Snowflake to Avalanche: A Novel Metastable Consensus Protocol Family for Cryptocurrencies

Team Rocket[†]
t-rocket@protonmail.com

Revision: 05/16/2018 21:51:26 UTC

Abstract

This paper introduces a new family of leaderless Byzantine fault tolerance protocols, built on a metastable mechanism. These protocols provide a strong probabilistic safety guarantee in the presence of Byzantine adversaries, while their concurrent nature enables them to achieve high throughput and scalability. Unlike blockchains that rely on proof-of-work, they are quiescent and green. Surprisingly, unlike traditional consensus protocols which require $O(n^2)$ communication, their communication complexity ranges from $O(kn \log n)$ to $O(kn)$ for some security parameter $k \ll n$.

The paper describes the protocol family, instantiates it in three separate protocols, analyzes their guarantees, and describes how they can be used to construct the core of an internet-scale electronic payment system. The system is evaluated in a large scale deployment. Experiments demonstrate that the system can achieve high throughput (1300 tps), provide low confirmation latency (4 sec), and scale well compared to existing systems that deliver similar functionality. For our implementation and setup, the bottleneck of the system is in transaction verification.

1 Introduction

Achieving agreement among a set of distributed hosts lies at the core of countless applications, ranging from Internet-scale services that serve billions of people [12, 30] to cryptocurrencies worth billions of dollars [1]. To date, there have been two main families of solutions to this problem. Traditional consensus protocols rely on all-to-all communication to ensure that all correct nodes reach the same decisions with absolute certainty. Because they require quadratic communication overhead and accurate knowledge of membership, they have been difficult to scale to large numbers of participants.

On the other hand, Nakamoto consensus protocols [9, 24, 25, 34, 40–43, 49–51] have become popular with the rise of Bitcoin. These protocols provide a probabilistic safety guarantee: Nakamoto consensus decisions may revert with some probability ϵ. A protocol parameter allows this probability to be rendered arbitrarily small, enabling high-value financial systems to be constructed on this foundation. This family is a natural fit for open, permissionless settings where any node can join the system at any time. Yet, these protocols are quite costly, wasteful, and limited in performance. By construction, these protocols cannot quiesce: their security relies on constant participation by miners, even when there are no decisions to be made. Bitcoin currently consumes around 63.49 TWh/year [22], about twice as all of Denmark [15]. Moreover, these protocols suffer from an inherent scalability bottleneck that is difficult to overcome through simple reparameterization [18].

This paper introduces a new family of consensus protocols. Inspired by gossip algorithms, this new family gains its safety through a deliberately metastable mechanism. Specifically, the system operates by repeatedly sampling the network at random, and steering the correct nodes towards the same outcome. Analysis shows that metastability is a powerful, albeit non-universal, technique: it can move a large network to an irreversible state quickly, though it is not always guaranteed to do so.

Similar to Nakamoto consensus, this new protocol family provides a probabilistic safety guarantee, using a tunable security parameter that can render the possibility of a consensus failure arbitrarily small. Unlike Nakamoto consensus, the protocols are green, quiescent and efficient; they do not rely on proof-of-work [23] and do not consume energy when there are no decisions to be made. The efficiency of the protocols stems from two sources: they require communication overheads ranging from $O(kn \log n)$ to $O(kn)$ for some small security parameter k, and they establish only a partial order among dependent transactions.

This combination of the best features of traditional and Nakamoto consensus involves one significant tradeoff: liveness for conflicting transactions. Specifically, the new family guarantees liveness only for *virtuous* transactions, i.e. those issued by correct clients and thus guaranteed not to conflict with other transactions. In a cryptocur-

[†] `0x 8a5d 2d32 e68b c500 36e4 d086 0446 17fe 4a0a 0296 b274 999b a568 ea92 da46 d533`

rency setting, cryptographic signatures enforce that only a key owner is able to create a transaction that spends a particular coin. Since correct clients follow the protocol as prescribed, they are guaranteed both safety and liveness. In contrast, the protocols do not guarantee liveness for *rogue* transactions, submitted by Byzantine clients, which conflict with one another. Such decisions may stall in the network, but have no safety impact on virtuous transactions. We show that this is a sensible tradeoff, and that resulting system is sufficient for building complex payment systems.

Overall, this paper makes one significant contribution: a brand new family of consensus protocols suitable for cryptocurrencies, based on randomized sampling and metastable decision. The protocols provide a strong probabilistic safety guarantee, and a guarantee of liveness for correct clients.

The next section provides intuition behind the new protocols, Section 3 provides proofs for safety and liveness, Section 4 describes a Bitcoin-like payment system, Section 5 evaluates the system, Section 6 presents related work, and finally, Section 7 summarizes our contributions.

2 Approach

We start with a non-Byzantine protocol, Slush, and progressively build up Snowflake, Snowball and Avalanche, all based on the same common metastable mechanism. Though we provide definitions for the protocols, we defer their formal analysis and proofs of their properties to the next section.

Overall, this protocol family achieves its properties by humbly cheating in three different ways. First, taking inspiration from Bitcoin, we adopt a safety guarantee that is probabilistic. This probabilistic guarantee is indistinguishable from traditional safety guarantees in practice, since appropriately small choices of ϵ can render consensus failure practically infeasible, less frequent than CPU miscomputations or hash collisions. Second, instead of a single replicated state machine (RSM) model, where the system determines a sequence of totally-ordered transactions $T_0, T_1, T_2, \ldots$ issued by any client, we adopt a *parallel consensus model* with authenticated clients, where each client interacts independently with its own RSMs and optionally transfers ownership of its RSM to another client. The system establishes only a partial order between dependent transactions. Finally, we provide no liveness guarantee for misbehaving clients, but ensure that well-behaved clients will eventually be serviced. These techniques, in conjunction, enable the system to nevertheless implement a useful Bitcoin-like cryptocurrency, with drastically better performance and scalability.

2.1 Model, Goals, Threat Model

We assume a collection of nodes, $\mathcal{N}$, composed of correct nodes $\mathcal{C}$ and Byzantine nodes $\mathcal{B}$, where $n = |\mathcal{N}|$. We adopt what is commonly known as Bitcoin's unspent transaction output (UTXO) model. In this model, *clients* are authenticated and issue cryptographically signed transactions that fully consume an existing UTXO and issue new UTXOs. Unlike nodes, clients do not participate in the decision process, but only supply transactions to the *nodes* running the consensus protocol. Two transactions *conflict* if they consume the same UTXO and yield different outputs. Correct clients never issue conflicting transactions, nor is it possible for Byzantine clients to forge conflicts with transactions issued by correct clients. However, Byzantine clients can issue multiple transactions that conflict with one another, and correct clients can only consume one of those transactions. The goal of our family of consensus protocols, then, is to *accept* a set of non-conflicting transactions in the presence of Byzantine behavior. Each client can be considered as a replicated state machine whose transitions are defined by a totally ordered list of accepted transactions.

Our family of protocols provide the following guarantees with high probability:

P1. Safety. No two correct nodes will accept conflicting transactions.

P2. Liveness. Any transaction issued by a correct client (aka virtuous transaction) will eventually be accepted by every correct node.

Instead of unconditional agreement, our approach guarantees that safety will be upheld with probability $1 - \epsilon$, where the choice of the security parameter ϵ is under the control of the system designer and applications.

We assume a powerful adaptive adversary capable of observing the internal state and communications of every node in the network, but not capable of interfering with communication between correct nodes. Our analysis assumes a synchronous network, while our deployment and evaluation is performed in a partially synchronous setting. We conjecture that our results hold in partially synchronous networks, but the proof is left to future work. We do not assume that all members of the network are known to all participants, but rather may temporarily have some discrepancies in network view. We assume a safe bootstrapping system, similar to that of Bitcoin, that enables a node to connect with sufficiently many correct nodes to acquire a statistically unbiased view of the network. We do not assume a PKI. We make standard cryptographic assumptions related to public key signatures and hash functions.

2.2 Slush: Introducing Metastability

The core of our approach is a single-decree consensus protocol, inspired by epidemic or gossip protocols. The

```
1: procedure onQuery(v, col')
2:     if col = ⊥ then col := col'
3:     respond(v, col)
4: procedure slushLoop(u, col_0 ∈ {R, B, ⊥})
5:     col := col_0 // initialize with a color
6:     for r ∈ {1 ... m} do
7:         // if ⊥, skip until onQuery sets the color
8:         if col = ⊥ then continue
9:         // randomly sample from the known nodes
10:        K := sample(N\u, k)
11:        P := [query(v, col)   for v ∈ K]
12:        for col' ∈ {R, B} do
13:            if P.count(col') ≥ α · k then
14:                col := col'
15:    accept(col)
```

Figure 1: Slush protocol. Timeouts elided for readability.

simplest metastable protocol, Slush, is the foundation of this family, shown in Figure 1. Slush is *not* tolerant to Byzantine faults, but serves as an illustration for the BFT protocols that follow. For ease of exposition, we will describe the operation of Slush using a decision between two conflicting colors, red and blue.

In Slush, a node starts out initially in an uncolored state. Upon receiving a transaction from a client, an uncolored node updates its own color to the one carried in the transaction and initiates a query. To perform a query, a node picks a small, constant sized (k) sample of the network uniformly at random, and sends a query message. Upon receiving a query, an uncolored node adopts the color in the query, responds with that color, and initiates its own query, whereas a colored node simply responds with its current color. If k responses are not received within a time bound, the node picks an additional sample from the remaining nodes uniformly at random and queries them until it collects all responses. Once the querying node collects k responses, it checks if a fraction $\geq \alpha k$ are for the same color, where $\alpha > 0.5$ is a protocol parameter. If the αk threshold is met and the sampled color differs from the node's own color, the node flips to that color. It then goes back to the query step, and initiates a subsequent round of query, for a total of m rounds. Finally, the node decides the color it ended up with at time m.

This simple protocol has some curious properties. First, it is almost *memoryless*: a node retains no state between rounds other than its current color, and in particular maintains no history of interactions with other peers. Second, unlike traditional consensus protocols that query every participant, every round involves sampling just a small, constant-sized slice of the network at random. Second, even if the network starts in the metastable state of a 50/50 red-blue split, random perturbations in sampling will cause one color to gain a slight edge and repeated samplings afterwards will build upon and amplify that imbalance. Finally, if m is chosen high enough,

```
1: procedure snowflakeLoop(u, col_0 ∈ {R, B, ⊥})
2:     col := col_0, cnt := 0
3:     while undecided do
4:         if col = ⊥ then continue
5:         K := sample(N\u, k)
6:         P := [query(v, col)   for v ∈ K]
7:         for col' ∈ {R, B} do
8:             if P.count(col') ≥ α · k then
9:                 if col' ≠ col then
10:                    col := col', cnt := 0
11:                else
12:                    if ++cnt > β then accept(col)
```

Figure 2: Snowflake.

Slush ensures that all nodes will be colored identically whp. Each node has a constant, predictable communication overhead per round, and we will show that m grows logarithmically with n.

If Slush is deployed in a network with Byzantine nodes, the adversary can interfere with decisions. In particular, if the correct nodes develop a preference for one color, the adversary can attempt to flip nodes to the opposite so as to keep the network in balance. The Slush protocol lends itself to analysis but does not, by itself, provide a strong safety guarantee in the presence of Byzantine nodes, because the nodes lack state. We address this in our first BFT protocol.

2.3 Snowflake: BFT

Snowflake augments Slush with a single counter that captures the strength of a node's conviction in its current color. This per-node counter stores how many consecutive samples of the network have all yielded the same color. A node accepts the current color when its counter exceeds β, another security parameter. Figure 2 shows the amended protocol, which includes the following modifications:

1. Each node maintains a counter cnt;
2. Upon every color change, the node resets cnt to 0;
3. Upon every successful query that yields $\geq \alpha k$ responses for the same color as the node, the node increments cnt.

When the protocol is correctly parameterized for a given threshold of Byzantine nodes and a desired ϵ-guarantee, it can ensure both safety (P1) and liveness (P2). As we later show, there exists a phase-shift point after which correct nodes are more likely to tend towards a decision than a bivalent state. Further, there exists a point-of-no-return after which a decision is inevitable. The Byzantine nodes lose control past the phase shift, and the correct nodes begin to commit past the point-of-no-return, to adopt the same color, whp.

