# The Honey Badger of BFT Protocols

Andrew Miller
*University of Maryland, College Park*

Yu Xia
*Tsinghua University*

Kyle Croman
*Cornell University*

Elaine Shi
*Cornell University*

Dawn Song
*University of California, Berkeley*

**Abstract**

The surprising success of cryptocurrencies has led to a surge of interest in deploying large scale, highly robust, Byzantine fault tolerant (BFT) protocols for mission-critical applications, such as financial transactions. Although the conventional wisdom is to build atop a (weakly) synchronous protocol such as PBFT (or a variation thereof), such protocols rely critically on network timing assumptions, and only guarantee liveness when the network behaves as expected. We argue these protocols are ill-suited for this deployment scenario.

We present an alternative, HoneyBadgerBFT, the first practical *asynchronous* BFT protocol, which guarantees liveness without making any timing assumptions. We base our solution on a novel atomic broadcast protocol that achieves optimal asymptotic efficiency. We present an implementation and experimental results to show our system can achieve throughput of tens of thousands of transactions per second, and scales to over a hundred nodes on a wide area network. We even conduct BFT experiments over Tor, without needing to tune any parameters. Unlike the alternatives, HoneyBadgerBFT simply does not care about the underlying network.

## 1 Introduction

Distributed fault tolerant protocols are promising solutions for mission-critical infrastructure, such as financial transaction databases. Traditionally, they have been deployed at relatively small scale, and typically in a single administrative domain where adversarial attacks might not be a primary concern. As a representative example, a deployment of Google's fault tolerant lock service, Chubby [14], consists of five nodes, any tolerates up to two crash faults.

In recent years, a new embodiment of distributed systems called "cryptocurrencies" or "blockchains" have emerged, beginning with Bitcoin's phenomenal success [40]. Such cryptocurrency systems represent a surprising and effective breakthrough [12], and open a new chapter in our understanding of distributed systems.

Cryptocurrencies systems challenge our traditional belief about the deployment environment for fault tolerance protocols. Unlike the classic "5 Chubby nodes within Google" environment, cryptocurrencies have revealed and stimulated a new demand for consensus protocols over a wide area network, among a large number of nodes that are mutually distrustful, and moreover, network connections can be much more unpredictable than the classical LAN setting, or even adversarial. This new setting poses interesting new challenges, and calls upon us to rethink the design of fault tolerant protocols.

**Robustness is a first-class citizen.** Cryptocurrencies demonstrate the demand for and viability of an unusual operating point that prioritizes robustness above all else, even at the expense of performance. In fact, Bitcoin provides terrible performance by distributed systems standards: a transaction takes on average 10 minutes to be committed, and the system as a whole achieves throughput on the order of 10 transactions per second. However, in comparison with traditional fault tolerant deployment scenarios, cryptocurrencies thrive in a highly adversarial environment, where well-motivated and malicious attacks are expected (if not commonplace). For this reason, many of Bitcoin's enthusiastic supporters refer to it as the "Honey Badger of Money" [38]. We note that the demand for robustness is often closely related to the demand for *decentralization* — since decentralization would typically require the participation of a large number of diverse participants in a wide-area network.

**Favor throughput over latency.** Most existing works on scalable fault tolerance protocols [6, 46] focus on optimizing scalability in a LAN environment controlled by a single administrative domain.

Since bandwidth provisioning is ample, these works often focus on reducing (cryptographic) computations and minimizing response time while under contention (i.e., requests competing for the same object).

In contrast, blockchains have stirred interest in a class of financial applications where response time and contention are not the most critical factors, e.g., payment and settlement networks [1]. In fact, some financial applications intentionally introduce delays in committing transactions to allow for possible rollback/chargeback operations.

Although these applications are not latency critical, banks and financial institutions have expressed interest in a *high-throughput* alternative of the blockchain technology, to be able to sustain high volumes of requests. For example, the Visa processes 2,000 tx/sec on average, with a peak of 59,000 tx/sec [1].

### 1.1 Our Contributions

**Timing assumptions considered harmful.** Most existing BFT implementations, even those called "robust," assume some variation of *weak synchrony*, where, roughly speaking, messages are guaranteed to be delivered after a certain bound $\Delta$, but $\Delta$ may be time-varying or unknown to the protocol designer. We argue that protocols based on timing assumptions are unsuitable for decentralized, cryptocurrency-like settings, where network links can be unreliable, network speeds change rapidly, and network delays may even be adversarially induced.

First, the liveness properties of weakly synchronous protocols can fail completely when the expected timing assumptions are violated (e.g., due to a malicious network adversary). To demonstrate this, we explicitly construct an adversarial "intermittently synchronous" network that violates the assumptions, such that existing weakly synchronous protocols such as PBFT [20] would grind to a halt (Section 3).

Second, even when the weak synchrony assumptions are satisfied in practice, weakly synchronous protocols degrade significantly in throughput when the underlying network is unpredictable. Ideally, we would like a protocol whose throughput closely tracks the network's performance even under rapidly changing network conditions. Unfortunately, weakly asynchronous protocols require timeout parameters that are finicky to tune, especially in cryptocurrency-like application settings; and when the chosen timeout values are either too long or too short, throughput can be hampered. As a concrete example, we show that even when the weak synchrony assumptions are satisfied, such protocols are slow to recover from transient network partitions (Section 3).

**Practical asynchronous BFT.** We propose HoneyBadgerBFT, the first BFT protocol to provide *optimal asymptotic efficiency* in the asynchronous setting, where messages are eventually delivered but no other timing assumption is made. Unlike existing weakly synchronous protocols where parameter tuning can be finicky, HoneyBadgerBFT does not care. Regardless of how network conditions fluctuate, HoneyBadgerBFT's throughput always closely tracks the network's available bandwidth. Imprecisely speaking, HoneyBadgerBFT eventually makes progress as long as messages eventually get delivered; moreover, it makes progress as soon as messages are delivered!

We formally prove the security and liveness of our HoneyBadgerBFT protocol, and show that it provides better throughput than the classical PBFT protocol [20] even in the optimistic case.

**Implementation and large-scale experiments.** We provide a full-fledged implementation of HoneyBadgerBFT, which will we release as free open source software in the near future. We demonstrate experimental results from an Amazon AWS deployment with more than 100 nodes distributed across 5 continents. To demonstrate its versatility and robustness, we also deployed HoneyBadgerBFT over the Tor anonymous relay network *without changing any parameters*, and present throughput and latency results.

**Technical highlights.** We construct an asynchronous *atomic broadcast* protocol, where clients can submit transactions, and honest nodes agree on a totally ordered log of transactions, as long as more than $\frac{2}{3}$ fraction of the nodes are honest. Our protocol guarantees fairness, i.e., a polynomially bounded adversary cannot significantly delay an honest party's request from being committed. To achieve this, we invent a novel reduction from *atomic broadcast* to the *asynchronous common subset* primitive, achieving high efficiency by splitting the workload evenly among all the nodes. We also show how to efficiently instantiate ACS by combining existing but overlooked techniques (reliable broadcast using erasure codes [18], and an ACS protocol from the multi-party computation literature). Our result is a significant improvement over prior-known asynchronous atomic broadcast protocols, decreasing the worst-case overhead from $O(N^3)$ to $O(N)$ for large batch sizes.

HoneyBadgerBFT's design is optimized for a cryptocurrency-like deployment scenario where network bandwidth is the scarce resource, and computation is relatively ample. This allows us to take advantage of cryptographic building blocks that would be considered too expensive in classical fault-tolerant database works where the primary goal is to minimize response time even under contention. For example, we show how to leverage an efficient threshold public-key encryption schemes such that an adversary cannot selectively censor specific transactions, thus attaining increased fairness without sacrificing efficiency.

### 1.2 Suggested Deployment Scenarios

Among numerous conceivable applications, we highlight two likely deployment scenarios that are sought after by banks, financial institutions, as well as advocates for fully decentralized cryptocurrencies.

**Confederation cryptocurrencies.** The success of decentralized cryptocurrencies such as Bitcoin has inspired banks and financial institutions to inspect their transaction processing and settlement infraustruture with a new light. "Confederation cryptocurrency" is an oft-cited vision [23,24,44], where a conglomerate of financial institutions jointly contribute to a Byzantine agreement protocol to allow fast and robust settlement of transactions. Passions are running high that this approach will streamline today's slow and clunky infrastructure for inter-bank settlement. As a result, several new open source projects aim to build a suitable BFT protocol for this setting, such as IBM's Open Blockchain and the Hyperledger project by the Linux foundation [37].

A confederation cryptocurrency would require a BFT protocol deployed over the wide-area network, possibly involving hundreds to thousands of consensus nodes. In this setting, enrollment can easily be controlled, . such that the set of consensus nodes are known a priori — often referred to as the "permissioned" blockchain. Clearly HoneyBadgerBFT is a natural candidate for use in such confederation cryptocurrencies.

**Applicability to permissionless blockchains.** By contrast, decentralized cryptocurrencies such as Bitcoin and Ethereum opt for a "permissionless" blockchain, where enrollment is open to anyone, and nodes may join and leave dynamically and frequently. To achieve security in this setting, known consensus protocols rely on proofs-of-work to defeat Sybil attacks, and pay an enormous price in terms of throughput and latency, e.g., Bitcoin commits transactions every $\sim$ 10 min, and its throughput limited by 7 tx/sec even when the current block size is maximized. Several recent works have suggested the promising idea of leveraging either a slower, external blockchain such as Bitcoin or economic "proof-of-stake" assumptions involving the underlying currency itself [32,34] to bootstrap faster BFT protocols, by selecting a random committee to perform BFT in every different epoch. These approaches promise to achieve the best of both worlds, security in an open enrollement, decentralized network, and the throughput and response time matching classical BFT protocols. Here too HoneyBadgerBFT is a natural choice since the randomly selected committee can be geographically heterogenous.

## 2 Background and Related Work

Our overall goal is to build a replicated state machine, where clients generate and submit transactions and a network of nodes receives and processes them. Abstracting away from application specific details (such as how to represent state and compute transitions), it suffices to build a totally globally-consistent, totally-ordered, append-only transaction log. Traditionally, such a primitive is called *total order* or *atomic broadcast* [15]; in modern cryptocurrency-inspired parlance (as we'll discuss shortly), we'd call it a *blockchain*.

Fault tolerant state machine replication protocols provide strong safety and liveness guarantees, allowing a distributed system to provide correct service in spite of network latency and the potential for some nodes to fail. Vast body of work, performance tradeoffs, tolerating different forms of failures and attacks, and making varying assumptions about the underlying network.

