BEAT: Asynchronous BFT Made Practical

Sisi Duan
UMBC
sduan@umbc.edu

Michael K. Reiter
UNC Chapel Hill
reiter@cs.unc.edu

Haibin Zhang
UMBC
hbzhang@umbc.edu

ABSTRACT

We present BEAT, a set of practical Byzantine fault-tolerant (BFT) protocols for completely asynchronous environments. BEAT is flexible, versatile, and extensible, consisting of five asynchronous BFT protocols that are designed to meet different goals (e.g., different performance metrics, different application scenarios). Due to modularity in its design, features of these protocols can be mixed to achieve even more meaningful trade-offs between functionality and performance for various applications. Through a 92-instance, five-continent deployment of BEAT on Amazon EC2, we show that BEAT is efficient: roughly, all our BEAT instances significantly outperform, in terms of both latency and throughput, HoneyBadgerBFT, the most efficient asynchronous BFT known.

CCS CONCEPTS

• **Security and privacy → Systems security**; **Distributed systems security**; • **Computer systems organization → Reliability**; **Availability**;

KEYWORDS

Byzantine fault tolerance, BFT, asynchronous BFT, blockchain, robustness, threshold cryptography

ACM Reference Format:
Sisi Duan, Michael K. Reiter, and Haibin Zhang. 2018. BEAT: Asynchronous BFT Made Practical. In *2018 ACM SIGSAC Conference on Computer and Communications Security (CCS '18), October 15–19, 2018, Toronto, ON, Canada.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3243734.3243812

1 INTRODUCTION

State machine replication (SMR) [64, 81] is a fundamental software approach to enabling highly available services in practical distributed systems and cloud computing platforms (e.g., Google's Chubby [20] and Spanner [29], Apache ZooKeeper [53]). Its Byzantine failure counterpart, Byzantine fault-tolerant SMR (BFT), has recently regained its prominence, as BFT has been regarded as *the* model for building permissioned blockchains where the distributed ledgers know each other's identities but may not trust one another. As an emerging technology transforming business models, there has been a large number of industry implementations of permissioned blockchains, including Hyperledger Fabric [7, 87], Hyperledger Iroha [56], R3 Corda [30], Tendermint [88], and many more. The Hyperledger umbrella [5], for instance, has become a global collaborative open-source project under the Linux Foundation, now with more than 250 members.

Asynchronous BFT protocols [14, 21, 23, 70] are arguably the most appropriate solutions for building high-assurance and intrusion-tolerant permissioned blockchains in wide-area (WAN) environments, as these asynchronous protocols are inherently more robust against timing and denial-of-service (DoS) attacks that can be mounted over an unprotected network such as the Internet. Asynchronous BFT ensures liveness of the protocol without depending on any timing assumptions, which is prudent when the network is controlled by an adversary. In contrast, partially synchronous BFT (e.g., PBFT [27]) guarantees liveness only when the network becomes synchronous (i.e., satisfies timing assumptions). For instance, it was shown in [70] that PBFT would achieve zero throughput against an adversarial asynchronous scheduler.

Challenges and opportunities in adopting asynchronous permissioned blockchains. While a recent asynchronous BFT protocol, HoneyBadgerBFT [70], significantly improves prior asynchronous BFT protocols [14, 21, 23, 70], there are still significant pain points and challenges that prevent it from being used in practice. Meanwhile, there are also new opportunities for asynchronous BFT with the rise of blockchains.

Performance (latency, throughput) issues. Compared to partially synchronous BFT protocols (e.g., PBFT [27]), HoneyBadgerBFT has significantly higher latency and lower throughput, in part due to its use of expensive threshold cryptography (specifically, threshold encryption [10] and threshold signatures [17]). This is particularly visible in cases where each replica has limited computation power.

These limitations are further exacerbated by various engineering issues. For example, HoneyBadgerBFT was evaluated at only 80-bit security and it will be even slower if implemented with now-standard 128-bit security. Moreover, due to its use of an erasure-coding library zfec [93], HoneyBadgerBFT can only support Reed-Soloman codes (for which better alternatives exist) and at most 2^8 servers.

No one-size-fits-all BFT. In partially synchronous environments, one-size-fits-all BFT protocols have been hard to achieve (as has been argued in various works, e.g., [8, 31, 59]). Indeed, a variety of partially synchronous BFT protocols [1, 8, 16, 27, 28, 31, 33, 59] have been proposed to meet different needs. For instance, chain-based BFT protocols, such as Aliph-Chain [8], BChain [33], and Shuttle [90], favor throughput over latency. Q/U [1] achieves fault-scalability that tolerates increasing numbers of faults without significantly decreasing performance. Zyzzyva [59] and Aliph [8] are *hybrid* protocols that have high performance in failure-free cases. Moreover, a large number of *robust* BFT protocols [4, 9, 16, 28, 91] aim

CCS '18, October 15–19, 2018, Toronto, ON, Canada

ACM ISBN 978-1-4503-5693-0/18/10…$15.00
https://doi.org/10.1145/3243734.3243812

to provide a trade-off between performance and liveness during attacks that affect the timing behavior of the network.

While robustness is natively achieved in asynchronous BFT, we still require different designs and trade-offs for different performance metrics. Unlike HoneyBadgerBFT, which was designed to optimize throughput only, BEAT aims to be flexible and versatile, providing protocol instances optimized for latency, throughput, bandwidth, or scalability (in terms of the number of servers).

Append-only ledger vs. smart contracts. We advocate distinguishing two different classes of blockchain applications: append-only ledgers and on-chain smart contracts. The former corresponds to append-only, linearizable storage systems (hereinafter, *BFT storage*), and the latter corresponds to *general SMR*. While they share security requirements (agreement, total order of updates, liveness), general SMR requires each replica to maintain a copy of all service state to support contracts that operate on that state. In contrast, BFT storage may leverage erasure coding to reduce overall storage by allowing servers to keep only fragments. (See Sec. 3 for formal definitions.) Both of the applications are rather popular. Applications such as food safety [92] and exchange of healthcare data [54] are examples of append-only ledgers, while AI blockchain [86] and financial payments [55] fall into the category of requiring smart contracts. Internet of things (IoT) with blockchains may be of either type, depending on the applications: if one just uses blockchains to store and distribute IoT data to avoid the single point of failure that the clouds may have, then we just need the distributed ledger functionality; if one additionally uses blockchains to consume and analyze the data, then we will additionally need smart contracts.

BFT storage may be extended to support off-chain smart contracts run among clients (e.g., Hyperledger Fabric [7]). While off-chain smart contracts have many benefits (e.g., achieving some level of confidentiality, as argued in [7]), they also have limitations: 1) they are less suited to running complex smart contract applications with power- and computation-restricted clients (e.g., IoT devices); 2) they require communication channels among clients; and 3) they do not support efficient cross-contract state update.

Some blockchain systems use BFT for building consensus ordering services (e.g., Hyperledger Fabric). We find that BFT storage may be used to model the consensus ordering service, and a more efficient BFT storage can lead to a more efficient ordering service.

When designing BEAT, we aimed to answer the following major question: Can we have asynchronous BFT storage that significantly outperforms asynchronous general SMR?

Flexible read. Some applications benefit from flexible reading, i.e., reading just a portion of a data block as needed (instead of the whole block). For example, in a blockchain that stores video, a user may only want to read the first portion of the stored video. This can be challenging when we use erasure-coding as the underlying storage mechanism. BEAT aims to achieve flexible read with significantly reduced bandwidth.

BEAT in a nutshell. We design, implement, and evaluate BEAT — a set of practical asynchronous BFT protocols that resolve the above challenges. First, BEAT leverages more secure and efficient cryptography support and more flexible and efficient erasure-coding support. Second, BEAT is flexible, versatile, and extensible; the BEAT family includes asynchronous BFT protocols that are designed to meet different needs. BEAT's design is modular, and it can be extended to provide *many more* meaningful trade-offs among functionality and performance. Third, BEAT is efficient. Roughly, all our BEAT instances significantly outperform, in terms of both latency and throughput, HoneyBadgerBFT.

The BEAT protocols. BEAT includes five BEAT instances (BEAT0–BEAT4). BEAT0, BEAT1, are BEAT2 are general SMR that can support both off-chain and on-chain smart contracts, while BEAT3 and BEAT4 are BFT storage that can support off-chain smart contracts only. We summarize the characteristics of the BEAT protocols in Table 1 as a series of improvements to HoneyBadgerBFT.

- BEAT0, our baseline protocol, incorporates a more secure and efficient threshold encryption [85], a direct instantiation of threshold coin-flipping [22] (instead of using threshold signatures [17]), and more flexible and efficient erasure-coding support.
- BEAT1 additionally replaces an erasure-coded broadcast (AVID broadcast) [24] used in HoneyBadgerBFT with a replication-based broadcast (Bracha's broadcast [19]). This helps reduce latency when there is low contention and the batch size is small.
- BEAT2 opportunistically moves the encryption part of the threshold encryption to the client, further reducing latency. BEAT2 does so at the price of achieving a weaker liveness notion, but can be combined with anonymous communication networks to achieve full liveness. Asynchronous BFT with Tor networks has been demonstrated in HoneyBadgerBFT.
  BEAT2 additionally achieves causal order [21, 35, 79], a rather useful property for many blockchain applications that process transactions in a "first come, first served" manner, such as stock trading and financial payments.
- BEAT3 is a BFT storage system. While HoneyBadgerBFT, BEAT0, BEAT1, and BEAT2 use Byzantine reliable broadcast [19, 24, 67], we find that replacing Byzantine reliable broadcast with a different and more efficient primitive — bandwidth-efficient asynchronous verifiable information dispersal (AVID-FP) [45] (using fingerprinted cross-checksum) suffices to build a BFT storage. The bandwidth consumption in BEAT3 is information-theoretically optimal. To order transactions of size B, the communication complexity of BEAT3 is $O(B)$, while the complexity for HoneyBadger and PBFT is $O(nB)$ (where n is the total number of replicas). This improvement is significant, as it allows running BEAT in bandwidth-restricted environments, allows more aggressive batching, and significantly improves scalability.
- BEAT4 further reduces read bandwidth. BEAT4 is particularly useful when it is common that clients frequently read only a fraction of stored transactions. We provide a generic framework to enable this optimization, and BEAT4 is a specific instantiation of the framework. Roughly, BEAT4 reduces the access overhead by 50% with around 10% additional storage overhead. To achieve this, we extend fingerprinted cross-checksums [45] to handle partial read and to the case of pyramid codes [51], and we design a novel erasure-coded asynchronous verifiable information dispersal protocol with reduced read bandwidth (AVID-FP-Pyramid). Both techniques may be of independent interest.

