Vault: Fast Bootstrapping for Cryptocurrencies

Derek Leung, Adam Suhl, Yossi Gilad, Nickolai Zeldovich
MIT CSAIL
{dtl, asuhl, yossigi, nickolai}@mit.edu

***Abstract*—Decentralized cryptocurrencies rely on participants to keep track of the state of the system in order to verify new transactions. As the number of users and transactions grows, this requirement places a significant burden on the users, as they need to download, verify, and store a large amount of data in order to participate.**

Vault is a new cryptocurrency designed to minimize these storage and bootstrapping costs for participants. Vault builds on Algorand's proof-of-stake consensus protocol and uses several techniques to achieve its goals. First, Vault decouples the storage of recent transactions from the storage of account balances, which enables Vault to delete old account state. Second, Vault allows sharding state across participants in a way that preserves strong security guarantees. Finally, Vault introduces the notion of stamping certificates that allow a new client to catch up securely and efficiently in a proof-of-stake system without having to verify every single block.

Experiments with a prototype implementation of Vault's data structures shows that Vault reduces the bandwidth cost of joining the network as a full client by 99.7% compared to Bitcoin and 90.5% compared to Ethereum when downloading a ledger containing 500 million transactions.

I. Introduction

Cryptocurrencies are a promising approach for decentralized electronic payments, smart contracts, and other applications. However, supporting a large number of users and transactions will require cryptocurrencies to address two crucial and related bottlenecks: *storage* (how much data every participant must store) and *bootstrapping* (how much data every participant has to download to join the system). For example, in Bitcoin [1], a new client that wishes to join the network and verify that it received the correct state must download about 150 GB of data, as of January 2018 [2]. Storage and bootstrapping costs are related because, in a decentralized design, existing nodes must store enough state to help new nodes join the system.

Designing a cryptocurrency whose storage and bootstrapping costs scale well with the number of users and transactions is difficult due to several challenges.

First, a cryptocurrency must prevent double-spending—that is, prevent a user from spending the same money twice or issuing the same transaction multiple times. This is typically done by keeping track of past transactions, but doing so is incompatible with good scalability. For instance, Bitcoin stores all past transactions, which does not scale well (costs grow linearly with the number of transactions). As another example, Ethereum [3] does not store all transactions, but instead keeps track of the sequence number ("nonce") of the last transaction issued from a given account [4]. This nonce must be stored even if the account has no remaining balance. As a result, this does not scale well either (costs grow linearly with the number of old accounts) and has caused problems for Ethereum, when a smart contract inadvertently created many zero-balance accounts [5], [6]. We measure the Ethereum ledger (§VII) and find that 38% of Ethereum accounts have a balance of zero.

Second, a cryptocurrency relies on all participants to check the validity of transactions. This requires the participants to have enough state to validate those transactions. Storing all account balances allows a participant to validate any transaction, but requires storage space that grows with the number of accounts. On the other hand, not storing all account balances runs the risk of having fewer participants vet transactions.

Third, proof-of-stake systems, such as Algorand [7], can provide high throughput and low latency for transactions. However, such proof-of-stake systems are particularly challenging in terms of bootstrapping cost. Convincing a new participant of the validity of a block in the blockchain requires first convincing them of the balances (stakes) of all users in an earlier block. Convincing a new user of the validity of the latest block thus requires convincing them of the balances of all users at all points in time, starting with the initial genesis block.

Finally, an appealing way to reduce storage and bootstrapping costs is to delegate the job of storing state and certifying future states to a committee whose participants are trusted in aggregate. However, existing systems that take this approach [8], [9], [10] rely on *long-standing* committees that are known to the adversary. As a result, an adversary may be able to corrupt the committee members, leading to security or availability attacks.

This paper presents Vault, a new cryptocurrency that addresses the storage and bootstrapping bottlenecks described above. In particular, Vault reduces the bandwidth cost of joining the network as a full client by 99.7% compared to Bitcoin and 90.5% compared to Ethereum when downloading a ledger containing 500 million transactions. Vault borrows the underlying proof-of-stake consensus protocol from Algorand and addresses the above challenges of storage and bootstrapping costs using several techniques:

First, Vault *decouples* the tracking of account balances from the tracking of double-spent transactions. Each Vault transaction is valid for a bounded window of time, expressed in terms of the position in the blockchain where the transaction can appear. This allows Vault nodes to keep track of just the transactions that appeared in recent blocks and to forget about all older transactions. The account balance state, on the other hand, is not directly tied to past transactions, and zero-balance

accounts can be safely evicted.

Second, Vault uses an *adaptive sharding scheme* that combines three properties: (1) it allows sharding the account state across nodes, so that each node does not need to store the state of all accounts; (2) it allows all transactions to be validated by all nodes, using a Merkle tree to store the balance information; and (3) it adaptively caches upper layers of the Merkle tree so that the bandwidth cost of transferring Merkle proofs grows gradually with the number of accounts.

Finally, Vault introduces *stamping certificates* to reduce the cost of convincing new users of a block's validity. The insight lies in trading off the liveness parameter used in selecting a committee to construct the certificate of a new block [7], [9].[1] Vault augments existing certificates in Algorand with its stamping certificates, which have a much lower probability of liveness (e.g., in many cases, Vault fails to find enough participants to construct a valid certificate) but requires fewer participants to form the certificate (thus significantly reducing their size) while still preserving the same safety guarantees (i.e., an adversary still has a negligible probability of corrupting the system). Building an extra layer of stamping certificates allows us to relax liveness for stamping without affecting the liveness of transaction confirmation. Vault's stamping certificates are also generated in a way that allows new clients to skip over many blocks in one verification step.

We implemented a prototype of Vault and used it to evaluate its design and individual techniques. The results show that Vault's storage and bootstrapping cost is 477 MB for 500 million transactions, compared to 5 GB for Ethereum and 143 GB for Bitcoin. Individual microbenchmarks also demonstrate that each of Vault's techniques are important in achieving its performance goals.

The contributions of this paper are:

- The design of Vault, a cryptocurrency that reduces storage and bootstrapping costs by 10.5–301× compared to Bitcoin and Ethereum and that allows sharding without weakening security guarantees.
- Techniques for reducing storage costs in a cryptocurrency, including the decoupling of account balances from double-spending detection and the adaptive sharding scheme.
- The stamping certificate technique for reducing bootstrapping costs in a proof-of-stake cryptocurrency.
- An evaluation of Vault's design that demonstrates its low storage and bootstrapping costs, as well as the importance of individual techniques.

The rest of this paper is organized as follows. We describe related work in §II. §III gives an overview of Vault's design and operation. The next three sections cover Vault's techniques in detail: §IV describes how Vault decouples account state from recent transactions, §V describes Vault's adaptive sharding, and §VI describes Vault's stamping certificates. §VII evaluates Vault's design and techniques, and §VIII concludes.

[1] Vault avoids the use of long-standing committees by using Algorand's cryptographic sortition and player-replaceable consensus.

II. Related Work

Vault's goal is to reduce the cost of storage and bootstrapping in a cryptocurrency. There are two significant aspects to this goal, corresponding to two broad classes of prior work.

The first is what we call the "width" of the ledger: how much data does each participant need to store in order to validate transactions (including detecting double-spending)? In the case of Bitcoin, for example, the "width" is the set of all past unspent transactions [1]. Techniques that address the width of a ledger focus on managing the substantial storage costs of keeping the history of all transactions on each client.

The second is what we call the "length" of the ledger: how much data has to be transmitted to a new participant as proof of the current state of the ledger? In Bitcoin's case, the proof consists of all block headers starting from the genesis block, chained together by hashes in the block headers, as well as all of the corresponding block contents (to prove which transactions have or have not been spent yet). Techniques addressing the length of the ledger typically allow clients to skip entries when verifying block headers, which reduces the total download cost.

Table I summarizes Vault's characteristics and compares them against other cryptocurrencies. Bitcoin and Ethereum fail to provide any formal guarantees on the correctness of the latest state. Permissioned cryptocurrencies like Stellar have low bootstrapping cost but are vulnerable to an adversary which compromises a quorum of permissioned nodes at any point. A system combining OmniLedger and Chainiac lacks single points of failure, but even then an adversary may adaptively compromise a selected committee. Algorand provides strong security guarantees, but its bootstrapping costs grow prohibitively quickly. Vault alone achieves cryptographic security against an adversary that can adaptively compromise users while scaling in both storage and bootstrapping costs.

A. Steady-State Savings: The "Width" Approach

Many cryptocurrencies observe that the transaction log becomes impractical to store and to transmit over time. They seek to reduce the size of this log, which both reduces the amount of bandwidth needed to join the protocol (as a verifier) and also the amount of storage needed to run the protocol.

Ethereum [3] supports the succinct summarization of account balances and other state into a short digest. In each block, ledger writers use *Patricia Merkle Trees* [11] to commit to the current set of balances. A Merkle Tree [12] allows a party to efficiently produce proofs of an object's membership in some set. These Merkle "checkpoints" allow new clients to obtain balance state from any untrusted node and then quickly verify this state against a known Merkle root. To prevent an attacker from replaying a transaction issued by a user, the users embed a sequence number (called the transaction *nonce*) in each transaction. Ethereum clients must track the last nonce issued by each account in the balance tree, even if the account is empty (i.e., its balance is 0); otherwise, an old transaction could be replayed (e.g., if an empty account receives a deposit in the future). As we note in §IV, this means

TABLE I
A COMPARISON OF VAULT TO OTHER CRYPTOCURRENCIES. UTXO REFERS TO UNSPENT TRANSACTION OUTPUTS; TX REFERS TO TRANSACTIONS.

