Enabling secure and resource-efficient blockchain networks with VOLT

Srinath Setty Soumya Basu⋆ Lidong Zhou Michael L. Roberts Ramarathnam Venkatesan
Microsoft Research ⋆Cornell University

MSR Technical Report MSR-TR-2017-38

Abstract

We consider the problem of creating secure and resource-efficient *blockchain networks* i.e., enable a group of mutually distrusting participants to efficiently share state and then agree on an append-only history of valid operations on that shared state. This paper proposes a new approach to build such blockchain networks. Our key observation is that an append-only, tamper-resistant ledger (when used as a communication medium for messages sent by participants in a blockchain network) offers a powerful primitive to build a simple, flexible, and efficient consensus protocol, which in turn serves as a solid foundation for building secure and resource-efficient blockchain networks.

A key ingredient in our approach is the abstraction of a *blockchain service provider (BSP)*, which oversees creating and updating an append-only, tamper-resistant ledger, and a new distributed protocol called *Caesar consensus*, which leverages the BSP's interface to enable members of a blockchain network to reach consensus on the BSP's ledger—even when the BSP or a threshold number of members misbehave arbitrarily. By design, the BSP is untrusted, so it can run on any untrusted infrastructure and can be optimized for better performance without affecting end-to-end security. We implement our proposal in a system called VOLT. Our experimental evaluation suggests that VOLT incurs low resource costs and provides better performance compared to alternate approaches.

1 Introduction

Blockchain technology—which underpins popular cryptocurrencies such as Bitcoin [69] and Ethereum [85]—has been touted to dramatically transform common business processes in many areas such as finance [86], health care [47], and the public sector [77]. The fundamental appeal of blockchain is its promise to enable verifiable, low-cost, and trustworthy business processes without involving expensive, trusted intermediaries, which are rampant in the aforementioned business areas. A key mechanism behind this technology is an append-only, tamper-resistant ledger shared by all participants in a blockchain network.

However, most blockchains fail to deliver this promise fully: they fall short in security, privacy, performance, or cost. For example, the security of public blockchain systems such as Bitcoin and Ethereum relies on game-theoretic and incentive-based mechanisms (e.g., mining based on proof-of-work) because they operate in a *permissionless* membership model where anyone can participate in the system. Furthermore, they are slow and expensive (due to mining) for many business applications (§2.2).

This has led to blockchains in a *permissioned* environment such as the cloud [6, 7], where participation is restricted to entities of a business process. Some of these solutions employ public blockchain protocols such as Ethereum, but for such systems, strong security requires deploying enormous computational resources for mining [44, 71]. Others propose systems such as Hyperledger [5], Tendermint [24], and Ripple [79], which use variants of Byzantine fault-tolerance (BFT) protocols [28, 32, 55] for consensus in a blockchain network.

Such BFT-based approaches are resource efficient compared to mining-based systems as they do not employ mining. However, members in such blockchains must trust the full implementation of the BFT protocol as well infrastructure where it is deployed. Furthermore, given the monolithic nature of these protocols, they are hard to optimize without risking subtle bugs.

We address these problems with a system called VOLT, which is the first system to enable secure and resource-efficient blockchain networks in a permissioned membership model such that the bulk of the work can be safely run on untrusted infrastructure. To achieve this, VOLT advocates a new architecture for blockchain networks, which introduces the concept of a *blockchain service provider (BSP)* and a novel distributed protocol called *Caesar consensus*[1] that, unlike a traditional consensus protocol, is run by member nodes participating in a blockchain network along with the BSP.

In VOLT, the BSP oversees the task of creating and updating a append-only, tamper-resistant ledger (where entries are governed by a state machine defined by the members of the blockchain network). Despite the BSP handling critical work, member nodes use Caesar consensus to detect misbehavior by the BSP, recover from misbehavior, and to reach consensus on the ledger. Caesar consensus does all these through end-to-end checks on properties of the BSP, by building atop prior works in untrusted storage [27, 41, 59, 62, 76], BFT systems [28, 32, 55, 60], and accountability [45, 46, 89].

A key observation in Caesar consensus is that member

[1]We name it after Julius Caesar, a dictator who ran the Roman Republic until he was assassinated by the Senate. Similarly, the BSP orchestrates our consensus protocol until the member nodes of a network replace it. We recognize naming conflicts with a system in another area [68].

nodes of a blockchain network can leverage the append-only, tamper-resistant ledger maintained by the BSP to reach agreement on a prefix of the ledger, *by writing special messages of a single type on the ledger itself.* This mechanism leads to a simple consensus algorithm for permissioned blockchain networks. Furthermore, those special messages are recorded on the ledger and processed by members in a network, so member nodes can use them to enforce a flexible class of policies to determine which prefix of the ledger is agreed-upon by the network. As an example, the blockchain network can enforce consistency levels ranging from strong to eventual, and fault models (for member nodes) from crash to Byzantine (§3.4,§8), while keeping the BSP oblivious to these policies.

The above is in stark contrast with the aforementioned BFT-based systems where nodes reach consensus on a set of transactions (using a protocol such as PBFT [28]) *before* writing them to a tamper-resistant ledger, thus embedding the consensus policy a priori into the system. Additionally, in VOLT, if the BSP or any malicious member node commits a safety violation and exposes it to an honest member node, it can eventually be blamed as malicious behavior by the BSP. Such a blame is cryptographically verifiable by other honest member nodes, and it does not require any threshold assumption.

Finally, this paper makes the following contributions.

- We propose a new foundation—a blockchain service provider and Caesar consensus—for designing permissioned blockchain networks.
- We architect a trustworthy BSP through a modular design that makes it easy to optimize the BSP for better performance and security.
- We implement VOLT using Azure service fabric, a framework for building distributed systems.
- Our experimental evaluation shows that VOLT achieves several thousands of transactions per second per ledger for a realistic blockchain application (which is 190$\times$ higher than an Ethereum-based baseline).

2 Setup, motivation, and goals

This section describes the high-level goals of VOLT and provides background on blockchain technology.