2.4 Snowball: Adding Confidence

Snowflake's notion of state is ephemeral: the counter gets reset with every color flip. While, theoretically, the proto-

```
1: procedure snowballLoop(u, col_0 ∈ {R, B, ⊥})
2:     col := col_0, lastcol := col_0, cnt := 0
3:     d[R] := 0, d[B] := 0
4:     while undecided do
5:         if col = ⊥ then continue
6:         K := sample(N\u, k)
7:         P := [query(v, col)  for v ∈ K]
8:         for col' ∈ {R, B} do
9:             if P.count(col') ≥ α · k then
10:                d[col']++
11:                if d[col'] > d[col] then
12:                    col := col'
13:                if col' ≠ lastcol then
14:                    lastcol := col', cnt := 0
15:                else
16:                    if ++cnt > β then accept(col)
```

Figure 3: Snowball.

```
1: procedure avalancheLoop
2:     while true do
3:         find T that satisfies T ∈ T ∧ T ∉ Q
4:         K := sample(N\u, k)
5:         P := Σ_{v∈K} query(v, T)
6:         if P ≥ α · k then
7:             c_T := 1
8:             // update the preference for ancestors
9:             for T' ∈ T : T' ←* T do
10:                if d(T') > d(P_T'.pref) then
11:                    P_T'.pref := T'
12:                if T' ≠ P_T'.last then
13:                    P_T'.last := T', P_T'.cnt := 0
14:                else
15:                    ++P_T'.cnt
16:        // otherwise, c_T remains 0 forever
17:        Q := Q ∪ {T}   // mark T as queried
```

Figure 4: Avalanche: the main loop.

col is able to make strong guarantees with minimal state, we will now improve the protocol to make it harder to attack by incorporating a more permanent notion of belief. Snowball augments Snowflake with *confidence counters* that capture the number of queries that have yielded a threshold result for their corresponding color (Figure 3). A node decides a color if, during a certain number of consecutive queries, its confidence for that color exceeds that of other colors. The differences between Snowflake and Snowball are as follows:

1. Upon every successful query, the node increments its confidence counter for that color.
2. A node switches colors when the confidence in its current color becomes lower than the confidence value of the new color.

Snowball is not only harder to attack than Snowflake, but is more easily generalized to multi-decree protocols.

2.5 Avalanche: Adding a DAG

Avalanche, our final protocol, generalizes Snowball and maintains a dynamic append-only Directed Acyclic Graph (DAG) of all known transactions. The DAG has a single sink that is the *genesis vertex*. Maintaining a DAG provides two significant benefits. First, it improves efficiency, because a single vote on a DAG vertex implicitly votes for all transactions on the path to the genesis vertex. Second, it also improves security, because the DAG intertwines the fate of transactions, similar to the Bitcoin blockchain. This renders past decisions difficult to undo without the approval of correct nodes.

When a client creates a transaction, it names one or more *parents*, which are included inseparably in the transaction and form the edges of the DAG. The parent-child relationships encoded in the DAG may, but do not need to, correspond to application-specific dependencies; for instance, a child transaction need not spend or have any relationship with the funds received in the parent transaction. We use the term *ancestor set* to refer to all transactions reachable via parent edges back in history, and *progeny* to refer to all children transactions and their offspring.

The central challenge in the maintenance of the DAG is to choose among *conflicting transactions*. The notion of conflict is application-defined and transitive, forming an equivalence relation. In our cryptocurrency application, transactions that spend the same funds (*double-spends*) conflict, and form a *conflict set*, out of which only a single one can be accepted. Note that the conflict set of a virtuous transaction is always a singleton. Shaded regions in Figure 7 indicate conflict sets.

Avalanche embodies a Snowball instance for each conflict set. Whereas Snowball uses repeated queries and multiple counters to capture the amount of confidence built in conflicting transactions (colors), Avalanche takes advantage of the DAG structure and uses a transaction's progeny. Specifically, when a transaction T is queried, all transactions reachable from T by following the DAG edges are implicitly part of the query. A node will only respond positively to the query if T and its entire ancestry are currently the preferred option in their respective conflict sets. If more than a threshold of responders vote positively, the transaction is said to collect a *chit*, $c_{uT} = 1$, otherwise, $c_{uT} = 0$. Nodes then compute their *confidence* as the sum of chit values in the progeny of that transaction. Nodes query a transaction just once and rely on new vertices and chits, added to the progeny, to build up their confidence. Ties are broken by an initial preference for first-seen transactions.

2.6 Avalanche: Specification

Each correct node u keeps track of all transactions it has learned about in set $\mathcal{T}_u$, partitioned into mutually

```
1: procedure INIT
2:     𝒯 ≔ ∅  // the set of known transactions
3:     𝒬 ≔ ∅  // the set of queried transactions
4: procedure onGenerateTx(data)
5:     edges ≔ {T′ ← T : T′ ∈ parentSelection(𝒯)}
6:     T ≔ Tx(data, edges)
7:     onReceiveTx(T)
8: procedure onReceiveTx(T)
9:     if T ∉ 𝒯 then
10:        if 𝒫_T = ∅ then
11:            𝒫_T ≔ {T}, 𝒫_T.pref ≔ T
12:            𝒫_T.last ≔ T, 𝒫_T.cnt ≔ 0
13:        else 𝒫_T ≔ 𝒫_T ∪ {T}
14:        𝒯 ≔ 𝒯 ∪ {T}, c_T ≔ 0.
```

Figure 5: Avalanche: transaction generation.

```
1: function isPreferred(T)
2:     return T = 𝒫_T.pref
3: function isStronglyPreferred(T)
4:     return ∀T′ ∈ 𝒯, T′ ←* T : isPreferred(T′)
5: function isAccepted(T)
6:     return
       ((∀T′ ∈ 𝒯, T′ ← T : isAccepted(T′))
         ∧ |𝒫_T| = 1 ∧ d(T) > β1)  // safe early commitment
       ∨𝒫_T.cnt > β2  // consecutive counter

7: procedure onQuery(j, T)
8:     onReceiveTx(T)
9:     respond(j, isStronglyPreferred(T))
```

Figure 6: Avalanche: voting and decision primitives.

exclusive conflict sets $\mathcal{P}_T, T \in \mathcal{T}_u$. Since conflicts are transitive, if T_i and T_j are conflicting, then $\mathcal{P}_{T_i} = \mathcal{P}_{T_j}$.

We write $T' \leftarrow T$ if T has a parent edge to transaction T', The "$\overset{*}{\leftarrow}$"-relation is its reflexive transitive closure, indicating a path from T to T'. Each node u can compute a confidence value, $d_u(T)$, from the progeny as follows:

$$d_u(T) = \sum_{T' \in \mathcal{T}_u, T \overset{*}{\leftarrow} T'} c_{uT'}.$$

In addition, it maintains its own local list of known nodes $\mathcal{N}_u \subseteq \mathcal{N}$ that comprise the system. For simplicity, we assume for now $\mathcal{N}_u = \mathcal{N}$, and elide subscript u in contexts without ambiguity. DAGs built by different nodes are guaranteed to be compatible. Specifically, if $T' \leftarrow T$, then every node in the system that has T will also have T' and the same relation $T' \leftarrow T$; and conversely, if $T' \not\leftarrow T$, then no nodes will end up with $T' \leftarrow T$.

Each node implements an event-driven state machine, centered around a query that serves both to solicit votes on each transaction and, simultaneously, to notify other nodes of the existence of newly discovered transactions. In particular, when node u discovers a transaction T through a query, it starts a one-time query process by sampling k random peers. A query starts by adding T to $\mathcal{T}$, initializing c_T to 0, and then sending a message to the selected peers.

Node u answers a query by checking whether each T' such that $T' \overset{*}{\leftarrow} T$ is currently preferred among competing transactions $\forall T'' \in \mathcal{P}_{T'}$. If every single ancestor T' fulfills this criterion, the transaction is said to be *strongly preferred*, and receives a yes-vote (1). A failure of this criterion at any T' yields a no-vote (0). When u accumulates k responses, it checks whether there are αk yes-votes for T, and if so grants the chit $c_T = 1$ for T. The above process will yield a labeling of the DAG with chit value c_T and associated confidence for each transaction T. The chits are the result of one-time samples and are immutable, while $d(T)$ can increase as the DAG grows. Because c_T values range from 0 to 1, confidence values are monotonic.

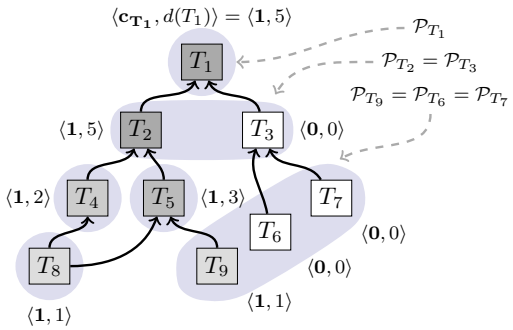

Figure 7: Example of chit and confidence values, labeled as tuples in that order. Darker boxes indicate transactions with higher confidence values.

Figure 7 illustrates a sample DAG built by Avalanche. Similar to Snowball, sampling in Avalanche will create a positive feedback for the preference of a single transaction in its conflict set. For example, because T_2 has larger confidence than T_3, its descendants are more likely collect chits in the future compared to T_3.

Similar to Bitcoin, Avalanche leaves determining the acceptance point of a transaction to the application. An application supplies an isAccepted predicate that can take into account the value at risk in the transaction and the chances of a decision being reverted to determine when to decide.

Committing a transaction can be performed through a *safe early commitment*. For virtuous transactions, T is accepted when it is the only transaction in its conflict set and has a confidence greater than threshold β_1. As in Snowball, T can also be accepted after a β_2 number of consecutive successful queries. If a virtuous transaction fails to get accepted due to a liveness problem with parents, it could be accepted if reissued with different parents.

Figure 5 shows how Avalanche performs parent selection and entangles transactions. Because transactions

```
1: initialize u.col ∈ {R, B} for all u ∈ 𝒩
2: for t = 1 to φ do
3:     u := SAMPLE(𝒩, 1)
4:     𝒦 := SAMPLE(𝒩\u, k)
5:     P := [v.col  for v ∈ 𝒦]
6:     for col' ∈ {R, B} do
7:         if P.COUNT(col') ≥ α · k then
8:             u.col := col'
```

Figure 8: Slush run by a global scheduler.

that consume and generate the same UTXO do not conflict with each other, any transaction can be reissued with different parents.

Figure 4 illustrates the protocol main loop executed by each node. In each iteration, the node attempts to select a transaction T that has not yet been queried. If no such transaction exists, the loop will stall until a new transaction is added to $\mathcal{T}$. It then selects k peers and queries those peers. If more than αk of those peers return a positive response, the chit value is set to 1. After that, it updates the preferred transaction of each conflict set of the transactions in its ancestry. Next, T is added to the set $\mathcal{Q}$ so it will never be queried again by the node. The code that selects additional peers if some of the k peers are unresponsive is omitted for simplicity.

Figure 6 shows what happens when a node receives a query for transaction T from peer j. First it adds T to $\mathcal{T}$, unless it already has it. Then it determines if T is currently strongly preferred. If so, the node returns a positive response to peer j. Otherwise, it returns a negative response. Notice that in the pseudocode, we assume when a node knows T, it also recursively knows the entire ancestry of T. This can be achieved by postponing the delivery of T until its entire ancestry is recursively fetched. In practice, an additional gossip process that disseminates transactions is used in parallel, but is not shown in pseudocode for simplicity.

3 Analysis

In this section, we analyze Slush, Snowflake, Snowball, and Avalanche.

Network Model We assume a synchronous communication network, where at each time step t, a global scheduler chooses a single correct node uniformly at random.

Preliminaries Let $c = |\mathcal{C}|$ and $b = |\mathcal{B}|$; let $u \in \mathcal{C}$ be any correct node; let $k \in \mathbb{N}_+$; and let $\alpha \in \mathbb{R} = (1/2, 1]$. We let R ("red") and B ("blue") represent two generic conflicting choices. Without loss of generality, we focus our attention on counts of R, i.e. the total number of nodes that prefer R.

Each network query of k peers corresponds to a sample without replacement out of a network of n nodes, also referred to as a hypergeometric sample. We let the random variable $\mathcal{H}^k_{\mathcal{N},x} \to [0, k]$ denote the resulting counts of R (ranging from 0 to k), where x is the total count of R in the population. The probability that the query achieves the required threshold of αk or more votes is given by:

$$P(\mathcal{H}^k_{\mathcal{N},x} \geq \alpha k) = \sum_{j=\alpha k}^{k} \frac{\binom{x}{j}\binom{n-x}{k-j}}{\binom{n}{k}} \tag{1}$$

Tail Bound We can reduce some of the complexity in Equation 1 by introducing a bound on the hypergeometric distribution induced by $\mathcal{H}^k_{\mathcal{C},x}$. Let $p = x/c$ be the ratio of support for R in the population. The expectation of $\mathcal{H}^k_{\mathcal{C},x}$ is exactly kp. Then, the probability that $\mathcal{H}^k_{\mathcal{C},x}$ will deviate from the mean by more than some small constant ψ is given by the Hoeffding tail bound [29], as follows,

$$P(\mathcal{H}^k_{\mathcal{C},x} \leq (p - \psi)k) \leq e^{-k\mathcal{D}(p-\psi,p)} \tag{2}$$

where $\mathcal{D}(p - \psi, p)$ is the Kullback-Leibler divergence, measured as

$$\mathcal{D}(a, b) = a \log \frac{a}{b} + (1 - a) \log \frac{1 - a}{1 - b} \tag{3}$$

3.1 Analysis of Slush

Slush operates in a non-Byzantine setting; that is, $\mathcal{B} = \varnothing$ and thus $\mathcal{C} = \mathcal{N}$ and $c = n$. In this section, we will show that (R1) Slush converges to a state where all nodes agree on the same color in finite time almost surely (i.e. with probability 1); (R2) provide a closed-form expression for speed towards convergence; and (R3) characterize the minimum number of steps per node required to reach agreement with probability $\geq 1 - \epsilon$.

