to recover the blockchain. In order to recover the blockchain, the total size of the downloaded data must be at least the size of the blockchain. Further, each of the n droplet nodes contributes $\texttt{size}(B)/\gamma$. Therefore, n should be at least γ. □

4.1 SeF Codes

In the following, we analyze the performance of SeF codes. First, we outline how an adversarial droplet node can behave while encoding with SeF codes.

Adversarial Behavior Against SeF Codes: In addition to staying silent when contacted by a bucket node, an adversarial droplet node can act maliciously in the following two ways:

- Store arbitrary values for C_l, v_l, and H. In particular, let $\mathbf{B}$ be a $k \times L$ binary matrix, in which the i-th row is the block B_i. Then, for an honest node j, v_j and C_j are such that $C_j = v_j\mathbf{B}$. On the other hand, an adversarial node l can store any values for C_l and v_l such that $C_l \neq v_l\mathbf{B}$. We refer to such a droplet as a *murky droplet*.
- Arbitrarily choose degree d, and arbitrarily choose d blocks to compute a droplet. We refer to such a droplet as an *opaque droplet*. This attack is essentially targeted at increasing the probability of decoding failure.

We refer to the droplets computed by honest nodes as *clear droplets*.

We make the following assumptions for simplicity: (i) the storage space required to store blocks in the current epoch and the header-chain is negligible as compared to the size of the blockchain;[10] (ii) we assume that a droplet node randomly samples degrees and neighbors in the first epoch for each of its droplets (see Step (2) of the encoder). The node then uses the same degree and neighbors in subsequent epochs; and (iii) as compared to k, it takes negligible number of arithmetic operations to compute the Merkle root for a block.

We begin with an important property of SeF codes which is crucial for characterizing their performance.

Lemma 1. *Suppose the bucket node collects n droplets. If these n droplet contain at least* $k+O\left(\sqrt{k}\ln^2(k/\delta)\right)$ *clear droplets, then the probability that the error-resilient peeling decoder fails to recover all the k blocks is at most* δ.

It is important to note that the number of droplet nodes that the bucket node would need to contact until it collects $k+O\left(\sqrt{k}\ln^2(k/\delta)\right)$ clear droplets will vary depending upon how many adversarial nodes it ends up contacting. However, as long as more than $\left(k+O\left(\sqrt{k}\ln^2(k/\delta)\right)\right)/s$ droplet nodes are honest, Lemma 1 ensures that it is possible to recover the blockchain with probability at least $1-\delta$. Therefore, SeF codes achieve the bootstrap cost of $K(k/s,\delta) = \left(k+O\left(\sqrt{k}\ln^2(k/\delta)\right)\right)/s$.

Further, note that a bucket node needs to download $O\left(\sqrt{k}\ln^2(k/\delta)\right)$ droplets in excess. Thus, the bandwidth overhead is $O\left(\ln^2(k/\delta)/\sqrt{k}\right)$. We summarize the performance of SeF codes in the theorem below.

Theorem 1. *SeF codes are decentralized and achieve the following performance measures:*

1. *Storage savings:* $\gamma = k/s$;

[10] Note that, since the blockchain is an ever growing data structure, this assumption can be easily justified.

2. *Bootstrap cost:* $K(k/s,\delta) = \frac{k+O\left(\sqrt{k}\ln^2(k/\delta)\right)}{s}$;
3. *Bandwidth overhead:* $\beta(k/s,\sigma) = O\left(\frac{\ln^2(k/\delta)}{(1-\sigma)\sqrt{k}}\right)$;
4. *Computation cost: encoding cost* = $O\left(\frac{s\ln(k/\delta)}{k}\right)$, *decoding cost* = $O\left(\frac{\ln(k/\delta)}{1-\sigma}\right)$.

4.2 Comparison with Random Sampling and Reed-Solomon Codes

Random Sampling: Consider a very simple scheme as follows. In each epoch, a droplet node stores s distinct blocks that are selected uniformly at random.[11] Note that this scheme achieves the storage savings of k/s, since the storage grows by s blocks when the blockchain grows by k blocks.

As noted in [12], random sampling can be considered as a special case of LT codes for the following degree distribution (referred to as *all-at-once* distribution).

$$\rho(d) = \begin{cases} 1 & \text{if } d = s \\ 0 & \text{otherwise.} \end{cases} \tag{6}$$

On the positive side, this simple scheme has small encoding cost $O(s/k)$ and decoding cost $O(k/(1-\sigma))$. However, a major limitation of the random sampling scheme is that it has poor security performance. To see this, consider $s = 1$ for simplicity[12], and focus on the first epoch. Notice that recovering the blockchain is equivalent to the coupon collector problem. It is well-know that, in order to recover the blockchain with probability at least $1-\delta$, it is necessary to have $k\ln(k/\delta)$ honest droplet nodes on average. Therefore, the security performance worsens multiplicatively as storage savings increase.