### 2.1 Robust BFT Protocols

While Paxos [33], Raft [42], and many other well-known protocols tolerate crash faults, Byzantine fault tolerant protocols (BFT), beginning with PBFT [20], tolerate even arbitrary (e.g., maliciously) corrupted nodes. Many subsequent protocols offer improved performance, often through *optimistic execution* that provides excellent performance when there are no faults, clients do not contend much, and the network is well-behaved, and at least some progress otherwise [2,5,30,36,48].

In general, BFT systems are evaluated in deployment scenarios where latency and CPU are the bottleneck [46], thus the most effective protocols re-

duce the number of rounds and minimize expensive cryptographic operations.

Clement et al. [22] initiated a recent line of work [4, 6, 10, 21, 22, 47] by advocating improvement of the *worst-case* performance, providing service quality guarantees even when the system is under attack — even if this comes at the expense of performance in the optimistic case. However, although the "Robust BFT" protocols in this vein gracefully tolerate compromised nodes, they still rely on timing assumptions about the underlying network. Our work takes this approach further, guaranteeing good throughput even in a fully asynchronous network.

### 2.2 Randomized Agreement

Deterministic asynchronous protocols are impossible for most tasks [27]. While the vast majority of practical BFT protocols steer clear of this impossibility result by making timing assumptions, randomness (and, in particular, cryptography) provides an alternative route. Indeed we know of asynchronous BFT protocols for a variety of tasks such as binary agreement, reliable broadcast, and more [13, 15, 16].

Our work is most closely related to SINTRA [17], a system implementation based on the asynchronous atomic broadcast protocol from Cachin et al. (CKPS01) [15]. This protocol consists of a reduction from atomic broadcast (ABC) to common subset agreement (ACS), as well as a reduction from ACS to multi-value validated agreement (MVBA). [1]

The key invention we contribute is a novel reduction from ABC to ACS that provides better efficiency (by an $O(N)$ factor) through batching, while using threshold encryption to preserve fairness (See Section 4.4). We also obtain better efficiency by cherry-picking from the literature improved instantiations of subcomponents. In particular, we sidestep the expensive MVBA primitive by using an alternative ACS [9] along with an efficient reliable broadcast (RBC) [18] as explained in Section 4.4.

Table 1 summarizes the asymptotic performance of HoneyBadgerBFT with several other atomic broadcast protocols. Here "Comm. compl." denotes the expected communication complexity (i.e., total bytes transferred) per committed transaction. Since PBFT relies on weak synchrony assumptions, it may therefore fail to make progress at all in an asynchronous network. Protocols KS02 [31] and RC05 [43] are optimistic, falling back to an expensive recovery mode based on MVBA. As mentioned the protocol of Cachin et al. (CKPS01) [15] can be improved using a more efficient ACS construction [9, 18]. We also obtain another $O(N)$ improvement through our novel reduction.

Table 1: Asymptotic communication complexity (per transaction, expected) for atomic broadcast protocols

| | Async? | Comm. compl. | |
|---|---|---|---|
| | | Optim. | Worst |
| PBFT | no | $O(N)$ | $\infty$ |
| KS02 [31] | yes | $O(N^2)$ | $O(N^3)$ |
| RC05 [43] | yes | $O(N)$ | $O(N^3)$ |
| CKPS01 [15] | yes | $O(N^3)$ | $O(N^3)$ |
| CKPS01 [15]+ [9, 18] | yes | $O(N^2)$ | $O(N^2)$ |
| **HoneyBadgerBFT (this work)** | **yes** | $\mathbf{O(N)}$ | $\mathbf{O(N)}$ |

Finally, King and Saia [28, 29] have recently developed agreement protocols with less-than-quadratic number of messages by routing communications over a sparse graph. However, extending these results to the asynchronous setting remains an open problem.

## 3 The Gap Between Asynchronous and Weakly Synchronous Network Models

Almost all modern BFT protocols rely on timing assumptions (such as *partial* or *weak synchrony*) to guarantee liveness. Purely asynchronous BFT protocols have received considerably less attention in recent years. Consider the following argument, which, if it held, would justify this narrowed focus:

[X] *Weak synchrony assumptions are unavoidable, since in any network that violates these assumptions, even asynchronous protocols would provide unacceptable performance.*

In this section, we present make two counterarguments that refute the premise above. First, we illustrate the theoretical separation between the asynchronous and weakly synchronous network models. Specifically we construct an adversarial network scheduler that violates PBFT's *weak synchrony* assumption (and indeed causes it to fail) but under which any purely asynchronous protocol (such as HoneyBadgerBFT) makes good progress. Second, we make a practical observation: even when their assumptions are met, weakly synchronous protocols are slow to recover from a network partition once it heals, whereas asynchronous protocols make progress as soon as messages are delivered.

### 3.1 Many Forms of Timing Assumptions

Before proceeding we review the various standard forms of timing assumptions. In an asyn-

chronous network, the adversary can deliver messages in any order and at any time, but nonetheless must *eventually* deliver every message sent between correct nodes. Nodes in an asynchronous network effectively have no use for "real time" clocks, and can only take actions based on the *ordering* of messages they receive.

The well-known FLP [27] result rules out the possibility of deterministic asynchronous protocols for atomic broadcast and many other tasks. A deterministic protocol must therefore make some stronger timing assumptions. A convenient (but very strong) network assumption is *synchrony*: a $\Delta$-synchronous network guarantees that every message sent is delivered after at most a delay of $\Delta$ (where $\Delta$ is a measure of real time).

Weaker timing assumptions come in several forms. In the *unknown*-$\Delta$ model, the protocol is unable to use the delay bound as a parameter. Alternatively, in the *eventually-synchronous* model, the message delay bound $\Delta$ is only guaranteed to hold after some (unknown) instant, called the "Global Stabilization Time." Collectively, these two models are referred to as *partial synchrony* [26]. Yet another variation is *weak synchrony* [26], in which the delay bound is time varying, but eventually does not grow faster than a polynomial function of time [20].

In terms of feasibility, the above are all equivalent — a protocol that succeeds in one setting can be systematically adapted for another. In terms of concrete performance, however, adjusting for *weak synchrony* means gradually increasing the timeout parameter over time (e.g., by an "exponential back-off" policy). As we show later, this results in delays when recovering from transient network partitions.

Protocols typically manifest these assumptions in the form of a timeout event. For example, if parties detect that no progress has been made within a certain interval, then they take a corrective action such as electing a new leader. Asynchronous protocols do not rely on timers, and make progress whenever messages are delivered, regardless of actual clock time.

**Counting rounds in asynchronous networks.** Although the guarantee of eventual delivery is decoupled from notions of "real time," it is nonetheless desirable to characterize the running time of asynchronous protocols. The standard approach (e.g., as explained in [19]) is for the adversary to assign each message a virtual round number, subject to the condition that every $(r-1)$-message between correct nodes must be delivered before any $(r+1)$-message is sent.

### 3.2 When Weak Synchrony Fails

We now proceed to describe why weakly synchronous BFT protocols can fail (or suffer from performance degradations) when network conditions are adversarial (or unpredictable). This motivates why such protocols are unsuited for the cryptocurrency-oriented application scenarios described in Section 1.

**A network scheduler that thwarts PBFT.** We use Practical Byzantine Fault Tolerance (PBFT) [20], the classic leader-based BFT protocol, a representative example to describe how an adversarial network scheduler can cause a class of leader-based BFT protocols [4, 6, 10, 22, 30, 47] to grind to a halt.

At any given time, the designated leader is responsible for proposing the next batch of transactions. If progress isn't made, either because the leader is faulty or because the network has stalled, then the nodes attempt to elect a new leader. The PBFT protocol critically relies on a weakly synchronous network for liveness. We construct an adversarial scheduler that violates this assumption, and indeed prevents PBFT from making any progress at all, but for which HoneyBadgerBFT (and, in fact, any asychronous protocol) performs well.

The intuition behind our scheduler is simple. First, we assume that a single node has crashed. Then, the network delays messages whenever a correct node is the leader, preventing progress and causing the next node in round-robin order to become the new leader. When the crashed node is the next up to become the leader, the scheduler immediately heals the network partition and delivers messages very rapidly among the honest nodes; however, since the leader has crashed, no progress is made here either.

This attack violates the weak synchrony assumption because it must delay messages for longer and longer each cycle, since PBFT widens its timeout interval after each failed leader election. On the other hand, it provides larger and larger periods of synchrony as well. However, since these periods of synchrony occur at inconvenient times, PBFT is unable to make use of them. Looking ahead, HoneyBadgerBFT, and indeed any asynchronous protocol, would be able to make progress during these opportunistic periods of synchrony.

To confirm our analysis, we implemented this malicious scheduler as a proxy that intercepted and delayed all view change messages to the new leader, and tested it against a 1200 line Python implemen-

tation of PBFT. The results and message logs we observed were consistent with the above analysis; our replicas became stuck in a loop requesting view changes that never succeeded. In the Appendix A we give a complete description of PBFT and explain how it behaves under this attack.

**Slow recovery from network partitions.** Even if the weak synchrony assumption is eventually satisfied, protocols that rely on it may also be slow to recover from transient network partitions. Consider the following scenario, which is simply a finite prefix of the attack described above: one node is crashed, and the network is temporarily partitioned for a duration of $2^D\Delta$. Our scheduler heals the network partition precisely when it is the crashed node's turn to become leader. Since the timeout interval at this point is now $2^{D+1}\Delta$, the protocol must wait for another $2^{D+1}\Delta$ interval before beginning to elect a new leader, despite that the network is synchronous during this interval.

**The tradeoff between robustness and responsiveness.** Such behaviors we observe above are not specific to PBFT, but rather are fundamentally inherent to protocols that rely on timeouts to cope with crashes. Regardless of the protocol variant, a practitioner must tune their timeout policy according to some tradeoff. At one extreme (eventual synchrony), the practitioner makes a specific estimate about the network delay $\Delta$. If the estimate is too low, then the system may make no progress at all; too high, and it underutilizes the available bandwidth. At the other extreme (weak synchrony), the practitioner avoids specifying any absolute delay, but nonetheless must choose a "gain" that affects how quickly the system tracks varying conditions. An asynchronous protocol avoids the need to tune such parameters.

## 4 The HoneyBadgerBFT Protocol

In this section we present HoneyBadgerBFT, the first asynchronous atomic broadcast protocol to achieve optimal asymptotic efficiency.

### 4.1 Problem Definition: Atomic Broadcast

We first define our network model and the atomic broadcast problem. Our setting involves a network of $N$ designated nodes, with distinct well-known identities ($\mathcal{P}_0$ through $\mathcal{P}_{N-1}$). The nodes receive transactions as input, and their goal is to reach common agreement on an ordering of these transactions. Our model particularly matches the deployment scenario of a "permissioned blockchain" or "distributed ledger" submitted by arbitrary clients.