System	Execution State	Proof Size	Bootstrap Security
Bitcoin [1]	UTXOs	Headers + TXs	Probabilistic (heaviest chain wins)
Ethereum [3]	All accounts	Headers + All accounts	Probabilistic (heaviest chain wins)
Permissioned (e.g., Stellar [8])	$\frac{\text{Live accounts}}{\text{Shards}}$	Majority of trust set's signatures	Cryptographic if majority never compromised; none otherwise
OmniLedger [10] + Chainiac [9]	$\frac{\text{UTXOs}}{\text{Shards}}$	$\frac{\text{Headers+Certificates}}{\text{Sparseness}} + \frac{\text{UTXOs}}{\text{Shards}}$	Cryptographic with static attacker; none with adaptive attacker
Algorand [7]	UTXOs	Headers + Certificates + TXs	Cryptographic
Vault	$\frac{\text{Live accounts}}{\text{Shards}}$	$\frac{\text{Headers+Certificates}}{\text{Sparseness}} + \frac{\text{Live accounts}}{\text{Shards}}$	Cryptographic

that Ethereum's storage cost grows with the number of all accounts that ever existed, which leaves Ethereum vulnerable to denial-of-service attacks that create many temporary accounts. By decoupling account balances from tracking double-spent transactions (§IV), Vault prevents storage costs from growing with the number of old accounts. We believe that Vault's decoupling can be adopted by Ethereum to avoid unbounded storage for old accounts.

OmniLedger [10] *shards* its ledger by users' public keys, running Byzantine agreement rounds on many ledgers in parallel. OmniLedger performs load balancing across each shard to improve throughput and reduce bandwidth and storage costs proportional to the number of shards. Sharding allows OmniLedger to scale horizontally under increased load. However, OmniLedger requires a long-standing committee to run the PBFT [13] protocol to establish consensus on the ledger's state; this leaves it vulnerable to a strong adversary which may quickly corrupt validators. Moreover, its shard size and thus scalability is sensitive to the proportion of malicious users. Vault's adaptive sharding (§V) reduces the storage cost per participant and remains secure against an adversary that can quickly corrupt users, but its throughput per unit of bandwidth cost does not increase with sharding.

An alternative approach to reduce the "width" of the ledger is to issue fewer transactions on the ledger. The Lightning Network [14] establishes payment channels between pairs of users which supports many off-ledger transactions, relying on incentives to prevent them from cheating. Participants in the channel post amounts of their stake as collateral and then exchange transactions off the ledger to record their debts. As a result, by posting only two transactions on the Bitcoin ledger, a pair of participants may process arbitrarily many off-ledger transactions in a payment channel as long as it contains a sufficient amount of *capacity* to absorb them. One advantage of this scheme is that participants do not need to broadcast transactions within a payment channel. However, it remains difficult to generalize this scheme over non-pairwise payment relationships, the amount of collateral that each participant posts limits channel capacity, and its incentive scheme assumes that participants always act to maximize their payout. In Vault, participants store account balances and a set of recent transactions. This storage cost depends on the total number of accounts and not the transaction rate, thus obviating the need for off-ledger transactions as a way of reducing storage cost.

MimbleWimble [15] uses an accumulator-like *signature sinking* scheme to "compact" blocks together according to the amount of work proved in the block header. Combined blocks eliminate transaction outputs which have been spent, reducing the state a verifier is required to download. Switching to a balance-based scheme like Vault's may allow MimbleWimble to further increase its compaction savings by committing not just to the set of unspent transactions but also to the current set of balances.

B. Short Proofs of State: The "Length" Approach

Other cryptocurrencies focus more specifically on the bandwidth costs of bootstrapping. They observe that a small block header is often sufficient evidence of a block's validity. Therefore, they reduce the cost of verifying the block header sequence by shortening it. This allows clients to efficiently prove the validity of their state at any particular point in time.

Like Vault's stamping certificates, Chainiac's [9] Collective Signing (CoSi) [16] scheme allows a committee of verifiers to jointly produce a proof that a particular block is correct. As in Vault, verifying committees for some block also certify the correctness of blocks into the future; upon observing a block confirmation, committees produce forward links to the block. Since these links are arranged in a skiplist-like configuration, they allow verifiers to quickly bootstrap to the current state. However, Chainiac's scheme is inherently vulnerable to an adversary that can adaptively corrupt users because its committees are not secret. Sometime after the protocol designates a committee, an adversary which compromises this committee can forge a proof that a false view of the ledger is valid and thus deceive new clients into accepting a bogus state. Since the committees that produce Vault's certificate signatures are secretly selected and emit exactly one message, Vault's certificates resist attacks from adversaries that can adaptively corrupt clients.

MimbleWimble also reduces the length of the ledger. Blocks with more work supersede prior blocks with less work; since an adversary must possess significant processing power to attack these blocks, the proof of work requirements increase the new verifiers' *confidence* in these blocks. As in Bitcoin, this approach does not produce a *proof* of blocks' correctness, since an adversary that controls the network can prevent a user from ever observing the block with the largest amount of

work. Vault builds on Algorand for reaching consensus, which ensures safety (no forks) even in the presence of network partitions.

We observe that in a permissioned cryptocurrency, where a supermajority of ledger writers are trusted, a signed checkpoint suffices to convince a new verifier that the state is correct [13]. Stellar [8] can be thought of in similar terms, where a core node will accept a checkpoint from nodes in its quorum set. Vault targets a permissionless setting where users do not configure trusted sets of known writers or trusted core nodes. As a result, Vault authenticates checkpoint signatures using cryptographic sortition, based on techniques from Algorand.

III. Overview

Vault is a *permissionless*, *proof-of-stake* cryptocurrency that significantly reduces new client bootstrapping costs relative to the state of the art by reducing both steady-state storage costs and the sizes of proofs needed to verify the latest state.

A. Objectives

Suppose Alice is a new participant in Vault who holds the correct genesis block. She wishes to catch up to the latest state and contacts Bob, an existing participant (or perhaps a set of participants). Vault should achieve the following main goals:

- *Bootstrap Efficiency*: If Bob is honest, he should be able to convince Alice that his state is correct and deliver this state using a minimal amount of bandwidth. Moreover, once Alice synchronizes with the protocol, she should be able to help other new clients catch up.
- *Safety*: If Bob is malicious, he should not be able to convince Alice that any forged protocol state is correct.
- *Storage Efficiency*: Bob must store a small amount of data to execute the Vault protocol correctly and to help Alice join the network.

Our design also confers additional benefits:

- *Charging for Storage*: Adversaries that wish to inflate the size of the protocol state must acquire a significant amount of stake to do so.
- *Availability*: Vault continues to operate even when some users disconnect from the network, despite sharding state across clients.

B. Threat Model

Vault should achieve its goals even in the face of adversarial conditions. However, many properties are unachievable given an arbitrarily strong attacker [17]. We therefore limit the attacker's power with the following assumptions, inherited from Algorand [7] (owing to the fact that Vault builds on Algorand's consensus protocol):

- *Bounded Malicious Stake*: At least some proportion h of the stake in Vault is controlled by honest clients at any time, where $h > \frac{2}{3}$. Stake sold off by any user counts towards this threshold for some period d (e.g., 48 hours) following the sale.
- *Cryptographic Security*: The adversary has high but bounded computation power. In particular, the adversary cannot break standard cryptographic assumptions.
- *Adaptive Corruptions*: The adversary may corrupt a particular user at any time (given that at no point it controls more than $1 - h$ of the stake in Vault).
- *Weak Synchrony*: At all times, the adversary may reschedule any message in a small window of time whose duration is fixed in advance (e.g., lasting 20 seconds) and drop a small number of these messages. In addition, the adversary may introduce *network partitions* lasting for a duration of at most b (e.g., 24 hours). During a network partition, the adversary may arbitrarily reschedule or drop any message. The minimum time between network partitions is nontrivial (e.g., 4 hours).

C. Algorand Background

Vault's consensus protocol is based on Algorand, which we briefly review here. All users' clients in Vault agree on an ordered sequence of signed transactions, and this sequence constitutes the cryptocurrency *ledger*. Vault is a permissionless proof-of-stake system, meaning that any user's client, identified by a cryptographic public key, may join the system, and the client of any user holding any amount of money may eventually be selected to append to the ledger. Honest clients listen for proposed transactions and append recent valid transactions to the ledger.

The frequency at which a user's client is selected is proportional to the user's stake. Ledger writers batch sets of transactions into *blocks*. Each block contains a *block header*, which in turn contains a cryptographic commitment to the transaction set. Block headers also contain the cryptographic hash of the previous block in the ledger. Since block headers are small, these hashes allow clients to quickly verify historical transaction data.

Additionally, block headers contain a special pseudorandom *selection seed* Q. Before a client proposes a block, it computes Q in secret, so Q is unpredictable by the rest of the network and partially resistant to adversarial manipulation. As in Algorand, Vault uses Q to seed *Verifiable Random Functions* (VRFs) [18] to implement *cryptographic sortition*. Cryptographic sortition produces a sample of the users in the system, weighted by the stake of their accounts. Each client's membership in the sample remains unknown to an adversary until the client emits a message because a VRF allows the client to compute this membership privately; since VRFs produce a proof of their correctness, any other client can verify this membership. To protect the system against adversaries which corrupt a user after that user is selected, clients sign their messages with ephemeral keys, which they delete before transmission.