Reed-Solomon (RS) Codes: Consider $L' \geq L$ such that L divides L'. Note that $\mathbb{F}_{2^{L'}}$ is an extension field of $\mathbb{F}_{2^L}$. For the simplicity of the analysis, we assume that $2^{L'}$ is comparable with N, the total number of droplet nodes.

Now, we describe the encoding of an RS code, focusing on the first epoch. A droplet node selects s points from $\mathbb{F}_{2^{L'}}$ uniformly at random, and stores the evaluations the following polynomial $B(x)$ on these points: $B(x) = B_1 + B_2 x + \cdots + B_i x^{i-1} + B_k x^{k-1}$.

Note that it is possible to interpolate $B(x)$ from its evaluations on any k distinct points. This implies that an RS code enables a bucket node to recover the blockchain from any k/s honest droplet nodes with high probability. Therefore, in principle, an RS code can achieve the *optimal* security performance of k/s. However, since a bucket node cannot distinguish an honest droplet node from a malicious one just by observing its stored droplets, achieving the security performance of k/s requires a bucket node to employ the following decoding scheme. For every subset of k/s droplet nodes, recover a candidate blockchain and check its validity using the header chain. This decoding scheme has a prohibitive decoding cost.

One can also use specific decoding schemes designed for RS codes to recover the blockchain in the presence of malicious droplet nodes. This allows one to tolerate $(N-k)/2$ adversarial droplets, resulting in the security performance of $(N+k)/2$. Moreover, the computational cost of best known theoretical decoding algorithm is $O(Npolylog(N))$.

[11] It is worth noting that a similar scheme is used in the Ripple blockchain, and is referred to as *history sharding* [17]. In history sharding, the transaction history of the XRP Ledger is partitioned into segments, called shards. A server that has enabled history sharding acquires and stores randomly selected shards.

[12] It is worth noting that, for $s > 1$, the random sampling scheme is equivalent to the coupon collector with group drawing problem, and the analysis is similar, see *e.g.*, [32].

5 Practical Issues

5.1 Tackling Variability in Block Size

Until now, we have assumed that all the blocks have the same size. On the other hand, popular blockchains such as Bitcoin and Ethereum produce blocks with variable size (see [33] and [34]), respectively). In this section, we discuss how to handle variability in block size.

In a blockchain with maximum limit on the block size, the simplest way to deal with variable block sizes is to zero pad every block to the maximum size during encoding. However, when the average block size is smaller than the maximum, this results in higher storage costs. In the following, we discuss two simple and efficient protocols to handle variable block size.

1. **Adaptive zero-padding:** Recall that in LT encoding a node first chooses a degree d using a degree distribution. Then, it chooses d distinct blocks from the epoch under consideration. Then, for computing the bit-wise XOR, the node zero-pads the blocks to the largest block among the d blocks. We refer to this procedure as adaptive zero-padding. Adaptive zero-padding performs well when the variance in block size is small. However, it can perform poorly when the variance in block size is large. To overcome this issue, we propose to concatenate several contiguous blocks in the following.

2. **Block Concatenation:** A natural way to reduce variance in block size is to first concatenate blocks to form *super-blocks* of approximately same size, and then perform encoding on the super-blocks. For simplicity, we assume here that the block header contains the size of the block. We define an epoch as the time required for the blockchain to grow by k super-blocks. The actual number of blocks produced in an epoch will vary depending on the block sizes. Let L denote the maximum block size, and let $L_s \geq L$ be a design parameter. For example, in our simulations we use $L_s = 1, 5,$ and $10MB$ for the Bitcoin blockchain, which has $L = 1MB$.

   The details are given in the following. For simplicity, we focus on the first epoch without loss of generality. The block concatenation procedure is the same for all subsequent epochs. We need the following notation: For two binary strings B_i and B_j, let $B_i \,||\, B_j$ denote their concatenation.

   Block concatenation procedure:

   (i) *Initialization:* Set super-block count $j = 1$ and block count $i = 1$.