	Discussed in	New Techniques	Features	Applicability
BEAT0	Section 6	more efficient labeled threshold encryption and coin flipping; more flexible and efficient erasure coding support	baseline protocol; outperform HoneyBadgerBFT in all metrics	general SMR
BEAT1	Section 7	replacing AVID broadcast with Bracha's broadcast	latency optimized	general SMR
BEAT2	Section 7	client-side labeled CCA threshold encryption	latency optimized additional achieve causal order	general SMR anonymous networks
BEAT3	Section 8	a novel protocol combining AVID-FP and ABA	storage, bandwidth, throughput optimized	BFT storage
BEAT4	Section 9	extending fpcc to single fragments and pyramid codes; a novel AVID-FP saving bandwidth	further saving read bandwidth	BFT storage

Table 1: Characteristics of BEAT protocols.

To our knowledge, all the erasure-coded systems against arbitrary failures in reliable distributed systems community [6, 25, 32, 41, 46] use conventional MDS (maximum distance separable) codes [69] such as Reed-Solomon codes [78] and they inherit the large bandwidth features of MDS codes. On the other hand, a large number of works aim to reduce the read bandwidth by designing new erasure coding schemes [42–44, 50–52, 58]. The systems using these codes work in synchronous environments only, and do not achieve any strong consistency goals even in the crash failure model (let alone Byzantine failures). It is our goal to blend these two disjoint communities and offer new insights to both, by designing novel Byzantine reliable broadcast and BFT protocols with reduced bandwidth.

- BEAT's design is modular, and features of these protocols can be mixed to achieve *even more* meaningful trade-offs among functionalities, performance metrics, and concrete applications.

2 RELATED WORK

The (subtle) differences between (BFT) SMR and (BFT) atomic registers. State machine replication [81] is a general technique to provide a fault-tolerant services using a number of server replicas. It can support arbitrary operations, not just read and write. In SMR, the servers need to communicate with each other and run an interactive consensus protocol to keep the servers in the same state.

Register specifications were introduced by Lamport in a series of papers [62, 65, 66], with atomic register as the strongest one. The notions of linearizability and wait-freedom for atomic registers were introduced by Herlihy and Wing [48] and Herlihy [47], respectively. Atomic registers can only support reads and writes.

Atomic registers can be realized in asynchronous distributed systems with failures. However, state machine replication cannot be achieved in asynchronous environments [38], unless it uses randomization to circumvent this impossibility result. HoneyBadgerBFT and BEAT fall into this category.

BFT SMR is suitable for a number of permissioned blockchain applications (e.g., on-chain smart contracts), while atomic registers are more suitable to model data-centric and cloud storage applications.

Comparison with erasure-coded Byzantine atomic registers. An active line of research studies erasure-coded Byzantine atomic registers, as erasure coding can be used to provide storage reduction and/or reduce bandwidth. Notable systems include Pasis [41], CT [25], M-PoWerStore [32], Loft [46], and AWE [6]. These systems have rather different properties from BEAT storage (i.e., BEAT3 and BEAT4). Loft has the same communication complexity as BEAT storage, but it only achieves obstruction-freedom, vs. BEAT's (randomized) wait-freedom. AWE, Pasis, CT, and M-PoWerStore have larger communication complexity. Additionally, while AWE achieves wait-freedom, it relies on an architecture that separates storage from metadata and therefore may rely on more servers.

Erasure-code choice in BEAT4 (Or: Why pyramid codes?). As discussed in Section 1, an ingredient in BEAT is a novel adaptation of fingerprinted cross-checksums [45] to accommodate pyramid codes. Pyramid codes and their derivatives have already been used in practice, although in a very different setting (data centers), and offer a significant performance boost [26, 51, 52]. We leverage them here to reduce bandwidth costs for fragments that contain real data. Its close competitor, Xorbas codes [80], reduces bandwidth cost for both data and redundant fragments, though we do not leverage them here. We also do not choose the (more complex) derivatives of basic pyramid codes such as generalized pyramid codes [51] and local reconstruction codes [52] that offer maximal recoverability and improve the fault tolerance of basic pyramid codes. For our designed protocol, these codes offer even greater recoverability than we need and hence would be overkill. Weaver codes [43], HoVer codes [44], and Stepped Combination codes [42] (belonging to LDPC codes) do not provide the bandwidth savings and flexibility that we need.

Another direction of research in code design is to read instead from more fragments (see [50, 58] and references therein), but less data from each. However, the bandwidth savings are only around 20%∼30%, much less than pyramid codes and its derivatives. In addition, these codes do not fit our setting where we assume a fixed number of servers may behave maliciously and we attempt to mask as many Byzantine servers as possible.

3 SYSTEM AND THREAT MODEL

Timing assumptions. Distributed systems can be roughly divided into three categories according to their timing assumption: asynchronous, synchronous, or partially synchronous. An asynchronous system makes no timing assumptions on message processing or transmission delays. If there is a known bound on message processing delays and transmission delays, then the corresponding system is synchronous. The partial synchrony model [37] lies in-between: messages are guaranteed to be delivered within a time bound, but the bound may be unknown to participants of the system.

In protocols for asynchronous systems, neither safety nor liveness can rely on timing assumptions. In contrast, a protocol built for a synchronous or partially synchronous system risks having its safety or liveness properties violated if the synchrony assumption on which it depends is violated. For this reason, protocols built for asynchronous systems are inherently more robust to timing and denial-of-service (DoS) attacks [70, 94].

BFT SMR. We consider asynchronous Byzantine fault-tolerant state machine replication (BFT SMR) protocols, where f out of n replicas can fail arbitrarily (Byzantine failures) and a computationally bounded adversary can coordinate faulty replicas.

The replicas collectively implement the abstraction of a key-value store. A replica *delivers operations*, each *submitted* by some client. All operations must be deterministic functions of the key-value store contents. The client should be able to compute a final response to its submitted operation from the responses it receives from replicas. Correctness for a secure BFT SMR protocol is specified as follows.

- **Agreement**: If any correct replica delivers an operation m, then every correct replica delivers m.
- **Total order**: If a correct replica has delivered $m_1, m_2, \cdots, m_s$ and another correct replica has delivered $m'_1, m'_2, \cdots, m'_{s'}$, then $m_i = m'_i$ for $1 \leq i \leq min(s, s')$.
- **Liveness**: If an operation m is submitted to $n-f$ correct replicas, then all correct replicas will eventually deliver m.

The liveness property has been referred to by other names, e.g., "fairness" in CKPS [21] and SINTRA [23], and "censorship resilience" in HoneyBadgerBFT [70]. We use them interchangeably.

We consider two types of BFT SMR services.

BFT storage. A BFT storage service implements only read(*key*) and write(*key*, *val*) operations. The former should return to the client the current value for *key* in the key-value store, and the latter should update the value of *key* in the key-value store to *val*.

General SMR. A general SMR service—which is our default concern, unless specified otherwise—supports operations that consist of arbitrary deterministic programs, or *transactions*, that operate on the key-value store.

To support operations that are arbitrary transactions, each replica will typically maintain the contents of the key-value store in its entirety. Then, *total order* and the determinism of transactions ensures that the key-value store contents remain synchronized at correct replicas (assuming they begin in the same state). BFT storage can be implemented in more space-efficient ways, e.g., with each replica storing only an erasure-coded fragment for the value of each key (e.g., [24, 41, 45, 46]).

Secure causal BFT protocols. One of the BEAT instances achieves causality, which we briefly recall as follows. Input causality prevents the faulty replicas from creating an operation derived from a correct client's but that is delivered (and so executed) before the operation from which it is derived. The problem of preserving input causality in BFT atomic broadcast protocols was first introduced by Reiter and Birman [79]. The notion was later refined by Cachin et al. [21] and recently generalized by Duan et al. [35].

4 BUILDING BLOCKS

This section reviews the cryptographic and distributed systems building blocks for BEAT.

Labeled threshold cryptosystems. We review robust labeled threshold cryptosystem (i.e., threshold encryption) [85] where a public key is associated with the system and a decryption key is shared among all the servers. Syntactically, a (t, n) threshold encryption scheme ThreshEnc consists of the following algorithms. A probabilistic key generation algorithm TGen takes as input a security parameter l, the number n of total servers, and threshold parameter t, and outputs (pk, vk, sk), where pk is the public key, vk is the verification key, and $sk = (sk_1, \cdots, sk_n)$ is a list of private keys. A probabilistic encryption algorithm TEnc takes as input a public key pk, a message m, and a label lb, and outputs a ciphertext c. A probabilistic decryption share generation algorithm ShareDec takes as input a private key sk_i, a ciphertext c, and a label lb, and outputs a decryption share σ. A deterministic share verification algorithm Vrf takes as input the verification key vk, a ciphertext c, a label lb, and a decryption share σ, and outputs $b \in \{0, 1\}$. A deterministic combining algorithm Comb takes as input the verification key vk, a ciphertext c, a label lb, a set of t decryption shares, and outputs a message m, or $\perp$ (a distinguished symbol).

We require the threshold encryption scheme to be chosen ciphertext attack (CCA) secure against an adversary that controls up to $t-1$ servers. We also require consistency of decryptions, i.e., no adversary that controls up to $t-1$ servers can produce a ciphertext and two t-size sets of valid decryption shares (i.e., where Vrf returns $b = 1$ for each share) such that they yield different plaintexts.

For our purpose, we require a *labeled* threshold encryption scheme [85]; threshold cryptosystems that do not support labels [10, 18] are not suitable.

Threshold PRF. We review threshold PRF (e.g., [22]), where a public key is associated with the system and a PRF key is shared among all the servers. A (t, n) threshold PRF scheme for a function F consists of the following algorithms. A probabilistic key algorithm FGen takes as input a security parameter l, the number n of total servers, and threshold parameter t, and outputs (pk, vk, sk), where pk is the public key, vk is the verification key, and $sk = (sk_1, \cdots, sk_n)$ is a list of private keys. A PRF share evaluation algorithm Eva takes a PRF input c, pk, and a private key sk_i, and outputs a PRF share y_i. A deterministic share verification algorithm Vrf takes as input the verification key vk, a PRF input c, and a PRF share y_i, and outputs $b \in \{0, 1\}$. A deterministic combining algorithm FCom takes as input the verification key vk, x, and a set of t valid PRF shares, and outputs a PRF value y.[1]

We require the threshold PRF value to be unpredictable against an adversary that controls up to $t-1$ servers. We also require the threshold PRF to be robust in the sense the combined PRF value for c is equal to $\mathsf{F}(c)$.

We can use a direct implementation of threshold PRF [22] or can use build a threshold PRF using threshold signatures [17, 83].

Erasure coding scheme. An (n, m) *erasure coding scheme* takes as input m *data fragments* and outputs n ($n \geq m$) same-size *coded fragments*. This essentially captures the encode algorithm of an

[1] Our syntactic description of threshold encryption and threshold PRF can be made more general and the algorithms are not necessarily non-interactive.

erasure code, but we (intentionally) leave the decode algorithm undefined.