Vault uses a Byzantine agreement scheme which operates in *rounds*. Each round, the protocol selects some block proposer which assembles the transaction set and header forming the block, which is broadcast via a peer-to-peer gossip network [19]. Subsequently, the protocol selects a committee

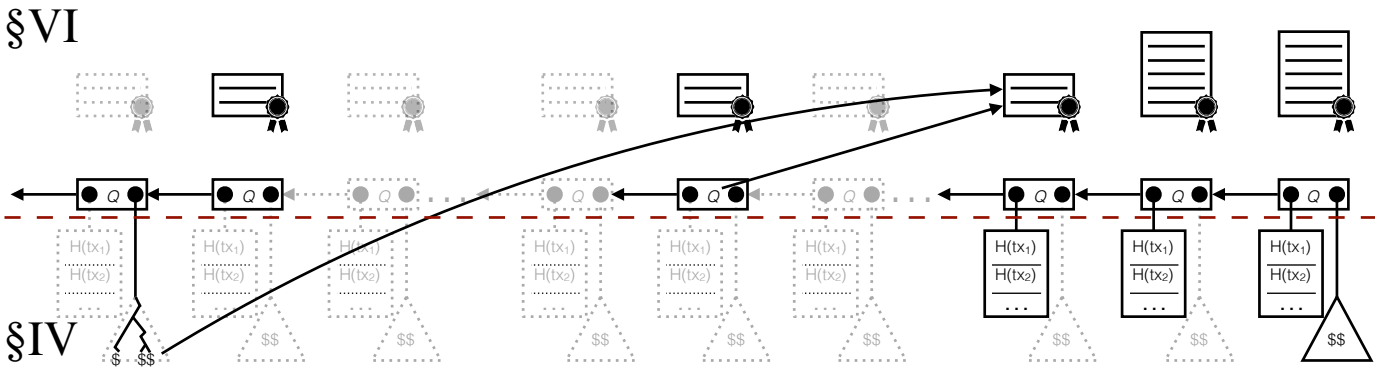

Fig. 1. An overview of the authenticated data structures used in Vault. In this figure, the objects each client stores locally on disk are outlined in solid black, while the objects it may discard are outlined with faint dots. The triangles annotated with "$$" represent the sparse Merkle trees containing account balances, while the bottom row of rectangles annotated with "$H(\mathrm{tx})$" represents the set of transaction hashes in each block. Both the transaction hash set and the balance set are committed to in block headers (the row of rectangles in the middle of the figure); the commitments are represented as solid black dots. In addition, each block header contains Q (i.e., the selection seed), which is computed pseudorandomly and seeded with the previous header's Q-value. The top row of rectangles and seals represent Vault's small stamping certificates and large final certificates; we draw arrows to illustrate how a particular certificate is verified by two block headers. Not shown is Vault's adaptive sharding (§V).

which verifies the correctness of the block. To sample users in a manner resistant to adversarial manipulation, committees from round r are seeded with the value of Q from round $r-1$ and weighted by proofs of stake from round $r-b$.

Once clients become confident of a block's confirmation, Vault uses sortition to select a subset of clients to certify the block by signing its receipt (i.e., Algorand's "final" round). The aggregation of these signatures past some secure threshold, along with proofs of stake for each signature, forms a *final certificate* which proves to any client that a block is valid: the Byzantine agreement protocol guarantees that for each round, at most one valid block (or an empty block if the proposer misbehaves) reliably receives this certificate. Given knowledge of only the *genesis* (i.e., the first) block, a new client is convinced that the block from round n is correct if a peer can produce $n-1$ block headers and the $n-1$ corresponding certificates of validity.

D. System Design

Figure 1 gives an overview of Vault's data structures, which are the key to Vault's lower storage and bootstrapping costs. The data structures are based around a chain of block headers, shown in the middle of the figure. Each block header consists of four elements: PREVBLOCK (the hash of the previous block), Q (the seed for cryptographic sortition), TXROOT (a Merkle tree commitment [12] to the list of transactions in the block), and BALROOT (a sparse Merkle tree commitment [20] to the balances of every user after applying the block's transactions).

Every block must follow certain rules in order to be considered valid:

1) Transactions in the block are not expired. Each transaction includes the first and last block number (in the blockchain) where it can appear.
2) After all transactions in the block are executed, no account ends up with a negative balance.
3) The transactions in the block have not been executed before (i.e., have not appeared previously on the ledger).
4) BALROOT correctly reflects all users' balances after applying the block's transactions to the previous block's balances.

In order to check that a new block follows these rules, clients maintain two pieces of state, shown in solid black (as opposed to grayed out) in the bottom half of Figure 1:

- The tree of account balances from the most recent block. This allows a client to ensure that new transactions have sufficient funds (rule 2), and to verify the correctness of the new balance tree (rule 4).
- The lists of transactions from the last few blocks. This allows a client to ensure that a transaction has not appeared previously (rule 3), by checking that a new transaction does not appear in any of the previous transaction lists. To minimize the storage required by these lists, TXROOT commits to a list of transaction hashes, rather than the transactions themselves.

  Clients can discard transaction lists older than a certain threshold, corresponding to the maximum validity interval of a transaction, which we denote $T_{\max}$. Transactions that appeared more than $T_{\max}$ blocks ago will be rejected by rule 1 and need not be tracked explicitly.

§IV describes in more detail how clients check these rules while using a minimal amount of storage. §V further describes Vault's adaptive sharding, which allows clients to store only a subset of the balance tree. These techniques combine to reduce the "width" of Vault's ledger.

Vault uses Algorand's consensus protocol to decide which valid block will be next in Vault's blockchain. The consensus protocol produces a *certificate* confirming agreement on that block, shown in the top half of Figure 1. These certificates allow a new client to securely join the system and determine which chain of blocks is authentic.

Each certificate consists of a set of signatures (of the block

header) by a committee of clients chosen pseudorandomly using cryptographic sortition. In order to verify a certificate, a new client must check that all of the signatures are valid (which is straightforward) and check that the clients whose signatures appear in the certificate were indeed members of the committee chosen by cryptographic sortition (which requires state). Verifying committee membership requires two pieces of state: the sortition seed Q, used to randomize the selection, and the balance tree at BALROOT, used to weigh clients by how much money their users have.

In Algorand's certificates, BALROOT comes from b blocks ago, while Q comes from the immediately previous block. This means that, in order to verify block n, the client must first verify block $n-1$, so that the client knows the correct Q for verifying block n's certificate. Furthermore, the committees used for Algorand's certificates are relatively large, so that with high probability there are enough committee members to form a certificate for each block. These certificates are shown with a big rectangle at the top of Figure 1.

Vault introduces a second kind of certificate, called a *stamping certificate*, which helps speed up bootstrapping. The stamping certificate differs in two important ways. First, instead of using Q from the immediately previous block, it uses Q from b blocks ago (for security, BALROOT must be chosen from b blocks before Q, so this means BALROOT now comes from $2b$ blocks ago). This allows clients to "vault" forward by b blocks at a time. Second, the stamping certificates use a smaller committee size. This makes the certificate smaller since it contains fewer signatures. The smaller rectangles at the top of Figure 1 represent these stamping certificates, along with the arrows reflecting the Q and BALROOT values needed to verify them.

Vault sets parameters so that the stamping certificate is just as hard for an adversary to forge as Algorand's original certificates. The trade-off, however, is that in some blocks, there may not be enough committee members to form a valid stamping certificate. To help new clients join the system, every Vault client keeps the stamping certificates for approximately every bth block since the start of the blockchain, along with full Algorand-style certificates for the blocks since the last stamping certificate. Other certificates are discarded (shown as grayed out in Figure 1). §VI-B describes Vault's stamping certificates in more detail, which help Vault shrink the "length" of its ledger.

IV. Efficient Double-Spending Detection

This section describes Vault's design for minimizing the amount of storage required by a client to verify new transactions. To understand the challenges in doing so, consider the key problem faced by a cryptocurrency: *double-spending*. Suppose Alice possesses a coin which she gives to both Bob and Charlie. A secure cryptocurrency must reject one of these transactions, as if both are accepted, Alice has double-spent her coin.

In Bitcoin, each transaction has a set of inputs and outputs. The inputs collect money from previous transactions' outputs, which can then be used by this transaction. The outputs define where the money goes (e.g., some may now be spendable by another user, and the rest remains with the same user). To detect double-spending in this scheme, Bitcoin must determine whether some output has been previously spent or not. Thus, clients must store the set of all unspent transaction outputs.

A more space-efficient approach is to store the balance associated with each user, rather than the set of unspent transactions. For example, Ethereum follows this approach. The cost savings from storing just the balances may be significant: for instance, there are ten times as many transactions in Bitcoin as there are addresses [21], [22].

Switching to a balance-based scheme introduces a subtle problem with transaction replay. If Alice sends money to Bob, Bob may attempt to re-execute the same transaction twice. In Bitcoin's design, this would be rejected, because the transaction already spent its inputs. However, in a naïve design that tracked only account balances, this transaction still appears to be valid (as long as Alice still has money in her account), and an adversary may be able to re-execute it many times to drain Alice's account.

To distinguish between otherwise identical transactions, Ethereum tags each account with a *nonce*, which acts as a sequence number. When an account issues a transaction, it tags the transaction with the account's current nonce, and when this transaction is processed, the account increments its nonce. The transactions issued by an account must have sequential nonces. Because of this design, Ethereum cannot delete accounts with zero balance; all clients must track the nonces of old accounts to prevent replay attacks, on the off chance that the account will receive money in the future.

The storage of empty accounts significantly increases the storage overhead of Ethereum. Our analysis of its ledger shows that approximately one-third of all Ethereum addresses have zero balance (§VII). Worse, the inability to garbage-collect old accounts constitutes a serious denial-of-service vulnerability: an adversary with a small amount of money may excessively increase the cryptocurrency's storage footprint by creating many accounts. In fact, in 2016 an Ethereum user inadvertently created many empty accounts (due to a bug in Ethereum's smart contract processing) [5], requiring the Ethereum developers to issue a hard fork to clean up the ledger [6].[2]

At a high level, Vault avoids the problem of storing empty accounts by forcing transactions to expire. The rest of this section describes Vault's solution in more detail.

A. Transaction Expiration

All transactions in Vault contain the fields t_{issuance} and t_{expiry}, which are round numbers delineating the validity of a transaction: blocks older than t_{issuance} or newer than t_{expiry}

[2]Currently, Ethereum transaction fees are high enough to make such attacks unlikely. However, proposed cryptocurrency designs like Algorand [7] aim to support orders of magnitude more throughput, which would lead to lower transaction fees, and which would in turn make such attacks worth considering.