The procedural version of Slush in Figure 1 made use of a parameter m, the number of rounds that a node executes Slush queries. To derive this parameter, we transform the protocol execution from a procedural and concurrent version to one carried out by a scheduler, shown in Figure 8. What we ultimately want to extract is the total number of rounds ϕ that the scheduler will need to execute in order to guarantee that the entire network is the same color, whp.

We analyze the system mainly using standard Markov chain techniques. Let $\mathbf{X} = \{X_1, X_2, \ldots X_\phi\}$ be a discrete-time Markov chain. Let s_i, $i \in [0, c]$, represent a state in this Markov chain, where the count of R is i and the count of B is $c - i$. Let M be the transition probability matrix for the Markov chain. Let $q(s_i) = M(s_i, s_{i-1})$, $p(s_i) = M(s_i, s_{i+1})$, and $r(s_i) = M(s_i, s_i)$, where $p(s_0) = q(s_0) = p(s_c) = q(s_c) = 0$. This is a birth-death process where the number of states is c and the two endpoints, s_0 and s_c, are absorbing states, where all nodes are red and blue, respectively.

To extract concrete values, we apply the following probabilities to $q(s_i)$ and $p(s_i)$, which capture the probability of picking a red node and successfully sampling

for blue, and vice versa.

$$p(s_i) = \frac{c-i}{c} P(\mathcal{H}^k_{\mathcal{C},i} \geq \alpha k)$$
$$q(s_i) = \frac{i}{c} P(\mathcal{H}^k_{\mathcal{C},c-i} \geq \alpha k)$$

Lemma 1 (R1)**.** *Slush reaches an absorbing state in finite time almost surely.*

Proof. Let s_i be a starting state where $i \leq c - \alpha k$. All such states s_i then have a non-zero probability of reaching absorbing state s_0 for all time-steps $t \geq i$. States where $i > c - \alpha$ have no possible threshold majority for B, and all timesteps $t \leq i$ do not allow enough transitions to ever reach s_0. Under these conditions, the probability of absorption is strictly > 0, and therefore Slush converges in finite steps. □

Lemma 2 (R2)**.** ***Expected Rounds to Deciding*** B**.** *Let U_{s_i} be a random variable that expresses the number of steps to reach s_0 from state s_i. Let*

$$\mathscr{Y}_c(h) = \sum_{l=s_h}^{s_{c-1}} \prod_{j=s_1}^{l} \frac{q(j)}{p(j)} \tag{4}$$

Then,

$$\begin{aligned}\mathbb{E}[U_{s_i}] = \mathbb{E}[U_{s_1}] &\sum_{l=s_0}^{s_{i-1}} \prod_{j=s_1}^{l} \frac{q(j)}{p(j)} \\ &- \sum_{l=s_0}^{s_{i-1}} \sum_{j=s_1}^{l} \frac{1}{p(j)} \prod_{v=j}^{l} \frac{q(v+1)}{p(v+1)}\end{aligned} \tag{5}$$

where

$$\mathbb{E}[U_{s_1}] = \mathscr{Y}_c(0)^{-1} \sum_{l=s_1}^{s_{c-1}} \sum_{j=s_1}^{l} \frac{1}{p(j)} \prod_{v=j}^{l} \frac{q(v+1)}{p(v+1)} \tag{6}$$

Proof. The expected time to absorption in s_0 can be modeled using the following recurrence equation:

$$\begin{aligned}\mathbb{E}[U_{s_i}] = q(s_i)\mathbb{E}[U_{s_{i-1}}] + p(s_i)\mathbb{E}[U_{s_{i+1}}] \\ + r(s_i)\mathbb{E}[U_{s_i}]\end{aligned} \tag{7}$$

for $i \in [1, c-1]$, with boundary condition $\mathbb{E}[U_{s_0}] = 0$. Solving for the recurrence relation yields the result. □

$\mathcal{C}$ Size	600	1200	2400	4800	9600
Exp. Conv.	12.66	14.39	15.30	16.43	18.61

Table 1: Expected number of per-node-iterations to convergence starting at worst-case (equal) $\mathcal{C}$ network split, in the case of $k = 10$, $\alpha = 0.8$. Standard deviation for all samples is ≤ 2.5.

Table 1 shows the latency (i.e. number of timesteps per-node) to reach the B deciding state starting from the metastable 50/50 split, in other words, value $\mathbb{E}[U_{s_{c/2}}]/c$. As the network size c doubles, expected convergence grows only linearly.

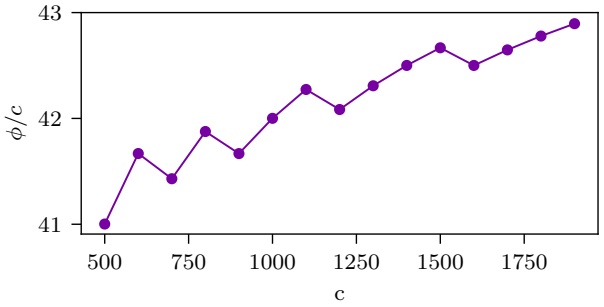

Figure 9: Necessary parameter ϕ/c required in order to achieve ϵ-convergence in the Slush protocol, i.e. every node has reached the same color whp. We fix $k = 10$, $\alpha = 0.8$.

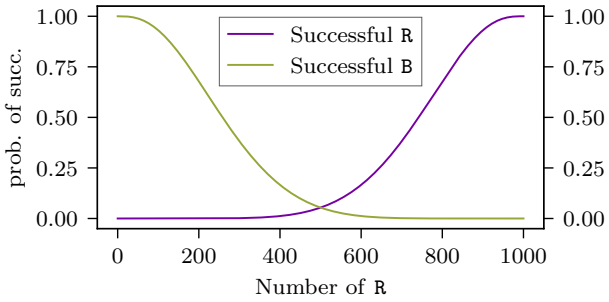

Figure 10: Probability of successful samples vs. network splits.

Finally, to demonstrate R3, we need to compute the first timestep of the Markov chain M that yields a probability of $\leq \epsilon$ for all transient states. This computation has no simple closed-form approximation, therefore we must iteratively raise the transition probability matrix M until the first timestep t that achieves probability $\leq \epsilon$ for all states s_i where $s_{\alpha k} < s_i < s_{c-\alpha k}$. Figure 9 shows the expected time to reach $s_{c-\alpha k}$ whp, over various network sizes, with fixed $k = 10, \alpha = 0.8$.

The next two additional results provide some useful intuition to the underlying stochastic processes of Slush and subsequent protocols. The probability of a single successful query as a function of the total number of R-supporting nodes in the network is shown in Figure 11. This probability decreases rapidly as the value of k increases. The network topples over faster if the value of k is smaller due to the larger amount of random query perturbance. Indeed, for Slush, $k = 1$ is optimal. Later we will see that for the Byzantine case, we will need a larger value for k.

Figure 10 shows the divergence of the probabilities of successful samples over various network splits. When the network is evenly split, the probabilities are equal for either color, but very rapidly diverge as the network becomes less balanced.

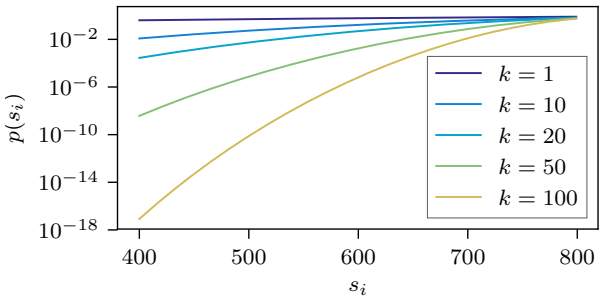

Figure 11: Probability of a single successful majority-red sample with varying k, $\alpha = 0.8$, and successes ranging from half the network to full-support.

3.2 Safety Analysis of Snowflake

Snowflake, whose pseudocode is shown in Figure 12, is similar to Slush, but has three key differences. First, the sampled set of nodes includes Byzantine nodes. Second, each node also keeps track of the total number of consecutive times it has sampled a majority of the same color. We also introduce a function called $\mathcal{A}$, the adversarial strategy, that takes as parameters the entire space of correct nodes as well as the chosen node u at time t, and as a side-effect, modifies the set of nodes $\mathcal{B}$ to some arbitrary configuration of colors.

The birth-death chain that models Snowflake, shown below, is nearly identical to that of Slush, with the added presence of Byzantine nodes.

$$p(s_i) = \frac{c-i}{c} P(\mathcal{H}^k_{\mathcal{C},i} \geq \alpha k),$$
$$q(s_i) = \frac{i}{c} P(\mathcal{H}^k_{\mathcal{C},c-i+b} \geq \alpha k)$$

In Slush, the metastable state of the network was a 50/50 split at Markov state $s_{c/2}$. In Snowflake, Byzantine nodes have the ability to compensate for deviations from $s_{c/2}$ up to a threshold proportional to their size in the network. Let the Markov chain state $s_{ps} > s_{c/2}$ be the state after which the probability of entering the R-absorbing state becomes greater than the probability of entering the B-absorbing state given the entire weight of the Byzantine nodes. Symmetrically, there exists a phase shift point for B as well. All states between these two phase-shift points are Byzantine controlled, whereas all other states are metastable.

In this section, we analyze the two conditions that ensure the safety of Snowflake. The first condition (C1) requires demonstrating that that there exists a point of no return $s_{ps+\Delta}$ after which the maximum probability under full Byzantine activity of bringing the system back to the phase-shift point is less than ϵ.

The second condition (C2) requires the construction of a mechanism to ensure that a process can only commit to a color if the system is beyond the minimal point of no return, whp. As we show later, satisfying these two

```
1:  initialize ∀u, u.col,
2:  initialize ∀u, u.cnt := 0
3:  for t = 1 to φ do
4:      u := SAMPLE(C, 1)
5:      A(u, C)
6:      K := SAMPLE(N\u, k)
7:      P := [v.col  for v ∈ K]
8:      for col' ∈ {R, B} do
9:          if P.COUNT(col') ≥ α · k then
10:             if col' ≠ u.col then
11:                 u.col := col', u.cnt := 0
12:             else
13:                 ++u.cnt
```

Figure 12: Snowflake run by a global scheduler.

conditions is necessary and sufficient to guarantee safety of Snowflake whp.

Properties of Sampling Parameter k. Before constructing and analyzing the two conditions that satisfy the safety of Snowflake, we introduce some important observations of the sample size k. The first observation of interest is that the hypergeometric distribution approximates the binomial distribution for large-enough network sizes. If x is a constant fraction of network size n, then:

$$\lim_{n\to\infty} P(\mathcal{H}^k_{\mathcal{N},x} \geq \alpha k) = \sum_{j=\alpha k}^{k} \binom{k}{j} (x/n)^k (1-(x/n))^{k-j} \tag{8}$$

This dictates that for large enough network sizes, the probability of success for any single sample becomes a function dependent only on the values of k and α, unaffected by further increases of network size. The result has consequences for the scalability of Snowflake and subsequent protocols. For sufficiently large and fixed k, the security of the system will be approximately equal for all network sizes $n > k$.

We now examine how the phase shift point behaves with increasing k. Whereas $k = 1$ is optimal for Slush (i.e. the network topples the fastest when $k = 1$), we see that Snowflake's safety is maximally displaced from its ideal position for $k = 1$, reducing the number of feasible solutions. Luckily, small increases in k have an exponential effect on s_{ps}, creating a larger set of feasible solutions. More formally:

Lemma 3. *s_{ps} approaches $s_{c/2+b/2}$ exponentially fast as k approaches n.*

Proof. At s_{ps}, the ratio $p(s_i)/q(s_i)$ is equal to 1, by definition. Substituting the solution $i = c/2 + b/2$ into this equation yields the result at $k = n$. Computing the differential in s_{ps} between any chosen k and $k+1$ yields the exponential trend. □

Figure 13 shows the exponential relationship between decreasing k and s_{ps}. The graph shows that small-but-not-tiny values of k lead to large feasible regions.