2.1 Goals

The principal goal of VOLT is to enable an emerging class of applications that employ blockchains to re-architect critical business processes [77, 86]. Examples include clearing and settlement [35], supply chain management [34], health care [13], asset registries [51], etc. We call such applications *secure multi-party collaborations (SMCs).*

In a nutshell, they involve mutually distrusting entities sharing state and then agreeing on a history of operations on that shared state. Given the mission-critical nature of these applications, we wish to build a system that:

- Ensures that all mutations to shared state are cryptographically verifiable by all participants in the system.
- Makes it computationally hard to tamper with the history of operations on shared state, or the state itself.
- Keeps the shared state and the operations on it private.
- Tolerates malice from participants in the system.

2.2 Blockchain technology

In principle, blockchain technology can support SMC, since, at its core, it makes a group of mutually distrusting parties maintain a shared ledger. We now elaborate this by focusing on two popular blockchain systems.

Bitcoin [69] is a cryptocurrency where users are identified by public keys in a digital signature scheme, and users transact using their private keys [12]. Bitcoin records all transactions in a *blockchain*, which is an append-only, cryptographically chained sequence of blocks. The blockchain is maintained by *miners* in a decentralized network, and they agree on the current state of blockchain via *mining*: miners solve computational puzzles to append new blocks to the chain [8]. Thus, Bitcoin creates a trustworthy ledger using a novel combination of puzzles and incentives [43, 70, 74]. Ethereum [85] extends Bitcoin's programmability[2]: users can store stateful computations on the blockchain. They can then mutate the state of such computations by submitting transactions, which are also stored in the same blockchain. Such a model enables building general applications atop blockchain.

Difficulties. It is well known that enterprises cannot directly use public blockchains for SMC, since they scale poorly [25, 36]. As an example, the maximum throughput of Bitcoin is only 7 transactions/second [1]. There are proposals to address the throughput limitations of public blockchains [14, 39, 53]. However, the blockchain is still secured via mining and incentives, which can lead to surprising attacks [38, 40, 50, 61, 72]. For example, Luu et al. [61] show that rational miners have incentives to skip verifying expensive transactions in new blocks. Additionally, public blockchains are expensive. For example, it costs \$0.18$\times 10^3$/KB of storage on Ethereum.[3] This has led to blockchains in a permissioned setting [5–7, 24, 79], but as Section 1 discusses, they are either resource-inefficient or require trusting the full implementation as well as infrastructure where they are deployed.

2.3 Our approach: VOLT

We take an alternate approach. We propose the notion of a blockchain service provider (BSP) as well as a new

[2]Each Bitcoin transaction includes a script that miners execute [12, 80].
[3]It costs 20,000 gas/256 bits [85]; the gas price is $\approx 10^{-6}$ ETH [3, 4].

distributed protocol, called Caesar consensus, which leverages the BSP and the cryptographic properties of a blockchain, to enable enterprises to create secure and efficient blockchain networks (§3). Additionally, through a modular design for the BSP, we show how to harden the BSP for better performance and security (§4).

3 VOLT: BSP and Caesar consensus

We first provide an overview of VOLT. We then describe details of the BSP and Caesar consensus. We finally discuss accountability properties of the BSP.

3.1 Overview of VOLT

VOLT targets permissioned blockchains, which limit participation to entities involved in a business application.[4] Figure 1 depicts VOLT's high level architecture, which shows two types of entities: the BSP and member nodes. The BSP can naturally be offered as a service on any untrusted infrastructure including the cloud by a cloud provider such as Amazon AWS and Microsoft Azure. Alternatively, the BSP can be instantiated in a decentralized manner on member nodes (§8). Regardless, the BSP is fully untrusted. Member nodes on the other hand are owned and operated by the participating entities in the blockchain network.

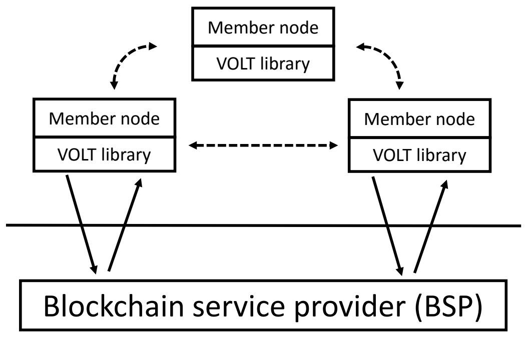

FIGURE 1—High level architecture of VOLT. Member nodes interact with the BSP using a VOLT library, which makes calls to a well-defined interface exposed by the BSP, to submit transactions and to learn the current state of the blockchain network. The dotted arrows represent optional communication channels.

Each member node uses the VOLT library to interact with the BSP i.e., to submit transactions and to learn the current state of the blockchain network. The VOLT library also implements the Caesar consensus protocol, without requiring any cooperation from the BSP beyond its interface (§3.4). For simplicity, we assume that only member nodes submit transactions.[5]

[4]Many enterprise applications of blockchain fundamentally involve entities with well-known identities.

[5]It is easy to extend VOLT where members specify an authentication policy that defines who can submit transactions (e.g., accept transactions from external users if they are authorized by one of the member nodes), and have the BSP enforce it.

Initialization and setup. When a VOLT blockchain network is initialized, member nodes collectively define rules that govern the network. This includes the current list of participants, initial state of the network, and rules to authenticate and validate transactions. The latter two are encoded in a *blockchain state machine (BSM)*,[6] which includes code executed by the network to process transactions. The next subsection discusses blockchain state machines and the programming model VOLT supports as well as where this initialization information is recorded.

Threat model. We make the following assumptions. We assume that the BSP and member nodes obey standard cryptographic hardness assumptions. We assume that the BSP's public key in a digital signature scheme is known to all member nodes, and that all member nodes in a blockchain network know the public keys of other members. We assume that the BSP and member nodes could arbitrarily deviate from their protocol—in particular, that they could be controlled by the same global adversary (we refine this assumption further in Section 3.4).

We assume that the network could be adversarial by dropping, reordering, and duplicating messages. However, we assume that correct entities (i.e., those that follow their prescribed protocol) in the system can eventually communicate with other correct entities. We also assume that the network is eventually synchronous for liveness [42].

3.2 Verifiable ledgers and programming model

A core ingredient in a VOLT network is the blockchain data structure from decentralized cryptocurrencies (§2.2). We generalize this data structure and call it a verifiable outsourced ledger (VOL).