Alice→Bob:	$30
Issuance:	550
Expiry:	574
Nonce:	8

Alice

Fig. 2. The format of a Vault transaction from Alice to Bob. In addition to the sender, receiver, and amount, the transaction contains $t_{\text{issuance}}, t_{\text{expiry}}$, and a nonce. A valid transaction contains the sender's digital signature.

may not contain the transaction. Moreover, we require that $0 \leq t_{\text{expiry}} - t_{\text{issuance}} \leq T_{\max}$ for some constant $T_{\max}$. This way, a verifying client may detect the replaying of a transaction simply by checking for its presence in the last $T_{\max}$ blocks. (Transactions still contain a nonce to distinguish between otherwise identical transactions; however, this nonce is ephemeral and needs not be stored.) As a result, clients do not need to track account nonces and can delete empty accounts from the balance tree. Figure 2 shows the format of one transaction.

Requiring transaction lifetimes to be finite means that, if a transaction fails to enter a block before it expires (e.g., because its transaction fee was lower than the current clearing rate), the issuer must reissue the transaction in order for the transaction to be executed. On the other hand, the expiration time ensures that old transactions that failed to enter a block when they were originally issued cannot be re-entered into a block at a much later time by an adversary; i.e., expiration bounds the outstanding liabilities of an account.

The choice of $T_{\max}$ affects two considerations. The first is that clients must store the last $T_{\max}$ blocks' worth of transactions to detect duplicates; a larger $T_{\max}$ increases client storage. (Clients can store transaction hashes instead of the transactions themselves to reduce this cost.) The second is that clients must reissue any transaction that fails to enter a block within $T_{\max}$ (if they still want to issue that transaction). In our experiments, we set $T_{\max}$ to the expected number of blocks in 4 hours (which, based on Algorand's throughput of ∼750 MB/hour [7], suggests at most a few hundred megabytes of recent transactions); we believe this strikes a balance between the two constraints.

B. Efficient Balance Commitments

To efficiently commit to the large set of balances in BALROOT, Vault clients build Merkle trees [12] over these sets. With a Merkle tree, clients may prove that some object exists in a given set using a witness of size $\mathcal{O}(\log n)$, where n is the total number of objects in the set. This allows them to efficiently construct proofs of stake. For example, for a balance set containing 100 million accounts (4 GB of on-disk storage), it suffices for a client to send 1 KB of data to prove its stake against a 32-byte BALROOT in a block header. It is important for the proofs of stake in Vault to be small since a certificate may contain thousands of these proofs (see §VI).

For clients to verify the validity of BALROOT for a new block, BALROOT must be deterministically constructed given a set of balances. As a result, the Merkle leaves are sorted before they are hashed together to create the root. Since a leaf may be deleted when an account balance reaches 0, Vault uses *sparse* Merkle trees [20] to perform balance commitments. A sparse Merkle tree possesses the structure of a prefix trie, which allows us to perform tree insertions and deletions with a Merkle path witness of size $\mathcal{O}(\log n)$.

In fact, a witness of size $\mathcal{O}(\log n)$ is sufficient for a client to securely update BALROOT *without* storing the corresponding Merkle tree. We exploit this self-verifying property of Merkle witnesses in §V.

V. SHARDING BALANCE STORAGE

As the number of accounts in Vault grows, the cost of storing balances becomes the primary bottleneck. Concretely, each account requires about 40 bytes (32 bytes for a public key and 8 bytes for the balance). This means that if there were 100 million accounts, every Vault client would need to store about 4 GB of data, which may be acceptable (e.g., it is less than Bitcoin's current storage cost). On the other hand, if Vault grew to 10× or 100× more accounts, the storage cost would likely be too high for many clients.

To address this problem, Vault implements sharding to split the tree of balances across clients, pseudorandomly distributed by the client user's key. Sharding allows clients to deal with large ledger sizes. As an extreme example, consider a system with 100 billion accounts and 1 million online clients. Setting the number of shards to 1,000 would require each client to store approximately 4 GB of data (i.e., 1/1000th of the balances), rather than the full 4 TB balance tree. Even with a large number of shards in this example, every shard's data is held by about 1000 clients, ensuring a high degree of availability.

One challenge in sharding is that fewer clients now have the necessary data to verify any given new transaction. Existing proposals (like OmniLedger [10]) implement sharding by restricting verifiers' responsibility of preventing double-spending attacks to their own shards. These proposals seem attractive because they reduce not just storage costs but also bandwidth and latency costs, allowing the system to scale throughput arbitrarily. Unfortunately, such schemes are vulnerable to adversaries which control a fraction of the currency. As shard size decreases, so do shards' replication factors, and as a result, transactions in a given shard are verified by a small number of clients. An adversary may own a critical fraction of the stake in a shard by chance, enabling it to control the entire shard. Thus, these systems require an undesirable trade-off between scaling and security, which in practice limits the degree of sharding.

Vault's design allows sharding without any reduction in security because all clients retain the ability to verify all transactions. The trade-off comes in terms of increased bandwidth costs during normal operation (which may reduce the maximum throughput): as we describe in the rest of this section, with Vault's adaptive sharding, each transaction must

include a partial Merkle proof, which grows proportionally with the size of the balance tree.

A. Secure Shard Witnesses

Vault shards the tree of account balances into 2^N pieces by assigning an account to a shard according to the top N bits of the account's public key. A client stores a shard by storing the balances for the accounts in that shard. Clients store the shard(s) corresponding to their user's public key(s). Clients that join the network, or create a new account, download the corresponding shards from existing peers.

The main challenge is to support sharding without a loss in security by ensuring that all clients can verify all transactions. Recall that the balance set is stored in a sparse Merkle tree (§IV-B) whose root is committed to in the block header. These trees support insertions, updates, and deletions with witnesses of size $\mathcal{O}(\log n)$. To allow a client to check the validity of any transaction in a proposed block (even if that transaction operates on accounts outside of this client's shard), Vault transactions include Merkle witnesses for the source and destination accounts in the previous block's BALROOT. Any client that possesses the previous block's BALROOT can use the Merkle witnesses to confirm the source and destination account balances and thus to verify the transaction.

Unfortunately, these witnesses increase transaction size. For example, if the transaction size is 250 bytes (on par with Bitcoin), and there are 100 billion accounts in the system, a single Merkle witness will hold 37 sibling nodes in expectation, which is 1.2 KB. Two witnesses would introduce 2.4 KB of overhead per transaction—almost an $11\times$ increase. The next subsection describes Vault's approach for mitigating this cost.

Note that the inclusion of Merkle witnesses increases *bandwidth* but not *storage* costs: since all blocks are certified by an honest committee, verifiers discard the witnesses after they recompute BALROOT. Thus, the trade-off applies only to the bandwidth costs of broadcasting transactions during rounds.

B. Adaptive Sharding: Truncating Witnesses

To manage the overhead of larger witnesses, clients store (in addition to their shards) an intermediate frontier that cuts across the Merkle tree—roughly speaking, the subset of tree nodes at some depth. Storing this frontier allows clients to verify *partial* witnesses, which prove the path from a leaf node to the frontier, rather than all the way to BALROOT. Figure 3 illustrates one such partial witness.

We can quantify the trade-off between transaction size and the cost of storing the frontier. First, we observe that moving the frontier up in the tree by one level (i.e., going from the nodes in the frontier to their parents) increases the length of a partial Merkle witness by a single sibling. Second, moving the frontier up in the tree by one level halves its size. Vault can thus tune the trade-off between the size of partial Merkle witnesses in each transaction and the amount of storage required for the frontier.

If the frontier lies in the dense region of the Merkle prefix tree (i.e., towards the top of the tree), the shape of the frontier

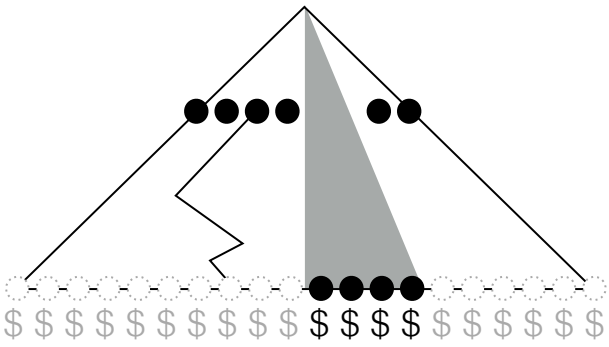

Fig. 3. An illustration of a single Vault shard and the balance Merkle tree. Dots in this image represent Merkle nodes, and the "$" symbols represent account balances. The solid black dots and dark "$" symbols represent the balances which are part of the shard (the shaded gray triangle), while those in gray represent the parts of the tree which are not. The row of black dots in the middle represent the frontier of Merkle nodes that is stored by all clients regardless of shard assignment. The jagged line connecting one of these nodes to an unstored leaf represents the Merkle witness necessary for performing a balance update.

is simple: it involves all the Merkle nodes at a given *level*. However, if the frontier lies near the leaves of the Merkle prefix tree (i.e., near the bottom), a client cannot simply store all the nodes at a given level, as the layers are larger than the balance set itself (owing to the sparseness of the Merkle tree). Instead, these frontiers assume a "jagged" shape; they are defined as the nodes which sit at a fixed *height* from the bottom of the tree.

To update a node in the frontier, it suffices for a client to observe a witness and follow these two rules: (1) if the witness increases the height of a frontier node, the client replaces that frontier node with its children (which were present in the witness); and (2) if the witness decreases the height of a frontier node, the client replaces that frontier node with its parent (if it did not previously store the parent). Note that the length of the witness alone is sufficient for determining whether an insertion, an update, or a deletion occurred.

By application of the coupon collector's problem [23], we see that if there are approximately $n \log n$ account balances, then in expectation the last dense layer is of depth n. For example, if there are 100 billion $\approx 2^{37}$ accounts, then the $n = 32$nd layer is the last dense one.