An (n, m) erasure coding scheme is *systematic* if the n coded fragments contain the original m data fragments and $g = n - m$ *redundant fragments*. Let d_i ($i \in [1..m]$) denote the data fragments, and d_i ($i \in [m+1..n]$) denote the redundant fragments. We have d_i ($i \in [1..n]$) to denote all the coded fragments.

An (n, m) erasure coding scheme is *maximum distance separable* (MDS) [69] if and only if *all* the data fragments can be recovered from any m-size subset of coded fragments.

A systematic (n, m) erasure coding scheme is *linear* if each redundant fragment d_i ($i \in [m+1..n]$) is a linear combination of the data fragments, i.e., $d_i = \sum_{j=1}^{m} b_{ij} d_j$, where b_{ij}'s are coding coefficients.

Basic pyramid codes. Huang et al. [51] introduced (basic) pyramid codes to *slightly* trade space for access efficiency in erasure-coded storage systems. For practical parameters, the pyramid codes can reduce the access overhead by 50% with around 10% additional storage overhead, compared to MDS erasure codes.

Pyramid codes can be efficiently built from any (n, m) systematic and MDS code that tolerates arbitrary $g = n - m$ erasures as defined above. Specifically, we divide the m data fragments into L equal-size[2] disjoint groups S_l ($l \in [1..L]$), each of which contains m/L data fragments. Next, we keep g_1 out of g redundant fragments unchanged. These fragments are *global redundant fragments*. Then, for each group S_l, we compute $g_0 = g - g_1$ *group redundant fragments* and add them to each group, where the j-th group redundant fragment, denoted $d_{j,l}$, is a *projection* of the j-th redundant block d_{m+j} in the original MDS code onto that group, i.e., $d_{j,l}$ is computed as d_{m+j} except setting all the coding coefficients that do not correspond to S_l to 0.

This yields an $(m + g_0 L + g_1, m)$ systematic and non-MDS code, which has m data fragments and $g_0 L + g_1$ redundant fragments, where each of group S_l has g_0 group redundant fragments. It is important to note that new code is also linear.

We briefly describe two useful properties of the basic pyramid codes [51]: (1) An $(m + g_0 L + g_1, m)$ pyramid code can tolerate arbitrary $g = g_0 + g_1$ erasures; (2) Each equal-size group S_l is an $(m/L + g_0, m/L)$ MDS code. To decode a data fragment, first try from the group level. For each group S_l, if the number of the unavailable fragments is less than g_0, any unavailable fragments can be recovered. Otherwise, first recover all the unavailable fragments from the group level, and then move to global level. One may needs to compute the fragments in the original MDS code that correspond to group redundant fragments, and then uses the conventional decoding algorithm of an MDS code to recover unavailable fragments.

Figure 1 uses an example to describe pyramid codes and shows how they can reduce read bandwidth and system I/O. The example builds a (10, 6) pyramid code from a (9, 6) linear, MDS code. In the (9, 6) MDS code, there are 6 data fragments and 3 redundant fragments. The redundant fragments (d_7, d_8, and d_9) are linear combinations of the data fragments. For instance, d_7 can be written as $d_7 = \sum_{j=1}^{6} b_{7j} d_j$, where b_{7j}'s are coding coefficients.

We then divide the data fragments equally (setting $L = 2$), and compute one redundant fragment for each group: $d_{7-1} = \sum_{j=1}^{3} b_{7j} d_j$ and $d_{7-2} = \sum_{j=4}^{6} b_{7j} d_j$, respectively. Clearly, we have $d_{7-1} + d_{7-2} = d_7$. They are local (group) redundant fragments, and d_8 and d_9 are global redundant fragments. If the fragment d_1 is not available, one can just use 3 fragments (d_2, d_3, and d_{7-1}) to recover d_1. If there is more than one failure in the local group, one would need to use the traditional MDS decoding algorithm to recover the faulty local fragment. One may need to compute the sum of d_{7-1} and d_{7-2} to recover d_7 if necessary.

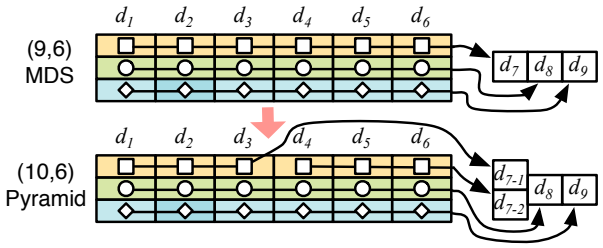


Figure 1: A basic (10,6) pyramid code from a (9,6) MDS code.

Fingerprinted cross-checksum. A fingerprinted cross-checksum [45] is data structure used by a server to verify that its fragment corresponds to a unique original data block. An (n, m) fingerprinted cross-checksum fpcc consists of an array fpcc.cc[ ] of n values and an array fpcc.fp[ ] of m values. The first array is a cross-checksum [40, 60] that contains the n hashes of the n coded fragments. The second array holds homomorphic fingerprints of data fragments that preserve the property of linear codes. Let h be a collision-resistant hash function and H be a hash function modeled as a random oracle. A homomorphic fingerprinting function fingerprint takes as input a random key and a data fragment and outputs a small field element. A fragment d is consistent with fpcc for index $i \in [1..n]$, if fpcc.cc[i] = h(d) and fingerprint(r, d) = encode(fpcc.fp[1], $\cdots$, fpcc.fp[m]), where r = H(fpcc.cc[1], $\cdots$, fpcc.cc[n]).

A central theorem in [45, 46] is that for an (n, m) systematic, MDS, and linear erasure coding scheme, no adversary $\mathcal{A}$ can produce two different sets of m fragments such that each fragment is consistent with fpcc for its index and they can be decoded into two different data blocks with non-negligible probability.

Asynchronous verifiable information dispersal using fingerprinted cross-checksum. In an asynchronous verifiable information dispersal (AVID) protocol [24], a client disperses a block M to n servers (where at most f of them might be faulty). The clients can later retrieve the full block M through the servers. The verifiability of the protocol ensures that any two clients retrieve the same block.

An (n, m)-asynchronous verifiable information dispersal scheme is a pair of protocols (disperse, retrieve) that satisfy the following with high probability:

- **Termination**: If a correct client initializes disperse(M), then all correct servers will eventually complete dispersal disperse(M).
- **Agreement**: If some correct server completes disperse(M), all correct servers eventually complete disperse(M).
- **Availability**: If $f + 1$ correct servers complete disperse(M), a correct client can run retrieve( ) to eventually reconstruct some block M'.

[2]While there is no need to require m divides L for pyramid codes, in practice one almost always uses equal-size sets. Typically, L can be set to 2.

- **Correctness**: If $f+1$ correct servers complete disperse(M), all correct clients that run retrieve( ) eventually retrieve the same block M'. If the client that initiated disperse(M) was correct, then $M' = M$.

Cachin and Tessaro [24] proposed an erasure-coded AVID, which we call AVID-CT, To broadcast a message M, the communication complexity of AVID-CT is $O(n|M|)$.

AVID-FP [45] is a bandwidth-efficient AVID using fingerprinted cross-checksum. In AVID-FP, given a block B to be dispersed, the dealer applies an (m, n) erasure coding scheme, where $m \geq f+1$ and $n = m + 2f$. Here f is the maximum number of Byzantine faulty servers that system can tolerate, and n is the total number of servers. Then it generates the corresponding fingerprinted cross-checksum for B with respect to the erasure coding scheme. Next, the client distributes the erasure-coded fragments and the same fingerprinted cross-checksum to the servers. Each server verifies the correctness of the fragment that it receives according to the fingerprinted cross-checksum and then, roughly speaking, leverages the (much smaller) fingerprinted cross-checksum in place of the fragment in the original AVID protocol. Different from AVID-CT, to disperse a message M, the communication complexity of AVID-FP is $O(|M|)$.

Byzantine reliable broadcast. Byzantine reliable broadcast (RBC), also known as the "Byzantine generals' problem," was first introduced by Lamport et al. [67]. An asynchronous reliable broadcast protocol satisfies the following properties:

- **Agreement**: If two correct servers deliver two messages M and M' then $M = M'$.
- **Totality**: If some correct server delivers a message M, all correct servers deliver M.
- **Validity**: If a correct sender broadcasts a message M, all correct servers deliver M.

Bracha's broadcast [19], one that assumes only authenticated channels, is a well-known implementation of Byzantine reliable broadcast. To broadcast a message M, its communication complexity is $O(n^2|M|)$. Cachin and Tessaro [24] proposed both an erasure-coded AVID (AVID-CT, mentioned above) and an erasure-coded variant of Bracha's broadcast — AVID broadcast, which reduces the cost to $O(n|M|)$ compared to that of Bracha's broadcast. Note that we explicitly distinguish among AVID-CT and AVID-FP (both of which are verifiable information dispersal protocols) and AVID broadcast (a RBC protocol).

5 REVIEWING HONEYBADGERBFT

This section provides an overview of HoneyBadgerBFT and related primitives. We begin by introducing asynchronous common subset (ACS).

Asynchronous common subset. HoneyBadgerBFT uses ACS [14, 21]. Formally, an ACS protocol satisfies the following properties:

- **Validity**: If a correct server delivers a set V, then $|V| \geq n - f$ and V contains the inputs of at least $n - 2f$ correct servers.
- **Agreement**: If a correct server delivers a set V, then all correct servers deliver V.
- **Totality**: If $n-f$ correct servers submit an input, then all correct servers deliver an output.

ACS can trivially lead to asynchronous BFT: each server can propose a subset of transactions, and deliver the *union* of the transactions in the agreed-upon vector; sequence numbers can be then assigned to the agreed transactions using any predefined but fixed order.

HoneyBadgerBFT in a nutshell. HoneyBadgerBFT essentially follows Ben-Or et al. [14], which uses reliable broadcast (RBC) and asynchronous binary Byzantine agreement (ABA) to achieve ACS. HoneyBadgerBFT cherry-picks a bandwidth-efficient, erasure-coded RBC (AVID broadcast) [24] and the most efficient ABA [72] to realize ACS. Specifically, HoneyBadgerBFT uses Boldyreva's threshold signature [17] to provide common coins for the randomized ABA protocol [72]. HoneyBadgerBFT favors throughput over latency by aggressively batching client transactions. It was shown that HoneyBadgerBFT can outperform PBFT when the number of servers exceeds 16 in terms of throughput in WANs, primarily because HoneyBadgerBFT distributes the network load more evenly than PBFT [27].