   (ii) If $j \leq k$, set super-block $\bar{B}_j = \texttt{NULL}$.

   a. If $\texttt{size}(\bar{B}_j \,||\, B_i) \leq L_s$,
   Set $\bar{B}_j \leftarrow \bar{B}_j \,||\, B_i$.
   Increment i.
   Go to Step (ii)-a.

   b. Else,
   Increment j.
   Go to Step (ii).

   LT encoding is then performed on super-blocks $\bar{B}_1, \bar{B}_2, \cdots, \bar{B}_k$. Note that the encoder may still need to use adaptive zero padding while XORing super-blocks. However, the size of a super-block is at least $L_s - L$. Thus, choosing L_s fairly larger than L ensures small variance in super-block sizes, reducing the overhead incurred by adaptive zero padding.

   In the error-resilient peeling decoder in Sec. 3.4, we modify Step 3 to check all the blocks in the singleton super-block. To be more precise, consider Step 2 in at which the bucket node finds a

singleton super-block, say C_l. As we assume that the header contains the block size, the bucket node knows from the header chain that the l-th super-block should be a concatenation of blocks $i+1$ to $i+p$ for some $i \geq 0$ and $p \geq 0$. In other words, if C_l is a clear droplet, then it will have the following structure: $C_l = \{\{H_{i+1}, T_{i+1}\}, \{H_{i+2}, T_{i+2}\}, \cdots, \{H_{i+p}, T_{i+p}\}\}$ for some i and p.

Assuming that the headers have the same size and the block-size is included in the header, it is possible to decompose C_l in the following form: $C_l = \{\{\hat{H}_{l_1}, \hat{T}_{l_1}\}, \{\hat{H}_{l_2}, \hat{T}_{l_2}\}, \cdots, \{\hat{H}_{l_p}, \hat{T}_{l_p}\}\}$. Then, in the Step 3, the singleton is accepted only if, for each $1 \leq j \leq p$, $\hat{H}_{l_j}$ matches with H_{i+j} and $\texttt{root}\left(\hat{T}_{l_j}\right)$ matches with the Merkle root in H_{i+j}. Otherwise, the singleton is rejected. The rest of the decoding algorithm remains the same.

5.2 Obtaining the Correct Header-Chain

While describing the error-resilient peeling decoder, we assumed that a bucket node has an access to the correct header-chain. In this section, we discuss how a bucket node can obtain the correct header chain. A bucket node first queries a large number of droplet nodes, and obtains the longest valid header-chain. Assuming that the majority of the mining power is honest, the longest valid header-chain is the correct header chain with high probability. Thus, as long as the bucket node can contact one honest droplet node, it obtains the correct header-chain.

It is worth noting that light (or SPV) clients, which are an integral part of several practical blockchain protocols like Bitcoin and Ethereum, are designed to obtain the longest header-chain. Thus, a bucket node can first act as a light client before starting to collect the droplets.

6 Simulation Results

In this section, we numerically analyze the performance for the proposed SeF codes. We consider two parameter settings, targeted at $1000\times$ storage savings: (i) ($k = 1000$, $s = 1$); and (ii) ($k = 10000$, $s = 10$). Recall that we make the following assumption about the network model during the bootstrap process: if a σ fraction of droplet nodes are malicious, then a droplet node contacted by a bucket node turns out to be malicious with probability σ.

First, we compute numerical results with the following assumptions: (a) all blocks are of the same size; and (b) the block size is sufficiently large so that we can ignore the overhead due to storing the binary-vector v_j associated with a droplet C_j.[13] With these assumptions, both $(k = 1000, s = 1)$ and $(k = 10000, s = 10)$ achieve the storage savings of $\gamma = 1000\times$.

We measure the bootstrap overhead as the percentage overhead in terms of the the number of blocks required to be downloaded in order to successfully recover the blockchain. In particular, considering one epoch, the bucket node keeps collecting droplets until the error-resilient peeling decoder can successfully recover the blockchain. The experiment is repeated 100 times to compute the statistics for the bootstrap overhead. We consider the following set of parameters for LT codes (cf. (2)): $c = \{0.01, 0.03, 0.1, 0.3\}$ and $\delta = \{0.1, 0.3, 0.5, 0.7\}$. We choose the values of c and δ that result in the smallest average bootstrap overhead. We plot the average bootstrap overhead versus the fraction of malicious droplet nodes σ in Fig. 5. Observer that $k = 10000$, $s = 10$ results in a smaller bootstrap overhead as compared to $k = 1000$, $s = 1$. This is because LT codes are more efficient for larger k. On the other hand, a drawback of selecting

[13] Note that, for $k = 1000$ and $k = 10000$, the size of v_j is 125 bytes and 1250 bytes, respectively. Thus, for the block size of over $1MB$, this assumption is reasonable.