VI. SUCCINCT LEDGER CERTIFICATES

Bootstrapping a new client in a proof-of-stake cryptocurrency, such as Algorand, requires transferring a significant amount of data to the new client. This is due to two factors. First, the selection of each block depends on the state of the system at the time the block was selected. For instance, as mentioned in §III-C, the Algorand committee that forms the final certificate of a block is selected based on the random seed Q from the previous block. Thus, to verify the correctness of block n, a new client must first verify the correctness of block $n-1$ in order to obtain the correct Q value for verifying block n. Second, in Algorand's design, the certificate confirming a block consists of a large number of signatures, reflecting the

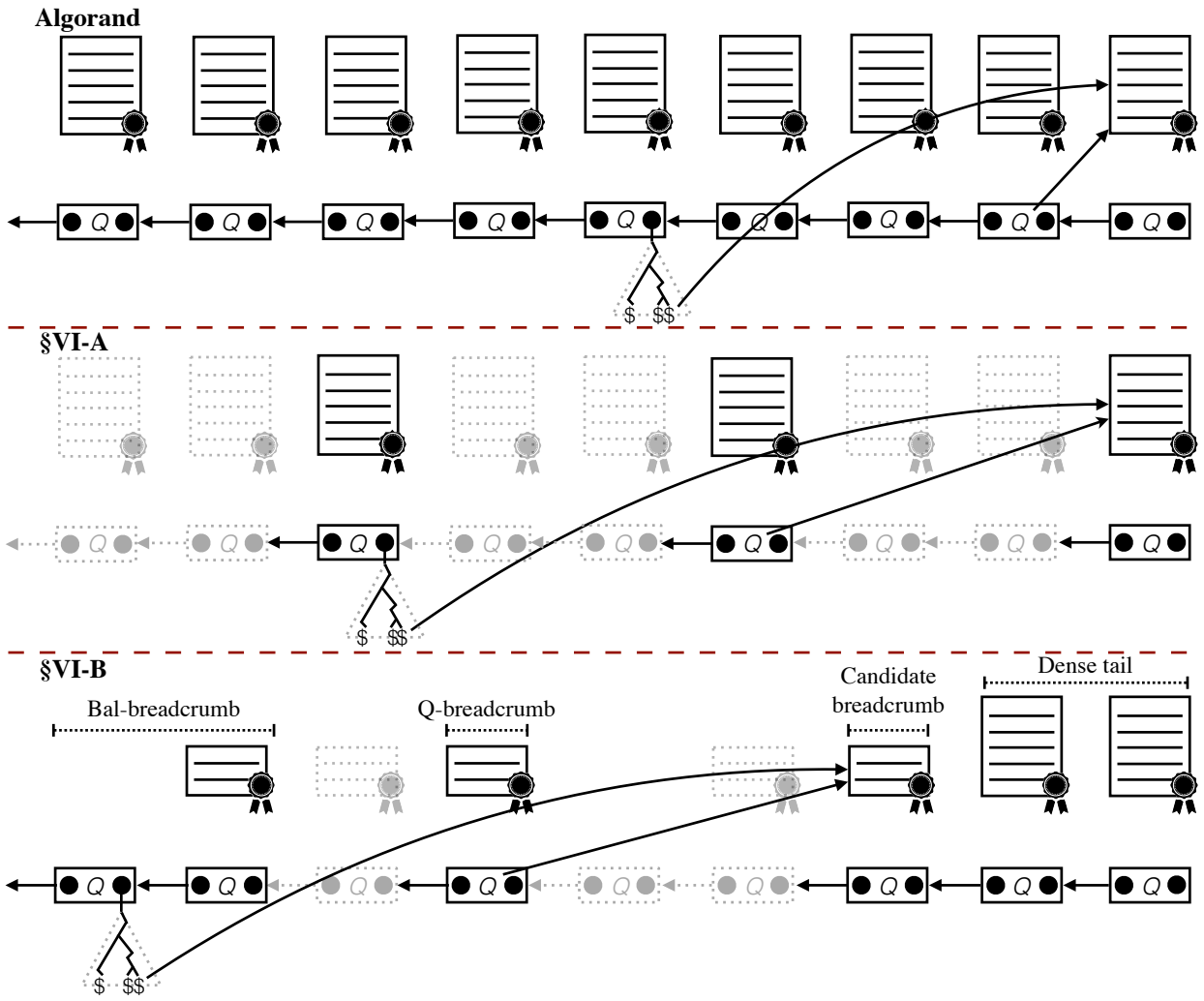

Fig. 4. Two optimizations used to reduce the bandwidth needed to prove validity of the latest state. In this figure, $b = c = 3$; as before, objects that clients can discard are outlined with dots. The top figure depicts the basic ledger data structure without any optimization: a large final certificate authenticates each block header, and each certificate depends on the Q value immediately before it and the proofs of stake b blocks ago. The next figure shows the additional stamping certificate chain with the leapfrog optimization: each leapfrogging certificate depends on the value of Q from b blocks ago and the balances from $2b$ blocks ago (§VI-A). The bottom figure shows stamping committee optimization used to reduce the size of certificates (§VI-B). It illustrates the candidate, Q-, and Bal-breadcrumbs which consist of a small stamping certificate and block header along with a tail of block headers. It shows the dense tail formed by the large final certificates that prove validity of the current block. (The figure omits the frozen breadcrumbs, which are farther back in the chain.)

large committee size. This arrangement is shown at the top of Figure 4.

Vault addresses this problem using a combination of two techniques. First, Vault introduces a *stamping* certificate that can be verified using state from b and $2b$ blocks ago rather than the state from 1 block ago. This allows clients to "leapfrog" by b blocks at a time instead of having to verify every single block in the blockchain. Vault uses Algorand's cryptographic sortition to privately select a committee for such stamping certificates in a way that does not reveal the committee membership to an adversary in advance. This ensures that an adversary cannot selectively corrupt members of this committee to falsify a certificate. Certificate signatures use ephemeral keys of each committee member, which are deleted by each committee member before they broadcast their signature. This is shown in the middle of Figure 4 and described in more detail in §VI-A.

Second, Vault uses a smaller committee size to generate the stamping certificates, which reduces the size of the certificates themselves (since they contain fewer signatures).[3] To ensure that a smaller committee does not give the adversary a higher probability of corrupting the committee, Vault requires a much larger fraction of expected committee members to vote in order for the stamping certificate to be valid. This means that, with significant probability, the committee fails to gather enough votes to form a stamping certificate. However, this is acceptable because new clients have two fallback options: they can either verify Algorand's full certificate, or they can verify

[3] Note that multi-signatures [24] would not significantly reduce the size of the certificates since the certificate needs to include a proof of cryptographic sortition (VRF) and a partial Merkle proof for each committee member whose signature appears in the certificate, which cannot be aggregated away.

a stamping certificate for a later block and backtrack using PREVBLOCK hashes in the block headers. This arrangement is shown at the bottom of Figure 4 and described in §VI-B.

A. Leapfrog Protocol

To allow leapfrogging, Vault constructs a sortition committee for the stamping certificate of block n using the seed Q from block $n-c$, where $c \geq 1$ is some constant. For security, the proof-of-stake balances must be selected from b blocks before the seed Q, so they are chosen from block $n-c-b$. Members of this committee wait for consensus on block n, and once consensus is reached, they broadcast signatures for that block (after deleting the corresponding ephemeral signing key), along with proofs of their committee selection. The set of these signatures forms the stamping certificate for BlockHeader_n.

As mentioned above, this committee is, in principle, known as soon as block $n-c$ has been agreed upon. However, the committee is selected in private using cryptographic sortition, and honest clients do not reveal their committee membership until they vote for block n, which prevents an adversary from adaptively compromising these committee members.

Now each certificate depends on two previous block headers: Certificate_n depends on Q from BlockHeader_{n-c} and BALROOT from $\text{BlockHeader}_{n-c-b}$. Moreover, Certificate_n validates BlockHeader_n, which itself contains the value of Q used for Certificate_{n+c} and the value of BALROOT used for $\text{Certificate}_{n+c+b}$.

To optimize for the case of a new client catching up on a long sequence of blocks starting with the genesis block, we set $c = b$, so that the client does not need to validate separate blocks for Qs and BALROOTs. This reduces the bootstrapping bandwidth by a factor of b, since a new client needs to download and authenticate every bth block header and certificate.

To ensure that any client can help a new peer bootstrap, all clients store the block header and certificate for blocks at positions that are a multiple of b. Additionally, to ensure that the base case is true, the first $2b$ blocks in Vault are predetermined to be empty. Finally, to quickly catch up after momentarily disconnecting from the network, clients keep the previous $2b$ block headers at all times.

Choosing b. Vault's choice of b trades off the weak synchrony assumption (i.e., partitions may not last for periods longer than b) against d, the speed at which stake that is sold becomes malicious. We briefly justify our choice of b below; we refer the reader to Algorand's security analysis [25] for a formal treatment.

On the one hand, suppose the adversary partitions the network for more than b blocks starting at round r'. Then the adversary may manipulate its public keys at round r' and the value of Q at round $r'+b$ such that at round $r'+b+1$, it engineers a proposer along with a committee whose members it wholly controls. In this way, the adversary gains total control of the ledger. Therefore, b must be large enough to tolerate complete partitions.

On the other hand, suppose a rich, honest user sells off 50% of the stake in Vault at round r'. A few rounds after the user completes the sale, a poor adversary corrupts this user, who by chance controls a supermajority of the committee at round $r'+b+1$. Then again the adversary gains control of the ledger. Although this adversary controls little of the system's *current* stake, it controls much of the system's *past* stake. As a result, b must be small enough to allow honest users to finish participating in Vault after selling off their stake.

Since c introduces an extra delay to certificate creation, for security we require that not $b \leq d$ but instead $b+c \leq d$, and since we set $c = b$ we require that $2b \leq d$. At Vault's highest level of throughput, $2b = d = 2880$ corresponds to about two days' worth of blocks.

B. Stamping Committees

Algorand's consensus protocol requires thousands or tens of thousands of signatures to produce a final certificate for a block. This threshold is very high because Algorand guarantees a very low rate of failure in terms of liveness and safety. A failure in liveness prevents a block from being confirmed, while a failure in safety may produce a ledger fork.