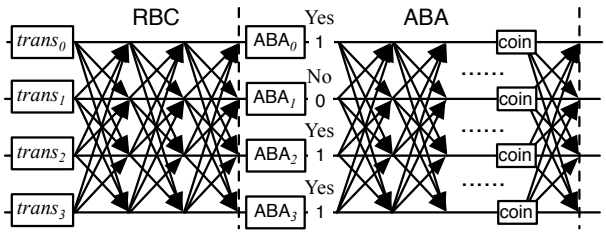


Figure 2: The HoneyBadgerBFT protocol.

As illustrated in Figure 2, the HoneyBadgerBFT protocol is composed of two subprotocols/phases: RBC and ABA. In the RBC phase, each replica first proposes a set of transactions and uses reliable broadcast to disseminate its proposal to all other replicas. In the second phase, n concurrent ABA instances are used to agree on an n-bit vector b_i for $i \in [1..n]$, where b_i indicates that if replica i's proposed transactions are included.

HoneyBadgerBFT proceeds in epochs. Let B be a batch size of client transactions. In each epoch, each replica will propose B/n transactions. Each epoch will commit $\Omega(B)$ transactions. To improve efficiency, HoneyBadgerBFT ensures that each replica proposes mostly disjoint sets of transactions. For this reason, it asks replicas to propose randomly selected transactions. To prevent adversary from censoring some particular transaction by excluding whichever replicas propose it, HoneyBadgerBFT requires replicas to use threshold encryption to encrypt transactions proposed to avoid censorship.

HoneyBadgerBFT contains four distributed algorithms: a threshold signature [17] that provides common coins for ABA, an ABA protocol [72] that has expected running time $O(1)$ (completing within $O(k)$ rounds with probability $1 - 2^{-k}$), a bandwidth-efficient reliable broadcast [24], and a threshold encryption [10] to avoid censorship and achieve liveness.

Roughly, the reliable broadcast dominates the bandwidth and guides the selection of batch size. The threshold encryption scheme

and the threshold signature scheme use expensive cryptographic operations, and they and the ABA dominate the latency of HoneyBadgerBFT.

While HoneyBadgerBFT is the most efficient asynchronous BFT protocol known, HoneyBadgerBFT favors throughput over other performance metrics (latency, bandwidth, scalability). For instance, HoneyBadgerBFT has rather high latency, which is particularly visible in local area networks (LANs) [89]. This makes it difficult to work in latency-critical applications. Indeed, it is desirable to have asynchronous BFT protocols that are designed for different goals (different performance metrics, different application scenarios).

6 BEAT0

This section describes BEAT0, our baseline protocol, that uses a set of generic techniques to improve HoneyBadgerBFT. Specifically, BEAT0 incorporates a more secure and efficient threshold encryption, a direct implementation of threshold coin flipping, and more flexible and efficient erasure-coding support.

BEAT0 specification. Instead of using CPA/CCA-secure threshold encryption that does not support labels, BEAT0 leverages a CCA-secure, labeled threshold encryption [85] to encrypt transactions while making the ciphertexts *uniquely identifiable*.

BEAT0 proceeds in epochs (i.e., rounds). Let r the current epoch number. Let n be the total number of replicas. Let ThreshEnc = (TGen, TEnc, ShareDec, Vrf, Comb) be a $(f+1, n)$ labeled threshold encryption scheme. Let pk and vk be threshold encryption public key and verification key, respectively. Let sk_i be the private key for replica $i \in [1..n]$. Let B be the batch size of BEAT0.

Each replica $i \in [1..n]$ randomly selects a set T of transactions of size B/n. It then computes a labeled threshold encryption ciphertext $(lb, c) \xleftarrow{\$} \mathsf{TEnc}_{pk}(lb, T)$ where $lb = (r, i)$. Next, each replica submits the labeled ciphertexts to ACS as input. Each replica i, upon receiving some labeled threshold ciphertexts (r, j', c) from some other replica j, does a sanity check to see if $j = j'$ and if there is already a different triple for the same r and j before proceeding. Namely, each replica i only stores and processes one ciphertext from the same j and the same r, and will discard ciphertexts subsequently received for the same j and r.

After getting output from ACS, a replica i can run ShareDec to decrypt the ciphertexts using its secret key sk_i, and broadcasts its decryption shares. When receiving $f+1$ valid shares (that pass the verification of Vrf), a replica can use Comb to combine the transactions.

Efficiently instantiating CCA secure labeled threshold encryption. We observe that much of the latency in HoneyBadgerBFT is due to usage of pairing-based cryptography, which is much slower than elliptic curve cryptography (cf. [71]). We thus implement our threshold encryption using the TDH2 scheme by Shoup and Gennaro [85] using the P-256 curve which provides standard 128-bit security. TDH2 is secure against chosen-ciphertext attacks, under the Decisional Diffie-Hellman (DDH) assumption in the random oracle model [13].

Jumping ahead, while we use a stronger and functionally more complex cryptographic scheme, our experiments show that doing so actually improves the latency of HoneyBadgerBFT greatly.

Directly instantiating common coin protocol. Instead of using a threshold signature to derive the common coins as in HoneyBadgerBFT and other multi-party computation protocols, we choose to directly use threshold coin flipping. Specifically, we use the scheme due to Cachin, Kursawe, and Shoup (CKS) [22] and implement it again using the P-256 curve that provides 128 bits of security. The threshold PRF scheme is proven secure under the Computational Diffie-Hellman (CDH) assumption in the random oracle model.

Enabling more efficient and more flexible erasure coding. HoneyBadgerBFT uses an erasure-coding library zfec [93] that supports Reed-Soloman codes only and supports at most 128 servers.

We integrate the C erasure coding library Jerasure 2.0 [73] with our BEAT framework. This allows us to remove the restriction that HoneyBadgerBFT can only support at most 128 replicas, use more efficient erasure-coding schemes (e.g., Cauchy Reed-Soloman codes [75]), and flexibly choose between erasure-coding scheme parameters to improve performance.

Distributed key generation. Our threshold encryption and threshold PRF are discrete-log based, and BEAT0 and all subsequent BEAT instances allow efficient distributed key generation [39, 57], which should be run during setup. The implementation of distributed key generation, however, is outside the scope of the present paper.

7 BEAT1 AND BEAT2 — LATENCY OPTIMIZED

This section presents two latency-optimized protocols in BEAT: BEAT1 and BEAT2.

BEAT1. Via a careful study of latency for each HoneyBadgerBFT subprotocol, we find that 1) most of latency comes from threshold encryption and threshold signatures, and 2) somewhat surprisingly, when the load is small and there is low contention, erasure-coded reliable broadcast (AVID broadcast) [24] causes significant latency. To test the actual latency overhead incurred by erasure-coded broadcast, we implement a variant of HoneyBadgerBFT, HB-Bracha, which replaces erasure-coded broadcast with a popular, replication-based reliable broadcast protocol — Bracha's broadcast [19]. We find that when the client load is small, HB-Bracha outperforms HoneyBadgerBFT in terms of latency by 20%∼60%. This motivates us to devise BEAT1.

BEAT1 replaces the AVID broadcast protocol in BEAT0 with Bracha's broadcast. It turns out that when the load is small, BEAT1 is consistently faster than BEAT0, though the difference by percentage is not as significant as that between HB-Bracha and HoneyBadgerBFT. However, when the load becomes larger, BEAT1 has significantly higher throughput, just as the case between HB-Bracha and HoneyBadgerBFT.

BEAT2. In BEAT0, our use of CCA-secure, labeled threshold encryption is at the server side, to prevent the adversary from choosing which servers' proposals to include. BEAT2 opportunistically moves the use of threshold encryption to the client side, while still using Bracha's broadcast as in BEAT1.

In BEAT2, when the ciphertexts are delivered, it is too late for the adversary to censor transactions. Thus, the adversary does not know what transactions to delay, and can only delay transactions from specific clients. BEAT2 can be combined with anonymous communication networks to achieve full liveness. BEAT2 additionally achieves causal order [21, 35, 79], which prevents the adversary

from inserting derived transactions before the original, causally prior transactions. Causal order is a rather useful property for blockchain applications that process client transactions in a "first come, first served" manner, such as trading services, financial payments, and supply chain management.

8 BEAT3 — BANDWIDTH OPTIMIZED BFT STORAGE

This section presents BEAT3, an asynchronous BFT storage system. BEAT3 significantly improves all performance metrics that we know of — latency (compared to HoneyBadgerBFT), bandwidth, storage overhead, throughput, and scalability.

Deployment scenarios. Recall that the safety and liveness properties of BFT storage remain the same as those of general SMR, with the only exception that the state may not be replicated at each server (but instead may be erasure-coded). BEAT3 can be used for blockchain applications that need append-only ledgers, and specific blockchains where the consensus protocol serves as an ordering service, such as Hyperledger Fabric [7, 87].

BEAT3. BEAT3 achieves better performance by using a novel combination of a bandwidth-efficient information dispersal scheme (AVID-FP [45]) and an ABA protocol [72]. In comparison, HoneyBadgerBFT, BEAT0, BEAT1, and BEAT2 use a combination of reliable broadcast and an ABA protocol.

AVID-FP has optimal bandwidth consumption which does *not* depend on the number of replicas. The bandwidth required to disperse a block M in AVID-FP is only $O(|M|)$, while the bandwidth in AVID broadcast (used in HoneyBadgerBFT) is $O(n|M|)$. Technically speaking, AVID-FP has a much smaller communication complexity than AVID-CT because replicas in AVID-FP agree upon a small constant-size fingerprinted cross-checksum instead of on the block itself (i.e., the bulk data).

Our basic idea is to replace AVID broadcast used in HoneyBadgerBFT with an (n, m) AVID-FP protocol, where $n = m + 2f$ and $m \geq f + 1$. Accordingly, at the end of the AVID-FP protocol, each replica now stores some fingerprinted cross-checksum and the corresponding erasure-coded fragment. There is, however, a challenge to use the approach. In AVID-FP, a correct replica cannot reconstruct its fragment if it is not provided by the AVID-FP client who proposes some transaction (here, some other replica in our protocol). Namely, as mentioned by Hendricks et al. [45], even with a successful dispersal, only $f + 1$ correct replicas, instead of all correct replicas, may have the corresponding fragments. However, ABA expects *all* correct replicas to deliver the transaction during the broadcast/dispersal stage (to correctly proceed). Note that we cannot trivially ask replicas in AVID-FP to reconstruct their individual fragment or reconstruct the whole transaction, which would nullify the bandwidth benefit of using AVID-FP.

We observe that AVID-FP actually agrees on the fingerprinted cross-checksum of the transaction. It is good enough for us to proceed to the ABA protocol once each replica delivers the fingerprinted cross-checksum. The consequence for BEAT3 is just as in AVID-FP: at least $f + 1$ correct replicas have their fragments, and some correct replicas may not have their fragments. This causes no problem, as the data is retrievable using $f + 1 = m$ correct fragments. Each replica just needs to send the client the fingerprinted cross-checksum and its fragment. The client can then reconstruct the transaction.