Each blockchain network in VOLT is associated with a VOL. A VOL is an append-only chain of blocks where each block consists of two components: a *metadata block* and a *data block*. A metadata block is a triple: $(id, prev, h)$ where *id* is the identity of the blockchain network represented by the VOL, *prev* is a cryptographic hash of the predecessor block (or `null` if no predecessor exists), and *h* is a cryptographic hash of the corresponding data block, which contains a list of transactions.

We make a simple extension: Each entry in a VOL is accompanied by a set of digital signatures (called receipts) signed by the private key of the BSP (§4). (We discuss details of these receipts in the next section.) Also, as in decentralized cryptocurrencies, entries in a VOL are governed by a program, which we call a blockchain state machine (BSM). The BSM of a blockchain network is stored at the first block (called the genesis block) in the

[6]We assume a static membership list and blockchain state machine for each network. Section 8 discusses how to evolve them.

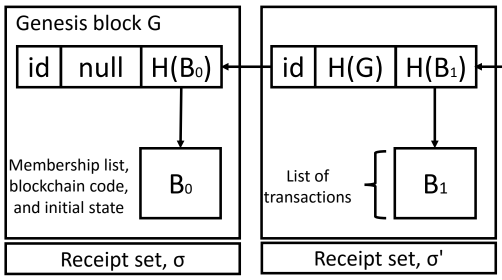

FIGURE 2—Overview of a VOL, which is an append-only data structure, similar to blockchain in decentralized cryptocurrencies (e.g., each block contains transactions and embeds a cryptographic hash H of its predecessor). In VOLT, a VOL also acts as a mechanism for disseminating a blockchain network's metadata (e.g., the first block in a VOL stores rules that govern the blockchain network it represents). Each entry in a VOL is also accompanied by a set of signatures using the BSP's private key.

VOL associated with the network. We now discuss VOLT's programming model to express BSMs.

Programming model. VOLT supports a programming model akin to Ethereum's smart contracts for expressing blockchain state machines Ψ. Ψ consists of (Ψ_c, Ψ_s), where Ψ_s is the internal state of a BSM and Ψ_c is the code that mutates Ψ_s by processing a state machine command (i.e., a blockchain transaction). Ψ_c is expected to be single threaded and deterministic. It however gets to invoke VOLT-provided cryptographic libraries (to perform digital signature checks, hashing, etc.), and it gets access to a simple key-value store to persist its internal state.[7]

3.3 Blockchain service provider (BSP)

The BSP uses a VOL to represent a blockchain network i.e., it records the initial state of the network and all transactions processed by the network on the same VOL. The BSP exposes the following three operations to member nodes: `Setup`, `Transact`, and `Sync`.

Interface. `Setup` takes information necessary to initialize a new blockchain network—blockchain state machine, list of member nodes, etc. It returns a new genesis block to the calling member node, which contains the setup information as well as the identity for the blockchain network just created. The genesis block is signed by the BSP so that the calling member node can distribute it to other member nodes and that they can verify the authenticity of the block using the BSP's public key.

`Transact` accepts the identity of a blockchain network and a signed transaction, which is a command that mutates the internal state of the blockchain network. The BSP validates the transaction using the blockchain code associated with the blockchain network, and appends it to the VOL associated with the blockchain network. For higher throughput, the BSP batches a configurable number of transactions in a single block.

`Sync` accepts the identity of the blockchain network and returns the VOL associated with the blockchain network. For better performance, `Sync` accepts an additional parameter that enables the member nodes to specify parts of the VOL that they have already downloaded, so that the BSP can simply return subsequent additions to the VOL.

Implementation. The above interface is straightforward to implement with a stateful process. Section 4 presents a modular design for the BSP.

3.4 Caesar consensus

While the BSP takes care of maintaining a VOL, all honest members must have a consistent view of the VOL i.e., they must all see the same set of blocks in the same order. We refer to this special form of agreement *Caesar consensus*. Unlike traditional consensus, our protocol involves not only n member nodes, but also the BSP, which is mostly correct and highly resourceful. It is not a general consensus either, but specifically for reaching agreement on an append-only, tamper-resistant ledger. It leverages the BSP to enable honest members to reach agreement as well as ensure that a "committed" prefix of a VOL is preserved—even if the BSP misbehaves arbitrarily.

We start by describing the normal-case Caesar consensus protocol, and then show how to configure our protocol to handle a wide range of fault models and consistency semantics. In particular, we show how to configure the protocol so that it can be mapped to a standard Byzantine fault tolerance (BFT) protocol such as PBFT [28].

Normal-case protocol. A key differentiating aspect of our Caesar consensus protocol is that it leverages the VOL abstraction, where the actual ledger is stored on the BSP and locally on each member node. As a result, the normal case of our protocol is deceptively simple. It just involves a special heartbeat transaction $(m, g, h)_\sigma$, where m is the identity of a member node (i.e., its public key), g is the *height*[8] of the VOL validated by the member node, h is the cryptographic hash of the metadata block at height g, and σ is a digital signature using m. The VOL at the current height is deemed *valid* iff

- The VOL is "well-formed": there must be a sequence of blocks where the first one is the genesis block and every subsequent block points cryptographically to the previous one.
- Every transaction included in each block is valid according to the associated blockchain state machine.

[7]We leave it as future work to enforce these requirements on Ψ_c via program verifiers such as Dafny [2, 57] and to support existing smart contract languages such as Solidity [10].

[8]The height of a VOL is the height of the last block in the VOL, which is its distance away from the genesis block of the VOL.

- For each heartbeat transaction $(m, g, h)_\sigma$ included in a block of the VOL, the cryptographic hash h matches the hash of the chain at height g.

In the normal case of our Caesar consensus, each member node m executes the following steps periodically:

1. Call the BSP's `Sync` API to learn the latest additions to the VOL, the current height g, and the cryptographic hash (h) of the metadata block at height g.
2. Check if the VOL remains valid with the latest additions, as described earlier. If the VOL is no longer valid, the BSP is considered compromised and results in a failover (§3.5).
3. Create a signed heartbeat transaction $(m, g, h)_\sigma$. Post the transaction to the BSP using the `Transact` API.
4. Compute the committed prefix of the VOL (i.e., committed height g') by processing heartbeat transactions of other member nodes included in the VOL, according to a policy. We elaborate this next.