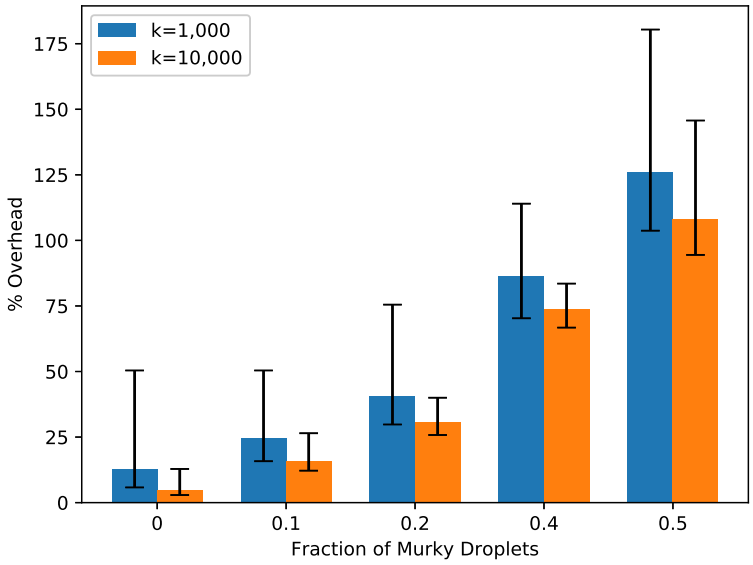

Figure 5: *Bootstrap overhead versus the fraction of malicious droplet nodes.*

Table 1: Results on the Bitcoin blockchain for $k = 1000$ and $s = 1$.

$k = 1000$, $s = 1$	Adaptive Zero Padding	Block Concatenation to 1MB	Block Concatenation to 5MB	Block Concatenation to 10MB
Storage Savings (γ)	749.26	895.97	961.24	978.94
Bootstrap Overhead (β) (All Honest)	50.55%	26.05%	17.51%	15.43%
Bootstrap Overhead (β) (10% Malicious)	66.79%	39.55%	30.09%	27.79%

larger k is that a droplet node needs larger buffer space to store the blocks of the current epoch before they can be encoded.

6.1 Simulations on the Bitcoin Blockchain

In this section, we describe experiments carried out on the Bitcoin blockchain. To tackle variability in block sizes, we use adaptive zero padding and block concatenation as discussed in Sec. 5.1. We list the storage savings γ and the bootstrap overhead β in Tables 1 and 2. Since the block size variability in the Bitcoin is fairly large, simply using adaptive zero padding does not yield a good performance. On the other hand, block concatenation successfully mitigates the block size variability. As we increase the super-block size from $1MB$ to $10MB$, the variance in the super-block size reduces. This results in the performance improvement.

Table 2: Results on the Bitcoin blockchain for $k = 10000$ and $s = 10$.

$k = 10000$, $s = 10$	Adaptive Zero Padding	Block Concatenation to 1MB	Block Concatenation to 5MB	Block Concatenation to 10MB
Storage Savings (γ)	745.17	894.71	958.68	976.64
Bootstrap Overhead (β) (All Honest)	40.54%	17.05%	9.25%	7.24%
Bootstrap Overhead (β) (10% Malicious)	55.36%	29.39%	20.78%	18.56%

A Toy Example for the Error-Resilient Peeling Decoder

We describe the decoder algorithm on the example shown in Fig. 6. There are $k = 6$ blocks, and the bucket node has collected 9 droplets, denoted as $C_1, C_2, \ldots, C_9$. The corresponding bipartite graph G is shown in Fig. 6. Suppose droplets C_2 and C_6 are murky. Note that the decoder does not know this at the beginning of the decoding. We assume that the bucket node has access to the correct header-chain $H = \{H_1, H_2, \ldots, H_6\}$.

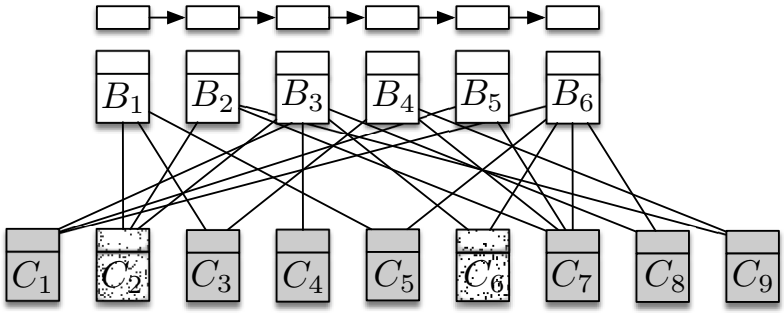

Figure 6: Toy example for the error-resilient peeling decoder for $k = 6$ blocks and $ns = 9$ droplets. The bipartite graph G at the initialization Step (1).