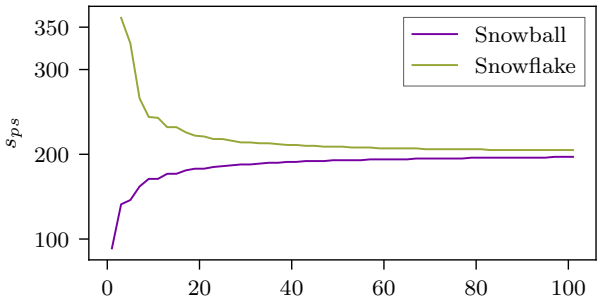

Figure 13: Phase shift state of Snowflake/Snowball vs. k values (x-axis), with $n = 2000$, $\alpha = 0.8$ and $b = 20\%$ of n. The phase-shift is indexed starting from $c/2$.

Satisfiability of Safety Criteria Let ϵ be a system security parameter of value $\leq 2^{-32}$, typically referred to as *negligible* probability.

We now formalize and analyze the safety condition C1. We refer to all states s_i that satisfy condition C1 as *states of feasible solutions*. For the system to guarantee safety, there must exist some state $s_{c/2+\Delta} > s_{c/2+b} > s_{ps}$ that implies ϵ-irreversibility in the birth-death chain. In other words, once the system reaches this state, then it reverts back with only negligible probability, under *any* Byzantine strategy. Figure 14 summarizes the region of interest. Our goal is to determine values for tunable system parameters to achieve this condition, given target network size n, maximum Byzantine component b, and security parameter ϵ, fixed by the system designer.

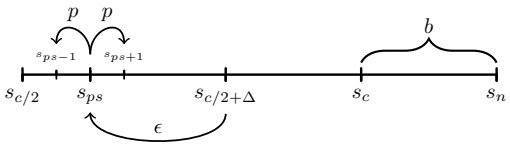

Figure 14: Representation of the space of feasible solutions. Outside the phase-shift point, the network builds a bias towards one end. At the Δ point, this bias is ϵ-irreversible.

To ensure C1, we must find Δ such that the following condition holds:

$$\exists \Delta, \text{where } s_{c/2+\Delta} > s_{ps},\ s.t.\ \forall t \leq \phi : \\ \epsilon \geq (V_c[s_{c/2+\Delta}, s_{ps}])^t \tag{9}$$

where V_c is the $(c+1)^2$ probability transition matrix for the system. In probability theory terminology, the statement above computes the t-step hitting probability, where the starting state is $s_{c/2+\Delta}$ and the hitting state is s_{ps}. Satisfying this first constraint, that is, moving the ensemble of nodes to a decision, is not sufficient. The nodes have to be able to assure *themselves* that they can safely decide, which means they must ensure condition C2. While there are multiple ways of constructing the predicate that ensures this condition, the one we adopt in Snowflake involves β consecutive successful queries, such that $P(\text{isAccepted} \mid s_i \leq s_{c/2+\Delta}) \leq \epsilon$. To achieve this threshold probability of failure for C2, we solve for the smallest β that meets inequality $P(\mathcal{G}(\mathtt{p}, \phi/c) \geq \beta) \leq \epsilon$ where $\mathtt{p} = P(\mathcal{H}^k_{\mathcal{N},i} \geq \alpha k)$ and $\mathcal{G}(\mathtt{p}, \phi/c)$ is a random variable which counts the largest number of consecutive same-color successes for a run over ϕ/c trials. Solving for the minimum i yields β.

The task of the system designer, finally, is to derive values for α and β, given desired n, b, k, ϵ. Choice of α immediately follows from b and is $(n-b)/n$, the maximum allowable size to tolerate $\leq b$ Byzantine nodes. Solving for Δ with a closed form expression would be ideal but unfortunately is difficult, hence we employ a numerical, iterative search. Since k and β are mutually dependent, a system designer will typically fix one and search for the other. Because β is a tunable parameter whose consideration is primarily latency vs. security, a designer can fix it at a desired value and search for a corresponding k. Since low k are desirable, we perform this by starting at $k = 1$, and successively evaluating larger values of k until a feasible solution is found. Depending on b and the chosen ϵ, such a solution may not exist. If so, this will be apparent during system design.

We finalize our analysis by formally composing C1 and C2:

Theorem 4. *If C1 and C2 are satisfied under appropriately chosen system parameters, then the probability that two nodes u and v decide on* R *and* B *respectively is strictly $< \epsilon$ over all timesteps $0 \leq t \leq \phi$.*

Proof. The proof follows in a straightforward manner from the core guarantees that C1 and C2 provide. Without loss of generality, suppose a single node u decides R at time $t < \phi$, and suppose that the network is at state s_j at the time of decision. By construction, isAccepted will return true with probability no greater than ϵ for all network states $s_i < s_{c/2+\Delta}$. Therefore, since u decided, it must be the case that $s_j > s_{c/2+\Delta}$, with high probability. Lastly, since the system will only revert back to a state with a majority of B nodes with negligible probability once it is past $s_{c/2+\Delta}$, the probability that a node v decides on B is strictly $< \epsilon$. □

For illustration purposes, we examine a network with $n = 2000$, $\alpha = 0.8$, $\epsilon \leq 2^{-32}$, and throughput of 10000tps. If we allow this system to run for ~4,000 years (which corresponds to $\phi = 10^{15}$), we choose β such that the probability of a node committing at state

$s_{c/2+\Delta-1}$ is $< \epsilon$. A safety failure then occurs when both a node commits R and the system reverts to B. Figure 15 shows the probability of failure and the corresponding β for different choices of k.

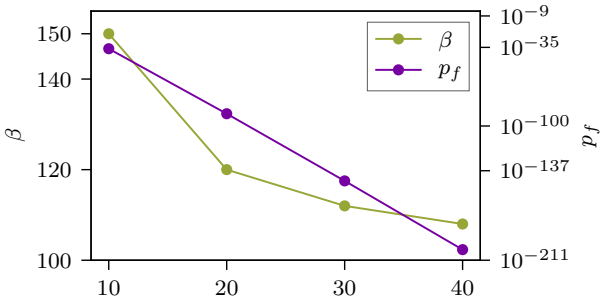

Figure 15: β and failure probability p_f vs. k values (x-axis), with $n = 2000$, $\alpha = 0.8$.

Note that a larger choice of the α parameter dictates a larger space of feasible solutions, which implies a larger tolerance of Byzantine presence. If the system designer chooses to allow a larger presence of Byzantine nodes, she may achieve this goal at the cost of liveness.

3.3 Safety Analysis of Snowball

In this section, we will demonstrate that Snowball provides strictly better safety than Snowflake. The key difference in protocol mechanics between Snowflake and Snowball is that correct nodes now monotonically increase their confidence in R and B. This limits the effect of random perturbations, as well as the impact of the Byzantine adversary.

To intuitively understand the behavioral differences between Snowball and its predecessor, we illustrate with an example, accompanied by some concrete numerical values. Suppose a network of size n where $b = (1/5)n$, $\alpha = (4/5)k$ (i.e. the maximum allowable threshold to tolerate the $(1/5)n$ presence of Byzantine nodes), and where k is chosen large enough to admit a feasible solution. Suppose that the network swings to state $s_{c/2+\Delta}$, such that correct nodes have a non-negligible probability of deciding on R, and suppose, for the sake of simplicity, that Δ is chosen as the highest possible value, specifically $c/2 + \Delta + b = \alpha n = (4/5)n$, the upper-threshold set up by the system designer.

In this configuration, at least $(4/5)n - b = (3/5)n$ (correct) nodes prefer R. These nodes perceive network support for R of at most $(4/5)n$ if the Byzantine nodes choose to also vote for R, and at least $(3/5)n$ if the Byzantine nodes choose to withhold support for R. Conversely, the rest of the $(1/5)n$ (correct) nodes that may prefer B perceive network support for B of at least $(1/5)n$ and at most $(2/5)n$, depending on the Byzantine strategy. Regardless of Byzantine strategy, however, all correct nodes that prefer R will have an expected growth of the R confidence to be strictly greater than the B confidence for the nodes that prefer B. This is because network support for R is at least $(3/5)n$, which is greater than the maximum network support for B of $(2/5)n$. Therefore, once the network swings to the minimum required state, the expected growth of the R-confidence will be greater than that of B-confidence, for all correct nodes and for all Byzantine strategies.

In the previous section, we provided the two conditions necessary for guaranteeing the safety of Snowflake. The second condition, C2, remains identical in Snowball, and therefore does not need to be modified and re-analyzed. Conversely, condition C1 (i.e. showing that the network is irreversible whp) needs to be re-analyzed. In this section, using the same intuition provided in the example above, we formally show how Snowball achieves irreversibility under strictly higher probability than in Snowflake.

Security of Snowball. Figure 16 illustrates the scheduler-based version of Snowball. In Snowball, a node u *prefers* R if $u.d[\mathtt{R}] > u.d[\mathtt{B}]$, and vice versa. At the start of the protocol, these preferences are initialized to $\langle 0, 0 \rangle$. This is in contrast to Snowflake, where nodes prefer a color based only on the latest successful sample. We can model the protocol with a Markov chain similar to that of Snowflake, and derive the parameters and feasibility for the protocol.

```
1:  initialize ∀u, u.col
2:  initialize ∀u, u.lastcol
3:  initialize ∀u, u.cnt := 0
4:  initialize ∀u, ∀col' ∈ {R, B}, u.d[col'] := 0
5:  for t = 1 to φ do
6:      u := SAMPLE(C, 1)
7:      A(u, C)
8:      K := SAMPLE(N\u, k)
9:      P := [v.col  for v ∈ K]
10:     for c' ∈ {R, B} do
11:         if P.COUNT(col') ≥ α · k then
12:             u.d[col']++
13:             if u.d[col'] > u.d[col] then
14:                 u.col := col'
15:             if col' ≠ u.lastcol then
16:                 u.lastcol := col', u.cnt := 0
17:             else
18:                 ++u.cnt
```

Figure 16: Snowball run by a global scheduler.

We fix a Δ such that, under the Markovian construction of Snowflake, if the system reaches $s_{c/2+\Delta}$, then whp it does not revert. We show that this choice of Δ provides an even stronger guarantee once the network switches protocol to Snowball. Without loss of generality, we cluster the correct nodes into two groups, and represent each group through two nodes u and v, where u represents

the set of correct nodes that prefer red and v those that prefer blue. Lastly, let the state of the system be initialized at $s_{c/2+\Delta}$.

The best possible confidence-configuration that the Byzantine nodes can attempt to force correct nodes into is where all R-preferring nodes, represented by u, are forced to maintain nearly equal confidence between the two colors, and where all of the B-preferring nodes, represented by v, gain as much confidence as possible for B.

Let $u.d[\mathtt{R}] = u.d[\mathtt{B}] + 1$, corresponding to the minimal viable color difference for the red-preferring nodes, and let $v.d[\mathtt{B}] - v.d[\mathtt{R}] = \kappa$ be the difference for the blue-preferring nodes. While $\kappa \geq 0$, then the expectation of preference growths at time t are:

$$\begin{aligned}
\mathbb{E}[u.d^t[\mathtt{R}]] &= \mathbb{E}[u.d^{t-1}[\mathtt{R}]] + \left(\frac{1}{2} + \frac{\Delta}{c}\right) P(\mathcal{H}^k_{\mathcal{N},c/2+\Delta+b} \geq \alpha k) \\
\mathbb{E}[u.d^t[\mathtt{B}]] &= \mathbb{E}[u.d^{t-1}[\mathtt{B}]] + \left(\frac{1}{2} + \frac{\Delta}{c}\right) P(\mathcal{H}^k_{\mathcal{N},c/2-\Delta+b} \geq \alpha k) \\
\mathbb{E}[v.d^t[\mathtt{R}]] &= \mathbb{E}[v.d^{t-1}[\mathtt{R}]] + \left(\frac{1}{2} - \frac{\Delta}{c}\right) P(\mathcal{H}^k_{\mathcal{N},c/2+\Delta} \geq \alpha k) \\
\mathbb{E}[v.d^t[\mathtt{B}]] &= \mathbb{E}[v.d^{t-1}[\mathtt{B}]] + \left(\frac{1}{2} - \frac{\Delta}{c}\right) P(\mathcal{H}^k_{\mathcal{N},c/2-\Delta+b} \geq \alpha k)
\end{aligned} \tag{10}$$

Note that we are implicitly using an indicator function when adding up the probability of success to the expected confidence growth, where the expected value of the indicator value is exactly the probability of success of a sample. Substituting for Hoeffding's tail bound, we get the rate at which κ decreases per time-step to be

$$\begin{aligned}
r = &\left(\left(\frac{1}{2} - \frac{\Delta}{c}\right) e^{-k\alpha \log\left(\alpha / \frac{c/2-\Delta+b}{c+b}\right)} \right. \\
&\qquad \left. e^{-k(1-\alpha)\log\left((1-\alpha)/\left(1-\frac{c/2-\Delta+b}{c+b}\right)\right)}\right) \\
-&\left(\left(\frac{1}{2} - \frac{\Delta}{c}\right) e^{-k\alpha \log\left(\alpha / \frac{c/2+\Delta+b}{c+b}\right)} \right. \\
&\qquad \left. e^{-k(1-\alpha)\log\left((1-\alpha)/\left(1-\frac{c/2+\Delta+b}{c+b}\right)\right)}\right)
\end{aligned} \tag{11}$$

Upon reaching $\kappa = -1$, the blue-preferring nodes will have flipped to red. From then on, all correct nodes are red, and the confidence of red grows significantly faster than that of blue. We now show that the rate of growth of R will always be larger than the rate of growth of B, over all correct nodes.