As with final certificates, a stamping committee threshold should be set sufficiently high such that an adversary cannot gather the signatures required to trick a new client into accepting a forged ledger fork with high probability. Since adversaries know when they are selected for a leadership in advance, and a certificate must be secure for all time, we must keep a strict safety threshold.

Although we cannot relax safety, we can greatly relax the liveness property. Suppose a new client has already verified the block headers for blocks r and $r+b$, using stamping certificates, but there was no stamping certificate produced for block $r+2b$ due to relaxed liveness requirements. If there was a stamping certificate produced for block $r+2b-1$, the new client can efficiently verify that stamping certificate and block instead.

Specifically, the new client can ask an existing peer for the headers of blocks $r-1$ and $r+b-1$ and efficiently verify them by checking PREVBLOCK hashes in blocks r and $r+b$ respectively. Since headers are relatively small, this costs the client little bandwidth. We use the term *breadcrumb* to denote this chain of PREVBLOCK pointers from a stamping certificate to an earlier block header. Figure 4's bottom row shows two such breadcrumbs: one that required backtracking by one block (for BALROOT), and one that did not require any backtracking (for Q).

If the stamping certificate at $r+2b-1$ also failed to form, Vault repeats this process to find the highest block below $r+2b$ that did have a stamping certificate. If no such block exists in a b-block interval, Vault falls back to a full Algorand certificate.

While Q is usually unpredictable and random, an adversary may introduce bias into its value during network partitions. Given this bias, Vault requires a safety failure rate of 2^{-100} for both its final and stamping certificates. However, with a

relaxed liveness assumption, we can decrease certificate size by at least an order of magnitude.

For example, with an honesty rate of $h = 80\%$, a final certificate requires a threshold of 7,400 signatures. If we allow stamping certificates to fail to form 65% of the time, then it suffices to have a threshold of 100 signatures (out of a suitably smaller committee). Applying the stamping optimizations allows clients in Vault to verify the latest block header in a 10-year old ledger by downloading 365 MB or less. Appendix §A analyzes stamping certificate size given other settings of the honesty and liveness failure rates.

Given that stamping certificate creation may occasionally fail, each breadcrumb must contain a small "tail" of block headers which are required to certify the two subsequent breadcrumbs produced at most b and $2b$ blocks ahead, respectively. Since block headers are relatively small (less than 256 bytes), the cost of storage here is low (less than 1.3 MB for $b = 1440$). As clients observe the confirmation of new blocks and the successful creation of new stamping certificates, they update their state so as to minimize the sizes of these tails. Clients must also hold a *dense tail* of block headers and final certificates at the end of the ledger for each block after the last header for which a stamping certificate was produced. Vault clients discard this dense tail whenever new stamping certificates are successfully created.

Proof components. For completeness, we describe the components of the proof sent to a new client to convince it that the ledger state is valid, and we describe the invariants that apply to each component of this proof. See Appendix §B for a description of one algorithm which achieves these invariants.

- *Dense tail*: The set of all headers and full final certificates since the candidate breadcrumb.
- *Candidate breadcrumb*: The breadcrumb with the last-observed stamping certificate. The candidate breadcrumb is tentative and may be overwritten by a "better" breadcrumb (i.e., a more recent breadcrumb which makes the candidate breadcrumb obsolete). This breadcrumb is never more than b blocks ahead of the Q-breadcrumb.
- *Q-breadcrumb*: The breadcrumb with the stamping certificate immediately preceding the candidate breadcrumb. This breadcrumb's certificate has been fixed as no subsequent certificate may be better than this. However, its tail of block headers may not yet be trimmed.
- *Bal-breadcrumb*: The breadcrumb with the stamping certificate immediately preceding the Q-breadcrumb. Like the Q-breadcrumb, its certificate is final and unchanging. Moreover, its tail remains "minimal" as new certificates are seen. In other words, it maintains the shortest tail such that the following conditions are true:
  1) It contains the block header needed to authenticate the Q-breadcrumb's certificate's Q-value.
  2) It contains the block header needed to authenticate the candidate breadcrumb's certificate's proofs of stake.
- *Frozen breadcrumbs*: The set of the rest of the breadcrumbs. These breadcrumbs have finalized both their certificates and tails. This set expands and absorbs the Bal-breadcrumb when the candidate breadcrumb "graduates" into a Q-breadcrumb, which in turn "graduates" into a Bal-breadcrumb.

VII. Evaluation

The primary question that our evaluation attempts to answer is, "How effective is Vault at reducing the bandwidth cost of helping a new client join the network?" §VII-B presents the results.

To understand why Vault achieves a reduction in bandwidth, we further answer three questions targeted at each of Vault's techniques, as follows. Recall that two components contribute to bootstrapping costs: the state needed to execute the protocol and the bandwidth required to convince a new client that this state is correct.

- *Balance Pruning*: How much does transaction expiration reduce storage cost by? (§VII-C)
- *Stamping Certificates*: What are the cost savings of using Vault's sparse sequence of stamping certificates for bootstrapping? (§VII-D)
- *Balance Sharding*: What are the trade-offs involved in sharding Vault's balance sets? (§VII-E)

A. Experimental Setup

To answer the above questions, we implemented the data structures needed to execute the Bitcoin, Ethereum, Algorand, and Vault protocols. However, we have not integrated these data structures into their respective systems. We vary transaction volumes between 50 and 500 million transactions, and we fill all blocks with the maximum number of transactions given some fixed block size. (As of February 2018, there are around 300 million transactions in Bitcoin [21] and around 150 million transactions in Ethereum [26].) We ignore the storage cost of auxiliary data structures required to efficiently update a protocol's state; for example, we do not implement database indexes.

Algorand uses a transaction format similar to Bitcoin's. We consider only simple transactions with the form of one input and two outputs (one to the receiver and the other to self).

The ratio of unique accounts to transactions on Ethereum is around 15% [27], [26] as of January 1, 2018. Additionally, we obtained the Ethereum ledger up to this date by synchronizing a Parity [28], [29] Ethereum client (in `fatdb` mode). Our analysis of the Ethereum state indicates that around 38% of all accounts have no funds and no storage/contract data (i.e., only an address and a nonce). For Ethereum and Vault, we fix the account creation rate at 15% and the churn rate at 38%. Other than to count the number of empty accounts, we do not consider the costs in Ethereum which result from per-account data storage or from smart contracts.

We instantiate the following parameters both in Algorand and in Vault:

- 80% of the stake in the system is honest ($h = 0.8$).

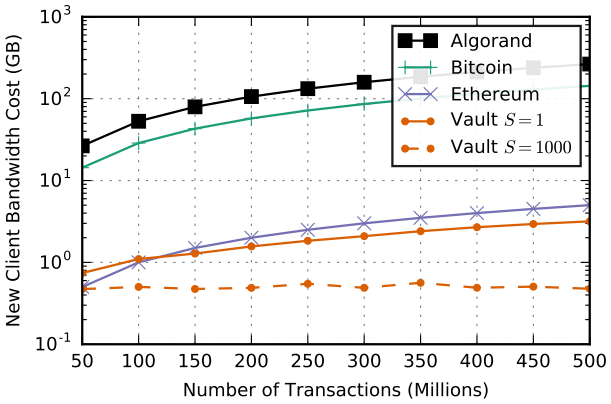

Fig. 5. An end-to-end comparison of the bootstrapping costs of the Bitcoin, Ethereum, Algorand, and Vault ledgers (with sharding factors of 1 and 1,000). Compared to Bitcoin and Algorand, Vault and Ethereum reduce storage costs by one to two orders of magnitude. Vault outperforms Ethereum at 150 million transactions because it can delete old accounts. Sharding Vault with a factor of 1,000 reduces the costs of storing balances to a negligible amount, and the total storage cost remains low (below 500 MB) even with 500 million transactions on the ledger. Note that the y-axis is logarithmic.

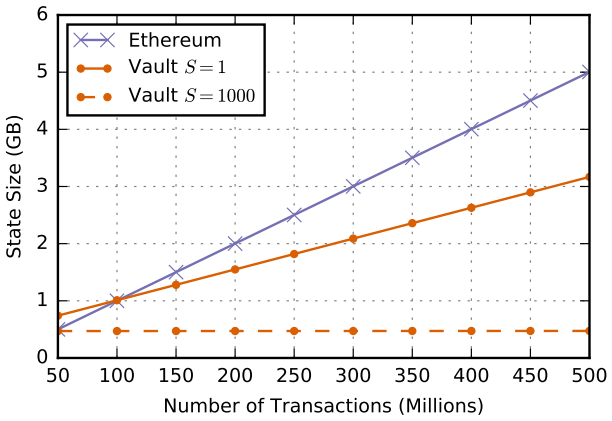

Fig. 6. A comparison of steady-state storage costs between Vault and Ethereum, given an account churn rate of 38%. Observe that the ability to prune empty balances allows Vault to keep a smaller balance tree than Ethereum past around 100 million transactions, even as Vault must pay the cost of storing its recent transaction log.

- Stake sold off by a later-corrupted user counts towards h for $d = 48$ hours.
- Network partitions last for at most 2 days. (Recall that during a network partition, an adversary may arbitrarily reschedule and drop any message.) This implies that the leapfrogging interval is $b = 1440$ rounds.
- The maximum transaction lifetime is $T_{max} = 4$ hours. This keeps the cost of storing the hashes of recent transactions to the hundreds of megabytes.
- Stamping certificates fail to form at a rate of 65%. This implies that a certificate contains $T_{\text{stamping}} = 100$ signatures, and a stamping sortition produces $\tau_{\text{stamping}} = 120$ committee members in expectation.
- The size of a block is 10 MB. (Lower block sizes are possible; these increase throughput and reduce latency.)

We use S in the rest of this section to denote the number of shards in Vault.