The atomic broadcast primitive allows us to abstract away any application-specific details, such as how transactions are to be interpreted (to prevent replay attacks, for example, an application might define a transaction to include signatures and sequence numbers). For our purposes, transactions are simply unique strings. In practice, clients would generate transactions and send them to nodes, and consider them committed after collecting signatures from a majority of nodes. To simplify our presentation, we do not explicitly model clients, but rather assume that transactions are chosen by the adversary and provided as input to the nodes. Likewise, a transaction is considered committed once it is output by a node.

Our system model makes the following assumptions:

- *(Purely asynchronous network)* We assume each pair of nodes is connected by a reliable authenticated point-to-point channel that does not drop messages. The delivery schedule is entirely determined by the adversary, but every message sent between correct nodes must eventually be delivered. We will be interested characterizing the running time of protocols based on the number of asynchronous rounds (as described in Section 2). As the network may queue messages with arbitrary delay, we also assume nodes have unbounded buffers and are able to process all the messages they receive.
- *(Static Byzantine faults)* The adversary is given complete control of up to $f$ faulty nodes, where $f$ is a protocol parameter. Note that $3f + 1 \leq N$ (which our protocol achieves) is the lower bound for broadcast protocols in this setting.
- *(Trusted setup)* We assume that nodes may interact with a trusted dealer during an initial protocol-specific setup phase, which we will use to establish public keys and secret shares.

The assumption that nodes are designated in advance and interact with a trusted dealer is a rather strong assumption, since it effectively implies a trusted central administrator. However, as mentioned earlier in Section 1.2, several works have suggested the possibility of bootstrapping an instance of "permissioned" consensus from a "permissionless" environment without such trusted setup assumptions.

**Definition 1.** *An atomic broadcast protocol must satisfy the following properties, all of which should hold with high probability in an asynchronous network and in spite of an arbitrary adversary:*

- *(Agreement)* If any correct node outputs a transaction `tx`, then every correct node outputs `tx`.

- *(Total Order)* If one correct node has output the sequence of transactions $\{\mathsf{tx}_0, \mathsf{tx}_1, ...\mathsf{tx}_j\}$ and another has output $\{\mathsf{tx}'_0, \mathsf{tx}'_1, ...\mathsf{tx}'_j\}$, then $\mathsf{tx}_i = \mathsf{tx}'_i$ for $i \leq j$.
- *(Validity)* If a transaction $\mathsf{tx}$ is input to $N-f$ correct nodes, then it is eventually output by every correct node.

The validity property could also be called *Censorship Resilience*, since it precludes an adversary from preventing even a single transaction from being committed.

**Performance metrics.** We will primarily be interested in analyzing the *efficiency* and *fairness* of our atomic broadcast protocol.

- *(Efficiency)* Assume that the input buffers of each honest node are sufficiently full $\Omega(\mathsf{poly}(N, \lambda))$. Then *efficiency* is the expected communication cost for each node amortized over all committed transactions.

The above definition of efficiency assumes the network is *under load*, reflecting that our primary goal to sustain high throughput while fully utilizing the network's available bandwidth. Since we achieve good throughput by batching, our system uses more bandwidth per committed transaction during periods of low demand when transactions arrive infrequently. A stronger definition without this qualification would be appropriate if our goal was to minimize costs (e.g., for usage-based billing).

In practice, network links have limited capacity, and if more transactions are submitted than the network can handle, a guarantee on confirmation time cannot hold in general. Therefore we define *fairness* below relative to the number of transactions that have been input *ahead* of the transaction in question.

- *(Fairness)* Suppose an adversary passes a transaction $\mathsf{tx}$ as input to $N-f$ correct nodes. Let $T$ be the difference between the total number of transactions previously input to any correct node and the number of transactions that have been committed. Then *fairness* is the expected number of asynchronous rounds before $\mathsf{tx}$ is output by every correct node.

### 4.2 Overview and Intuition

In HoneyBadgerBFT, nodes receive transactions as input and store them in their (unbounded) buffers. The protocol proceeds in epochs, where after each epoch, a new batch of transactions is appended to the committed log. At the beginning of each epoch, nodes choose a subset of the transactions in their buffer (by a policy we'll define shortly), and provide them as input to an instance of a randomized agreement protocol. At the end of the agreement protocol, the final set of transactions for this epoch is chosen.

At this high level, our approach is similar to existing asynchronous atomic broadcast protocols, and in particular to Cachin et al. [15], the basis for a large scale transaction processing system (SINTRA). Like ours, Cachin et al.'s protocol is centered around an instance of the Asynchronous Common Subset (ACS) primitive. [2] Roughly speaking, the ACS primitive allows each node to propose a value, and guarantees that every node node outputs a common vector containing the inputs values of at least $N-2f$ correct nodes. It is trivial to build atomic broadcast from this primitive — each node simply proposes a subset of transactions from the front its queue, and outputs the union of the elements in the agreed-upon vector. However, there are two important challenges.

**Challenge 1: Achieving fairness.** The cost of ACS depends directly on size of the transaction sets proposed by each node. Since the output vector contains at least $N-f$ such sets, we can therefore improve the overall efficiency by ensuring that nodes propose *mostly disjoint* sets of transactions, thus committing more distinct transactions in one batch for the same cost. Therefore instead of simply choosing the first element(s) from its buffer (as in CKPS01 [15]), each node in our protocol proposes a randomly-chosen sample, such that each transaction is, on average, proposed by only one node.

However, implemented naïvely, this optimization would compromise fairness, since the ACS primitive allows the adversary to choose *which* nodes' proposals are ultimately included. The adversary could selectively censor a transaction excluding whichever node(s) propose it. We avoid this pitfall by using threshold encryption, which prevents the adversary from learning which transactions are proposed by which nodes, until after agreement is already reached. The full protocol will be described in Section 4.3.

**Challenge 2: Practical throughput.** Although the theoretical feasibility of asynchronous ACS and atomic broadcast have been known [9, 15, 17], their practical performance is not. To the best of our knowledge, the only other work that implemented ACS was by Cachin et al. [15], who showed that they could attain a throughput of 0.4 tx/sec over a wide area network. Therefore, an interesting question is

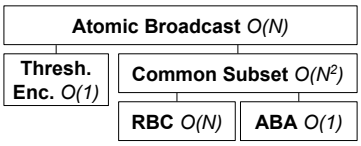


Figure 1: Construction stack for HoneyBadgerBFT

whether such protocols can attain high throughput in practice.

In this paper, we show that by stitching together a carefully chosen array of sub-components, we can efficiently instantiate ACS and attain much greater throughput both asymptotically and in practice. Notably, we improve the asymptotic cost (per node) of ACS from $O(N^2)$ (as in [15, 17]) to $O(1)$. Since the components we cherry-pick have not been presented together before (to our knowledge), we provide a self-contained description of the whole construction in Section 4.4.

**Modular Protocol Composition.** We are now ready to present our constructions formally. Before doing so, we make a remark about the style of our presentation. We define our protocols in a modular style, where each protocol may run several instances of other (sub)protocols. The outer protocol can provide input to and receive output from the subprotocol. A node may begin executing a (sub)protocol even before providing it input (e.g., if it receives messages from other nodes).

It is essential to isolate such (sub)protocol instances to ensure that messages pertaining to one instance cannot be replayed in another. This is achieved in practice by associating to each (sub)protocol instance a unique string (a session identifier), tagging any messages sent or received in this (sub)protocol with this identifier, and routing messages accordingly. We suppress these message tags in our protocol descriptions for ease of reading. We use brackets to distinguish between tagged instances of a subprotocol. For example, $\mathsf{RBC}[i]$ denotes an $i^{th}$ instance of the RBC subprotocol.

To distinguish different message types sent between parties within a protocol, we use a label in `typewriter` font (e.g., $\texttt{VAL}(m)$ indicates a message $m$ of type VAL).

Our overall construction stack is depicted in Figure 1.

### 4.3 Constructing HoneyBadgerBFT from Asynchronous Common Subset

**Building block: ACS.** Our main building block is a primitive called asynchronous common subset (ACS). The theoretical feasibility of constructing ACS has been demonstrated in several works [9, 15]. In this section, we will present the formal definition of ACS and use it as a blackbox to construct HoneyBadgerBFT. Later in Section 4.4, we will show that by combining several constructions that were somewhat overlooked in the past, we can instantiate ACS efficiently!

More formally, an ACS protocol satisfies the following properties:

- *(Validity)* If a correct node outputs a set $\mathbf{v}$, then $|\mathbf{v}| \geq N - f$ and $\mathbf{v}$ contains the inputs of at least $N - 2f$ correct nodes.
- *(Agreement)* If a correct process outputs $\mathbf{v}$, then every process outputs $\mathbf{v}$.
- *(Totality)* If $N - f$ correct processes receive an input, then all correct processes produce an output.

**Building block: threshold encryption.** A *threshold encryption* scheme $\mathsf{TPKE}$ is a cryptographic primitive that allows any party to encrypt a message to a master public key, such that the network nodes must work together to decrypt it. Once $f+1$ correct nodes compute and reveal *decryption shares* for a ciphertext, the plaintext can be recovered; until at least one correct node reveals its decryption share, the attacker learns nothing about the plaintext. A threshold scheme provides the following interface:

- $\mathsf{TPKE.Setup}(1^\lambda) \to \mathsf{PK}, \{\mathsf{SK}_i\}$ generates a public encryption key $\mathsf{PK}$, along with secret keys for each party $\mathsf{SK}_i$
- $\mathsf{TPKE.Enc}(\mathsf{PK}, m) \to C$ encrypts a message $m$
- $\mathsf{TPKE.DecShare}(\mathsf{SK}_i, C) \to \sigma_i$ produces the $\mathsf{i}^{th}$ share of the decryption (or $\perp$ if $C$ is malformed)
- $\mathsf{TPKE.Dec}(\mathsf{PK}, C\{i, \sigma_i\}) \to m$ combines a set of decryption shares to obtain the plaintext $m$ (or $\perp$, unless $C$ the set contains at least $f+1$ valid shares and $C$ is valid).

Decryption is also robust, in the sense that even for an adversarially generated ciphertext $C$, at most one plaintext (besides $\perp$) can be recovered. Note that $\mathsf{TPKE.Dec}$ effectively discards *invalid* decryption shares among the set. Finally, the scheme satisfies the obvious correctness properties, as well as a threshold version of the *IND-CCA* game. In our con-

crete instantiation, we use the threshold encryption scheme of Baek and Zheng [7].

**Atomic broadcast from ACS.** We now describe in more detail our atomic broadcast protocol, defined in Figure 2.

As mentioned, this protocol is centered around an instance of ACS. In order to obtain scalable efficiency, we choose a batching policy. We let $B$ be a batch size, and will commit $O(B)$ transactions in each epoch. Each node proposes $B/N$ transactions from its queue. To ensure that nodes propose mostly distinct transactions, we randomly select these transactions from the first $B$ in each queue.