Letting $X(u.d^t[\mathtt{R}])$ be a random variable that outputs the confidence of red for u at timestep t, and letting $\bar{X}(u.d^t[\mathtt{R}]) = X(u.d^1[\mathtt{R}]) + \cdots + X(u.d^t[\mathtt{R}])$, we apply Hoeffding's concentration inequality to get that $P(\bar{X}(u.d^t[\mathtt{R}]) - \mathbb{E}[\bar{X}(u.d^t[\mathtt{R}])] \geq l) \leq e^{-2l^2 t}$. At time t, the expected confidence of red for u is

$$u.d^0[\mathtt{R}] + t\left(\frac{1}{2} + \frac{\Delta}{c}\right)\left(\frac{\frac{c}{2} + \Delta + b}{c+b}\right) \tag{12}$$

And conversely the expected confidence of blue for u is

$$u.d^0[\mathtt{B}] + t\left(\frac{1}{2} + \frac{\Delta}{c}\right)\left(\frac{\frac{c}{2} - \Delta + b}{c+b}\right) \tag{13}$$

Then, the probability that the expected value of red confidence is close to the expected value of the blue confidence is

$$e^{-2t^3(1/2+\Delta/c)^2(2\Delta/(c+b))^2} \tag{14}$$

Solving for t that leads to such probability being negligible leads to:

$$t = \left(\frac{\log(\epsilon)}{-2 \cdot (1/2 + \Delta/c)^2 (2\Delta/(c+b))^2}\right)^{1/3} \tag{15}$$

Therefore, at only a small number of timesteps t, u has a red-confidence centered around its mean, with probability ϵ close to the mean of the blue-confidence. An identical analysis follows for v. In other words, after a small number of time-steps, the red-confidence of u grows large enough such that the probability of this value being close to that of the blue-confidence is negligible. Further timesteps amplify this distance and further decrease the probability of the two confidences being close. We conclude our result with the following Theorem:

Theorem 5. *Over all viable network parameters n, b, and for all appropriately chosen system parameters k and α, the probability of violating safety in Snowball is strictly less than the probability of violating safety in Snowflake.*

Proof. Under the construction of Snowball in comparison to Snowflake, we see that safety criterion C2 remains the same. In other words, the consecutive-successes decision predicate has the same guarantees. On the other hand, the probabilistic guarantees of C1 change, meaning that the reversibility probabilities of the system are different. However, as we determined in Equation 15, over a few timesteps u's confidence difference between R and B grows large enough to guarantee that this confidence difference will revert back with only negligible probability. As time progresses, the probability of being close decreases poly-exponentially, as shown in Equation 14. The same results follow for v, but inversely, meaning that the means get closer to each other rather than deviate. This continues until v flips color. □

Snowball has strictly stronger security guarantees than Snowflake, which implies that appropriately chosen parameters chosen for Snowflake automatically apply for Snowball. Using the same techniques as before, the system designer chooses appropriate k and β values that ensure the desired ϵ safety guarantees.

3.4 Safety Analysis of Avalanche

The key difference between Avalanche and Snowball is that in Avalanche, queries on the DAG on transaction T_i are used to implicitly query the entire ancestry of T_i. In particular, a transaction T_i is preferred by u *if and only if* all ancestors are also preferred. Suppose that T_i and T_j are in the same conflict set. We can now infer two things. First, we can consider the entire ancestry of T_i and T_j as a single decision instance of Snowball, where the ancestry of T_i can be considered to be the R decision, and the ancestry of T_j can be considered to be the B decision. Second, since T_i must be preferred if a child of T_i is to be preferred, then we can collapse the progeny of T_i into a single urn which repeatedly adds an R color to the confidence whenever a child of T_i gets a chit. Consequently, Avalanche maps to an instance of Snowball, with the previously shown safety properties.

We note, however, since a decision on a virtuous transaction is dependent on its parents, Avalanche's liveness guarantees do not mirror that of Snowball. We address this in the next two sub-sections.

3.5 Safe Early Commitment

As we reasoned previously, each conflict set in Avalanche can be viewed as an instance of Snowball, where each progeny instance iteratively votes for the entire path of the ancestry. This feature provides various benefits; however, it also can lead to some virtuous transactions that depend on a rogue transaction to suffer the fate of the latter. In particular, rogue transactions can interject in-between virtuous transactions and reduce the ability of the virtuous transactions to ever reach the required isAccepted predicate. As a thought experiment, suppose that a transaction T_i names a set of parent transactions that are all decided, as per local view. If T_i is sampled over a large enough set of successful queries without discovering any conflicts, then, since by assumption the entire ancestry of T_i is decided, it must imply that at least $c/2 + \Delta$ correct nodes also vote for T_i, achieving irreversibility.

To then statistically measure the assuredness that T_i has been accepted by a large percentage of correct nodes without any conflicts, we make use of a one-way birth process, where a birth occurs when a new correct node discovers the conflict of T_i. Necessarily, deaths cannot exist in this model, because a conflicting transaction cannot be unseen once a correct node discovers it. Let $t = 0$ be the time when T_j, which conflicts with T_i, is introduced to a single correct node u. Let s_x, for $x = 1$ to c, be the state where the number of correct nodes that know about T_j is x, and let $p(s_x)$ be the probability of birth at state s_x. Then, we have:

$$p(s_x) = \frac{c - x}{c} \left(1 - \frac{\binom{n-x}{k}}{\binom{n}{k}} \right) \tag{16}$$

Solving for the expected time to reach the final birth state provides a lower bound to the β_1 parameter in the isAccepted fast-decision branch. The table below shows an example of the analysis for $n = 2000$, $\alpha = 0.8$, and various k, where $\epsilon \ll 10^{-9}$, and where β is the minimum required size of $d(T_i)$. Overall, a very small number of iterations are sufficient for the safe early commitment predicate.

k	10	20	30	40
β	10.87625	10.50125	10.37625	10.25125

3.6 Liveness

Slush. Slush is a non-BFT protocol, and we have demonstrated that it terminates within a finite number of rounds almost surely.

Snowflake & Snowball. Both protocols make use of a counter to keep track of consecutive majority support. Since the adversary is unable to forge a conflict for a virtuous transaction, initially, all correct nodes will have color red or $\perp$. A Byzantine node cannot respond to any query with any answer other than red since it is unable to forge conflicts and $\perp$ is not allowed by protocol. Therefore, the only misbehavior for the Byzantine node is refusing to answer. Since the correct node will re-sample if the query times out, by expected convergence equation in [2], all correct nodes will terminate with the unanimous red value within a finite number of rounds almost surely.

Avalanche. Avalanche introduces a DAG structure that entangles the fate of unrelated conflict sets, each of which is a single-decree instance. This entanglement embodies a tension: attaching a virtuous transaction to undecided parents helps propel transactions towards a decision, while it puts transactions at risk of suffering liveness failures when parents turn out to be rogue. We can resolve this tension and provide a liveness guarantee with the aid of the following mechanisms:

Eventually good ancestry. Virtuous transactions can be retried by picking new parents, selected from a set that is more likely to be preferred. Ultimately, one can always attach a transaction to decided parents to completely mitigate this risk. A simple technique for parent selection is to select new parents for a virtuous transaction at successively lower heights in the DAG, proceeding towards the genesis vertex. This procedure is guaranteed to terminate with uncontested, decided parents, ensuring that the transaction cannot suffer liveness failure due to rogue transactions.

Sufficient chits. A secondary mechanism is necessary to ensure that virtuous transactions with decided ancestry will receive sufficient chits. To ensure this, correct nodes examine the DAG for virtuous non-nop transactions that lack sufficient progeny and emit nop transactions to help

increase their confidence. A nop transaction has just one parent and no application side-effects, and can be issued by any node. They cannot be abused by Byzantine nodes because, even though nops trigger new queries, they do not automatically grant chits.

With these two mechanisms in place, it is easy to see that, at worst, Avalanche will degenerate into separate instances of Snowball, and thus provide the same liveness guarantee for virtuous transactions.

3.7 Churn and View Updates

Any realistic system needs to accommodate the departure and arrival of nodes. Up to now, we simplified our analysis by assuming a precise knowledge of network membership. We now demonstrate that Avalanche nodes can admit a well-characterized amount of churn, by showing how to pick parameters such that Avalanche nodes can differ in their view of the network and still safely make decisions.

Consider a network whose operation is divided into epochs of length τ, and a view update from epoch t to $t+1$ during which γ nodes join the network and $\bar{\gamma}$ nodes depart. Under our static construction, the state space s of the network had a key parameter Δ^t at time t, induced by c^t, b^t, n^t and the chosen security parameters. This can, at worst, impact the network by adding γ nodes of color B, and remove $\bar{\gamma}$ nodes of color R. At time $t+1$, $n^{t+1} = n^t + \gamma - \bar{\gamma}$, while b^{t+1} and c^{t+1} will be modified by an amount $\leq \gamma - \bar{\gamma}$, and thus induce a new Δ^{t+1} for the chosen security parameters. This new Δ^{t+1} has to be chosen such that $P(s_{c^{t+1}/2+\Delta^{t+1}-\gamma} \rightarrow s_{ps}) \leq \epsilon$, to ensure that the system will converge under the previous pessimal assumptions. The system designer can easily do this by picking an upper bound on $\gamma, \bar{\gamma}$.

The final step in assuring the correctness of a view change is to account for a mix of nodes that straddle the τ boundary. We would like the network to avoid an unsafe state no matter which nodes are using the old and the new views. The easiest way to do this is to determine Δ^t and Δ^{t+1} for desired bounds on $\gamma, \bar{\gamma}$, and then to use the conservative value Δ^{t+1} during epoch t. In essence, this ensures that no commitments are made in state space s^t unless they conservatively fulfill the safety criteria in state space s^{t+1}. As a result, there is no possibility of a node deciding red at time t, the network going through an epoch change and finding itself to the left of the new point of no return Δ^{t+1}.

This approach trades off some of the feasibility space, to add the ability to accommodate $\gamma, \bar{\gamma}$ node churn per epoch. Overall, if τ is in excess of the time required for a decision (on the order of minutes to hours), and nodes are loosely synchronized, they can add or drop up to $\gamma, \bar{\gamma}$ nodes in each epoch using the conservative process described above. We leave the precise method of determining the next view to a subsequent paper, and instead rely on a membership oracle that acts as a sequencer and γ-rate-limiter, using technologies like Fireflies [31].

3.8 Communication Complexity

Since liveness is not guaranteed for rogue transactions, we focus our message complexity analysis solely for the case of virtuous transactions. For the case of virtuous transactions, Snowflake and Snowball are both guaranteed to terminate after $O(kn \log n)$ messages. This follows from the well-known results related to epidemic algorithms [21], and is confirmed by Table 1.

Communication complexity for Avalanche is more subtle. Let the DAG induced by Avalanche have an expected branching factor of p, corresponding to the width of the DAG, and determined by the parent selection algorithm. Given the β decision threshold, a transaction that has just reached the point of decision will have an associated progeny $\mathcal{Y}$. Let m be the expected depth of $\mathcal{Y}$. If we were to let the Avalanche network make progress and then freeze the DAG at a depth y, then it will have roughly py vertices/transactions, of which $p(y-m)$ are decided in expectation. Only pm recent transactions would lack the progeny required for a decision. For each node, each query requires k samples, and therefore the total message cost per transaction is in expectation $(pky)/(p(y-m)) = ky/(y-m)$. Since m is a constant determined by the undecided region of the DAG as the system constantly makes progress, message complexity per node is $O(k)$, while the total complexity is $O(kn)$.