Consider the first iteration. The decoder begins with finding a droplet, called singleton, that is connected to exactly one node in $G^0 = G$. The only singleton in G^0 is C_4, and is connected to B_3 (see Fig. 7).

The decoder then compares the header of C_4 with H_3 from the header-chain, and then verifies whether the Merkle root of the payload of C_4 matches with the Merkle root stored in H_3. Since C_4 is clear, the decoder would accept it, and decodes $\hat{B}_3 = C_4$. Then, it adds C_4 to the neighbors of B_3 excluding C_4, namely C_1, C_2, C_6, and C_8. We refer to this as updating the other neighbors of B_3. It then removes the edges from B_3 to obtain G^1 as shown in Fig. 8.

In iteration 2, there are two singletons C_6 and C_8. Suppose the decoder selects C_6. Since the droplet is murky, the matching fails for either the header or the Merkle root (or both), and the decoder rejects C_6. It deletes C_6 along with its edges from G^1 to obtain G^2 as shown in Fig. 9.

In iteration 3, the only singleton droplet is C_8 that is connected to B_6. Since the droplet is clear, the headers and the Merkle roots would match. The decoder accepts C_8 and decodes $\hat{B}_6 = C_8$. It updates the

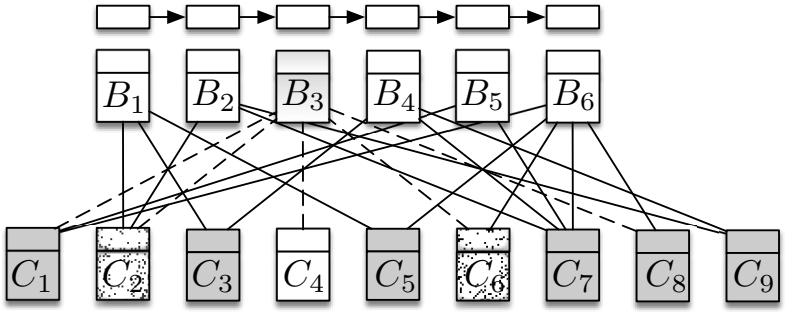

Figure 7: Iteration 1 with the bipartite graph G^0: the decoder accepts C_4 and decodes B_3.

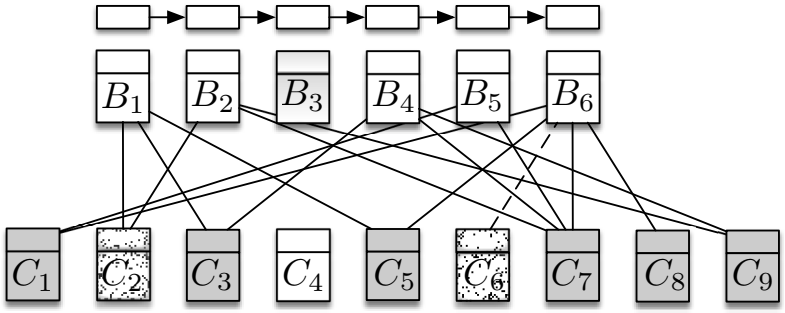

Figure 8: Iteration 2 with the bipartite graph G^1: the decoder rejects C_6.

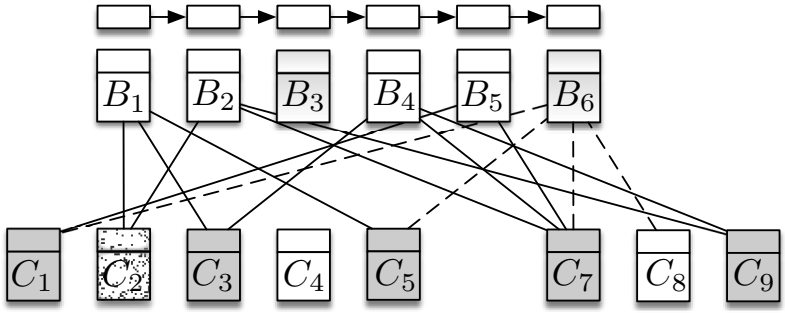

Figure 9: Iteration 3 with the bipartite graph G^2: the decoder accepts C_8 and decodes B_6.

other neighbors of B_6, and removes the edges from B_6 to obtain G^3 as shown in Fig. 10.