As we will see in Section 4.4, our ACS instantiation has a total communication cost of $O(N^2|\mathbf{v}| + \lambda N^3 \log N)$, where $|\mathbf{v}|$ bounds the size of any node's input. We therefore choose a batch size $B = O(\lambda N^2 \log N)$ so that the contribution from each node $(B/N)$ absorbs this additive overhead.

In order to prevent the adversary from influencing the outcome we use a thershold encryption scheme, as described below. In a nutshell, each node chooses a set of transactions, and then encrypts it. Each node then passes the encryption as input to the ACS subroutine. The output of ACS is therefore a vector of ciphertexts. The ciphertexts are decrypted once the ACS is complete. This guarantees that the set of transactions is fully determined before the adversary learns the particular contents of the proposals made by each node. This guarantees that an adversary cannot selectively prevent a transaction from being committed once it is in the front of the queue at enough correct nodes.

**Algorithm HoneyBadgerBFT (for process $\mathcal{P}_i$)**

Let $B = O(\lambda N^2 \log N)$ be the batch size parameter.

Let $\mathsf{PK}$ be the public key received from the $\mathsf{TPKE.Setup}$ (executed by a trusted dealer), and let $\mathsf{SK}_i$ be the secret key for $\mathcal{P}_i$.

Let $\mathsf{buf} := [\,]$ be a FIFO queue of input transactions.

Proceed in consecutive epochs numbered $r$:

*// Step 1: Random selection and encryption*

- let $\mathsf{proposed}$ be a random selection of $\lfloor B/N \rfloor$ transactions from the first $B$ elements of $\mathsf{buf}$
- encrypt $x := \mathsf{TPKE.Enc}(\mathsf{PK}, \mathsf{proposed})$

*// Step 2: Agreement on ciphertexts*

- pass $x$ as input to $\mathsf{ACS}[r]$
- receive $\{v_j\}_{j \in S}$, where $S \subset [1..N]$, from $\mathsf{ACS}[r]$

*// Step 3: Decryption*

- for each $j \in S$:
  - let $e_j := \mathsf{TPKE.DecShare}(\mathsf{SK}_i, v_j)$
  - multicast $\mathtt{DEC}(r, j, i, e_j)$
  - wait to receive at least $t+1$ messages of the form $\mathtt{DEC}(r, j, k, e_{j,k})$
  - decode $y_j := \mathsf{TPKE.Dec}(\mathsf{PK}, \{(k, e_{j,k})\})$
- let $\mathsf{block}_r := \mathsf{sorted}(\cup_{j \in S}\{y_j\})$, such that $\mathsf{block}_r$ is sorted in a canonical order (e.g., lexicographically)
- set $\mathsf{buf} := \mathsf{buf} - \mathsf{block}_r$

Figure 2: **HoneyBadgerBFT.**

### 4.4 Instantiating ACS Efficiently

Cachin et al. present a protocol that (implicitly) reduces ACS to *multi-valued validated Byzantine agreement* (MVBA) [15]. Roughly speaking, MVBA allows nodes to propose values satisfying a predicate, one of which is ultimately chosen. The reduction is simple: the validation predicate says that the output must be a vector of signed inputs from at least $N - f$ parties. Unfortunately, the MVBA primitive agreement becomes a bottleneck, because the only construction we know of incurs an overhead of $O(N^3|\mathbf{v}|)$.

We avoid this bottleneck by using an alternative instantiation of ACS that sidesteps MVBA entirely. The instantiation we use is due to Ben-Or, et al. [9] and has, in our view, been somewhat overlooked. In fact, it predates CKPS01 [15], and was initially developed for a mostly unrelated purpose (as a tool for achieving efficient asynchronous multi-party computation [9]). This protocol is a reduction from ACS to *reliable broadcast* (RBC) and *asynchronous binary Byzantine agreement* (ABA). Only recently do we know of efficient constructions for these subcomponents, which we explain shortly.

At a high level, the ACS protocol proceeds in two main phases. In the first phase, each node $\mathcal{P}_i$ uses RBC to disseminate its proposed value to the other nodes, followed by ABA to decide on a bit vector that indicates which RBCs have successfully completed.

We now briefly explain the RBC and ABA constructions before explaing the Ben-Or protocol in more detail.

**Algorithm RBC (for party $\mathcal{P}_i$, with sender $\mathcal{P}_{\mathsf{Sender}}$)**

- upon input($v$) (if $\mathcal{P}_i = \mathcal{P}_{\mathsf{Sender}}$):
  let $\{s_j\}_{j\in[N]}$ be the blocks of an $(N-2f,N)$-erasure coding scheme applied to $v$
  let $h$ be a Merkle tree root computed over $\{s_j\}$
  send $\texttt{VAL}(h,b_j,s_j)$ to each party $\mathcal{P}_j$, where $b_j$ is the $j^{th}$ Merkle tree branch
- upon receiving $\texttt{VAL}(h,b_i,s_i)$ from $\mathcal{P}_{\mathsf{Sender}}$,
  multicast $\texttt{ECHO}(h,b_i,s_i)$
- upon receiving $\texttt{ECHO}(h,b_j,s_j)$ from party $\mathcal{P}_j$,
  check that $b_j$ is a valid merkle branch for root $h$ and leaf $s_j$, and otherwise discard
- upon receiving valid $\texttt{ECHO}(h,\cdot,\cdot)$ messages from $N-f$ distinct parties,
  - interpolate $\{s'_j\}$ from any $N-2f$ leaves received
  - recompute Merkle root $h'$ and if $h' \neq h$ then abort
  - if $\texttt{READY}(h)$ has not yet been sent, multicast $\texttt{READY}(h)$
- upon receiving $f+1$ matching $\texttt{READY}(h)$ messages, if READY has not yet been sent, multicast $\texttt{READY}(h)$
- upon receiving $2f+1$ matching $\texttt{READY}(h)$ messages, wait for $N-2f$ ECHO messages, then decode $v$

Figure 3: Reliable broadcast algorithm, adapted from Bracha's broadcast [13], with erasure codes to improve efficiency [18]

**Communication-Optimal Reliable Broadcast.** An asynchronous reliable broadcast channel satisfies the following properties:

- *(Agreement)* If any two correct nodes deliver $v$ and $v'$, then $v = v'$.
- *(Totality)* If any correct node delivers $v$, then all correct nodes deliver $v$
- *(Validity)* If the sender is correct and inputs $v$, then all correct nodes deliver $v$

While Bracha's [13] classic reliable broadcast protocol requires $O(N^2|v|)$ total communications for a message of size $|v|$, Cachin and Tessaro [18] observed that erasure coding can reduce this cost to merely $O(N|v| + \lambda N^2 \log N)$, even in the worst case. This is a significant improvement for large messages (i.e., when $|v| \gg \lambda N \log N$), which, (looking back to Section 4.3) guides our choice of batch size. The use of erasure coding here induces at most a small constant factor of overhead, equal to $\frac{N}{N-2f} < 3$.

If the sender is correct, the total running time is three (asynchronous) rounds; and in any case, at most two rounds elapse between when the first correct node outputs a value and the last outputs a value. The reliable broadcast algorithm from [18] is shown in Figure 3.

**Binary Agreement.** Binary agreement is a standard primitive that allows nodes to agree on the value of a single bit. More formally, binary agreement guarantees three properties:

- *(Agreement)* If any correct node outputs the bit $b$, then every correct node outputs $b$.
- *(Termination)* If all correct nodes receive input, then every correct node outputs a bit.
- *(Validity)* If any correct node outputs $b$, then at least one correct node received $b$ as input.

The validity property implies *unanimity*: if all of the correct nodes receive the same input value $b$, then $b$ must be the decided value. On the other hand, if at any point two nodes receive different inputs, then the adversary may force the decision to either value even before the remaining nodes receive input.

We instantiate this primitive with a protocol from Moustefaoui et al. [39], which is based on a cryptographic common coin. We defer explanation of this instantiation to the Appendix. Its expected running time is $O(1)$, and in fact completes within $O(k)$ rounds with probability $1-2^{-k}$. The communication complexity per node is $O(N\lambda)$, which is due primarily to threshold cryptography used in the common coin.

**Agreeing on a subset of proposed values.** Putting the above pieces together, we use a protocol from Ben-Or, et al. [9] to agree on a set of values containing the entire proposals of at least $N-f$ nodes.

At a high level, this protocol proceeds in two main phases. In the first phase, each node $\mathcal{P}_i$ uses Reliable Broadcast to disseminate its proposed value to the other nodes. In the second stage, $N$ concurrent instances of binary Byzantine agreement are used to agree on a bit vector $\{b_j\}_{j\in[1..N]}$, where $b_j = 1$ indicates that $\mathcal{P}_j$'s proposed value is included in the final set.

Actually the simple description above conceals a subtle challenge, for which Ben-Or provide a clever solution.

A naïve attempt at an implementation of the above sketch would have each node to wait for the

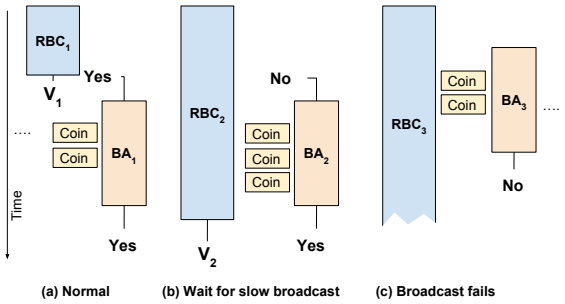


Figure 4: Each execution of our protocol involves running $N$ concurrent instances of reliable broadcast (RBC), as well as $N$ of byzantine agreement (BA), which in turn use an expected constant number of common coins. We illustrate several possible examples of how these instances play out (possibly all within the same run), from the viewpoint of Node 0. (a) In the ordinary case, Node 0 receives value $V_1$ (Node 1's proposed value) from the reliable broadcast at index 1. Node 0 therefore provides input "Yes" to $\mathsf{BA}_1$, which outputs "Yes." (b) $\mathsf{RBC}_2$ takes too long to complete, and Node 0 has already received $(N-f)$ "Yes" outputs, so it votes "No" for $\mathsf{BA}_2$. However, other nodes have seen $\mathsf{RBC}_2$ complete successfully, so $\mathsf{BA}_2$ results in "Yes" and Node 0 must wait for $V_2$. (c) $\mathsf{BA}_3$ concludes with "No" before $\mathsf{RBC}_3$ completes.

first $(N-f)$ broadcasts to complete, and then propose 1 for the binary agreement instances corresponding to those and 0 for all the others. However, correct nodes might observe the broadcasts complete in a different order. Since binary agreement only guarantees that the output is 1 if all the correct nodes unanimously propose 1, it is possible that the resulting bit vector could be empty.