4 Implementation

We have fully ported Bitcoin transactions to Avalanche, to yield a bare-bones payment system. Deploying a full cryptocurrency involves bootstrapping, minting, staking, unstaking, and inflation control. While we have solutions for these issues, their full discussion is beyond the scope of this paper. In this section, we focus on how Avalanche can support the value transfer primitive at the center of cryptocurrencies.

UTXO Transactions. In addition to the DAG structure in Avalanche, a UTXO graph that captures spending dependency is used to realize the ledger for the payment system. To avoid ambiguity, we denote the transactions that encode the data for money transfer *transactions*, while we call the transactions ($T \in \mathcal{T}$) in Avalanche's DAG *vertices*.

Each transaction represents a money transfer that takes several inputs from source accounts and generates several outputs to destinations. As a UTXO-based system that keeps a decentralized ledger, balances are kept by the *unspent outputs* of transactions.

More specifically, a transaction $\texttt{TX}_a$ maintains a list of inputs: $\texttt{In}_{a1}$, $\texttt{In}_{a2}$, $\cdots$. Each input has two fields: the

reference to an unspent transaction output and a spend script. The unspent transaction output uniquely refers to an output of a previously made transaction. The script snippet will be prepended to the script from the referred output forming a complete computation. It could typically be a cryptographic proof of validity, but could also be any Turing-complete computation in general. Each output $\texttt{Out}_{a1}$, $\texttt{Out}_{a2}$, $\cdots$, contains some amount of money and a script which typically contains cryptographic verification that takes the proof from the future input and verifies validity.

In our payment system, there are *addresses* representing different accounts by cryptographic keys. The public key is used as the identity for recipients in the output scripts, while the private key is for creating signatures in the input scripts, spending the available funds. Only the key owner is able spend the unspent output by creating an input with the signature in a new transaction.

Due to the possibility of double spending by the private key owner, cryptocurrencies such as Bitcoin use a blockchain as the linear log to reject the transaction that comes later in the log and tries to spend some output twice. Instead, in our payment system, we use Avalanche to resolve the double-spend conflicts in each conflict set, without maintaining a linear log.

If we could assume each transaction can only have one single input, the initialization of Avalanche would be straightforward. We let each vertex on the underlying DAG be one transaction. The conflict set in Avalanche is the set of transactions that try to spend the same unspent output. The conflict sets are disjoint because each transaction only has one input spending one unspent output, and thus belongs to exactly one set.

Multi-input transactions consume multiple UTXOs, and in Avalanche, may appear in multiple conflict sets. To account for these correctly, we represent *transaction-input* pairs (e.g. $\texttt{In}_{a1}$) as an Avalanche vertex, and use the conjunction of isAccepted for all inputs of a transaction to ensure that no transaction will be accepted unless all its inputs are accepted. Since the acceptance for each pair is meaningful for the payment system only if all pairs from the same transaction are accepted, we can tie the fate of these pairs from the same transaction together by a single, bundled query: the queried node will only answer "yes" if all of the pairs are strongly preferred according to the DAG. This more conservative predicate will not undermine safety because merely introducing transactions that gather no chits will not increase confidence value in the protocol.

Figure 17 demonstrates the actual implementation where the DAG is built at transaction granularity, whereas Figure 18 shows the equivalent logic of the underlying protocol, where vertices are at transaction-input granularity.

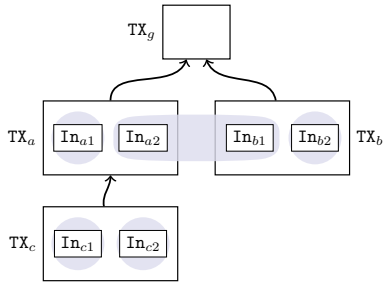

Figure 17: The actual DAG implementation at transition granularity.

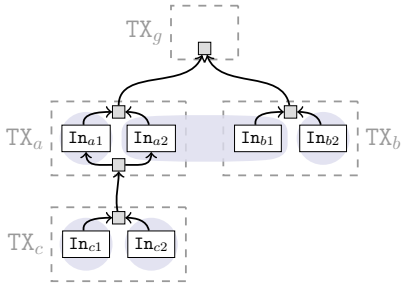

Figure 18: The underlying logical DAG structure used by Avalanche. The tiny squares with shades are dummy vertices which just help form the DAG topology for the purpose of clarity, and can be replaced by direct edges. The rounded gray regions are the conflict sets.

Following this idea, we finally implement the DAG of transaction-input pairs such that multiple transactions can be batched together per query.

Parent Selection. The goal of the parent selection algorithm is to yield a well-structured DAG that maximizes the likelihood that virtuous transactions will be quickly accepted by the network. While this algorithm does not affect the safety of the protocol, it affects liveness and plays a crucial role in determining the shape of the DAG. A good parent selection algorithm grows the DAG in depth with a roughly steady "width." The DAG should not diverge like a tree or converge to a chain, but instead should provide concurrency so nodes can work on multiple fronts.

There are inherent trade-offs in the parent selection algorithm: selecting well-accepted parents makes it more likely for a transaction to find support, but can lead to vote dilution. Further, selecting more recent parents at the frontier of the DAG can lead to stuck transactions, as the parents may turn out to be rogue and remain unsupported. In the following discussion, we illustrate this dilemma. We assume that every transaction will select a small number, p, of parents. We focus on the selection of eligible parent set, from which a subset of size p can be chosen at random.

Perhaps the simplest idea is to mint a fresh transaction with parents picked uniformly at random among those transactions that are currently strongly preferred. Specif-

ically, we can adopt the predicate used in the voting rule to determine eligible parents on which a node would vote positively, as follows:

$$\mathcal{E} = \{T : \forall T \in \mathcal{T}, \text{isStronglyPreferred}(T)\}.$$

But this strategy will yield large sets of eligible parents, consisting mostly of historical, old transactions. When a node samples the transactions uniformly from $\mathcal{E}$, the resulting DAG will have large, ever-increasing fan-out. Because new transactions will have scarce progenies, the voting process will take a long time to build the required confidence on any given new transaction.

In contrast, efforts to reduce fan-out and control the shape of the DAG by selecting the recent transactions at the decision frontier suffer from another problem. Most recent transactions will have very low confidence, simply because they do not have enough descendants. Further, their conflict sets may not be well-distributed and well-known across the network, leading to a parent attachment under a transaction that will never be supported. This means the best transactions to choose lie somewhere near the frontier, but not too far deep in history.

The adaptive parent selection algorithm chooses parents by starting at the DAG frontier and retreating towards the genesis vertex until finding an eligible parent.

Otherwise, the algorithm tries the parents of the transactions in $\mathcal{E}$, thus increasing the chance of finding more stabilized transactions as it retreats. The retreating search is guaranteed to terminate when it reaches the genesis vertex. Formally, the selected parents in this adaptive selection algorithm is:

1: **function** parentSelection($\mathcal{T}$)
2: $\quad\mathcal{E}' := \{T : |\mathcal{P}_T| = 1 \vee d(T) > 0, \forall T \in \mathcal{E}\}$.
3: $\quad$**return** $\{T : T \in \mathcal{E}' \wedge \forall T' \in \mathcal{T}, T \leftarrow T', T' \notin \mathcal{E}'\}$.

Figure 19: Adaptive parent selection.

Optimizations We implement some optimizations to help the system scale. First, we use *lazy updates* to the DAG, because the recursive definition for confidence may otherwise require a costly DAG traversal. We maintain the current $d(T)$ value for each active vertex on the DAG, and update it only when a descendant vertex gets a chit. Since the search path can be pruned at accepted vertices, the cost for an update is constant if the rejected vertices have limited number of descendants and the undecided region of the DAG stays at constant size. Second, the conflict set could be very large in practice, because a rogue client can generate a large volume of conflicting transactions. Instead of keeping a container data structure for each conflict set, we create a mapping from each UTXO to the preferred transaction that stands as the representative for the entire conflict set. This enables a node to quickly determine future conflicts, and the appropriate response to queries. Finally, we speed up the query process by terminating early as soon as the αk threshold is met, without waiting for k responses.

5 Evaluation

We have fully implemented the proposed payment system in around 5K lines of C++ code. In this section, we examine its throughput, scalability, and latency through a large scale deployment on Amazon AWS, and provide a comparison to known results from other systems.

5.1 Setup

We conduct our experiments on Amazon EC2 by running from hundreds to thousands of virtual machine instances. We use `c5.large` instances, which provide two virtual CPU cores per instance that accommodate two processes, each of which simulates an individual node. AWS provides bandwidth of up to 2 Gbps, though the Avalanche protocol utilizes at most 36 Mbps.

Our implementation uses the transaction data format, contract interpretation, and secp256k1 signature code directly from Bitcoin 0.16. We simulate a constant flow of new transactions from users by creating separate client processes, each of which maintains separated wallets, generates transactions with new recipient addresses and sends the requests to Avalanche nodes. We use several such client processes to max out the capacity of our system. The number of recipients for each transaction is tuned to achieve average transaction sizes of around 600 bytes (2–3 inputs/outputs per transaction on average and a stable UTXO size), the current average transaction size of Bitcoin. To utilize the network efficiently, we batch up to 10 transactions during a query, but maintain confidence values at individual transaction granularity.

All reported metrics reflect end-to-end measurements taken from the perspective of all clients. That is, clients examine the total number of confirmed transactions per second for throughput, and, for each transaction, subtract the initiation timestamp from the confirmation timestamp for latency. Each throughput experiment is repeated for 5 times and standard deviation is indicated in each figure. Because we saturate the capacity of the system in all runs, some transactions will have much higher latency than most, we use the 1.5×IQR rule commonly used to filter out outliers. Take the geo-replicated experiment as an example, 2.5% data are the outliers having 13 second latency on average. They fall out of 1.5×IQR (approximately 3σ) range and are filtered out. There are very few outliers when the system is not saturated. All reported latencies (including maximum) are those not filtered. As for security parameters, we pick $k = 10$, $\alpha = 0.8$, $\beta_1 = 11$, $\beta_2 = 150$, which yields an MTTF of ~10^{24} years.

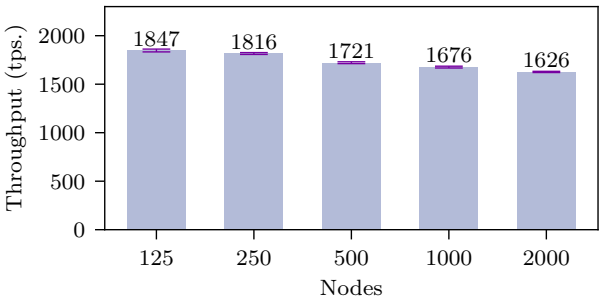

Figure 20: Throughput vs. network size. The x-axis contains the number of nodes and is logarithmic, while the y-axis is confirmed transactions per second.

5.2 Throughput

We first measure the throughput of the system by saturating it with transactions and examining the rate at which transactions are confirmed in the steady state. For this experiment, we first run Avalanche on 125 nodes (63 VMs) with 10 client processes, each of which maintains 300 outstanding transactions at any given time.

As shown in the first bar of Figure 20, the system achieves above 1800 transactions per second (tps). As a comparison, the crypto and transaction code we use from Bitcoin 0.16 can only generate 2.4K tps, and verify 12.3K tps, on a single core.

5.3 Scalability

To test whether the system is scalable in terms of the number of nodes participating in Avalanche consensus, we run the system with identical settings and vary the number of nodes from 125 up to 2000.

Figure 20 shows that overall throughput degrades about 10% to 1626 tps when the network grows by a factor of 16 to $n = 2000$. The degradation is caused by the increased time to gossip transactions. Note that the x-axis is logarithmic, and thus throughput degradation is sublinear.

Maintaining a partial order that just captures the spending relations allows for more concurrency in processing than a classic BFT log replication system where all transactions have to be linearized. Also, the lack of a leader naturally avoids bottlenecks.

5.4 Latency

The latency of a transaction is the time spent from the moment of its submission until it is confirmed as accepted. Figure 21 demonstrates the latency distribution histogram using the same setup as for the throughput measurements with 2000 nodes. The x-axis is the time in seconds while the y-axis is the percentage of transactions that are finalized within the corresponding time period.

This experiment shows that all transactions are confirmed within approximately 1 second. Figure 21 also outlines the cumulative distribution function by accumulating the number of finalized transaction over time. The most common latencies are around 620 ms and variance is low, indicating that nodes converge on the final value as a group around the same time. The vertical line shows the maximum latency we have observed, which is around 1.1 seconds.