B. End-to-end Evaluation

Figure 5 presents the results of an end-to-end evaluation of Bitcoin, Ethereum, Algorand, and Vault (with sharding factors of $S = 1$ and $S = 1000$).

Algorand's storage cost exceeds that of Bitcoin. Every transaction that Bitcoin stores must also be stored by Algorand. In addition, being able to execute secure bootstrapping in Algorand incurs an additional cost ranging from 4 to 47 GB, growing linearly with the number of confirmed transactions in the system.

Figure 5 shows clear gains in storing the set of account balances rather than the set of transactions. Vault and Ethereum, which both store account balances, outperform Algorand and Bitcoin by 1 to 2 orders of magnitude. This holds both because the set of balances is much smaller than the set of all transactions, and also because an individual balance entry is smaller than a transaction itself. Given that we only consider simple transactions with one input and two outputs, we expect more complex transactions to amplify this effect.

Moreover, we see that after 150 million transactions, Vault begins to outperform Ethereum even without sharding. This follows from the fact that Vault may delete accounts with no balance, which reduces overall storage cost by about 38%. However, before 150 million transactions, the cost of storing the recent transaction log imposes a fixed cost. We note that throttling the throughput of Vault or reducing T_{max} can easily decrease this cost.

Finally, we observe that sharding Vault reduces storage even more significantly. However, sharding is no "free lunch"; it increases the sizes of transactions and thus the steady-state bandwidth cost of propagating them to the entire network (§VII-E).

C. Balance Pruning

To evaluate the efficiency of Vault's balance pruning technique, we compare the storage footprint of Vault's balance set (again sharded at factors of 1 and 1000) against Ethereum's. Since Vault also requires a log of the recent transaction history to detect double-spending, we include these costs as well.

Figure 6 shows that the ability to prune the balance tree significantly reduces the ledger's storage costs at scale. Initially, Vault clients must hold the past 9.6 million transaction hashes to enforce transaction expiration, which costs around 307 MB of overhead (if transaction expiration T_{max} is set to correspond to 4 hours). However, past 150 million transactions, holding the set of account balances dominates the cost of detecting double-spending. Since Ethereum clients cannot garbage collect the 38% of empty accounts in their balance trees, they must store these accounts in perpetuity.[4] Maintaining a log of recent transactions constitutes a constant storage cost, while

[4] We speculate that the use of Ethereum's smart contracts to programmatically create temporary accounts only exacerbates this problem. Efficient garbage collection implies cheap temporary account creation.

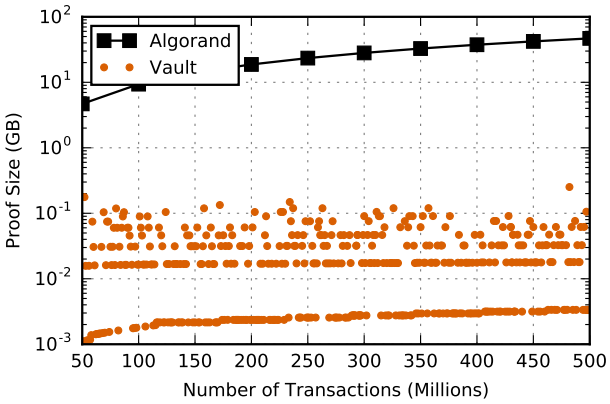

Fig. 7. A comparison between the certificate chain sizes in both Vault and Algorand. The proof size is 2—4 orders of magnitude smaller in Vault, and its size does not exceed 1 GB up to 500 million transactions. Usually, the size of this proof remains under 100 MB. In the plot, proof sizes cluster around several bands, which correspond to the number of final certificates present in the dense tail. The lowest band grows linearly with the number of stamping certificates that were formed. Note that the y-axis is logarithmic.

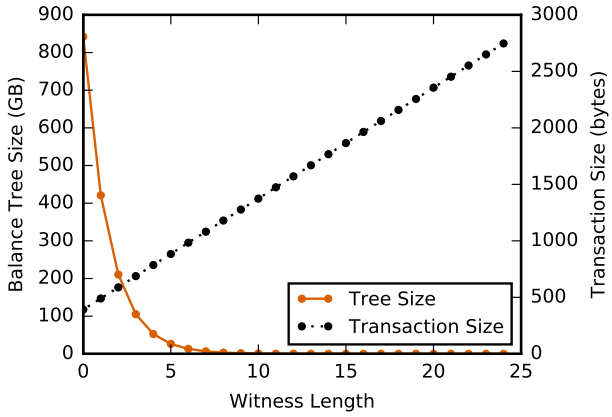

Fig. 8. Cost of storing sharded account balances and transaction sizes given some setting of the witness length. The tree here stores 10 billion accounts, divided into $S = 1000$ shards. Increasing transaction size linearly (i.e. extending the Merkle witness) enables clients to decrease their storage overhead by a factor of 2, which illustrates an exponential relationship between transaction size and storage cost. Note that the figure contains a shared x-axis but two y-axes: the left axis shows storage size while the right axis shows transaction size.

the overhead of storing empty accounts grows linearly as the system continually processes new transactions.

D. Stamping Certificates

Next, we evaluate how efficiently a client can prove the validity of its state to a new peer. We measure the amount of data transferred for the stamping certificate chain in Vault and compare it against the data transferred for the final certificate chain in Algorand. Since the creation of stamping certificates in Vault is non-deterministic, we evaluate the amount of data transferred using fine-grained steps on the x-axis (number of transactions processed) to illustrate these effects.

Figure 7 reveals that the overhead of the certificate and header storage cost becomes significant in Algorand. To catch up to a ledger with 500 million transactions, a client must download around 47 GB of data.

In contrast, Vault's proofs are much smaller even though the use of a balance-based ledger increases the size of certificates (by including partial Merkle proofs); these proofs are almost always less than 100 MB in size. Two factors decrease the size of these proofs. First, the chain of certificates is much sparser. On average, downloading an extra stamping certificate allows a client to validate an additional $b = 1440$ blocks. Second, each individual stamping certificate is small. Instead of 7,400 signatures, each certificate is made up of 100 signatures.

Finally, this experiment demonstrates that, without the stamping certificate optimization, certificates would dominate the data required for bootstrapping. A 3.4 GB state size for balances matters little if 47 GB is necessary to prove its validity. Reducing the proof overhead to less than 100 MB allows Vault to securely bootstrap new clients with modest bandwidth cost.

E. Balance Sharding

Under sharding, we would like to determine how decreasing the overhead of storing the intermediate frontier in the balance tree (§V-B) increases the size of transactions. We fix the number of accounts to be 10 billion and the number of shards to be $S = 1000$. Figure 8 illustrates this interaction.

We see that shard size is not the limiting factor. With $S = 1000$, each client stores only 10 million accounts per shard, which costs less than 500 MB each. Instead, sharding costs are dwarfed by the overhead of keeping the internal Merkle nodes on the frontier that allow clients to verify transactions in other shards without needing to receive the entire Merkle path as part of the transaction. On the one hand, all clients may simply store all leaf Merkle nodes, which adds nothing to transaction overhead but also reduces storage cost by only a small amount: storing the set of balance along with the Merkle frontier costs each client almost 1 TB. On the other hand, the exponential fanout of the sparse Merkle tree provides diminishing returns on storing each subsequent layer; extending the Merkle witness by one hash halves the storage footprint of the Merkle nodes. Eventually, storage costs converge to the size of a shard.

VIII. Conclusion

Vault is a new cryptocurrency designed to reduce storage and bootstrapping costs. Vault achieves its goals using three techniques: (1) transaction expiration, which helps Vault decouple storage of account balances from recent transactions, and thus delete old account state; (2) adaptive sharding, which allows Vault to securely distribute the storage of account balances across participants; and (3) stamping certificates, which allow new clients to avoid verifying every single block header, and which reduce the size of the certificate. Experiments demonstrate that Vault achieves its goals, reducing the storage and bootstrapping cost for 500 million transactions to 477 MB, compared to 5 GB for Ethereum and 143 GB for Bitcoin.

References

[1] S. Nakamoto, "Bitcoin: A peer-to-peer electronic cash system," https://bitcoin.org/bitcoin.pdf, 2008.

[2] Blockchain Luxembourg S.A., "Blockchain size," https://blockchain.info/charts/blocks-size, 2018.

[3] Ethereum Foundation, "Ethereum," 2016, https://www.ethereum.org/.

[4] G. Wood, "Ethereum: A secure decentralised generalised transaction ledger," *Ethereum Project Yellow Paper*, vol. 151, pp. 1–32, 2014.

[5] V. Buterin, "Security alert [11/24/2016]: Consensus bug in geth v1.4.19 and v1.5.2," https://blog.ethereum.org/2016/11/25/security-alert-11242016-consensus-bug-geth-v1-4-19-v1-5-2/, 2016.

[6] G. Wood, "State trie clearing (invariant-preserving alternative)," https://github.com/ethereum/EIPs/blob/master/EIPS/eip-161.md, 2016.

[7] Y. Gilad, R. Hemo, S. Micali, G. Vlachos, and N. Zeldovich, "Algorand: Scaling Byzantine agreements for cryptocurrencies," in *Proceedings of the 26th ACM Symposium on Operating Systems Principles (SOSP)*, Shanghai, China, Oct. 2017, pp. 51–68.

[8] D. Mazières, "The Stellar consensus protocol: A federated model for internet-level consensus," https://www.stellar.org/papers/stellar-consensus-protocol.pdf, 2014.

[9] K. Nikitin, E. Kokoris-Kogias, P. Jovanovic, N. Gailly, L. Gasser, I. Khoffi, J. Cappos, and B. Ford, "Chainiac: Proactive software-update transparency via collectively signed skipchains and verified builds," in *Proceedings of the 26th USENIX Security Symposium*, Vancouver, Canada, Aug. 2017, pp. 1271–1287.