To avoid this problem, nodes abstain from proposing 0 until it is certain that the final vector will have at least $N-f$ bits set. To provide some intuition for the flow of this protocol, we narrate several possible scenarios in Figure 4. The algorithm from Ben-Or et al. [9] is given in Figure 5. The running time is $O(\log N)$ in expectation, since it must wait for all binary agreement instances to finish. [3] When instantiated with the reliable broadcast and binary agreement constructions described above, the total communication complexity is $O(N^2|v| + \lambda N^3 \log N)$ assuming $|v|$ is the largest size of any node's input.

**Algorithm ACS (for party $\mathcal{P}_i$)**

Let $\{\mathsf{RBC}_i\}_N$ refer to $N$ instances of the reliable broadcast protocol, where $\mathcal{P}_i$ is the sender of $\mathsf{RBC}_i$. Let $\{\mathsf{BA}_i\}_N$ refer to $N$ instances of the binary byzantine agreement protocol.

- Upon receiving input $v_i$, input $v_i$ to $\mathsf{RBC}_i$.
- Upon delivery of $v_j$ from $\mathsf{RBC}_j$, if input has not yet been provided to $\mathsf{RBC}_j$, then provide input 1 to $\mathsf{BA}_j$
- Upon delivery of value 1 from at least $N-f$ instances of $\mathsf{BA}$, provide input 0 to each instance of $\mathsf{BA}$ that has not yet been provided input
- Once all instances of $\mathsf{BA}$ have completed, let $C \subset [1..N]$ be the indexes of each $\mathsf{BA}$ that delivered 1. Wait for $\{v_j\}$ for each $j$ such that the output of $\mathsf{BA}$ for each $j$. Finally output $\cup_{j \in C} v_j$.

Figure 5: Common Subset Agreement protocol (from Ben-Or, et al. [9])

### 4.5 Analysis

First we observe that the agreement and total order properties follow immediately from the definition of ACS and robustness of the $\mathsf{TPKE}$ scheme.

**Theorem 1.** (Agreement and total order). *The HoneyBadgerBFT protocol satisfies the agreement and total order properties, except for negligible probability.*

*Proof.* ACS guarantees that all process agree on a set of ciphertexts, and the robustness of $\mathsf{TPKE}$ guarantees that all nodes decrypt ciphertexts to consistent values. □

**Theorem 2** (Complexity)**.** *Assuming a batch size $B = O(\lambda N^2 \log N)$, the running time for each HoneyBadgerBFT epoch is $O(\log N)$ in expectation, and the total expected communication complexity is $O(B)$.*

*Proof.* The cost and running time of ACS is explained in Section 4.4. The $N$ instances of threshold decryption incur one additional round and an additional cost of $O(\lambda N^2)$, which is absorbed. □

The HoneyBadgerBFT protocol may commit up to $B$ transactions in a single epoch. However, the actual number may be less than this, since some correct nodes may propose overlapping transaction sets, others may respond too late, and corrupted nodes may propose an empty set. Fortunately, we prove (in the Appendix) that assuming each correct node's queue is full, then $B/4$ serves as a lower bound.

**Theorem 3.** (Efficiency). *Assuming each correct node's queue contains at least B distinct transactions, then the expected number of transactions committed in an epoch is at least $\frac{B}{4}$, resulting in constant efficiency.*

Finally, we prove (in the Appendix) that the adversary cannot significantly delay the commit of any transaction.

**Theorem 4.** (Fairness). *Suppose an adversary passes a transaction* tx *as input to $N-f$ correct nodes. Let $T$ be the size of the "backlog", i.e. the difference between the total number of transactions previously input to any correct node and the number of transactions that have been committed. Then* tx *is commited within $O(T/B+\lambda)$ epochs except for negligible probability.*

## 5 Implementation and Evaluation

In this section we carry out several experiments and performance measurements using a prototype implementation of the HoneyBadgerBFT protocol. Unless otherwise noted, numbers reported in this section are by default for the *optimistic* case where all nodes are behaving honestly.

First we demonstrate that HoneyBadgerBFT is indeed scalable by performing an experiments in a wide area network, including up to 128 nodes in five continents. Even under these conditions, HoneyBadgerBFT can reach peak throughputs of thousands of transactions per second. Furthermore, by a comparison with PBFT, a representative partially synchronous protocol, HoneyBadgerBFT performs only a small constant factor worse. Finally, we demonstrate the feasibility of running asynchronous BFT over the Tor anonymous communication layer.

**Implementation details.** We developed a prototype implementation of HoneyBadgerBFT in Python, using the gevent library for concurrent tasks.

For instantiating the common coin primitive, we implement Boldyreva's pairing-based threshold signature scheme [11]. For threshold encryption of transaction payloads, we use Baek and Zheng's scheme [7] to encrypt an 256-bit ephemeral key, followed by AES-256 in CBC mode over the actual payload. We implement these threshold cryptography schemes using the Charm [3] Python wrappers for PBC library [35]. For threshold signatures, we use the MNT224 curve, resulting in signatures (and signature shares) of only 65 bytes, and heuristically providing 112 bits of security [41]. Our threshold encryption scheme requires a symmetric bilinear group: we therefore use the SS512 group, which heuristically provides 80 bits of security [41]. For deterministic erasure coding, we use the zfec library [49], which implements Reed-Solomon codes.

### 5.1 Bandwidth Breakdown and Evaluation

We first analyze the bandwidth costs of our system. In all experiments, we assume a constant transaction size of $m_{\mathsf{T}} = 250$ bytes each, which would admit an ECDSA signature, two public keys, as well as a comfortably sized transaction payload (i.e., approximately the size of a typical Bitcoin transaction). Our experiments use the parameter $N = 4f$, and each party proposes a batch of $B/N$ transactions. To model the worst case scenario, nodes begin with identical queues of size $B$. We record the running time as the time from the beginning of the experiment to when the $(N-f)$-th node outputs a value.

**Bandwidth and breakdown findings.** The overall bandwidth consumed by each node consists of a fixed additive overhead as well as a transaction dependent overhead. For all parameter values we considered, the additive overhead is dominated by an $O(\lambda N^2)$ term resulting from the threshold cryptography in the ABA phases and the decryption phase that follows. The ABA phase involves each node transmitting $4N^2$ signature shares in expectation. Only the RBC phase incurs a transaction-dependent overhead, equal to the erasure coding expansion factor $r = \frac{N}{N-2f}$. The RBC phase also contributes $N^2 \log N$ hashes to the overhead because of Merkle tree branches included in the ECHO messages. The total communication cost (per node) is estimated as:

$$m_{\mathsf{all}} = r(Bm_{\mathsf{T}} + Nm_{\mathsf{E}}) + N^2((1+\log N)m_{\mathsf{H}} + m_{\mathsf{D}} + 4m_{\mathsf{S}})$$

where $m_{\mathsf{E}}$ and $m_{\mathsf{D}}$ are respectively the size of a ciphertext and decryption share in the TPKE scheme, and $m_{\mathsf{S}}$ is the size of a TSIG signature share.

The system's effective throughput increases as we increase the proposed batch size $B$, such that the transaction-dependent portion of the cost dominates. As Figure 6 shows, for $N = 128$, for batch sizes up to 1024 transactions, the transaction-independent bandwidth still dominates to overall cost. However, when when the batch size reaches 16384, the transaction-dependent portion begins to dominate — largely resulting from the RBC.ECHO stage where nodes transmit erasure-coded blocks.

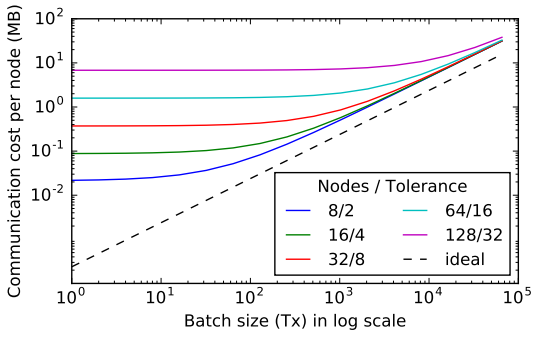


Figure 6: Estimated communication Cost (per node) over varying batch sizes. For small batch sizes, the fixed grows with $O(N^2 \log N)$. At saturation, the overhead factor approaches $\frac{N}{N-2f} < 3$.

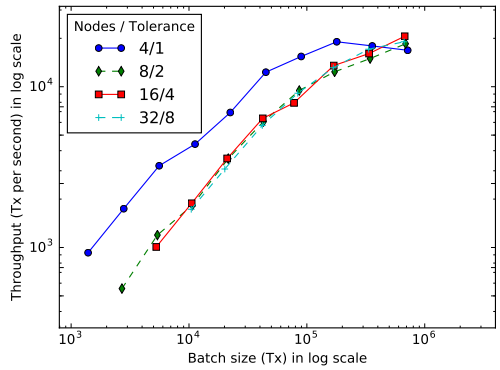


Figure 7: Throughput (transactions committed per second) vs number of transactions proposed, for different parameterizations.

### 5.2 Experiments on Amazon EC2

To see how practical our design is, we deployed our protocol on Amazon EC2 services and comprehensively tested its performance. We ran HoneyBagderBFT on 8, 16, 32, 64 and 128 Amazon EC2 *t2.medium* instances uniformly distributed throughout its 8 regions spanning 5 continents. In our experiments, we varied the batch size such that each node proposed 256, 512, 1024, 2048, 4096, 8192, 16384, 32768, 65536, or 131072 transactions.

**Throughput.** Throughput is defined as the number of transactions committed per unit of time. In our experiment, we use "confirmed transactions per second" as our measure unit if not specified otherwise.

Figure 7 shows the relationship between throughput and total number of transactions proposed by all $N$ parties for $N = 8, 16, 32, 64, 128$. The fault tolerance parameter is set to be $f = N/4$.

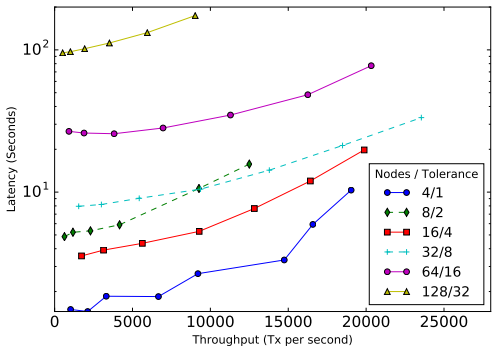


Figure 8: Latency vs. throughput for experiments over a wide area networks.

*Findings.* From Figure 7 we can see for each setting, the throughput increases as the number of proposed transactions increases. We achieve throughputs exceeding 20,000 transactions per second for medium size networks ranging from 8 to 64 nodes. For a large 128 node network, we attain more than 7,000 transactions per second. Given an infinite batch size, all network sizes would eventually converge to a common upper bound, limited only by available bandwidth. Although the total bandwidth consumed in the network increases (linearly) with each additional node, the additional nodes also contribute additional bandwidth capacity.