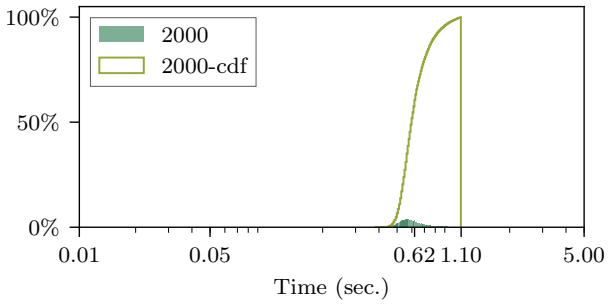

Figure 21: Transaction latency histogram for $n = 2000$. The x-axis is the transaction latency in log-scaled seconds, while the y-axis is the percentage of transactions that fall into the confirmation time. The solid histogram bars show the distribution of all transactions, along with a curve showing the CDF of transaction latency.

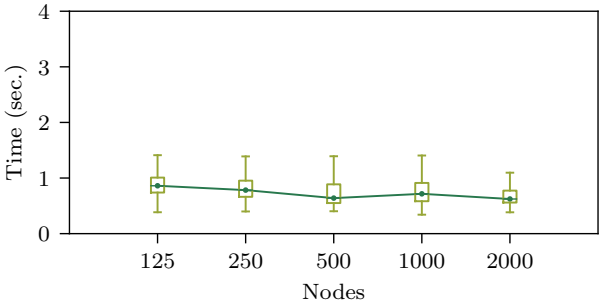

Figure 22: Transaction latency vs. network size.

Figure 22 shows transaction latencies for different numbers of nodes. The horizontal edges of boxes represent minimum, first quartile, median, third quartile and maximum latency respectively, from bottom to top. Crucially, the experimental data show that median latency is more-or-less independent of network size.

5.5 Misbehaving Clients

We next examine how rogue transactions issued by misbehaving clients that double spend unspent outputs can affect the latency for virtuous transactions created by other

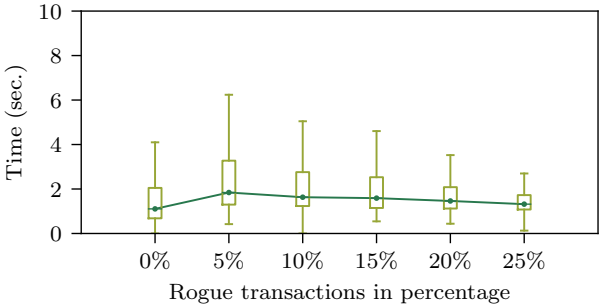

Figure 23: Latency vs. ratio of rogue transactions.

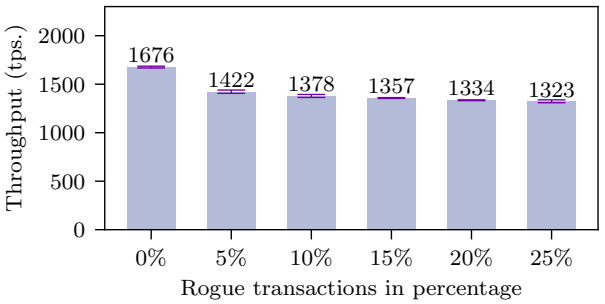

Figure 24: Throughput vs. ratio of rogue transactions.

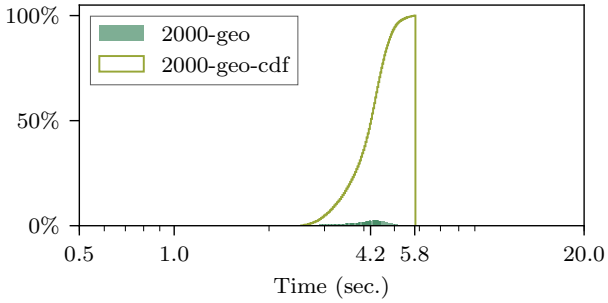

Figure 25: Latency histogram for $n = 2000$ in 20 cities.

honest clients. We adopt a strategy to simulate misbehaving clients where a fraction (from 0% to 25%) of the pending transactions conflict with some existing ones. The client processes achieve this by designating some double spending transaction flows among all simulated pending transactions and sending the conflicting transactions to different nodes. We use the same setup with $n = 1000$ as in the previous experiments, and only measure throughput and latency of confirmed transactions.

Avalanche's latency is only slightly affected by misbehaving clients, as shown in Figure 23. Surprisingly, latencies drop a bit when the percentage of rogue transactions increases. This behavior occurs because, with the introduction of rogue transactions, the overall *effective* throughput is reduced and thus alleviates system load. This is confirmed by Figure 24, which shows that throughput (of virtuous transactions) decreases with the ratio of rogue transactions. Further, the reduction in throughput appears proportional to the number of misbehaving clients, that is, there is no leverage provided to the attackers.

5.6 Geo-replication

We also evaluated the payment system in an emulated geo-replicated scenario with significantly higher latencies than in prior measurements. We selected 20 major cities that appear to be near substantial numbers of reachable Bitcoin nodes, according to [10]. The cities cover North America, Europe, West Asia, East Asia, Oceania, and also cover the top 10 countries with the highest number of reachable nodes. We use the latency and jittering matrix crawled from [53] and emulate network packet latency in the Linux kernel using `tc` and `netem`. 2000 nodes are distributed evenly to each city, with no additional network latency emulated between nodes within the same city. We assign a client process to each city, maintaining 300 outstanding transactions per city at any moment.

Our measurements show an average throughput of 1312 tps, with a standard deviation of 5 tps. As shown in Figure 25, the median transaction latency is 4.2 seconds, with a maximum latency of 5.8 seconds.

5.7 Batching

Batching is a critical optimization that can improve throughput by amortizing consensus overhead over greater numbers of useful transactions. Avalanche uses batching by default, to carry out 10 transactions per query event, that is, per vertex on the DAG.

To test the performance gain of batching, we performed an experiment where batching is disabled. Surprisingly, the batched throughput is only $2\times$ as large as the unbatched case, and increasing the batch size further does not increase throughput.

The reason for this is that the implementation is bottlenecked by transaction verification. Our current implementation uses an event-driven model to handle a large number of concurrent messages from the network. After commenting out the `verify()` function in our code, the throughput rises to 8K tps, showing that either contract interpretation or cryptographic operations involved in the verification pose the main bottleneck to the system. Removing this bottleneck by offloading transaction verification to a GPU is possible. Event without GPU optimization, 1312 tps is far in excess of extant blockchains.

5.8 Comparison to Other Systems

The parameters for our experiments were chosen to be comparable to Algorand [26], and we use Bitcoin [40] as a baseline.

Algorand uses a verifiable random function to elect committees, and maintains a totally-ordered log while Avalanche establishes only a partial order. Algorand is leader-based and performs consensus by committee, while Avalanche is leader-less. Both evaluations use a decision network of size 2000 on EC2. Our evaluation uses `c5.large` with 2 vCPU, 2 Gbps network per VM, while Algorand uses `m4.2xlarge` with 8 vCPU, 1 Gbps network per VM. The CPUs are approximately the same speed, and our system is not bottlenecked by the network, making comparison possible. The security parameters chosen in our experiments guarantee a safety violation probability below 10^{-9} in the presence of 20% Byzantine nodes, while Algorand's evaluation guarantees a violation probability below 5×10^{-9} with 20% Byzantine nodes.

The throughput is 3-7 tps for Bitcoin, 364 tps for Algorand (with 10 Mbyte blocks), and 159 tps (with 2 Mbyte

blocks). In contrast, Avalanche achieves over 1300 tps consistently on up to 2000 nodes. As for latency, finality is 10–60 minutes for Bitcoin, around 50 seconds for Algorand with 10 Mbyte blocks and 22 seconds with 2 Mbyte blocks, and 4.2 seconds for Avalanche.

6 Related Work

Bitcoin [40] is a cryptocurrency that uses a blockchain based on proof-of-work (PoW) to maintain a ledger of UTXO transactions. While techniques based on proof-of-work [6, 23], and even cryptocurrencies with minting based on proof-of-work [45, 52], have been explored before, Bitcoin was the first to incorporate PoW into its consensus process. Unlike more traditional BFT protocols, Bitcoin has a probabilistic safety guarantee and assumes honest majority computational power rather than a known membership, which in turn has enabled an internet-scale permissionless protocol. While permissionless and resilient to adversaries, Bitcoin suffers from low throughput (~3 tps) and high latency (~5.6 hours for a network with 20% Byzantine presence and 2^{-32} security guarantee). Furthermore, PoW requires a substantial amount of computational power that is consumed only for the purpose of maintaining safety.

Countless cryptocurrencies use PoW [6, 23] to maintain a distributed ledger. Like Bitcoin, they suffer from inherent scalability bottlenecks. Several proposals for protocols exist that try to better utilize the effort made by PoW. Bitcoin-NG [24] and the permissionless version of Thunderella [43] use Nakamoto-like consensus to elect a leader that dictates writing of the replicated log for a relatively long time so as to provide higher throughput. Moreover, Thunderella provides an optimistic bound that, with 3/4 honest computational power and an honest elected leader, allows transactions to be confirmed rapidly. ByzCoin [34] periodically selects a small set of participants and then runs a PBFT-like protocol within the selected nodes. It achieves a throughput of 393 tps with about 35 second latency.

Another category of blockchain protocols proposes to get rid of PoW and replace it with *Proof-of-Stake* (PoS). PoS eliminates the computational cost of PoW by requiring a node to put its money on hold in exchange for participation in consensus. Consensus protocols based on PoS replace the honest majority computational power by honest majority stake values. Snow White [19] and Ouroboros [33] are some of the earliest provably secure PoS protocols. Ouroboros uses a secure multiparty coin-flipping protocol to produce randomness for leader election. The follow-up protocol, Ouroboros Praos [20] provides safety in the presence of fully adaptive adversaries.

Protocols based on Byzantine agreement [36, 44] typically make use of quorums and require precise knowledge of membership. PBFT [14], a well-known representative, requires a quadratic number of message exchanges in order to reach agreement. The Q/U protocol [3] and HQ replication [17] use a quorum-based approach to optimize for contention-free cases of operation to achieve consensus in only a single round of communication. However, although these protocols improve on performance, they degrade very poorly under contention. Zyzzyva [35] couples BFT with speculative execution to improve the failure-free operation case. Aliph [28] introduces a protocol with optimized performance under several, rather than just one, cases of execution. In contrast, Ardvark [16] sacrifices some performance to tolerate worst-case degradation, providing a more uniform execution profile. This work, in particular, sacrifices failure-free optimizations to provide consistent throughput even at high number of failures. Past work in permissioned BFT systems typically requires at least $2b + 1$ replicas tolerance. However, CheapBFT [32] showed how to leverage trusted hardware components to construct a protocol that uses $b + 1$ replicas.

Other work attempts to introduce new protocols under redefinitions and relaxations of the BFT model. Large-scale BFT [46] modifies PBFT to allow for arbitrary choice of number of replicas and failure threshold, providing a probabilistic guarantee of liveness for some failure ratio but protecting safety with high probability. In another form of relaxation. Zeno [48] introduces a BFT state machine replication protocol that trades consistency for high availability. More specifically, this paper guarantees eventual consistency rather than linearizability, meaning that participants can be inconsistent but eventually agree once the network stabilizes. By providing an even weaker consistency guarantee, namely fork-join-causal consistency, Depot [37] describes a protocol that guarantees safety under $b + 1$ replicas.

NOW [27] is, to our best knowledge, the first to use the idea of sub-quorums to drive smaller instances of consensus. The insight of this paper is that small, logarithmic-sized quorums can be extracted from a potentially large set of nodes in the network, allowing smaller instances of consensus protocols to be run in parallel.

HoneyBadger [39] provides good liveness in a network with heterogeneous latencies and achieves over 341 tps with 5 minute latency on 104 nodes. Tendermint [11] rotates the leader for each block and has been demonstrated with as many as 64 nodes. Ripple [47] has low latency by utilizing collectively-trusted sub-networks in a large network. The Ripple company provides a slow-changing default list of trusted nodes, which renders the system essentially centralized. In the synchronous and authenticated setting, the protocol in [4] achieves constant-3-round commit in expectation, at the cost of quadratic message complexity.

Stellar [38] uses Federated Byzantine Agreement in

which *quorum slices* enable heterogeneous trust for different nodes. Safety is guaranteed when transactions can be transitively connected by trusted quorum slices.

Algorand [26] uses a verifiable random function to select a committee of nodes that participate in a novel Byzantine consensus protocol. It achieves over 360 tps with 50 second latency on an emulated network of 2000 committee nodes (500K users in total) distributed among 20 cities. To prevent Sybil attacks, it uses a mechanism like proof-of-stake that assigns weights to participants in committee selection based on the money in their accounts.