[10] E. Kokoris-Kogias, P. Jovanovic, L. Gasser, N. Gailly, and B. Ford, "OmniLedger: A secure, scale-out, decentralized ledger," Cryptology ePrint Archive, Report 2017/406, Feb. 2018, http://eprint.iacr.org/.

[11] D. E. Knuth, *The art of computer programming*. Pearson Education, 1997, vol. 3.

[12] R. C. Merkle, "A digital signature based on a conventional encryption function," in *Proceedings of the 7th Annual International Cryptology Conference (CRYPTO)*, Santa Barbara, CA, Aug. 1987, pp. 369–378.

[13] M. Castro and B. Liskov, "Practical Byzantine fault tolerance and proactive recovery," *ACM Transactions on Computer Systems*, vol. 20, no. 4, Nov. 2002.

[14] J. Poon and T. Dryja, "The Bitcoin Lightning network: Scalable off-chain instant payments," Jan. 2016, https://lightning.network/lightning-network-paper.pdf.

[15] A. Poelstra, "Mimblewimble," Oct. 2016, https://download.wpsoftware.net/bitcoin/wizardry/mimblewimble.pdf.

[16] E. Syta, I. Tamas, D. Visher, D. I. Wolinsky, P. Jovanovic, L. Gasser, N. Gailly, I. Khoffi, and B. Ford, "Keeping authorities "honest or bust" with decentralized witness cosigning," in *Proceedings of the 37th IEEE Symposium on Security and Privacy*, San Jose, CA, May 2016, pp. 526–545.

[17] M. J. Fischer, N. A. Lynch, and M. S. Paterson, "Impossibility of distributed consensus with one faulty process," in *Proceedings of the 2nd ACM SIGACT-SIGMOD Symposium on Principles of Database Systems (PODS)*, Atlanta, GA, Mar. 1983, pp. 1–7.

[18] S. Micali, M. O. Rabin, and S. P. Vadhan, "Verifiable random functions," in *Proceedings of the 40th Annual IEEE Symposium on Foundations of Computer Science (FOCS)*, New York, NY, Oct. 1999.

[19] K. R. Fall and W. R. Stevens, *TCP/IP Illustrated, Volume 1: The Protocols*. Addison-Wesley, 2011.

[20] R. Dahlberg, T. Pulls, and R. Peeters, "Efficient sparse Merkle trees: Caching strategies and secure (non-)membership proofs," in *Proceedings of the 21st Nordic Conference on Secure IT Systems*, Nov. 2016, pp. 199–215.

[21] Blockchain Luxembourg S.A., "Total number of transactions," https://blockchain.info/charts/n-transactions-total, 2018.

[22] BitInfoCharts, "Bitcoin rich list," https://bitinfocharts.com/top-100-richest-bitcoin-addresses.html, 2018.

[23] G. Blom, L. Holst, and D. Sandell, *Problems and Snapshots from the World of Probability*. Springer Science & Business Media, 2012.

[24] M. Bellare and G. Neven, "Multi-signatures in the plain public-key model and a general forking lemma," in *Proceedings of the 13th ACM Conference on Computer and Communications Security (CCS)*, Alexandria, VA, Oct.–Nov. 2006, pp. 390–399.

[25] Y. Gilad, R. Hemo, S. Micali, G. Vlachos, and N. Zeldovich, "Algorand: Scaling Byzantine agreements for cryptocurrencies," Cryptology ePrint Archive, Report 2017/454, May 2017, http://eprint.iacr.org/.

[26] Etherscan, "Ethereum transaction chart," https://etherscan.io/chart/tx, 2018.

[27] ——, "Ethereum unique address growth rate," https://etherscan.io/chart/address, 2018.

[28] Parity Technologies, "Parity," https://www.parity.io/, 2017.

[29] ——, "Github - paritytech/parity: Fast, light, robust Ethereum implementation," https://github.com/paritytech/parity, 2018.

Appendix

A. Stamping certificate security analysis

Recall that the security of a certificate is equivalent to the security of the committee that produced it. We first define two desirable properties of certificates.

Definition. A certificate has a *safety failure rate* of f_s if for all committees produced by cryptographic sortition, the probability that an adversary can obtain two distinct and validating certificates for a given block is f_s.

Definition. A certificate has a *liveness failure rate* of f_l if for all committees produced by cryptographic sortition, the probability that the honest users fail to produce a certificate for a given block is f_l.

Vault's stamping committees are secure if they satisfy the properties of liveness and safety. In Vault's stamping committees, an honest verifier does not release its signature until it sees a block confirmation. Because confirmed blocks are fork-safe, we are guaranteed that if one honest verifier sees a block, all other honest verifiers have seen that block. Thus, we have the following observation:

Theorem. *In Vault, it suffices to have one honest signature in a stamping certificate to prove that it is valid.*

Now let T_{stamping} be the threshold of signatures needed to produce a valid stamping certificate, and let τ_{stamping} be the number of committee members elected in expectation to produce this certificate. Moreover, let γ and β be the actual number of honest and malicious users elected to some committee. We can translate the theorem into two desirable properties, as follows:

Corollary. *For Vault to produce certificates with a safety failure rate of f_s and liveness failure rate of f_l, we must set τ_{stamping} as follows:*

$$\Pr[\gamma < T_{\text{stamping}}] \leq f_l \tag{1}$$

$$\Pr[\beta \geq T_{\text{stamping}}] \leq f_s \tag{2}$$

Suppose that the number of currency units in the system is U. For simplicity, let each user in the system own one unit of currency. If h is the proportion of honest users in the system, τ is the expected number of selected users following a cryptographic sortition, and U is arbitrarily large, we have that the chance of sampling exactly k honest users is

$$\Pr[\gamma = k] = \frac{(h\tau)^k}{k!} e^{h\tau}$$

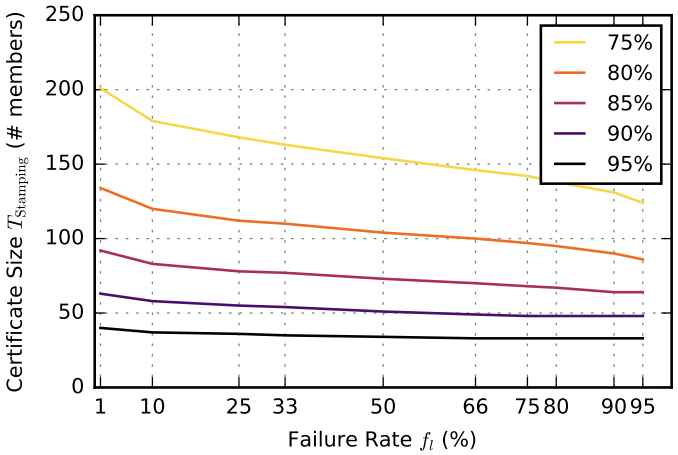

Fig. 9. Certificate sizes required to guarantee various liveness failure rates for a fixed safety failure rate of 2^{-100}, given assumptions on the amount of honest stake in the system. Honesty assumptions have a significant effect on certificate size, which in turn has a significant influence on the liveness failure rate.

while the chance of sampling exactly k malicious users is

$$\Pr[\beta = k] = \frac{((1-h)\tau)^k}{k!} e^{h\tau}.$$

(This analysis follows from the application of the binomial theorem.)

From (Equation 1) and (Equation 2) it follows that the following conditions must both hold:

$$\sum_{k=0}^{T_{\text{stamping}}-1} \frac{(h\tau_{\text{stamping}})^k}{k!} e^{h\tau_{\text{stamping}}} \leq f_l \quad (3)$$

$$\sum_{k=T_{\text{stamping}}}^{\infty} \frac{((1-h)\tau_{\text{stamping}})^k}{k!} e^{(1-h)\tau_{\text{stamping}}} \leq f_s \quad (4)$$

Then it is evident that $\tau_{\text{stamping}} = 120, T_{\text{stamping}} = 100$ satisfy these conditions with $h = 0.8, f_s = 2^{-100}, f_l = 0.65$.

Figure 9 illustrates the effects of changing f_l for various values of h, fixing $f_s = 2^{-100}$.

B. Stamping certificate algorithm

In this appendix we provide an algorithm which maintains the invariants required for a valid proof of the ledger's latest state.

All clients maintain a proof of the ledger's latest block. A client mutates its proof on observing two events from the cryptocurrency network:

- When a client observes a block and a full final certificate, it appends it to its dense tail.
- When a client observes a new stamping certificate later than its candidate breadcrumb, it deletes the final certificates in its dense tail up to and including this certificate. Moreover, it moves ownership of these block headers to the new stamping certificate, which becomes the new breadcrumb. Next,
  - If the breadcrumb index is not more than b blocks greater than the Q-breadcrumb, the client replaces its candidate breadcrumb with the new breadcrumb, transferring over the block headers of the tail.
  - Otherwise, the client:
    1) Freezes the Bal-breadcrumb, adding it to the set of frozen breadcrumbs.
    2) Sets the Q-breadcrumb as the new Bal-breadcrumb. This breadcrumb's tail becomes trimmable.
    3) Sets the candidate breadcrumb as the new Q-breadcrumb. This breadcrumb's position is now optimal and fixed (assuming that stamping certificates are received in order).
    4) Sets the new breadcrumb as the candidate breadcrumb.

  Finally, it "trims" the tail of the Bal-breadcrumb to keep its length minimal.

We summarize these procedures in pseudocode below.

```
function OnBlockReceive(block, finalCert)
    balances.Apply(block.transactions)
    denseTail.Append((block.header, finalCert))
end function
function OnStampReceive(stampCert)
    if stampCert.index − b ≤ qBreadcrumb.index then
        lastBreadcrumb.Update(stampCert, denseTail)
    else
        frozen.Append(balBreadcrumb)
        balBreadcrumb ← qBreadcrumb
        qBreadcrumb ← lastBreadcrumb
        lastBreadcrumb ← Combine(stampCert, denseTail)
    end if
    denseTail.Clear()
    balBreadcrumb.Trim(qBreadcrumb, lastBreadcrumb)
end function
```