The normal-case Caesar consensus protocol essentially uses heartbeat transactions as messages typically exchanged in a consensus protocol. Doing so brings several benefits. The protocol leverages the existing `Transact` API and the VOL, and is therefore straightforward to implement. More importantly, because all the heartbeat transactions are recorded in the VOL, we can easily instantiate different types of consensus (under different failure assumptions) by defining only the condition under which a prefix of the VOL is considered committed. For example, a member node might trust secure execution environments [78, 92] at other member nodes, so it might be satisfied with a crash-only failure assumption and consider a transaction t committed as soon as it sees heartbeat transactions validating t from a majority of members.

In fact, these policies can be configured on a per-member and per-transaction basis based on their respective risk and performance profiles. As an example, for a low-value transaction, a member node might be willing to consider it committed as soon as it gets logged in the VOL (so that it can respond immediately), but it brings a risk: such a transaction can disappear in the (unlikely) event that the BSP is compromised. For a high-value transaction, a member node might instead require that every other member node validate the transaction and then specify that they have done so via a heartbeat transaction, which might increase latency. This additional flexibility is nevertheless practically valuable and even critical in some business applications of blockchains.

It is worth noting that the standard practice (as typically used in permissioned blockchains such as Hyperledger [5]) requires member nodes to run a Byzantine fault tolerance (BFT) protocol on proposed transactions before committing them to a ledger. Such a design cannot accommodate the type of flexibility Caesar consensus offers because the policy is hardwired upfront in the BFT protocol in those approaches. Executing BFT outside of a ledger also adds complexity, which must be trusted.

Achieving Byzantine fault tolerance. BFT protocols such as PBFT [28] tolerate up to f Byzantine faults with $n = 3f + 1$ replicas [28]. The normal-case protocol involves a primary initiating the protocol with a `pre-prepare` message, followed by two rounds of multicast, one with `prepare` messages and another with `commit` messages, before committing a proposal. A replica must receive messages from at least $2f + 1$ replicas (including itself) in each round before proceeding. Correctness hinges on the fact that there exists at least one correct replica at the intersection of any two sets of $2f + 1$ replicas.

We can configure Caesar consensus with $n = 3f + 1$ member nodes to tolerate up to f Byzantine faults. We can then map the heartbeat transactions on a valid VOL to messages in PBFT, as follows. A block b on the VOL at height g is equivalent to a primary in PBFT multicasting a `pre-prepare` message for value b in the consensus instance for height g. A heartbeat transaction $(m, g_m, h)_\sigma$ with $g_m \geq g$ serves the purpose of a `prepare` message from member node m for value b at height g because it implicitly echoes this decision in the VOL.

Let g_p be the lowest height at which there exist heartbeat transactions $(m, g_m, h)_\sigma$ with $g_p \geq g_m \geq g$ from $2f + 1$ member nodes. At this point, block b at height g is considered prepared. A heartbeat transaction $(m, g_m, h)_\sigma$ with $g_m \geq g_p$ serves the purpose of a `commit` message from member node m for value b at height g because it implicitly echoes a `prepare` message from $2f + 1$ member nodes. Let g_c be the lowest height at which there exist heartbeat transactions $(m, g_m, h)_\sigma$ with $g_c \geq g_m \geq g_p$ from $2f + 1$ member nodes m. At this point, block b at height g is considered committed.

For simplicity, we present the mapping to BFT from the perspective of a single consensus instance in BFT i.e., in our mapping, each height on the VOL corresponds to a consensus instance in BFT. However, each heartbeat transaction actually participates in many consensus instances simultaneously, since each heartbeat transaction validates a linearly-expanding prefix. So if the VOL has a block prepared (or committed) at height g, then all blocks at height less than g are already prepared (or committed). Therefore, given the current VOL at a certain height, there exists a highest prepared (or committed) height g, at which a block is prepared (or committed) at any height $g' \leq g$. We refer to this height as the current prepared (or committed) point. Figure 3 depicts an example.

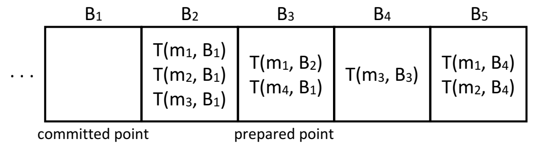

FIGURE 3—Overview of Caesar consensus. We use $T(m_i, B_j)$ to denote a heartbeat transaction from a member node m_i for block B_j (§3.4). Suppose there are four member nodes and they use BFT with $f=1$. Observe that $2f+1$ members (i.e., m_1, m_2, m_3) have sent heartbeat transactions for block B_3 or B_4, thus all blocks prior to and including B_3 are `prepared`. To find the committed block with the heighest height, members run the same algorithm assuming that the VOL ends at the highest prepared height. This ensures that at least $2f+1$ members have witnessed the block at the highest prepared height. Thus, all blocks up to and including B_1 are `committed`.

3.5 The role and accountability of the BSP

Recall that the BSP is often instantiated in a cloud, which is resourceful and hardened in terms of reliability, availability, and security. Thus, when the BSP is correct, Caesar consensus executes efficiently.

We also make the BSP accountable, as in the general accountability frameworks [45, 46, 89]. In our case, this task is simpler due to the structure of a VOL (§3.2). We also want to ensure that faulty member nodes cannot frame the BSP when the BSP is correct. We consider accountability in terms of safety, fairness, and liveness.

The BSP can violate a safety guarantee e.g., by creating a fork in a VOL by assigning two different blocks to the same height. (It can also create an ill-formed VOL, which is trivial to detect.) The existence of any such fork signed by the BSP provides sufficient evidence to blame the BSP because no member nodes can forge the BSP's signature. Also, observe that member nodes in VOLT cannot create a fork as long as the BSP is correct.

In addition to safety violations, the BSP can also violate fairness or liveness e.g., by unfairly dropping or delaying transactions from certain member nodes, or by preventing transactions from being added to the VOL. Doing so could also help evade safety-violation detection by trying to maintain forked views among correct member nodes. For fairness, a member node with trouble getting its transactions accepted by the BSP can broadcast its transactions to all other member nodes who can then piggyback them with their transactions. Thus, unless the BSP drops transactions from all correct member nodes (which turns a fairness violation to a liveness violation), any discrimination against a correct member node will eventually be detected by correct member nodes.