More formally, validity, agreement, and totality of the ACS using AVID-FP follow directly from the properties of asynchronous verifiable information dispersal, just as the case of using reliable broadcast. The only difference is that the ACS using AVID-FP now delivers a fingerprinted cross-checksum. We just need to prove that our ACS is functionally correct. This follows easily from correctness of asynchronous verifiable information dispersal: if a fingerprinted cross-checksum is delivered, then the corresponding data (i.e., transaction) is retrievable, and all clients are able to retrieve the data and the data was previously proposed by some server.

Bandwidth comparison. To order transactions of size B, the communication complexity of BEAT1, BEAT2, and HB-Bracha is $O(n^2B)$, the complexity of HoneyBadgerBFT and BEAT0 is $O(nB)$, while the communication complexity of BEAT3 is only $O(B)$. This improvement is significant, as it allows running BEAT in bandwidth-restricted environments, allows more aggressive batching, and greatly improves scalability.

9 BEAT4 — FLEXIBLE READ

This section presents a general optimization for erasure-coded BEAT instances that significantly reduce read bandwidth. For many blockchain applications, particularly data-intensive ones, it is common for clients to read only a fraction of the data block. Additionally, for many applications using smart contracts, clients may be interested in seeing the first few key terms of a large contract instead of the lengthy, detailed, and explanatory terms.

Our technique relies on a novel erasure-coded reliable broadcast protocol, AVID-FP-Pyramid, that reduces read bandwidth. AVID-FP-Pyramid uses pyramid codes [51]. As reviewed in Sec. 4, a $(m+g_0L+g_1, m)$ pyramid code can tolerate arbitrary $g = g_0 + g_1$ erasures. Let $n = m+g_0L+g_1$. We define for a $(m+g_0L+g_1, m)$ pyramid code a tailored fingerprinted cross-checksum. Our $(m+g_0L+g_1, m)$ fingerprinted cross-checksum fpcc consists of an array fpcc.cc[ ] that holds the hashes of all n coded fragments. The second array fpcc.fp[ ] *still* contains m values that are fingerprints of the first m data fragments, and because pyramid codes are linear, all the fingerprints of coded fragments can be derived by these m fingerprints, just as all the coded fragments can be derived by the original m fragments.

We say a fragment d is consistent with fpcc for index $i \in [1..n]$, if fpcc.cc[i] = h(d) and fingerprint(r, d)= encode (fpcc.fp[1], $\cdots$, fpcc.fp[m]), where r = H(fpcc.cc[1], $\cdots$, fpcc.cc[n]).

We extend the central theorem used in [45, 46] to the case of pyramid codes *and* to the case for fragments. We derive the following new lemma.

Lemma 9.1. *For an $(m+g_0L+g_1, m)$ fingerprinted cross-checksum fpcc, any probabilistic adversary $\mathcal{A}$ can produce with negligible probability a target data fragment index (resp., data fragment indexes) and two sets of fragments (that may have different sizes) such that each fragment is consistent with fpcc for its index and they can be decoded into two different data fragments for the target index (resp., different sets of fragments for the target indexes).*

The target data fragment index(es) may be an index of one of data fragment, indexes of all data fragments, or any number of

indexes in between. The two set of fragments that $\mathcal{A}$ provides can be of different sizes, and the decoding approaches for two sets may differ (may it be a group level or global level decoding).

The proof the lemma is an adaptation to the one due to Hendricks et al. [45, Theorem 3.4]. In proving Theorem 3.4 [45], the key claim is that two different sets of m fragments for the same fragment indexes and the same consistent fingerprinted cross-checksum imply that at least one fragment from the two sets is different, which is the starting point of their proof. Following the same argument, we can show that the probability that two fragments with the same index are different is bounded by $\epsilon' + q \cdot \epsilon$, where ϵ' is the advantage of attacking the hash function, q is the total number random oracle queries, and ϵ is the probability of the collisions in the fingerprinting function. The proof applies to any linear erasure-coding schemes, including pyramid codes.

AVID-FP-Pyramid. Now we describe AVID-FP-Pyramid, an asynchronous verifiable information dispersal protocol that compared to AVID-FP, further reduces read bandwidth. Instead of using a conventional MDS erasure code, AVID-FP-Pyramid uses a pyramid code. In an MDS code, m valid fragments can be used to reconstruct the original block. In a pyramid code, we need in general $m + g_0(L-1)$ valid fragments to reconstruct the block. Therefore, we have to make sure that in our new AVID protocol at least $m+g_0(L-1)+f$ servers receive consistent fragments, of which f servers might be faulty. Moreover, one needs to make sure that $m + g_0L + g_1 \geq m + g_0(L-1) + 2f$, i.e., $f \leq (g_0 + g_1)/2$, which ensures that the total number of replicas do not overflow.

Given a pyramid code (n, m) where $n = m + g_0L + g_1$ that can tolerate arbitrary $g = g_0 + g_1$ erasures, we construct AVID-FP-Pyramid where $f \leq m$ and $f \leq (g_0 + g_1)/2$. Specifically, AVID-FP-Pyramid consists of a triple of protocols (disperse, retrieve, read) which are described as follows.

Dispersal. To disperse a block B, a client applies the (n, m) pyramid code to generate n fragments $\{d_i\}_{i=1}^n$ and the fingerprinted cross-checksum fpcc. The server then sends each server i its fragment d_i and fpcc.

Upon receiving a disperse message, a server i verifies that the fragment d_i is consistent with fpcc. (Concretely, server i checks if fpcc.cc[i] = h(d) and fingerprint(r, d)= encode (fpcc.fp[1], $\cdots$, fpcc.fp[m]), where r = H(fpcc.cc[1], $\cdots$, fpcc.cc[n]).) If this is true, the server stores the fragment and sends an echo message containing fpcc (and only fpcc) to all servers.

Upon receiving $m + g_0(L-1) + f$ echo messages with matching fingerprinted cross-checksum fpcc, a server sends a ready message containing fpcc to all servers.

If receiving $f + 1$ ready with matching fingerprinted cross-checksum fpcc, and if a server does not yet send a ready message, it sends a ready message to all other servers.

Upon receiving $2f + 1$ ready messages with matching fpcc, it stores and delivers fpcc.

Retrieval. The retrieval protocol is almost the same as that in AVID-FP, with only a parameter difference. To retrieve a block, a client retrieves a fragment and fingerprinted cross-checksum from each server, waiting for matching fingerprinted cross-checksums from $f + 1$ servers and consistent fragments from $m + g_0(L-1)$ servers. These fragments are then decoded and the resulting block is returned.

Read. To read a single fragment d_i, one could choose one of the following two options. In the first option, which we term as *the optimistic mode*, a client requests from all servers the fingerprinted cross-checksum and only the target server i for the fragment. If it does not receive the fragment in time (set arbitrarily by the client), it queries the servers at the group level that contains the server i, and all servers in the local group should send their fragments. The client will repeat the procedure from the group level until it receives $m + g_0(L-1)$ fragments with matching fpcc and then recovers the fragment. In the second, which we term as *the balanced mode*, a client directly queries all servers at the group level, expecting the fragments from these group level servers.

Definition and security. While we could be more general, we provide a definition for AVID-FP-Pyramid that is specifically tailored for our purpose.

An (n, m)-asynchronous verifiable information dispersal scheme is a triple of protocols (disperse, retrieve, read) that satisfy the following with high probability:

- **Termination**: If a correct client initializes disperse(M) then all correct servers will eventually complete dispersal disperse(M).
- **Agreement**: If some correct server completes disperse(M), all correct servers eventually complete disperse(M).
- **Availability**: If $f + 1$ correct servers complete disperse(M), a correct client can run retrieve( ) to eventually reconstruct some block M'. Additionally, if $f + 1$ correct servers complete disperse(M), a correct client can run read(i) where $i \in [1..m]$ to eventually obtain a fragment d_i.
- **Correctness**: If $f + 1$ correct servers complete disperse(M), all correct clients that run retrieve( ) eventually retrieve the same block M'. If the client that initiated disperse(M) was correct, then $M' = M$. Additionally, if $f + 1$ correct servers complete disperse(M), all correct clients that run read(i) for $i \in [1..m]$ eventually obtain the same fragment d_i'. If the client that initiated disperse(M) was correct, then $d_i' = d_i$, where d_i is the i-th data fragment of M.

Theorem 9.2. *AVID-FP-Pyramid is an asynchronous verifiable information dispersal protocol as defined above.*

BEAT-FR. Replacing the AVID-FP protocol in BEAT3 with our AVID-FP-Pyramid protocol, we obtain a new BFT storage protocol — BEAT-FR which has reduced read bandwidth.

Corollary 9.3. *BEAT-FR is a BFT storage.*

Instantiating BEAT-FR: BEAT4. BEAT-FR is a generic asynchronous BFT framework that reduces read bandwidth. BEAT4 is an instantiation to BEAT-FR for concrete parameters. In BEAT4, we set $L = 2$, m is even, and $g_0 = 1$, which allows us to tolerate one failure within the local group, and reduces the read bandwidth by 50%. In BEAT4, we have $n = m + 2f + 1$, $m = f + 1$, and $n = 3m - 1$. Note that the number of echo messages which a replica has to wait before it can send ready message in BEAT4 is $m + f$.

Technique applicability. We comment that our technique presented in the section is general. While it is described for the setting of AVID-FP, it can be applied to all erasure-coded asynchronous

verifiable information dispersal and erasure-coded reliable broadcast protocols, including AVID-CT [24] and AVID broadcast [24]. Therefore, the technique can be used to improve both erasure-coded BFT storage (BEAT3) and general SMR (BEAT0).

10 IMPLEMENTATION AND EVALUATION

10.1 Implementation

We utilize the HoneyBadgerBFT prototype as the baseline to implement six asynchronous BFT protocols, including five BEAT protocols (BEAT0 to BEAT4) and HB-Bracha. HB-Bracha is implemented to understand the latency overhead caused by erasure coding. HB-Bracha replaces the underlying erasure-coded reliable broadcast (AVID broadcast) with Bracha's Broadcast [19], with the rest of the components intact.

Each of the six protocols involves 6,000 to 8,000 lines of code in Python. The underlying erasure-coding schemes (Reed-Soloman codes and pyramid codes) and fingerprinted cross-checksum, however, are implemented in C. The design and implementation of BEAT is modular, and we have implemented the following building blocks for the protocols.

Erasure coding support. HoneyBadgerBFT is 100% Python, and uses the zfec library to implement the Reed-Soloman code, an MDS erasure code. The zfec library, while popular in Python projects, suffers from both efficiency and usability issues: it supports only the traditional Reed-Soloman code implementation and supports only a word size (finite field size, a key tunable parameter in erasure coding for efficiency) of 8. Moreover, due to the usage of an erasure-coding library zfec [93], HoneyBadgerBFT supports at most 2^8 replicas.