In iteration 4, there are two singletons C_1 and C_5. Suppose the decoder selects C_5. Since the droplet is clear, the headers and the Merkle roots would match. The decoder accepts C_5 and decodes $\hat{B}_1 = C_5$. It updates the other neighbors of B_1, removes the edges from B_1 to obtain G^4 as shown in Fig. 11.

In iteration 5, there are three singletons C_1, C_2, and C_3. Suppose the decoder selects C_2. Since the droplet is murky, the matching fails for either the header or the Merkle root (or both), and the decoder rejects C_2. It deletes C_2 to obtain G^5 as shown in Fig. 12.

In iteration 6, out of the two singletons C_1 and C_3, suppose the decoder selects C_3. Since the droplet is clear, the headers and the Merkle roots would match. The decoder accepts C_3 and decodes $\hat{B}_5 = C_3$. It updates the other neighbors of B_5, and removes the edges from B_5 to obtain G^6 as shown in Fig. 13.

In iteration 7, the decoder chooses the singleton C_9. It accepts it, and decodes $\hat{B}_2 = C_9$. It updates

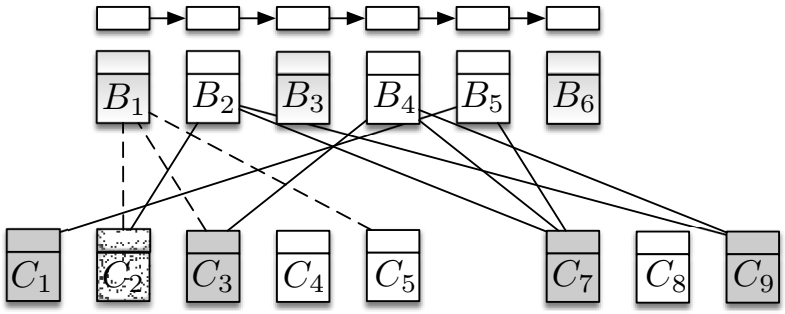

Figure 10: Iteration 4 with the bipartite graph G^3: the decoder accepts C_5 and decodes B_1.

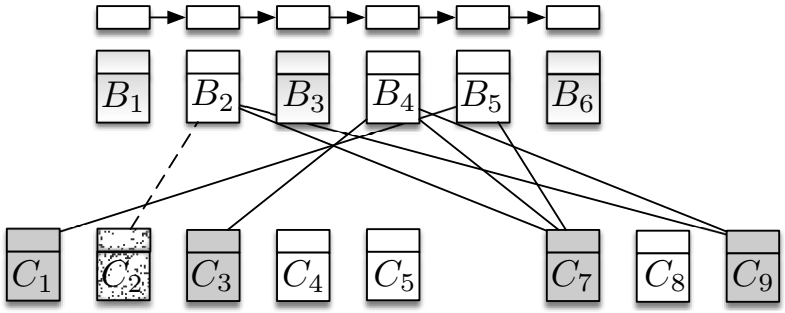

Figure 11: Iteration 5 with the bipartite graph G^4: the decoder rejects C_2.

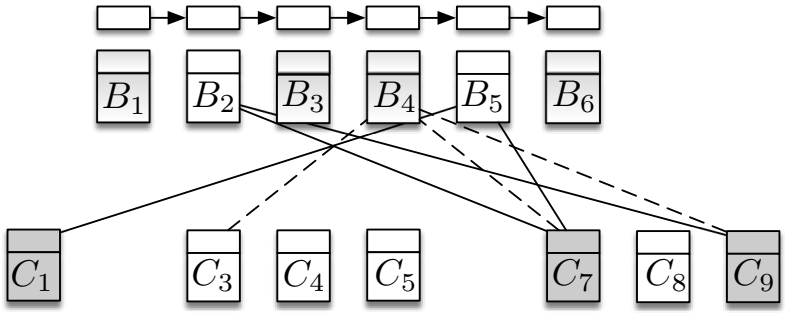

Figure 12: Iteration 6 with the bipartite graph G^5: the decoder accepts C_3 and decodes B_4.

the other neighbors of B_2. The graph G^7 after removing edges from B_2 is shown in Fig. 14.