**Throughput vs. latency tradeoff.** Latency is defined as the time interval between the the time the first node receives a client request and when the $(N-f)$-th node finishes the consensus protocol. This is reasonable because the $(N-f)$-th node finishing the protocol implies the accomplishment of the consensus for the honest parties.

Figure 8 shows the relationship between latency and throughput for different choices of $N$ and $f = N/4$. The positive slopes indicate that we have not yet fully saturated the available bandwidth, and would attain better throughput even with larger batch sizes. Figure 8 also shows that latency increases as the number of nodes increases, largely stemming from the ABA phase of the protocol. In fact, at $N = 128$, for the range of batch sizes we tried, our system is CPU bound rather than bandwidth bound because our implementation is single threaded and must verify $O(N^2)$ threshold signatures. Regardless, our largest experiment with 128 nodes completes in under 3 minutes.

**Comparison with PBFT.** Figure 9 shows a comparison with the PBFT protocol, a classic BFT proto-

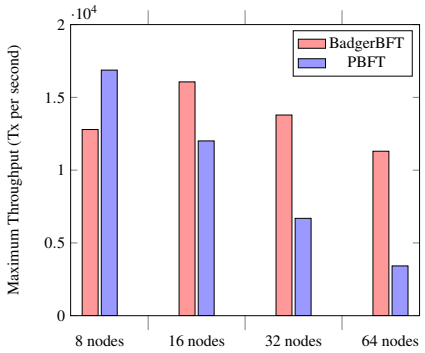


Figure 9: Comparison with PBFT on EC2s

col for partially synchronous networks. We use the Python implementation from Croman et al. [23], running on 8,16,32,64 nodes evenly distributed among Amazon AWS regions. Batch sizes were chosen to saturate the network's available bandwidth.

Fundamentally, while PBFT and our protocol have the same asymptotic communication complexity *in total*, our protocol distributes this load evenly among the network links, whereas PBFT bottlenecks on the leader's available bandwidth. Thus PBFT's attainable throughput diminishes with the number of nodes, while HoneyBadgerBFT's remains roughly constant.

Note that this experiment reflects only the optimistic case, with no faults or network interruptions. Even for small networks, HoneyBadgerBFT provides significantly better robustness under adversarial conditions as noted in Section 3. In particular, PBFT would achieve 0 **throughput** against an adversarial asynchronous scheduler, whereas HoneyBadgerBFT would complete epochs at a regular rate.

### 5.3 Experiments over Tor

Tor is a popular anonymous relay network [25]. Running consensus nodes over Tor gives better robustness against targeted attacks, since it allows nodes to conceal their IP addresses and to avoid censorship by oppressive governments. We therefore conducted experiments over Tor using hidden services. As far as we know, this is the first instance of a fault tolerant protocol experiment carried out over an anonymous network!

**Brief background on Tor.** The Tor network consists of approximately $6,500$ relays, which are listed in a public directory service. Tor enables "hidden services", which are servers that accept connections via Tor in order to conceal its location. When a client establishes a connection to a hidden service, both the client and the server construct 3-hop circuits to a common "rendezvous point." Thus each connection to a hidden service routes data through 5 randomly chosen relays, which . Tor provides a means for relay nodes to advertise their capacity and utilization, and these self-reported metrics are aggregated by the Tor project. According to these metrics, [4] the total capacity of the network is $\sim$145Gbps, and the current utilization is $\sim$65Gbps.

**Tor experiment setup.** We design our experiment setup such that we could run all $N$ HoneyBadgerBFT nodes on a single desktop machine running the Tor daemon software, while being able to realistically reflect Tor relay paths. To do this, we run all $N$ HoneyBadgerBFT nodes on a single desktop machine running a single instance of the Tor daemon software, configured to listen on $N$ hidden services (one service per node). Since each HoneyBadgerBFT node forms a connection to each other node, we manually construct a total of $N^2$ Tor circuits per experiment, beginning and ending with our machine, and passing through 5 random relays. In summary, all pairwise overlay links traverse real Tor circuits consisting of *random* relay nodes, and therefore the performance we obtain is representative of typical consensus performance over Tor (despite all nodes running on a single machine).

Since Tor provides a critical public service for many users, it is important to ensure that research experiments conducted on the live network do not adversely impact it. We formed connections from only a single vantage point (and thus avoid receiving), and ran experiments of short duration (several minutes) and with small parameters (only 256 circuits formed in our largest experiment). In total, our experiments involved the transfer of approximately five gigabytes of data through Tor – less than a 1E-5 fraction of its daily utilization.

Figure 10 shows how latency changes with throughput. In contrast to our EC2 experiment where nodes have ample bandwidth, Tor circuits are limited by the slowest link in the circuit. We attain a maximum throughput of over 800 transactions per second of Tor.

In general, messages transmitted over Tor's relay network tends to have significant and highly variable latency. For instance, during our experiment on 8 parties proposing 16384 transactions per party, a single message can be delayed for 316.18 seconds and the delay variance is over 2208 while the average

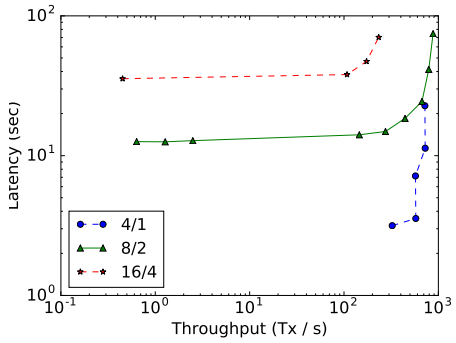


Figure 10: Latency vs throughput for experiments running HoneyBadgerBFT over Tor.

delay is only 12 seconds. We stress that our protocol did not need to be "tuned" for such network conditions, as would a traditional eventually synchronous protocol.

## 6 Conclusion

We have presented HoneyBadgerBFT, the first efficient and high-throughput asynchronous BFT protocol. Through our implementation and experimental we demonstrate that HoneyBadgerBFT can be a suitable component for incipient cryptocurrency-inspired deployments of fault tolerant transaction processing systems. More generally, we believe our work demonstrates the promise of building dependable and transaction processing systems based on asynchronous protocol.

**Acknowledgements.** We thank Jay Lorch and Jonathan Katz for their helpful suggestions, and especially Dominic Williams for several excellent discussions that inspired us to tackle this problem. This work is supported in part by NSF grants CNS-1314857, CNS-1453634, CNS-1518765, CNS-1514261, CNS-1518899, a Packard Fellowship, a Sloan Fellowship, two Google Faculty Research Awards, and a VMWare Research Award. This work was done in part while a subset of the authors were visiting the Simons Institute for the Theory of Computing, supported by the Simons Foundation and by the DIMACS/Simons Collaboration in Cryptography through NSF grant CNS-1523467.

## References

[1] How a Visa transaction works. `http://web.archive.org/web/20160121231718/http://apps.usa.visa.com/merchants/become-a-merchant/how-a-visa-transaction-works.jsp`, 2015.

[2] Michael Abd-El-Malek, Gregory R Ganger, Garth R Goodson, Michael K Reiter, and Jay J Wylie. Fault-scalable byzantine fault-tolerant services. *ACM SIGOPS Operating Systems Review*, 39(5):59–74, 2005.

[3] Joseph A Akinyele, Christina Garman, Ian Miers, Matthew W Pagano, Michael Rushanan, Matthew Green, and Aviel D Rubin. Charm: a framework for rapidly prototyping cryptosystems. *Journal of Cryptographic Engineering*, 3(2):111–128, 2013.

[4] Yair Amir, Brian Coan, Jonathan Kirsch, and John Lane. Prime: Byzantine replication under attack. *Dependable and Secure Computing, IEEE Transactions on*, 8(4):564–577, 2011.

[5] Yair Amir, Claudiu Danilov, Danny Dolev, Jonathan Kirsch, John Lane, Cristina Nita-Rotaru, Josh Olsen, and David Zage. Steward: Scaling byzantine fault-tolerant replication to wide area networks. *Dependable and Secure Computing, IEEE Transactions on*, 7(1):80–93, 2010.

[6] Pierre-Louis Aublin, Sonia Ben Mokhtar, and Vivien Quéma. Rbft: Redundant byzantine fault tolerance. In *Distributed Computing Systems (ICDCS), 2013 IEEE 33rd International Conference on*, pages 297–306. IEEE, 2013.

[7] Joonsang Baek and Yuliang Zheng. Simple and efficient threshold cryptosystem from the gap diffie-hellman group. In *Global Telecommunications Conference, 2003. GLOBECOM'03. IEEE*, volume 3, pages 1491–1495. IEEE, 2003.

[8] Michael Ben-Or and Ran El-Yaniv. Resilient-optimal interactive consistency in constant time. *Distributed Computing*, 16(4):249–262, 2003.

[9] Michael Ben-Or, Boaz Kelmer, and Tal Rabin. Asynchronous secure computations with optimal resilience. In *Proceedings of the thirteenth annual ACM symposium on Principles of distributed computing*, pages 183–192. ACM, 1994.

[10] Alysson Bessani, João Sousa, and Eduardo EP Alchieri. State machine replication for the masses with bft-smart. In *Dependable Systems and Networks (DSN), 2014 44th Annual IEEE/IFIP International Conference on*, pages 355–362. IEEE, 2014.

[11] Alexandra Boldyreva. Threshold signatures, multisignatures and blind signatures based on

the gap-diffie-hellman-group signature scheme. In *Public key cryptographyPKC 2003*, pages 31–46. Springer, 2002.

[12] Joseph Bonneau, Andrew Miller, Jeremy Clark, Arvind Narayanan, Joshua Kroll, and Edward W. Felten. Research perspectives on bitcoin and second-generation digital currencies. In *2015 IEEE Symposium on Security and Privacy*. IEEE, 2015.

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

[14] Mike Burrows. The chubby lock service for loosely-coupled distributed systems. In *Proceedings of the 7th symposium on Operating systems design and implementation*, pages 335–350. USENIX Association, 2006.

[15] Christian Cachin, Klaus Kursawe, Frank Petzold, and Victor Shoup. Secure and efficient asynchronous broadcast protocols. In *Advances in Cryptology – Crypto 2001*, pages 524–541. Springer, 2001.

[16] Christian Cachin, Klaus Kursawe, and Victor Shoup. Random oracles in constantipole: Practical asynchronous byzantine agreement using cryptography. In *Proceedings of the Nineteenth Annual ACM Symposium on Principles of Distributed Computing*, pages 123–132. ACM, 2000.

[17] Christian Cachin, Jonathan Poritz, et al. Secure intrusion-tolerant replication on the internet. In *Dependable Systems and Networks, 2002. DSN 2002. Proceedings. International Conference on*, pages 167–176. IEEE, 2002.