In BEAT, we instead use Jerasure 2.0 [73], a C library for erasure-coding, to implement the underlying erasure-coding schemes (including Reed-Soloman codes and pyramid codes). Jerasure 2.0 supports a variety of other coding schemes (including Cauchy Reed-Soloman codes [75]), and allows fine-grained parameter tuning.

Fingerprinted cross-checksum. We observe that for efficiency reasons one cannot separate the implementation of fingerprinting functions from the underlying erasure-coding support. The only implementation of fingerprinting is due to Hendricks et al. [45, 46]. They implemented their own erasure coding scheme using Rabin's information dispersal scheme [77] and the corresponding fingerprinted cross-checksum using Shoup's NTL [84]. While their fingerprinted cross-checksum is efficient, the erasure coding scheme is rather slow.

In contrast, we use GF-Complete [74], the Jerasure's underlying Galois Field library using Intel SSID, to implement the fingerprinted cross-checksum primitive. Erasure coding schemes have three parameters n, m, and w, where n is the number of fragments (also the number of replicas), m is the number of data fragments (where $m = f + 1$ in our protocols), and w is the word size (the index size of the Galois Field GF(2^w)). It is required that $n + m < 2^w$ and therefore $n < 2^w$. The word size w is typically set to be between 4 and 16 for efficiency, and indeed $w = 32$ is the largest value supported by Jerasure. However, for our applications, we need to use larger $w = 64$ or 128 for the security of fingerprinted cross-checksum. We therefore extend Jerasure to include these large w's.

The specific fingerprinting function we implemented is the *evaluation fingerprinting* [82]. Currently, we apply Horner's rule to evaluate the polynomial directly, without leveraging faster lookup tables. While the implementation can be further improved, we find that the implementation can already improve all performance metrics significantly. We implement fingerprinted cross-checksum in C, with 3,500 lines of code.

Finally, we use Cython [11] to wrap the C code in Python and support functions including Reed-Solomon codes, pyramid codes, matrix generation, coding padding, and fingerprinted cross-checksum. The implementation involves around 1,000 lines of code in Python.[3]

Threshold cryptography. We use the TDH2 scheme [85] for CCA-secure labeled threshold encryption and the threshold PRF scheme [22] for distributed coin flipping. We implement both schemes using the Charm [2] Python library. We use NIST recommended P-256 curve to implement both schemes to provide standard 128-bit security.

10.2 Evaluation

Overview. We deploy and test our protocols on Amazon EC2 utilizing up to 92 nodes from ten different regions in five different continents. Each node is a general purposed *t2.medium* type with two virtual CPUs and 4GB memory. We evaluate our protocols in both LAN and WAN settings, where the LAN nodes are selected from the same Amazon EC2 region, and the WAN nodes are uniformly selected from different regions. We evaluate the protocols under different network sizes (number of replicas) and contention levels (batch sizes). For each experiment, we use f to represent the network size, where $3f + 1$ nodes are launched in total for BEAT0 to BEAT3, HB-Bracha, and HoneyBadgerBFT (abbreviated as HB in the figures), and $3f + 2$ nodes are used for BEAT4. We vary the batch size where nodes propose 1 to 20,000 transactions at a time.

Bandwidth. The protocols mentioned above have rather different communication complexity. To order transactions of size B, the communication complexity of BEAT1, BEAT2, and HB-Bracha is $O(n^2B)$, the communication complexity of HoneyBadgerBFT and BEAT0 is $O(nB)$, while the communication complexity of BEAT3 is only $O(B)$. The consequence for throughput is significant: with the same bandwidth, BEAT3 and BEAT4 can process an order of magnitude more batched transactions, leading to significantly higher throughput. Our evaluation, however, does not set the bandwidth this way, but rather assumes the bandwidth is ample and assumes all protocols use the same batch size. *The readers should be aware that BEAT3 and BEAT4 have much higher throughput if using a larger batch size.*

Latency. We first evaluate the latency in the LAN setting with $f = 1, 2, 5, 10$, and 15, respectively. We examine and compare the average latency under no contention where each node proposes a single transaction (with variable size) at a time and no concurrent requests are sent by the clients. In the LAN setting, network latency is relatively small, so the overhead is mainly caused by the protocols themselves. We report our result for $f = 1, 2$ in Figure 3.

[3] PyECLib [76] is popular python library for erasure-coding: it has a Python interface but implements C based library, Liberasurecode [61], which allows us to use existing erasure-coding library such as Jerasure[73] and Intel(R) ISA-L. We choose not to use PyECLib, primarily because the underlying Liberasurecode has implemented data structures that are not necessary for our purpose. We therefore (have to) write our own wrapper for Jerasure and fingerprinted cross-checksum using Cython [11].

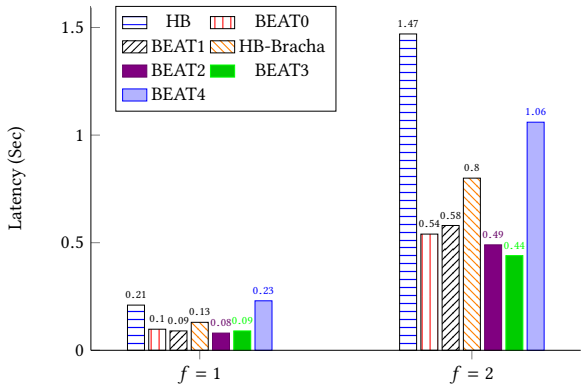


Figure 3: Latency of the protocols in the LAN setting under no contention.

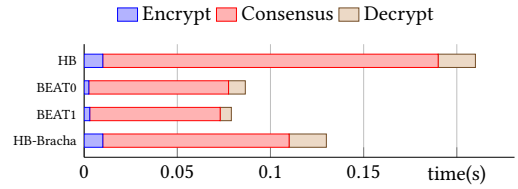


Figure 4: Latency breakdown in the LAN setting with $f = 1$.

When $f = 1$, BEAT0, BEAT1, BEAT2, and BEAT3 are around 2× faster than HoneyBadger, and when f becomes larger, they are even faster than HoneyBadger. When $f = 1$, BEAT4 is about as fast as HoneyBadger. This is primarily because BEAT4 has one more node, and the added overhead for the underlying consensus protocols and threshold cryptography is particularly visible when f is small. As f increases, HoneyBadger is much slower than BEAT4. Meanwhile, the difference between BEAT3 and BEAT4 becomes smaller; when f is 15, we barely notice the difference between them (not shown).

The differences among BEAT0, BEAT1, and BEAT2 are rather small when the batch size is 1, but becomes much more visible when the batch size becomes larger. However, the difference between BEAT1 and BEAT2 is not as large as the difference between HoneyBadger and HB-Bracha. Meanwhile, when the batch size exceeds 1,000, BEAT0 becomes faster than BEAT1 (not shown).

We further assess the latency breakdown for HoneyBadgerBFT, BEAT0, BEAT1, and HB-Bracha in order to better understand why we have these results. As illustrated in Figure 4, we evaluate the time for encrypting transactions, consensus protocols, and decrypting and combining transactions. We find the encryption and decryption for BEAT0 and BEAT1 are about three times faster than those in HoneyBadger and HB-Bracha. In addition, BEAT0 and BEAT1 use threshold PRF to produce the common coins for the consensus, and the latency of the consensus is also reduced by about 50%. HB-Bracha also achieves lower latency than HoneyBadgerBFT due to the use of latency-optimized Bracha's broadcast. This also explains why BEAT1 has lower latency than BEAT0 when the batch size is small.

Throughput. We evaluate the throughput of the protocols under different contention levels. We present the results in the LAN setting in Figure 5(a) and the the result in the WAN setting in Figure 5(b). Both cases set $f = 1$. We also show latency vs. throughput in Figure 5(c).

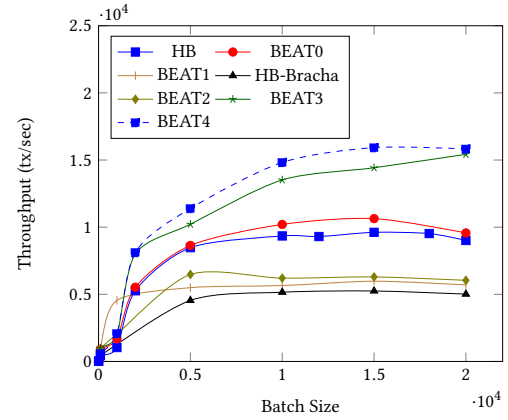
We first notice that BEAT0 slightly outperforms HoneyBadgerBFT in both settings. This is expected since BEAT0 employs optimized threshold cryptography. This also matches the result for the latency under no contention. In comparison, while BEAT1, BEAT2, and HB-Bracha are latency optimized, they do not outperform HoneyBadgerBFT in terms of throughput. We observe that in both the LAN setting and WAN setting, BEAT1, BEAT2, and HB-Bracha achieve higher throughput than HoneyBadgerBFT when the batch size is small. However, when batch size is higher than 5000, all the three protocols have 20% to 30% lower throughput than HoneyBadgerBFT. This is mainly because HB-Bracha consumes higher network bandwidth, which causes degradation when the batch size is large. This underscores the wisdom in designing HoneyBadgerBFT.

BEAT3 and BEAT4 outperform HoneyBadgerBFT consistently. They also outperform BEAT0, BEAT1, and BEAT2 consistently, though under low contention in the LAN setting, BEAT1 has larger throughput than the other protocols. These results also meet our expectation since BEAT3 and BEAT4 are bandwidth optimized. Again, we stress that we compare the performance of the protocols under the same batch size. BEAT3 and BEAT4 actually use much lower network bandwidth than the other protocols, and so for the same bandwidth budget, BEAT3 and BEAT4 (with more aggressive batching) will achieve much better throughput compared with other protocols.

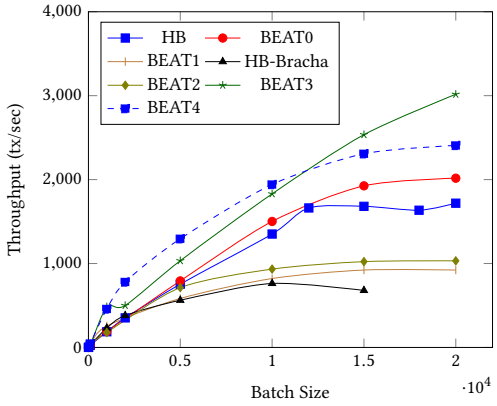
Scalability. We evaluate the scalability of BEAT0, BEAT3, and HoneyBadger by varying f from 1 to 30. We report our comparison between BEAT3 and HoneyBadger in Figure 5(d) (without BEAT0, for ease of illustration). We observe that the throughput for both protocols is in general higher when the number of replicas is smaller. Peak throughput for BEAT3 is reached in all the cases when the batch size is greater than 15,000. In the most extreme case for our experiment, where $f = 30$ and batch size is 20,000, the average latency is about 1.5 minutes. As we can see in the figure, BEAT3 outperforms HoneyBadgerBFT in all the cases. However, the difference between BEAT3 and HoneyBadgerBFT becomes smaller as the number of replicas grows. This is in part due to the fact that in large-scale networks, network latency may dominate the overhead of the protocol. BEAT0 has performance between BEAT3 and HoneyBadger, and again when f increases their difference becomes smaller.