Finally, iteration 8, the the decoder chooses the singleton C_7. It accepts it, and decodes $\hat{B}_5 = C_7$. As all the 6 blocks are decoded, the decoder stops.

References

[1] N. Teslya and I. Ryabchikov, "Blockchain-based platform architecture for industrial iot," in *2017 21st Conference of Open Innovations Association (FRUCT)*, Nov 2017, pp. 321–329.

[2] A. Azaria, A. Ekblaw, T. Vieira, and A. Lippman, "Medrec: Using blockchain for medical data access and permission management," in *2016 2nd International Conference on Open and Big Data (OBD)*, Aug 2016, pp. 25–30.

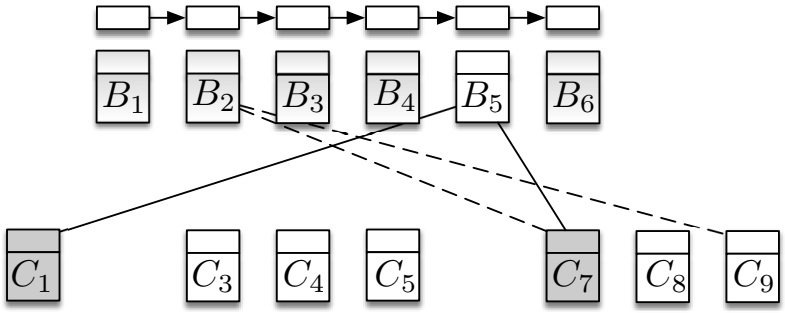

Figure 13: Iteration 7 with the bipartite graph G^6: the decoder accepts C_9 and decodes B_2.

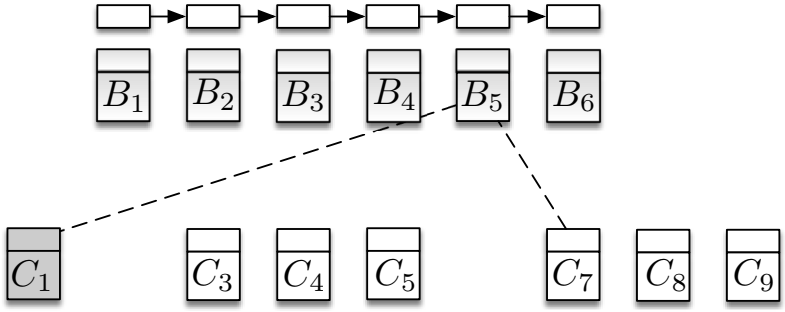

Figure 14: Iteration 8 with the bipartite graph G^7: the decoder accepts C_7 and decodes B_5.

[3] M. Mettler, "Blockchain technology in healthcare: The revolution starts here," in *2016 IEEE 18th International Conference on e-Health Networking, Applications and Services (Healthcom)*, Sept 2016, pp. 1–3.

[4] M. J. Casey and P. Wong, "Global supply chains are about to get better, thanks to blockchain," *Harvard Business Review*, Mar 2017. [Online]. Available: https://hbr.org/2017/03/global-supply-chains-are-about-to-get-better-thanks-to-blockchain

[5] "Blockchain Luxembourg S.A." https://www.blockchain.com/charts/blocks-size, [Online; Accessed on 04/15/2019].

[6] Ripple Documentation, "Capacity planning," https://developers.ripple.com/capacity-planning.html, [Online; Accessed on 04/15/2019].

[7] S. Nakamoto, "Bitcoin: A peer-to-peer electronic cash system," 2009. [Online]. Available: http://www.bitcoin.org/bitcoin.pdf

[8] BitcoinCore Documentation, "Running a full node," https://bitcoin.org/en/full-node#what-is-a-full-node, [Online; Accessed on 04/15/2019].

[9] G. Karame and E. Audroulaki, *Bitcoin and Blockchain Security*. Norwood, MA, USA: Artech House, Inc., 2016.

[10] Bitcoin Wiki, "Full node," https://en.bitcoin.it/wiki/Full_node, Feb 2019, [Online; Accessed on 04/15/2019].

[11] J. W. Byers, M. Luby, M. Mitzenmacher, and A. Rege, "A digital fountain approach to reliable distribution of bulk data," in *Proceedings of the ACM SIGCOMM '98 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication*, ser. SIGCOMM '98. New York, NY, USA: ACM, 1998, pp. 56–67.

[12] M. Luby, "LT codes," in *43rd Symposium on Foundations of Computer Science (FOCS 2002), 16-19 November 2002, Vancouver, BC, Canada, Proceedings*, 2002, p. 271.