It is hard to detect liveness violations without additional synchrony assumptions. Any member node experiencing unreasonably long delays for getting its transaction accepted (even after a broadcast) can file a signed complaint with other member nodes. If $f+1$ members complain, the BSP is malicious, since at least one of those complaints is by a correct member node.

Changing the BSP. When the BSP is considered compromised with cryptographically verifiable blames, member nodes can replace the BSP with a new one. However, in practice, this a rare event if the BSP is hosted on a reputable cloud provider. To switch to a new BSP, there are several options. The new BSP could be predetermined when the blockchain network is set up, be introduced by a trusted authority when needed, or be selected in a consensus protocol run by the member nodes. Note that the committed state of a ledger must be preserved when transferring to a new BSP.

When member nodes use the above BFT setup, we can simply use BFT's view change protocol. We sketch how we adapt view changes in our context, and leave the full description and correctness proof to a technical report due to space constraints. We first associate a view number with each BSP, which is included in every heartbeat transaction. A new BSP will always have a higher view number that its predecessor, and in Caesar consensus, member nodes stop sending heartbeat transactions in older views (i.e., to an old BSP) after getting a view change message with a higher view number.

During a view change, the new BSP starts by gathering local state (i.e., a copy of VOL) from $2f+1$ member nodes. Since the previous BSP might have been compromised, VOLs from the member nodes might not be consistent (where consistency is defined below), the new BSP must construct a consistent VOL.

Definition 3.1. Two VOLs c and c' are *consistent* iff, for $0 \leq i \leq min(g_c, g_{c'})$, block i on c is the same as block i on c'. Here, g_c and $g_{c'}$ are the heights of c and c'.

Constructing a consistent VOL. Let g_n be the highest prepared point of valid VOLs from a set C of $2f+1$ member nodes. The new BSP then chooses the VOL up to height g_n, which we now show is correct. First, the VOL constructed by the BSP is consistent with every valid VOL received because each block up to g_n is prepared, and is therefore validated by a set C' of $2f+1$ member nodes via heartbeat transactions. Thus, there will not be two conflicting blocks at the same height such that both are prepared because a correct member node in the intersection of C and C' would have validated both. Second, every committed block in the prior network will be included in the VOL constructed by the BSP. This is because, by definition, any committed block b must have been prepared at a set of C'' of $2f+1$ member nodes. Furthermore, there exists at least one correct member node in the intersection of C and C'', so block b must be before g_n.

4 Architecting a trustworthy BSP

This section discusses how to architect the BSP so that it is mostly trustworthy.[9] Our approach is to decompose the functionality of the BSP into modular, well-defined components, each with narrow responsibilities and interfaces. Such a modular design makes it possible to harden each component of the BSP separately for performance and security, with different mechanisms.

The BSP's functionality includes: storage of blocks, processing transactions using Ψ_c to construct new blocks, and chaining them to construct a blockchain. We now discuss a separate component for each of these responsibilities as well as a coordinator that ties them together.

4.1 Verifiable store with receipts ($\mathcal{S}$)

The first component of our BSP is a storage system that relies on a well-known substrate in most untrusted storage systems [41, 59, 62]: a storage system where data items are named by their cryptographic hashes. This enables verifiability i.e., a client of such a system can locally check if it got correct data from the system by comparing the name of the data with the cryptographic hash of the returned data. $\mathcal{S}$ stores both data blocks and metadata blocks of a VOL.[10]

Additionally, $\mathcal{S}$ provides a *storage receipt* when someone stores a blob of data in it. The purpose of a storage receipt is to enable $\mathcal{S}$ to convey—over an untrusted channel—to entities (that may trust $\mathcal{S}$ for availability and liveness) that $\mathcal{S}$ is responsible for storing a data item. $\mathcal{S}$ owns a key pair of a digital signature scheme, (PK_s, SK_s),[11] and it exports the following interface.

- `Deposit(Blob b)`: stores b in $\mathcal{S}$ and returns a tuple (h, σ), where $h = H(b)$ and $\sigma = \text{Signature}(SK_s, b)$.
- `Retrieve(Hash h)`: returns b such that $h = H(b)$

4.2 Chaining service ($\mathcal{C}$)

$\mathcal{C}$ is responsible for creating new VOLs as well as for maintaining a linear history of blocks of a VOL, stored in $\mathcal{S}$. Perhaps surprisingly, we find that $\mathcal{C}$ only needs to maintain a constant amount of state to accomplish these two tasks, making it easier for a cloud provider to implement $\mathcal{C}$ correctly. $\mathcal{C}$ holds keys of a digital signature scheme (PK_c, SK_c). It exposes the following operations (Figure 4 depicts the corresponding pseudocode).

- `Init`: assigns an identity for the blockchain network.
- `Chain`: appends a new block stored at $\mathcal{S}$ to a previously initialized VOL.

```
def Init(Hash h, Receipt s):
  if verifyReceipt(PK_s, h, s) == false:
    return "Storage receipt check failed"
  id = random() # random 128-bit number
  if state[id] != null:
    return "Internal error; retry"

  state[id] := (id, h, null, s)
  return (state[id], signature(SK_c, state[id]))

def Chain(Guid id, Hash h, Receipt s, Receipt r):
  if state[id] == null:
    return "Uninitialized VOL"

  if verifyReceipt(PK_s, state[id], r) == false:
    return "Storage receipt check failed"
  if verifyReceipt(PK_s, h, s) == false:
    return "Storage receipt check failed"

  state[id] = (id, h, H(state[id]), s, r)
  return (state[id], signature(SK_c, state[id]))
```

FIGURE 4—Pseudocode for VOLT's chaining service ($\mathcal{C}$) internals. `Init` initializes a new VOL and assigns a random identity, which $\mathcal{C}$ remembers in the `state` map. `Chain` appends a new block to an already initialized VOL. Note that $\mathcal{C}$ relies on $\mathcal{S}$'s receipts to ensure that it does not append a block not already stored by $\mathcal{S}$. Of course, if $\mathcal{S}$'s signing key is compromised, $\mathcal{C}$ can end up appending a block not stored by $\mathcal{S}$.