11 LESSONS LEARNED

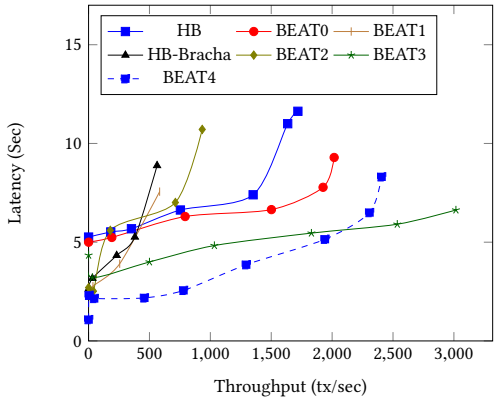
We implemented six new protocols (BEAT instances and HB-Bracha). While many of these protocols use similar components, maintaining, deploying, and comparing different BEAT instances takes tremendous effort. While one of our goals is to make BEAT modular and extensible, in practice it is still challenging to develop all the variants of the protocols. This is in part because even for the same function (e.g., threshold encryption), different APIs need to maintained. In fact, changing a small function in a BEAT instance may need to touch a large number of related functions accordingly.



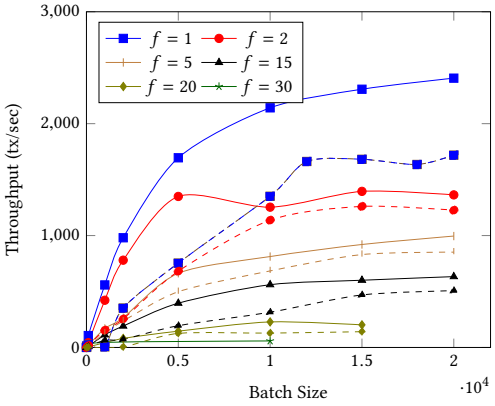
(a) Throughput for $f = 1$ where the nodes are from the same Amazon EC2 region.



(b) Throughput for $f = 1$ where the nodes are from 4 Amazon EC2 regions in different continents.



(c) Latency vs. Throughput in the WAN setting with $f = 1$.



(d) Scalability of BEAT3 and HoneyBadgerBFT. Solid lines represent the BEAT3. Dashed lines with the same mark represent the result for HoneyBadgerBFT with the same number of replicas.

Figure 5: Performance of the protocols. (The pictures are best viewed in color.)

On the other hand, we find that perhaps surprisingly, it may be easier to develop and deploy asynchronous BFT than partially synchronous BFT, for at least two reasons. First, protocols assuming partial synchrony rely on view change subprotocols, which are very difficult to implement well from our own experience and from the fact that a significant number of academic papers choose not to implement the view change protocols. Second, because of native robustness against timing and liveness attacks for asynchronous BFT, we simply do not need to take further measures to ensure robustness.

12 CONCLUSION

We describe the design and implementation of BEAT, a family of practical asynchronous BFT protocols that are efficient, flexible, versatile, and extensible. We deploy and evaluate the five BEAT protocols using 92 instances on Amazon EC2, and we show BEAT protocols are significantly more efficient than HoneyBadgerBFT, the most efficient asynchronous BFT known. We also develop new distributed system ingredients, including generalized fingerprinted cross-checksum and new asynchronous verifiable information dispersal, which might be of independent interest.

ACKNOWLEDGMENT

The authors are indebted to our shepherd Haibo Chen and the CCS reviewers for their helpful comments that greatly improve our paper.

REFERENCES

[1] M. Abd-El-Malek, G. Ganger, G. Goodson, M. K. Reiter, and J. Wylie. Fault-scalable Byzantine fault-tolerant services. *SOSP 2005*.

[2] J. A. Akinyele, et al. Charm: a framework for rapidly prototyping cryptosystems. *Journal of Cryptographic Engineering*, 3(2):111–128, 2013.

[3] Amazon Web Services (AWS). https://aws.amazon.com/

[4] Y. Amir, B. Coan, J. Kirsch, and J. Lane. Prime: Byzantine replication under attack. *IEEE TDSC*, 8(4):564–577, 2011.

[5] Hyperledger Whitepaper: An introduction to Hyperledger. https://www.hyperledger.org/wp-content/uploads/2018/08/HL_Whitepaper_IntroductiontoHyperledger.pdf

[6] E. Androulaki, C. Cachin, D. Dobre, and M. Vukolic. Erasure-coded Byzantine storage with separate metadata. *OPODIS 2014*, pp. 76–90, 2014.

[7] E. Androulaki et al. Hyperledger Fabric: a distributed operating system for permissioned blockchains. *EuroSys 2018*.

[8] P-L. Aublin, R. Guerraoui, N. Knezevic, V. Quema, and M. Vukolic. The next 700 BFT protocols. *TOCS*, vol. 32, issue 4, January 2015.

[9] P-L. Aublin, S. Mokhtar, and V. Quema. RBFT: Redundant Byzantine fault tolerance. *ICDCS 2013*.

[10] J. Baek and Y. Zheng. Simple and efficient threshold cryptosystem from the gap Diffie-Hellman group. *GLOBECOM '03*, pp. 1491–1495, 2003.

[11] S. Behnel, et al. Cython: The best of both worlds. *Computing in Science & Engineering*, 13(2:31–39, 2011.

[12] M. Bellare, R. Canetti, and H. Krawczyk. Keying hash functions for message authentication. *CRYPTO 1996*.

[13] M. Bellare and P. Rogaway. Random oracles are practical: A paradigm for designing efficient protocols. *CCS 93*, 1993.

[14] M. Ben-Or, B. Kelmer, and T. Rabin. Asynchronous secure computations with optimal resilience. *PODC 94*.

[15] A. Bessani, E. Alchieri, M. Correia, and J. Fraga. DepSpace: A Byzantine fault-tolerant coordination service. *EuroSys '08*.

[16] A. Bessani, J. Sousa, and E. Alchieri. State machine replication for the masses with BFT-SMART. *DSN '14*.

[17] A. Boldyreva. Efficient threshold signature, multisignature and blind signature schemes based on the gap-Diffie-Hellman-group signature scheme. *PKC 2003*.

[18] D. Boneh, X. Boyen, and S. Halevi. Chosen ciphertext secure public key threshold encryption without random oracles. *CT-RSA*, 2006.

[19] G. Bracha. Asynchronous Byzantine agreement protocols. *Information and Computation* 75, pp. 130–143, 1987.

[20] M. Burrows. The Chubby lock service for loosely-coupled distributed systems. *OSDI*, 2006.

[21] C. Cachin, K. Kursawe, F. Petzold, and V. Shoup. Secure and efficient asynchronous broadcast protocols (extended abstract). *CRYPTO 2001*.

[22] C. Cachin, K. Kursawe, and V. Shoup. Random oracles in Constantinople: Practical asynchronous Byzantine agreement using cryptography. *Journal of Cryptology* 18(3), 219–246.

[23] C. Cachin and J. Poritz. Secure Intrusion-tolerant replication on the Internet. *DSN 2002*, pp. 167–176, 2002.

[24] C. Cachin and S. Tessaro. Asynchronous verifiable information dispersal. *SRDS 2005*.

[25] C. Cachin and S. Tessaro. Optimal resilience for erasure-coded Byzantine distributed storage. *DSN-DCCS 2006*, pp. 115–124, 2006.

[26] B. Calder et al. Windows Azure Storage: A highly available cloud storage service with strong consistency. *ACM SOSP*, 2011.

[27] M. Castro and B. Liskov. Practical Byzantine fault tolerance and proactive recovery. *ACM Trans. Comput. Syst*, 20(4): 398–461, 2002.

[28] A. Clement, E. Wong, L. Alvisi, M. Dahlin, and M. Marchetti. Making Byzantine fault tolerant systems tolerate Byzantine faults. *NSDI* 2009.

[29] J. Corbett et al. Spanner: Google's globally-distributed database. *OSDI*, 2012.

[30] Corda. https://github.com/corda/corda

[31] J. Cowling et al. HQ replication: A hybrid quorum protocol for Byzantine fault tolerance. *OSDI* 2006.

[32] D. Dobre, G. Karame, W. Li, M. Majuntke, N. Suri, and M. Vukolic. PoWerStore: Proofs of writing for efficient and robust storage. *ACM CCS*, 2013.

[33] S. Duan, H. Meling, S. Peisert, and H. Zhang. BChain: Byzantine replication with high throughput and embedded reconfiguration. *OPODIS 2014*.

[34] Sisi Duan, Sean Peisert, and Karl Levitt. hBFT: Speculative Byzantine fault tolerance with minimum cost. *IEEE Transaction on Dependable and Secure Computing*, 12(1): 58–70, 2015.

[35] S. Duan, M. K. Reiter, and H. Zhang. Secure causal atomic broadcast, revisited. *DSN 2017*.

[36] S. Duan and H. Zhang. Practical state machine replication with confidentiality. *SRDS*, 2016.

[37] C. Dwork, N. Lynch, and L. Stockmeyer. Consensus in the presence of partial synchrony. *J. ACM* 35(2): 288–323, 1988.

[38] M. Fischer, N. Lynch, and M. Paterson. Impossibility of distributed consensus with one faulty process. *J. ACM* 32(2): 374–382, 1985.

[39] R. Gennaro, S. Jarecki, H. Krawczyk, and T. Rabin. Secure distributed key generation for discrete-log based cryptosystems. *J. Cryptology* 20(1): 51–83 (2007)

[40] L. Gong. Securely replicating authentication services. *ICDCS*, pp. 85–91, IEEE Computer Society, 1989.

[41] G. R. Goodson, J. J. Wylie, G. R. Ganger, and M. K. Reiter. Efficient Byzantine-tolerant erasure-coded storage. *DSN-DCCS 2004*, pp. 135–144, 2004.

[42] K. M. Greenan, X. Li, and J. J. Wylie. Flat XOR-based erasure codes in storage systems: Constructions, efficient recovery, and tradeoffs. *IEEE Mass Storage Systems and Technologies*, 2010.

[43] J. L. Hafner. Weaver codes: Highly fault tolerant erasure codes for storage systems. *USENIX FAST*, 2005.