[13] D. J. C. MacKay, "Fountain codes," *IEE Proceedings - Communications*, vol. 152, no. 6, pp. 1062–1068, Dec 2005.

[14] A. Shokrollahi and M. Luby, "Raptor codes," *Foundations and Trends in Communications and Information Theory*, vol. 6, no. 3–4, pp. 213–322, 2011. [Online]. Available: http://dx.doi.org/10.1561/0100000060

[15] V. Buterin, "State tree pruning," https://blog.ethereum.org/2015/06/26/state-tree-pruning/, Jun 2015, [Online; Accessed on 04/15/2019].

[16] P. Szilágyi, "Pruning historical chain segments," https://gist.github.com/karalabe/60be7bef184c8ec286fc7ee2b35b0b5b, Nov 2018, [Online; Accessed on 04/15/2019].

[17] R. Documentation, "History sharding," https://developers.ripple.com/history-sharding.html, [Online; Accessed on 04/15/2019].

[18] A. G. Dimakis, K. Ramchandran, Y. Wu, and C. Suh, "A survey on network codes for distributed storage," *Proceedings of the IEEE*, vol. 99, no. 3, pp. 476–489, March 2011.

[19] J. S. Plank, "Erasure codes for storage systems: A brief primer," *;login: the Usenix magazine*, vol. 38, no. 6, December 2013.

[20] C. Huang, H. Simitci, Y. Xu, A. Ogus, B. Calder, P. Gopalan, J. Li, and S. Yekhanin, "Erasure coding in windows azure storage," in *Presented as part of the 2012 USENIX Annual Technical Conference (USENIX ATC 12)*, Boston, MA, 2012, pp. 15–26.

[21] D. Perard, J. Lacan, Y. Bachy, and J. Detchart, "Erasure code-based low storage blockchain node," *CoRR*, vol. abs/1805.00860, 2018. [Online]. Available: http://arxiv.org/abs/1805.00860

[22] M. Dai, S. Zhang, H. Wang, and S. Jin, "A low storage room requirement framework for distributed ledger in blockchain," *IEEE Access*, vol. 6, pp. 22 970–22 975, 2018.

[23] R. K. Raman and L. R. Varshney, "Dynamic distributed storage for scaling blockchains," *CoRR*, vol. abs/1711.07617, 2017. [Online]. Available: http://arxiv.org/abs/1711.07617

[24] S. Li, M. Yu, S. Avestimehr, S. Kannan, and P. Viswanath, "Polyshard: Coded sharding achieves linearly scaling efficiency and security simultaneously," *CoRR*, vol. abs/1809.10361, 2018. [Online]. Available: http://arxiv.org/abs/1809.10361

[25] F. MacWilliams and N. Sloane, *The Theory of Error-Correcting Codes*, 2nd ed. North-holland Publishing Company, 1978.

[26] O. Etesami and A. Shokrollahi, "Raptor codes on binary memoryless symmetric channels," *IEEE Transactions on Information Theory*, vol. 52, no. 5, pp. 2033–2051, May 2006.

[27] M. G. Luby and M. Mitzenmacher, "Verification-based decoding for packet-based low-density parity-check codes," *IEEE Transactions on Information Theory*, vol. 51, no. 1, pp. 120–127, Jan 2005.

[28] R. Karp, M. Luby, and A. Shokrollahi, "Verification decoding of raptor codes," in *Proceedings. International Symposium on Information Theory, 2005. ISIT 2005.*, Sep. 2005, pp. 1310–1314.

[29] A. Juels, J. Kelley, R. Tamassia, and N. Triandopoulos, "Falcon codes: Fast, authenticated lt codes (or: Making rapid tornadoes unstoppable)," in *Proceedings of the 22Nd ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '15, 2015, pp. 1032–1047.

[30] "Ethereum fast synchronization," https://github.com/ethereum/go-ethereum/pull/1889, Oct 2015, [Online; Accessed on 04/15/2019].

[31] T. Richardson and R. Urbanke, *Modern Coding Theory*. New York, NY, USA: Cambridge University Press, 2008.

[32] W. Stadje, "The collector's problem with group drawings," *Advances in Applied Probability*, vol. 22, no. 4, pp. 866–882, 1990.

[33] "Blockchain Luxembourg S.A." https://www.blockchain.com/en/charts/avg-block-size, [Online; Accessed on 04/15/2019].

[34] "Etherscan: Ethereum block size history," https://etherscan.io/chart/blocksize, [Online; Accessed on 04/15/2019].