If $\mathcal{C}$ does not lose the `state` map (Figure 4) and SK_c, then it maintains the *no-fork* property: for every metadata block m produced by $\mathcal{C}$, there exists at most one metadata block m' such that $\texttt{predecessor}(m') = H(m)$, where `predecessor` returns the second field in a metadata block triple (§3.2).

4.3 State transition service ($\mathcal{T}$).

$\mathcal{T}$ is responsible for executing blockchain code to processing transactions. It holds keys of a digital signature scheme (PK_t, SK_t), and it exposes the following API.

- `Init(Guid id, Blob b)`: accepts the identity of a VOL and a blob containing blockchain code and initial state, and setups environment to process transactions.
- `CreateBlock(Guid id, list<Transaction> l)`: validates each transaction in l using Ψ_c and generates a new block along with a *computation receipt* signed by SK_t. The computation receipt contains a signature on the accepted transactions as well as the hash of the prior list of transactions processed by $\mathcal{T}$.

4.4 Coordinator ($\mathcal{R}$)

$\mathcal{R}$ implements the BSP's interface (§3.3). It consists of a Web server that receives HTTP requests from member nodes, coordinates various components of the BSP, and responds back to member nodes. $\mathcal{R}$ is different from the

[9]If a cloud provider offers the BSP as a commercial cloud service (e.g., Azure BaaS [7], IBM blockchain [6]), then the cloud provider has interest to make sure its cloud service is trustworthy.

[10]An alternate design: $\mathcal{S}$ issues storage receipts only for the tail of a VOL. But this requires $\mathcal{S}$ to understand the structure of a VOL).

[11]We assume that other entities relying on $\mathcal{S}$ already know PK_s.

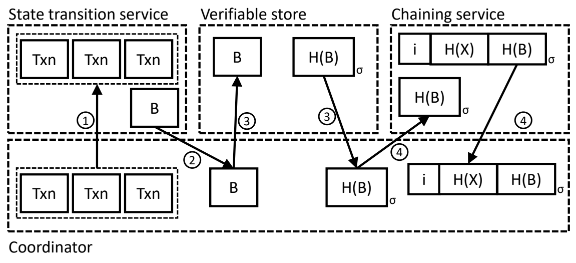

FIGURE 5—Life of a transaction in VOLT. ① The coordinator ($\mathcal{R}$) batches transactions and sends them to the state transition service ($\mathcal{T}$). ② $\mathcal{T}$ responds to $\mathcal{R}$ with a block B. ③ $\mathcal{R}$ stores B on the verifiable store with receipts ($\mathcal{S}$), which issues a storage receipt. ④ $\mathcal{R}$ sends the storage receipt to the chaining service, which appends B to a VOL and creates a new metadata block.

other services in that its correctness does not affect the BSP's correctness. It can only affect liveness. Figure 5 depicts how $\mathcal{R}$ implements `Transact`.

4.5 Discussion

We now discuss rationale for our design. $\mathcal{T}$'s computation receipt constructs a cryptographically chained sequence of transactions grouped into blocks. However, $\mathcal{T}$ runs code submitted by member nodes, so any exploitable bug in that code can potentially output a computation receipt that creates a fork in a VOL. Of course, Caesar consensus will detect such safety violations. But, it is prudent from the perspective of the cloud operator running the BSP to avoid such safety violations. $\mathcal{R}$ can prevent such violations, but we ask: is there a smaller component that can ensure the BSP never creates a fork? Our answer is the chaining service. We focus on designing a minimal component so the BSP can ensure safety with a small trusted computing base (TCB).

5 Implementation

We implement the BSP in 4,800 lines of C# and the VOLT library in 1,500 lines of Python. We build the BSP atop Azure service fabric as it simplifies creating reliable cloud services. We architect each component of the BSP (§4) as a microservice, so the fabric runtime takes care of isolating these microservices, recovering from failures, and load balancing across a cluster of machines. The microservice that implements verifiable store with receipts ($\mathcal{S}$) uses Azure DocumentDB, which is a NoSQL key-value store (akin to Amazon DynamoDB) to persist its state reliably. Also, the coordinator microservice exposes the BSP's interface (§3.3) to member nodes as REST APIs. Finally, our prototype uses SHA-256 for a hash function, and RSA for digital signatures.

Partitioned ledgers. Our design focuses on how to initialize and update a single VOL. An inherent bottleneck is that an application's throughput is limited by the number of entries the BSP can append to single VOL per second. We mitigate this problem by employing the well-studied partitioning technique, which naturally extends to our context. In particular, applications in VOLT create multiple VOLs where each VOL is responsible for a shard of the application state. Thus, two transactions that do not mutate the same shard of the application state can be handled in parallel without having to serialize them.

This raises the question: how do member nodes achieve a consistent view of multiple VOLs? Each heartbeat transaction is now a vector of entries, with an entry for each VOL. But, the rest of Caesar consensus works as before. Also, for simplicity, VOLT uses a separate VOL to store all the heartbeat transactions of a blockchain network.[12]

6 Experimental evaluation

Our experimental evaluation answers the following questions. First, what is the concrete performance of VOLT? Second, how long does it take for blocks to commit under Caesar consensus? Third, what are the resource costs imposed on participants of a VOLT blockchain network?

Experimental setup. We deploy the BSP on a cluster of ten machines on Microsoft Azure cloud, each with the following configuration: Windows Server 2016 running on Azure Standard_F2s VMs (i.e., an Intel Xeon E5-2673 v3 processor, 2 cores, 4 GB RAM, and 8 GB local storage). In our experiments, we reserve 250 GB for storage and 10,000 request units per second on Azure's DocumentDB for $\mathcal{S}$, which allows a few thousand operations per second, with medium-sized (i.e., a few KBs) blobs.

6.1 Benchmark applications

We implement two applications to evaluate VOLT.

(1) Centrally-issued cryptocurrency. The first application is a realistic application atop blockchains. It is a simple cryptocurrency where coins are issued by trusted entities, which is similar in spirit to applications in prior works [24, 37]. In this application, member nodes are financial institutions with the ability to mint coins. This easily extends to other digitally-issued assets [86].