[44] J. L. Hafner. HoVer erasure codes for disk arrays. *DSN*, 2006.

[45] J. Hendricks, G. R. Ganger, and M. K. Reiter. Verifying distributed erasure-coded data. *PODC 2007*, pp. 139–146, 2007.

[46] J. Hendricks, G. R. Ganger, and M. K. Reiter. Low-overhead Byzantine fault-tolerant storage. *SOSP 2007*, 2007.

[47] M. Herlihy. Wait-free synchronization. *ACM Transactions on Programming Languages and Systems*, 13(1):124–149, 1991.

[48] M. Herlihy and J. Wing. Linearizability: a correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems*, 12(3):463–492, 1990.

[49] M. Herlihy, V. Luchangco, and M. Moir. Obstruction-free synchronization: Double-ended queues as an example. *Proceedings of the 23rd International Conference on Distributed Computing Systems*, pp. 522–529, IEEE Computer Society, 2003.

[50] Y. Hu, H. Chen, P. Lee, and Y. Tang. NCCloud: Applying network coding for the storage repair in a Cloud-of-Clouds. *USENIX FAST*, 2012.

[51] C. Huang, M. Chen, and J. Li. Pyramid codes: Flexible schemes to trade space for access efficiency in reliable data storage systems. *ACM Transactions on Storage (TOS)*, Volume 9 Issue 1, March 2013. Earlier version in *NCA 2007*.

[52] C. Huang, H. Simitci, Y. Xu, A. Ogus, B. Calder, P. Gopalan, J. Li, and S. Yekhanin. Erasure coding in Windows Azure Storage. *USENIX ATC'12*, 2012.

[53] P. Hunt, M. Konar, F. P. Junqueira, and B. Reed. ZooKeeper: Wait-free coordination for Internet-scale systems. *USENIX ATC 2010*.

[54] IBM Watson Health Announces Collaboration to Study the Use of Blockchain Technology for Secure Exchange of Healthcare Data. https://www-03.ibm.com/press/us/en/pressrelease/51394.wss

[55] IBM Announces Major Blockchain Solution to Speed Global Payments. https://www-03.ibm.com/press/us/en/pressrelease/53290.wss

[56] Iroha. https://github.com/hyperledger/iroha

[57] A. Kate, Y. Huang, and I. Goldberg. Distributed key generation in the wild. IACR Cryptology ePrint Archive 2012: 377 (2012).

[58] O. Khan, R. Burns, J. Plank, W. Pierce, and C. Huang. Rethinking erasure codes for cloud file systems: Minimizing I/O for recovery and degraded reads. *USENIX FAST*, 2012.

[59] R. Kotla, L. Alvisi, M. Dahlin, A. Clement, and E. Wong. Zyzzyva: Speculative Byzantine fault tolerance. *SOSP 2007*.

[60] H. Krawczyk. Distributed fingerprints and secure information dispersal. *Proceedings of the 12th ACM Symposium on Principles of Distributed Computing*, pp. 207–218, ACM Press, 1993.

[61] Liberasurecode. https://github.com/openstack/liberasurecode

[62] L. Lamport. Concurrent reading and writing. *Communications of the ACM 11(20)*, 806–811, 1977.

[63] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Comm. ACM 21*, 7 (July), 558–565, 1978.

[64] L. Lamport. Using time instead of timeout for fault-tolerant distributed systems. *Trans. Prog. Lang. and Systems* 6(2):254–280, 1984.

[65] L. Lamport. On interprocess communication. Part I: Basic formalism. *Distrib. Comput. 1, 2*, 77–85, 1986.

[66] L. Lamport. On interprocess communication. Part II: Algorithms. *Distrib. Comput. 1, 2*, 86–101, 1986.

[67] L. Lamport, R. Shostak, and M. Pease. The Byzantine generals problem. *ACM Trans. on Programming Languages and Systems* 4(3): 382–401, 1982.

[68] Q. Lian, W. Chen, and Z. Zhang. On the impact of replica placement to the reliability of distributed brick storage systems. *ICDCS 2005*, pp. 187–196, 2005.

[69] F. J. MacWilliams and N. J. A. Sloane. *The Theory of Error Correcting Codes*. Amsterdam, North-Holland, 1977.

[70] A. Miller, Y. Xia, K. Croman, E. Shi, and D. Song. The honey badger of BFT protocols. *ACM CCS 16*, 2016.

[71] D. Moody, R. Peralta, R. Perlner, A. Regenscheid, A. Roginsky, and L. Chen. Report on pairing-based cryptography. *Journal of Research of the National Institute of Standards and Technology*, 2015.

[72] A. Mostefaoui, H. Moumen, and M. Raynal. Signature-free asynchronous Byzantine consensus with $t < n/3$ and $O(n^2)$ messages. *PODC 2014*.

[73] J. Plank and K. Greenan. Jerasure 2.0. http://jerasure.org/jerasure-2.0/

[74] J. Plank, K. Greenan, and E. Miller. Screaming fast Galois field arithmetic using Intel SIMD instructions. *FAST 2013*, 2013. Latest version: http://lab.jerasure.org/jerasure/gf-complete

[75] J. Plank and L. Xu. Optimizing Cauchy Reed-Solomon codes for fault-tolerant network storage applications. *NCA 2006.*
[76] PyECLib. https://pypi.python.org/pypi/PyECLib
[77] M. O. Rabin. Efficient dispersal of information for security, load balancing, and fault tolerance. *Journal of the ACM*, 36(2):335–348, 1989.
[78] I. S. Reed and G. Solomon. Polynomial codes over certain finite fields. *J. Soc. Industrial Appl. Math*, 1960.
[79] M. K. Reiter and K. Birman. How to securely replicate services. *ACM TOPLAS*, vol. 16 issue 3, pp. 986–1009, ACM, 1994.
[80] M. Sathiamoorthy. et al. XORing elephants: novel erasure codes for big data. *Journal Proceedings of the VLDB Endowment* volume 6, issue 5, pp. 325–336, 2013.
[81] F. Schneider. Implementing fault-tolerant services using the state machine approach: A tutorial. *ACM Comput. Surveys* 22(4): 299–319, 1990.
[82] V. Shoup. On fast and provably secure message authentication based on universal hashing. *CRYPTO '96*, pages 313–328, 1996.
[83] V. Shoup. Practical threshold signatures. *EUROCRYPT 2000.*
[84] V. Shoup. NTL: A library for doing number theory. http://shoup.net/ntl
[85] V. Shoup and R. Gennaro. Securing threshold cryptosystems against chosen ciphertext attack. *EUROCRYPT '98.*
[86] SingularityNET. https://singularitynet.io/
[87] J. Sousa, A. Bessani, and M. Vukolic. A Byzantine fault-tolerant ordering service for the Hyperledger Fabric blockchain platform. *DSN 2018.*
[88] Tendermint core. https://github.com/tendermint/tendermint
[89] H. Turki, F. Salgado, J. M. Camacho. HoneyLedgerBFT: Enabling Byzantine fault tolerance for the Hyperledger platform. Available: https://www.semanticscholar.org/
[90] R. van Renesse, C. Ho, and N. Schiper. Byzantine chain replication. *OPODIS 2012.*
[91] G. S. Veronese, M. Correia, A. N. Bessani, and L. C. Lung. Spin one's wheels? Byzantine fault tolerance with a spinning primary. *SRDS 2009.*
[92] Walmart, JD.com, IBM and Tsinghua University Launch a Blockchain Food Safety Alliance in China. https://www-03.ibm.com/press/us/en/pressrelease/53487.wss
[93] Z. Wilcox-O'Hearn. Zfec 1.5.2. https://pypi.python.org/pypi/zfec
[94] L. Zhou, F. B. Schneider, R. van Renesse. APSS: proactive secret sharing in asynchronous systems. *ACM Trans. Inf. Syst. Secur.* 8(3): 259–286 (2005)

A CORRECTNESS PROOF

Proof of Theorem 9.2. Termination is simple, as in AVID-FP. If a correct server initiates disperse, the server erasures codes the transaction, and sends fragments and the fingerprinted cross-checksum to each server. As the server initiating disperse is correct, at least $n - f \geq m + g_0(L-1) + f$ correct servers receive disperse messages, and send echo messages to all servers. Each server will eventually receive $m + g_0(L-1) + f$ echo messages, and then sends a ready message, if it has not done so. Each correct server will eventually receive at lest $2f + 1$ ready messages, and will then store the fingerprinted cross-checksum and complete.

Agreement follows *exactly* as in AVID-FP. If some correct server completes disperse(M), then the server must have received $2f + 1$ ready messages and at least $f + 1$ ready messages much have come from correct servers. This means that all correct servers will eventually receive ready messages from these correct servers. As our protocol implements the amplification step as in all other Bracha's broadcast like broadcast, all correct servers will send ready messages, and all of them will eventually receive at least $2f + 1$ ready messages. Agreement thus follows.

We first prove the first part of availability. In our protocol, if a correct server completes disperse, it must have received $2f+1$ ready messages, and at least one correct server received $m + g_0(L-1) + f$ echo messages. Therefore, at least $m + g_0(L-1)$ correct servers stored consistent fragments. According to the property of pyramid codes, these fragments can be used to reconstruct the original block. Accordingly, if $f + 1$ correct servers complete disperse, any client that initiates retrieve will receive $m + g_0(L-1)$ consistent fragments and $f + 1$ matching fingerprinted cross-checksums. The client can then decode the fragments to generate some block.

We now prove the second part of availability. Following an analogous line of the above argument, if a correct server completes disperse, at least $m + g_0(L-1)$ correct servers stored consistent fragments. If $f + 1$ correct servers complete disperse, any client that initiates read(i) for $i \in [1..m]$ will receive $f + 1$ matching fingerprinted cross-checksums. If the fragment i happens to be available or there is less than g_0 failures in the local group, the fragment will be available for the client. Otherwise, another round of interaction is needed, and the client will obtain $m + g_0(L-1)$ consistent fragments and reconstruct the fragment needed.

We now prove correctness. We first claim that if some correct server delivers fpcc_1 and some correct server delivers fpcc_2, then $\text{fpcc}_1 = \text{fpcc}_2$. The proof is quite "standard" for a quorum based protocol: if fpcc_1 is delivered then $m + g_0(L-1) + f$ servers echoed fpcc_1, of which at least $m + g_0(L-1)$ is correct. The same applied to fpcc_2. As a correct server will only echo once, there are at least $2m + 2g_0(L-1) + f$ servers echoed, which is larger than the total server (note that $L \geq 2$ and $2f \leq (g_0 + g_1)$). This leads to a contradiction. Thus, any block decoded during retrieve or any fragment during read is consistent with the same fpcc. By Theorem 3.4 in [45] and by Lemma 9.1, the probability that clients do not obtain the same block or fragment(s) is negligible. ∎