[18] Christian Cachin and Stefano Tessaro. Asynchronous verifiable information dispersal. In *Reliable Distributed Systems, 2005. SRDS 2005. 24th IEEE Symposium on*, pages 191–201. IEEE, 2005.

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

[20] Miguel Castro, Barbara Liskov, et al. Practical byzantine fault tolerance. In *OSDI*, volume 99, pages 173–186, 1999.

[21] Allen Clement, Manos Kapritsos, Sangmin Lee, Yang Wang, Lorenzo Alvisi, Mike Dahlin, and Taylor Riche. Upright cluster services. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, pages 277–290. ACM, 2009.

[22] Allen Clement, Edmund L Wong, Lorenzo Alvisi, Michael Dahlin, and Mirco Marchetti. Making byzantine fault tolerant systems tolerate byzantine faults. In *NSDI*, volume 9, pages 153–168, 2009.

[23] Kyle Croman, Christian Decker, Ittay Eyal, Adem Efe Gencer, Ari Juels, Ahmed Kosba, Andrew Miller, Prateek Saxena, Elaine Shi, Emin Gun Sirer, Dawn Song, and Roger Wattenhofer and. On scaling decentralized blockchains — a position paper. Manuscript, 2015.

[24] George Danezis and Sarah Meiklejohn. Centrally banked cryptocurrencies. *arXiv preprint arXiv:1505.06895*, 2015.

[25] Roger Dingledine, Nick Mathewson, and Paul Syverson. Tor: The second-generation onion router. Technical report, DTIC Document, 2004.

[26] Cynthia Dwork, Nancy Lynch, and Larry Stockmeyer. Consensus in the presence of partial synchrony. *Journal of the ACM (JACM)*, 35(2):288–323, 1988.

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

[28] Valerie King and Jared Saia. From almost everywhere to everywhere: Byzantine agreement with $o(n^{3/2})$ bits. In *Distributed Computing*, pages 464–478. Springer, 2009.

[29] Valerie King and Jared Saia. Breaking the $o(n^2)$ bit barrier: scalable byzantine agreement with an adaptive adversary. *Journal of the ACM (JACM)*, 58(4):18, 2011.

[30] Ramakrishna Kotla, Lorenzo Alvisi, Mike Dahlin, Allen Clement, and Edmund Wong. Zyzzyva: speculative byzantine fault tolerance. In *ACM SIGOPS Operating Systems Review*, volume 41, pages 45–58. ACM, 2007.

[31] Klaus Kursawe and Victor Shoup. Optimistic asynchronous atomic broadcast. In *in the Proceedings of International Colloqium on Automata, Languages and Programming (ICALP05)(L Caires, GF Italiano, L. Monteiro, Eds.) LNCS 3580*. Citeseer, 2001.

[32] Jae Kwon. TenderMint: Consensus without Mining, August 2014.

[33] Leslie Lamport. The part-time parliament. *ACM Transactions on Computer Systems (TOCS)*, 16(2):133–169, 1998.

[34] Loi Luu, Viswesh Narayanan, Kunal Baweja, Chaodong Zheng, Seth Gilbert, and Prateek Saxena. Scp: A computationally-scalable byzantine consensus protocol for blockchains. Cryptology ePrint Archive, Report 2015/1168, 2015. `http://eprint.iacr.org/`.

[35] Ben Lynn. On the implementation of pairing-based cryptography. *The Department of Computer Science and the Committee on Graduate Studies of Stanford University*, 2007.

[36] Yanhua Mao, Flavio Paiva Junqueira, and Keith Marzullo. Mencius: building efficient replicated state machines for wans. In *OSDI*, volume 8, pages 369–384, 2008.

[37] Robert McMillan. Ibm bets big on bitcoin ledger. Wall Street Journal.

[38] Robert McMillan. How bitcoin became the honey badger of money. Wired Magazine, `http://www.wired.com/2013/12/bitcoin_honey/`, 2013.

[39] Achour Mostefaoui, Hamouma Moumen, and Michel Raynal. Signature-free asynchronous byzantine consensus with t¡ n/3 and o (n 2) messages. In *Proceedings of the 2014 ACM symposium on Principles of distributed computing*, pages 2–9. ACM, 2014.

[40] Satoshi Nakamoto. Bitcoin: A peer-to-peer electronic cash system. `http://bitcon.org/bitcoin.pdf`, 2008.

[41] NIST. Sp 800-37. *Guide for the Security Certification and Accreditation of Federal Information Systems*, 2004.

[42] Diego Ongaro and John Ousterhout. In search of an understandable consensus algorithm. In *Proc. USENIX Annual Technical Conference*, pages 305–320, 2014.

[43] HariGovind V Ramasamy and Christian Cachin. Parsimonious asynchronous byzantine-fault-tolerant atomic broadcast. In *OPODIS*, pages 88–102. Springer, 2005.

[44] David Schwartz, Noah Youngs, and Arthur Britto. The Ripple Protocol Consensus Algorithm, September 2014.

[45] Victor Shoup. Practical threshold signatures. In *EUROCRYPT*, pages 207–220. Springer, 2000.

[46] Atul Singh, Tathagata Das, Petros Maniatis, Peter Druschel, and Timothy Roscoe. Bft protocols under fire. In *Proceedings of the 5th USENIX Symposium on Networked Systems Design and Implementation*, NSDI'08, pages 189–204, Berkeley, CA, USA, 2008. USENIX Association.

[47] Giuliana Santos Veronese, Miguel Correia, Alysson Neves Bessani, and Lau Cheuk Lung. Spin one's wheels? byzantine fault tolerance with a spinning primary. In *Reliable Distributed Systems, 2009. SRDS'09. 28th IEEE International Symposium on*, pages 135–144. IEEE, 2009.

[48] Giuliana Santos Veronese, Miguel Correia, Alysson Neves Bessani, and Lau Cheuk Lung. Ebawa: Efficient byzantine agreement for wide-area networks. In *High-Assurance Systems Engineering (HASE), 2010 IEEE 12th International Symposium on*, pages 10–19. IEEE, 2010.

[49] Zooko Wilcox-O'Hearn. Zfec 1.4. 0. *Open source code distribution:* `http://pypi.python.org/pypi/zfec`, 2008.

## Notes

[1] The decomposition of Atomic Broadcast by way of ACS is not explicit in Cachin et al. [15], though it is easily seen. For completeness, we present this decomposition in the Appendix.

[2] The decomposition of Cachin et al.'s protocol into Asynchronous Common Subset is not explicit, although quite easy to see. To be self-contained, we present the protocol this way in the Appendix.

[3] The expected running time can be reduced to $O(1)$ by [8] running several instances in parallel, though this comes at the expense of throughput.

[4] From `https://metrics.torproject.org/bandwidth.html` on Nov 10, 2015

## A Attacking PBFT

**PBFT.** The PBFT protocol consists of two main workflows: a "fast path" that provides good perfor-

mance in optimistic case (when the network is synchronous and the leader functions correctly), and a "view-change" procedure to change leaders.

The fast path consists of three rounds of communication: PRE_PREPARE, PREPARE, and COMMIT. The leader of a given view is responsible for totally ordering all requests. Upon receiving a client request, the leader multicasts a PRE_PREPARE message specifying the request and a sequence number to all other replicas, who respond by multicasting a corresponding PREPARE message. Replicas multicast a COMMIT message on receipt of $2f$ prepare messages and the PREPARE message, and execute the request on receipt of $2f+1$ COMMIT messages (including their own).

Replicas increment their view number and multicast a VIEW_CHANGE message to elect a new leader when a request takes too long to execute (i.e., longer than a timeout interval), a previously initiated view change has taken too long, or it receives $f+1$ VIEW_CHANGE messages with a higher view number. The leader of the next view is determined by the view number modulo the number of replicas (thus, leadership is transferred in a round-robin manner). The new leader multicasts a NEW_VIEW message once it receives $2f+1$ VIEW_CHANGE messages and includes them as proof of a valid view. A replica accepts the NEW_VIEW mesage if its number is equal to or greater than its own current view number, and resumes processing messages as normal; however messages with lower view numbers are ignored. The timeout interval is initialized to a fixed value ($\Delta$), but increases by a factor of 2 with each consecutive unsuccessful leader election.

**An intermittently synchronous network that thwarts PBFT.** The scheduler does not drop or reorder any messages, but simply delays delivering messages to whichever node is the current leader. In particular, whenever the current leader is a faulty node, this means that messages among all honest nodes are delivered immediately. In the Appendix, we provide a detailed illustration of the PBFT protocol behaves under our attack.

To confirm our analysis, we implemented this malicious scheduler as a proxy that intercepted and delayed all view change messages to the new leader, and tested it against a 1200 line Python implementation of PBFT. The results and message logs we observed were consistent with the above analysis; our replicas became stuck in a loop requesting view changes that never succeeded.

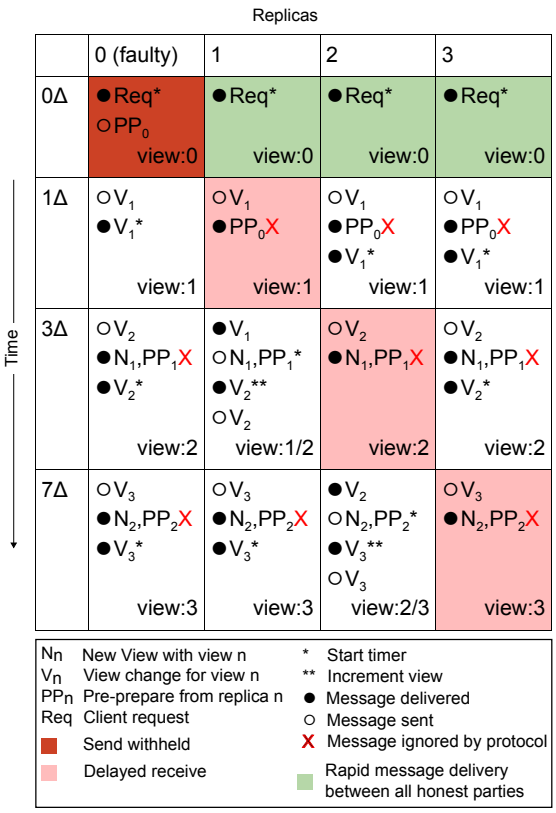


Figure 11: An *intermittently synchronous* scheduler that violates PBFT's assumptions and indeed prevents it from making progress. Only the first four phases are shown - the behavior continues to repeat indefinitely. In the pink regions, messages to the leader are delayed (for longer than the timeout $\Delta$, thus violating the eventual-synchrony assumption). However, all other messages are delivered at the ordinary rate between honest parties, hence "intermittently synchronous."

Since this scheduler is intermittently synchronous, any purely asynchronous protocol (including HoneyBadgerBFT) would make good progress during periods of synchrony, regardless of preceding intervals .