The internal state of the blockchain state machine, Ψ_s, includes the list of public keys that can issue currency, and account balances of users identified by public keys. The blockchain code, Ψ_c, processes two types of transactions. Member nodes can create currency by creating an `issue` transaction, which provides coins to a user, and then a user can spend coins by creating a `spend` transaction, which transfers coins from one public key to another. To prevent an adversary from replaying transactions, we employ a standard technique: transactions include monotonically increasing per-user sequence numbers.

[12]This could reduce the overall parallelism, but we expect that heartbeat transactions are sent less frequently compared to other transactions.

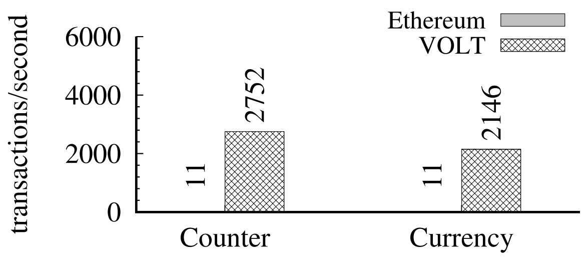

FIGURE 6—Transaction processing throughput of VOLT and Ethereum for two benchmark applications. VOLT does not rely on mining, so its throughput is orders of magnitude higher, but is comparable to BFT-based approaches.

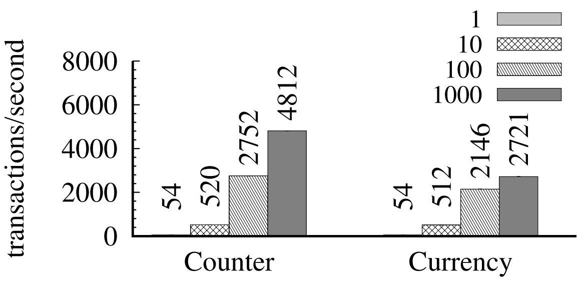

FIGURE 7—Throughput of VOLT's `Transact` with varying batch sizes. We find that at VOLT achieves peak throughput with batch sizes between 100–1000. Beyond 1000 transactions in a block (not depicted), throughput benefits due to batching reduce.

(2) Counter. Ψ_s for this application is simply a 64-bit integer, which is incremented during each transaction.

6.2 Performance of the BSP

To understand the performance of the BSP, we create blockchain networks initialized with the above applications. Our primary performance metrics include throughput (in transactions per second) and latency (in seconds) as experienced by member nodes.

Throughput. To measure VOLT's throughput for processing transactions, we use a client process that submits transactions to a given blockchain network using the BSP's `Transact` API: the client process creates multiple threads where each thread acts as a member node and submits transactions in a closed loop. We then measure the number of transactions processed per second by the BSP and the latency for each transaction from the perspective of the client process. We configure the BSP to batch 100 transactions into each data block of a VOL (we report the effect of varying batch sizes below). To determine BSP's peak throughput, we progressively increase the number of threads in the client process. In our experiments, the client process submits at least 10,000 transactions. As a baseline, we implement our benchmark applications in Solidity [10], and deploy a private Ethereum network using Azure Blockchain as a Service (BaaS) [7].

Figure 6 depicts our results. We must take this comparison to Ethereum with a grain of salt, since Ethereum and VOLT are based on completely different mechanisms. However, VOLT's throughput is several orders of magnitude higher than the throughput of Ethereum. The primary computational costs in VOLT include validating transactions to construct a block, reliably storing a block in $\mathcal{S}$, generating storage receipts, and sequencing blocks into a VOL using $\mathcal{C}$. On the other hand, Ethereum must also solve computational puzzles in addition to doing most of the above tasks. Finally, note we present the performance of a single VOL on a 20-core cluster. The throughput increases nearly linearly with multiple VOLs operating in parallel, with more cores and storage resources allocated to the BSP. We can improve the throughput of a single VOL with faster signatures [20]. Nevertheless, our current BSP scales to realistic workloads [36].

Throughput with varying batch sizes. To understand the effect of the number of transactions inside a data block (i.e., a batch size) on the throughput of the BSP, we run experiments where we vary the batch sizes from 1 to 1,000 in powers of 10. Figure 7 depicts our results.

We make two observations. First, increasing the batch size reduces the per-transaction overheads of the BSP. Thus, the throughput of the BSP increases with the batch size. Second, we find that the BSP's throughput is bottlenecked by signature operations it executes. Even though our two benchmark programs have vastly different execution times (e.g., the currency application executes a signature check whereas the counter application executes a simple increment), the overall throughput at smaller batch sizes is similar for both applications. As we increase batch sizes beyond 100, the difference in throughput is visible.

Latency. We also measure the latency of `Transact` as well as `Setup` for both benchmark applications as experienced by a member node. Note that this does not represent commit latencies for a transaction; we report commit latencies in the next subsection.

To measure the `Setup` latency, we run an experiment in which a client process executes `Setup` in a closed loop at least 10,000 times. The client then picks one of those networks and submit transactions in a closed loop. We configure the BSP to batch 100 transactions in a single data block. We measure latency of these operations as experienced by the client process using wall-clock time. (We run the client process in the same datacenter as the BSP to avoid the wide-area latency from dominating the BSP's latency.) Figure 8 reports average latencies. As expected, VOLT's operations achieve sub-second latencies (`Setup` does not involve batching, but it must compile blockchain code). The `Transact` latencies can be lowered with smaller batch sizes, at the cost of reducing the BSP's throughput (Figure 7).

operation	Counter	Currency
`Setup`	0.13 s	0.15 s
`Transact`	0.06 s	0.24 s

FIGURE 8—Latency of `Setup` and `Transact` under VOLT for the two benchmark applications.

6.3 Commit latency in Caesar consensus

VOLT's *commit latency*—time it takes for a transaction processed by the BSP to be committed via Caesar consensus (§3.4)—depends on various parameters: number of member nodes, frequency with which member nodes synchronize with the BSP using `Sync` and send heartbeat transactions, failure modes of participants, etc. To keep the effects of these parameters easy to understand, we fix the number of member nodes to be 4, and create a blockchain network initialized with the Counter application (we set the batch size for the network to be 1).