Some protocols use a Directed Acyclic Graph (DAG) structure instead of a linear chain to achieve consensus. Instead of choosing the longest chain as in Bitcoin, GHOST [50] uses a more efficient chain selection rule that allows transactions not on the main chain to be taken into consideration, increasing efficiency. SPECTRE [49] uses transactions on the DAG to vote recursively with PoW to achieve consensus, followed up by PHANTOM [51] that achieves a linear order among all blocks. Avalanche is different in that the voting result is a one-time chit that is determined by a query, while the votes in PHANTOM are purely determined by transaction structure. Similar to Thunderella, Meshcash [9] combines a slow PoW-based protocol with a fast consensus protocol that allows a high block rate regardless of network latency, offering fast confirmation time. Hashgraph [7] is a leader-less protocol that builds a DAG via randomized gossip. It requires full membership knowledge at all times, and, similar to the Ben-Or [8], it suffers from at least exponential message complexity [5, 13].

7 Conclusion

This paper introduced a new family of leaderless, metastable, and PoW-free BFT protocols. These protocols do not incur quadratic message cost and can work without precise membership knowledge. They are lightweight, quiescent, and provide a strong safety guarantee, though they achieve these properties by not guaranteeing liveness for conflicting transactions. We have illustrated how they can be used to implement a Bitcoin-like payment system, that achieves 1300tps in a geo-replicated setting.

References

[1] Crypto-currency market capitalizations. `https://coinmarketcap.com`. Accessed: 2017-02.

[2] Snowflake to Avalanche: A metastable protocol family for cryptocurrencies. `https://www.dropbox.com/s/t5h2weaws1vo3c7/paper.pdf`, 2018.

[3] M. Abd-El-Malek, G. R. Ganger, G. R. Goodson, M. K. Reiter, and J. J. Wylie. Fault-scalable byzantine fault-tolerant services. In *ACM SIGOPS Operating Systems Review*, volume 39, pages 59–74. ACM, 2005.

[4] I. Abraham, S. Devadas, D. Dolev, K. Nayak, and L. Ren. Efficient synchronous byzantine consensus. 2017.

[5] J. Aspnes. Randomized protocols for asynchronous consensus. *Distributed Computing*, 16(2-3):165–175, 2003.

[6] J. Aspnes, C. Jackson, and A. Krishnamurthy. Exposing computationally-challenged byzantine impostors. 2005.

[7] L. Baird. Hashgraph consensus: fair, fast, byzantine fault tolerance. Technical report, Swirlds Tech Report, 2016.

[8] M. Ben-Or. Another advantage of free choice (extended abstract): Completely asynchronous agreement protocols. In *Proceedings of the second annual ACM symposium on Principles of distributed computing*, pages 27–30. ACM, 1983.

[9] I. Bentov, P. Hubácek, T. Moran, and A. Nadler. Tortoise and Hares Consensus: the Meshcash framework for incentive-compatible, scalable cryptocurrencies. *IACR Cryptology ePrint Archive*, 2017:300, 2017.

[10] Bitnodes. Global Bitcoin nodes distribution. `https://bitnodes.earn.com/`. Accessed: 2018-04.

[11] E. Buchman. *Tendermint: Byzantine fault tolerance in the age of blockchains*. PhD thesis, 2016.

[12] M. Burrows. The chubby lock service for loosely-coupled distributed systems. In *7th Symposium on Operating Systems Design and Implementation (OSDI'06), November 6-8, Seattle, WA, USA*, pages 335–350, 2006.

[13] C. Cachin and M. Vukolic. Blockchain consensus protocols in the wild. *CoRR*, abs/1707.01873, 2017.

[14] M. Castro and B. Liskov. Practical byzantine fault tolerance. In *Proceedings of the Third USENIX Symposium on Operating Systems Design and Implementation (OSDI), New Orleans, Louisiana, USA, February 22-25, 1999*, pages 173–186, 1999.

[15] Central Intelligence Agency. The world factbook. `https://www.cia.gov/library/`

publications/the-world-factbook/geos/da.html. Accessed: 2018-04.

[16] A. Clement, E. L. Wong, L. Alvisi, M. Dahlin, and M. Marchetti. Making byzantine fault tolerant systems tolerate byzantine faults. In *Proceedings of the 6th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2009, April 22-24, 2009, Boston, MA, USA*, pages 153–168, 2009.

[17] J. Cowling, D. Myers, B. Liskov, R. Rodrigues, and L. Shrira. Hq replication: A hybrid quorum protocol for byzantine fault tolerance. In *Proceedings of the 7th symposium on Operating systems design and implementation*, pages 177–190. USENIX Association, 2006.

[18] K. Croman, C. Decker, I. Eyal, A. E. Gencer, A. Juels, A. E. Kosba, A. Miller, P. Saxena, E. Shi, E. G. Sirer, D. Song, and R. Wattenhofer. On scaling decentralized blockchains - (a position paper). In *Financial Cryptography and Data Security - FC 2016 International Workshops, BITCOIN, VOTING, and WAHC, Christ Church, Barbados, February 26, 2016, Revised Selected Papers*, pages 106–125, 2016.

[19] P. Daian, R. Pass, and E. Shi. Snow white: Provably secure proofs of stake. Cryptology ePrint Archive, Report 2016/919, 2016. https://eprint.iacr.org/2016/919.

[20] B. David, P. Gazi, A. Kiayias, and A. Russell. Ouroboros Praos: An adaptively-secure, semi-synchronous proof-of-stake blockchain. In *Advances in Cryptology - EUROCRYPT 2018 - 37th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Tel Aviv, Israel, April 29 - May 3, 2018 Proceedings, Part II*, pages 66–98, 2018.

[21] A. Demers, D. Greene, C. Hauser, W. Irish, J. Larson, S. Shenker, H. Sturgis, D. Swinehart, and D. Terry. Epidemic algorithms for replicated database maintenance. In *Proceedings of the sixth annual ACM Symposium on Principles of distributed computing*, pages 1–12. ACM, 1987.

[22] Digiconomist. Bitcoin energy consumption index. https://digiconomist.net/bitcoin-energy-consumption. Accessed: 2018-04.

[23] C. Dwork and M. Naor. Pricing via processing or combatting junk mail. In *Advances in Cryptology - CRYPTO '92, 12th Annual International Cryptology Conference, Santa Barbara, California, USA, August 16-20, 1992, Proceedings*, pages 139–147, 1992.

[24] I. Eyal, A. E. Gencer, E. G. Sirer, and R. van Renesse. Bitcoin-ng: A scalable blockchain protocol. In *13th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2016, Santa Clara, CA, USA, March 16-18, 2016*, pages 45–59, 2016.

[25] J. A. Garay, A. Kiayias, and N. Leonardos. The Bitcoin Backbone Protocol: Analysis and applications. In *Advances in Cryptology - EUROCRYPT 2015 - 34th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Sofia, Bulgaria, April 26-30, 2015, Proceedings, Part II*, pages 281–310, 2015.

[26] Y. Gilad, R. Hemo, S. Micali, G. Vlachos, and N. Zeldovich. Algorand: Scaling byzantine agreements for cryptocurrencies. In *Proceedings of the 26th Symposium on Operating Systems Principles, Shanghai, China, October 28-31, 2017*, pages 51–68, 2017.

[27] R. Guerraoui, F. Huc, and A.-M. Kermarrec. Highly dynamic distributed computing with byzantine failures. In *Proceedings of the 2013 ACM symposium on Principles of distributed computing*, pages 176–183. ACM, 2013.

[28] R. Guerraoui, N. Knežević, V. Quéma, and M. Vukolić. The next 700 bft protocols. In *Proceedings of the 5th European conference on Computer systems*, pages 363–376. ACM, 2010.

[29] W. Hoeffding. Probability inequalities for sums of bounded random variables. *Journal of the American statistical association*, 58(301):13–30, 1963.

[30] P. Hunt, M. Konar, F. P. Junqueira, and B. Reed. Zookeeper: Wait-free coordination for internet-scale systems. In *2010 USENIX Annual Technical Conference, Boston, MA, USA, June 23-25, 2010*, 2010.

[31] H. D. Johansen, R. van Renesse, Y. Vigfusson, and D. Johansen. Fireflies: A secure and scalable membership and gossip service. *ACM Trans. Comput. Syst.*, 33(2):5:1–5:32, 2015.

[32] R. Kapitza, J. Behl, C. Cachin, T. Distler, S. Kuhnle, S. V. Mohammadi, W. Schröder-Preikschat, and K. Stengel. Cheapbft: resource-efficient byzantine fault tolerance. In *Proceedings of the 7th ACM european conference on Computer Systems*, pages 295–308. ACM, 2012.

[33] A. Kiayias, A. Russell, B. David, and R. Oliynykov. Ouroboros: A provably secure proof-of-stake

blockchain protocol. In *Advances in Cryptology - CRYPTO 2017 - 37th Annual International Cryptology Conference, Santa Barbara, CA, USA, August 20-24, 2017, Proceedings, Part I*, pages 357–388, 2017.

[34] E. Kokoris-Kogias, P. Jovanovic, N. Gailly, I. Khoffi, L. Gasser, and B. Ford. Enhancing Bitcoin security and performance with strong consistency via collective signing. In *25th USENIX Security Symposium, USENIX Security 16, Austin, TX, USA, August 10-12, 2016.*, pages 279–296, 2016.

[35] R. Kotla, L. Alvisi, M. Dahlin, A. Clement, and E. L. Wong. Zyzzyva: Speculative byzantine fault tolerance. *ACM Trans. Comput. Syst.*, 27(4):7:1–7:39, 2009.

[36] L. Lamport, R. E. Shostak, and M. C. Pease. The byzantine generals problem. *ACM Trans. Program. Lang. Syst.*, 4(3):382–401, 1982.

[37] P. Mahajan, S. Setty, S. Lee, A. Clement, L. Alvisi, M. Dahlin, and M. Walfish. Depot: Cloud storage with minimal trust. *ACM Transactions on Computer Systems (TOCS)*, 29(4):12, 2011.

[38] D. Mazieres. The Stellar consensus protocol: A federated model for internet-level consensus. *Stellar Development Foundation*, 2015.

[39] A. Miller, Y. Xia, K. Croman, E. Shi, and D. Song. The Honey Badger of BFT protocols. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, Vienna, Austria, October 24-28, 2016*, pages 31–42, 2016.

[40] S. Nakamoto. Bitcoin: A peer-to-peer electronic cash system, 2008.

[41] R. Pass, L. Seeman, and A. Shelat. Analysis of the blockchain protocol in asynchronous networks. In *Advances in Cryptology - EUROCRYPT 2017 - 36th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Paris, France, April 30 - May 4, 2017, Proceedings, Part II*, pages 643–673, 2017.

[42] R. Pass and E. Shi. Fruitchains: A fair blockchain. *IACR Cryptology ePrint Archive*, 2016:916, 2016.

[43] R. Pass and E. Shi. Thunderella: Blockchains with optimistic instant confirmation. In *Advances in Cryptology - EUROCRYPT 2018 - 37th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Tel Aviv, Israel, April 29 - May 3, 2018 Proceedings, Part II*, pages 3–33, 2018.

[44] M. C. Pease, R. E. Shostak, and L. Lamport. Reaching agreement in the presence of faults. *J. ACM*, 27(2):228–234, 1980.

[45] R. Rivest and A. Shamir. Payword and micromint: Two simple micropayment schemes. In *Security protocols*, pages 69–87. Springer, 1997.

[46] R. Rodrigues, P. Kouznetsov, and B. Bhattacharjee. Large-scale byzantine fault tolerance: Safe but not always live.

[47] D. Schwartz, N. Youngs, A. Britto, et al. The Ripple protocol consensus algorithm. *Ripple Labs Inc White Paper*, 5, 2014.

[48] A. Singh, P. Fonseca, P. Kuznetsov, R. Rodrigues, P. Maniatis, et al. Zeno: Eventually consistent byzantine-fault tolerance.

[49] Y. Sompolinsky, Y. Lewenberg, and A. Zohar. SPECTRE: A fast and scalable cryptocurrency protocol. *IACR Cryptology ePrint Archive*, 2016:1159, 2016.

[50] Y. Sompolinsky and A. Zohar. Secure high-rate transaction processing in Bitcoin. In *Financial Cryptography and Data Security - 19th International Conference, FC 2015, San Juan, Puerto Rico, January 26-30, 2015, Revised Selected Papers*, pages 507–527, 2015.

[51] Y. Sompolinsky and A. Zohar. PHANTOM: A scalable blockdag protocol. *IACR Cryptology ePrint Archive*, 2018:104, 2018.

[52] V. Vishnumurthy, S. Chandrakumar, and E. G. Sirer. Karma: A secure economic framework for peer-to-peer resource sharing.

[53] WonderNetwork. Global ping statistics: Ping times between wondernetwork servers. `https://wondernetwork.com/pings`. Accessed: 2018-04.