**How PBFT behaves under attack.** In Figure 11, we illustrate our attack on PBFT. The scheduler does not drop or reorder any messages, but simply delays delivering messages to whichever node is the current leader. In particular, whenever the current leader is a faulty node, this means that messages among all honest nodes are delivered immediately.

We abbreviate client requests as "Req," NEW_VIEW messages as "N," VIEW_CHANGE messages as "V," and PRE_PREPARE messages as "PP." The sub-

script on a message indicates the view in which it was sent. Here, ◯ followed by a message indicates that this message has been broadcast to all other nodes (called replicas) by the replica specified by the column number, at the time specified by the row number multiplied by the fixed timeout interval Δ. Similarly, • followed by a message indicates that this message has been delivered to the replica specified by the column number, at the time specified by the row. As multiple VIEW_CHANGE messages for a given view are sent to each individual node, $\bullet V_n$ indicates the delivery of all VIEW_CHANGE messages with view number $n$. A red "X" appended to a delivered message indicates that the messages is ignored because the view number does not match that replica's current view. A "*" indicates that a timer has been started as a result of the delivered message. "**" indicates that a replica's view number has incremented as a result of the delivered message(s). A red region indicates that all broadcast operations from this replica at this time will be delayed by Δ. A pink region indicates that the receipt of all messages will be delayed by Δ.

In this example, the faulty replica 0 is initially the leader and withholds a PRE_PREPARE message for longer than the timeout period Δ. This triggers all nodes to increment their view counter and multicast a VIEW_CHANGE message for view number 1. The scheduler then delays the delivery of all VIEW_CHANGE messages to replica 1 (the leader in view 1). The view change operation for the remaining nodes times out, as they do not receive a valid NEW_VIEW message from replica 1. Nodes 0,2, and 3 then increment their view counters to 2, and multicast another VIEW_CHANGE message. At this point, the VIEW_CHANGE messages for view 1 are delivered to replica 1, which responds by multicasting a NEW_VIEW and a PRE_PREPARE message in view 1. These messages are then delivered and subsequently ignored by all other nodes, as they have progressed to view number 2. Replica 1 will then receive the VIEW_CHANGE messages for view 2, and increments its view counter accordingly. The scheduler then delays the delivery of all VIEW_CHANGE messages to replica 2, ensuring that the view change operation of all other nodes times out again. This process will continue until the faulty replica 0 is again elected leader, at which point the scheduler will deliver all messages at an accelerated rate while replica 0 withholds the corresponding NEW_VIEW and PRE_PREPARE messages to trigger another view change and repeat this cycle. The cycle may continue indefinitely so long as the scheduler withholds VIEW_CHANGE messages from the intended non-faulty leader for longer than the (exponentially increasing) timeout interval, preventing any view changes from succeeding and stopping the protocol from making any progress, despite the fact that at time intervals where replica 0 is the leader (0Δ,8Δ,64Δ...) all non-faulty replicas are able to communicate without any interference.

**Intermittently synchronous networks.** To more clearly illustrate the difference between asynchronous networks, we introduce a new network performance assumption, Δ-*intermittently synchrony*, which is strictly weaker than even *weak synchrony*. The idea is that a Δ-intermittently synchronous network approximates a Δ-synchronous network in the sense that *on average* it delivers messages at a rate of $1/\Delta$. However, the delivery rate may be unevenly distributed in time (e.g., "bursty"), delivering no messages at all during some time intervals and delivering messages rapidly during others.

**Definition 2.** *A network is* Δ-intermittently synchronous *if for any initial time $T_0$, and for any duration $\underline{D}$, there exists an interval $[T_0, T_1]$ such that $T_1 - T_0 \geq \underline{D}$ and the number of asynchronous rounds advanced during $[T_0, T_1]$ is at least $(T_1 - T_0)/\Delta$.*

It is clearly the case that every Δ-synchronous network is also Δ-intermittently synchronous, since for every interval of duration Δ, messages sent prior to that interval are delivered by the end of that interval. It is also clear that any intermittently synchronous network guarantees eventual delivery (i.e., it is no weaker than the asynchronous model).

Asynchronous protocols make progress whenever rounds of messages are delivered. Since an intermittently-synchronous network guarantees messages are delivered *on average* within Δ, this means *any asynchronous protocol also makes progress at an average rate of* Δ.

## B Asynchronous Binary Byzantine Agreement

**Realizing binary agreement from a common coin.** Binary agreement allows nodes to agree on the value of a single bit. More formally, binary agreement guarantees three properties:

- *(Agreement)* If any correct node outputs the bit $b$, then every correct node outputs $b$.
- *(Termination)* If all correct nodes receive input, then every correct node outputs a bit.
- *(Validity)* If any correct node outputs $b$, then at least one correct node received $b$ as input.

The validity property implies *unanimity*: if all of the correct nodes receive the same input value $b$, then $b$ must be the decided value. On the other hand, if at any point two nodes receive different inputs, then the adversary may force the decision to either value even before the remaining nodes receive input.

We instantiate this primitive with a protocol based on cryptographic common coin, which essentially act as synchronizing gadgets. The adversary only learns the value of the next coin after a majority of correct nodes have committed to a vote — if the coin matches the majority vote, then that is the decided value. The adversary can influence the majority vote each round, but only until the coin is revealed.

The Byzantine agreement algorithm from Moustefaoui et al. [39] is shown in Figure 12. Its expected running time is $O(1)$, and in fact completes within $O(k)$ rounds with probability $1-2^{-k}$. When instantiated with the common coin defined below, the total communication complexity is $O(\lambda N^2)$, since it uses a constant number of common coins.

**Realizing a common coin from a threshold signature scheme.** A common coin is a protocol that satisfies the following properties:

- If $f+1$ parties call $\mathsf{GetCoin}()$, then all parties eventually receive a common value, $s$.
- The value $s$ is uniformly sampled in the range $\{0,1\}^\lambda$, and cannot be influenced by the adversary.
- Until at least one party calls $\mathsf{GetCoin}()$, no information about $s$ is revealed to the adversary.

Following [16], a common coin can be realized from a unique threshold signature scheme. An $(N,f)$-threshold signature scheme involves distributing shares of a signing key $\mathsf{sk}_i$ to each of N parties. Given a message, a party using secret key $\mathsf{sk}_i$ can compute a *signature share* on an arbitrary message $m$. Given $f+1$ such signature shares for message $m$, anyone can combine the shares to produce a valid signature, which verifies under the public key $\mathsf{pk}$. With fewer than $f+1$ shares, (i.e., unless at least one honest party deliberately computes and reveals a share), the adversary learns nothing. We rely on an additional *uniqueness* property, which guarantees that for a given public key $\mathsf{pk}$, there exists exactly one valid signature on each message $m$.

The idea of Cachin et al. [16] is simply to use the threshold signature as a source of random bits, by signing a string that serves as the "name" of the coin. This naturally allows the protocol to be used to generate a sequence (or random-access table) of coins, and makes it convenient to use in modular subprotocols.

**Algorithm BA (for party $\mathcal{P}_i$)**

- upon receiving input $b_{\mathsf{input}}$, set $\mathsf{est}_0 := b_{\mathsf{input}}$ and proceed as follows in consecutive epochs, with increasing labels $r$:
  - multicast $\mathtt{BVAL}(\mathsf{est}_r)$
  - $\mathsf{bin\_values}_r := \{\}$
  - upon receiving $\mathtt{BVAL}_r(b)$ messages from $f+1$ processes, if $\mathtt{BVAL}_r(b)$ has not been sent, multicast $\mathtt{BVAL}_r(b)$
  - upon receiving $\mathtt{BVAL}_r(b)$ messages from $2f+1$ processes, $\mathsf{bin\_values}_r := \mathsf{bin\_values}_r \cup \{b\}$
  - wait until $\mathsf{bin\_values}_r \neq$, then
    - multicast $\mathtt{AUX}_r(w)$ where $w \in \mathsf{bin\_values}_r$
    - wait until at least $(N-f)$ $\mathtt{AUX}_r$ messages have been received, such that the set of values carried by these messages, vals are a subset of $\mathsf{bin\_values}_r$ (note that $\mathsf{bin\_values}_r$ may continue to change $\mathtt{BVAL}_r$ messages are received, thus this condition may be triggered upon arrival of either an $\mathtt{AUX}_r$ or a $\mathtt{BVAL}_r$ message)
    - $s \leftarrow \mathsf{Coin}_r.\mathsf{GetCoin}()$
    - if $\mathsf{vals} = \{b\}$, then
      - $\mathsf{est}_{r+1} := b$
      - if $(b = s\%2)$ then output $b$
    - else $\mathsf{est}_{r+1} := s\%2$
- continue looping until both a value $b$ is output in some round $r$, and the value $\mathsf{Coin}_{r'} = v$ for some round $r' > r$

Figure 12: Binary Byzantine Agreement from a Common Coin. Note that in the algorithm, $b$ ranges over $\{0,1\}$. This protocol makes use of a sequence of common coins, labeled $\mathsf{Coin}_r$.

We assume that $\mathsf{ThresholdCombine}$ is robust, in the sense that if it is run with a set of more than $f+1$ signature shares, it rejects any invalid ones. In particular, if $2f+1$ shares are provided, certainly a valid subset of $f+1$ is among them. In practice, any incorrect shares detected this way can be used as evidence to incriminate a node.

Concretely, we use an efficient threshold scheme [11] based on bilinear groups and the Gap Diffie Hellman assumption. We use $\mathsf{TSIG}$ to refer to this scheme. The common coin requires only one asynchronous round to complete, and the communication cost is $O(N\lambda)$ per node.

**Algorithm** $\mathsf{Coin}_{\mathsf{sid}}$ **for party** $\mathcal{P}_i$

$\mathsf{sid}$ is assumed to be a unique nonce that serves as "name" of this common coin

- (Trusted Setup Phase): A trusted dealer runs $\mathsf{pk}, \{sk_i\} \leftarrow \mathsf{ThresholdSetup}$ to generate a common public key, as well as secret key shares $\{\mathsf{sk}_i\}$, one for each party (secret key $\mathsf{sk}_i$ is distributed to party $\mathcal{P}_i$).
- on input $\mathsf{GetCoin}$, multicast $\mathsf{ThresholdSign}_{\mathsf{pk}}(\mathsf{sk}_i, \mathsf{sid})$
- upon receiving at least $f+1$ shares, attempt to combine them into a signature:

  $\mathsf{sig} \leftarrow \mathsf{ThresholdCombine}_{\mathsf{pk}}(\{j, s_j\})$

  if $\mathsf{ThresholdVerify}_{\mathsf{pk}}(\mathsf{sid})$ then deliver $\mathsf{sig}$

Figure 13: A common coin based on threshold signatures [45]