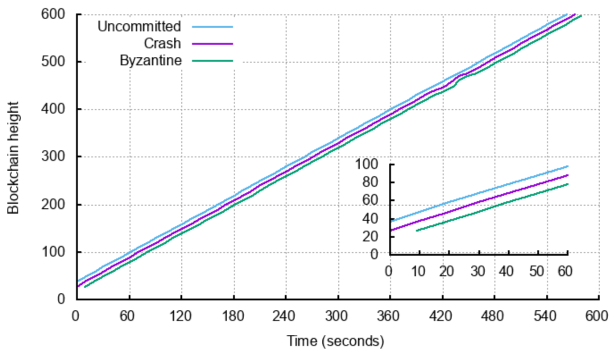

FIGURE 9—Commit points under different failure modes as well as uncommitted blockchain height over time under VOLT's Caesar consensus (see the text for details).

We then have each member node execute the Caesar consensus protocol every epoch, which we set to 5, 10 and 15 seconds in our experiments.[13] We also have one of the member nodes submit a transaction every second.

We run these experiments for 10 minutes and at the end of each epoch, each member nodes outputs commit points. Since VOLT allows flexible policies to determine commit points, we compute commit points for both Byzantine and crash failure modes. Figure 9 depicts our results when the time epoch is 10 seconds (the results for other experiments are similar). As we expect, every 10 seconds, member nodes sync and send heartbeats, so the commit point under crash failures is roughly 10 blocks away from the uncommitted height since we add a block to the VOL every second. Since the Byzantine failure model requires two phases to commit, commit points are 10 blocks away from that of the crash failure mode.

[13]We choose these time epochs based on Ethereum's default block-generation frequency, which is 15 seconds.

CPU costs	
generate heartbeat (1 VOL)	4.0 ms
generate heartbeat (2 VOLs)	7.3 ms
generate heartbeat (4 VOLs)	18 ms
verify heartbeat (1 VOL)	0.15 ms
verify heartbeat (2 VOLs)	0.17 ms
verify heartbeat (4 VOLs)	0.20 ms
sync VOL (100 blocks, batch=100)	1.41 s
sync VOL (200 blocks, batch=100)	3.14 s
sync VOL (400 blocks, batch=100)	7.22 s
validate VOL (100 blocks, batch=100)	0.9 s
validate VOL (200 blocks, batch=100)	1.7 s
validate VOL (400 blocks, batch=100)	3.3 s
Storage costs	
metadata block	1043 bytes
data block (batch=100)	7495 bytes
Network costs	
send heartbeat (1 VOL)	638 bytes
send heartbeat (2 VOLs)	741 bytes
send heartbeat (4 VOLs)	947 bytes

FIGURE 10—Resource costs of members in a VOLT network for the counter application.

6.4 Resource costs on participants

Resource costs that member nodes of a VOLT network incur depends on various parameters: how often member nodes participate in Caesar consensus, how many transactions are submitted to the network, etc. We thus run a set of microbenchmarks where we measure the CPU, storage, and network costs of actions that member nodes take. Figure 10 summarizes our results. As we expect, member nodes spend little CPU-time (even with our Python code) and storage costs (even when they store the entire VOL and validate it). Heartbeats use JSON encoding, which has a fixed overhead that does not increase significantly as we add more VOLs to a blockchain network.

7 Related work

We discuss closely related works in Sections 1 and 2. We now discuss other related works. (Bonneau et al. [22] provide an excellent survey of many of these works.)

Permissionless blockchains. Following the rise of cryptocurrencies [69, 85], most works on blockchains focus on a permissionless membership model, allowing anyone to participate. Since Sybil attacks are easy to mount in this model, consensus requires incentives (provided in the form of a cryptocurrency by the protocol) in addition to proof-of-work [14, 39, 53] (or alternate models such as proof-of-stake [18, 19, 52, 90], or proof-of-elapsed-time [9], or randomness [66]). As a result, the security guarantees of these systems are based on incentives and game theory. Compared to these works, VOLT targets enterprise applications where entities have well-known identities and do not require an open, permissionless model.

Untrusted storage. A line of work with a strong influence on VOLT is untrusted storage [23, 27, 41, 59, 62, 76, 81], which use hash chains to prevent untrusted participants in the system from tampering with operation histories. These systems target high availability and low latency (e.g., they limit client-to-client coordination), so they only provide weaker consistency guarantees such as variants of fork consistency [65]. Since VOLT targets mission-critical applications with financial implications (§2), VOLT prefers stronger safety and higher throughput over low latency and high availability. Thus VOLT enforces linearizability, with threshold assumptions on member nodes (which might be realistic in our context).

While these systems use hash chains to implement various storage interfaces such as file systems [59], key-value stores [41, 62, 76, 81], VOLT is more general since it exposes the abstraction of a VOL, which it uses to store a general-purpose state machine (i.e., a BSM) and transactions. Cachin proposes a similar abstraction atop a ledger, but retains the weaker fork-linearizability [26].

VOLT enforces linearizability via Caesar consensus, which relies on a mechanism similar to heartbeats in SUNDR [59] and beacons in Depot [62]. In SUNDR and Depot, these messages are used to detect forks upon which nodes reconcile histories and to enforce freshness guarantees. However, two clients can have views of the system that are inconsistent with each another at the same time. Whereas, VOLT's member nodes declare a transaction as committed only when they are guaranteed to be preserved even with a BSP failover (§3). Thus, any two honest member nodes will always have a consistent view of a VOL at all times. Finally, we investigate how to architect the BSP for a smaller TCB, which is not covered by these systems. A2M [30] and Trinc [58] explore small trusted primitives for building trustworthy distributed systems, similar to our chaining service, but they do not propose end-to-end checks as in Caesar consensus.

BFT. Following PBFT [28], there is a long line of work to improve performance [48, 55, 56, 63], robustness [29, 33], confidentiality [87, 88], and fault models [15] of BFT protocols. But, these protocols and systems were developed for replicating mission-critical services in datacenters. If we apply them to blockchains, they face trust choices similar to Hyperledger (§1). Due to commercial interest in blockchains, there are several recent works that improve on these existing BFT protocols in the context of blockchains, but still retain the deployment model. Sleepy consensus [75] loosens the definition of a correct node, providing consensus guarantees even if a node is only online sporadically. Miller et al. [67] describe a new BFT protocol that makes progress in adversarial network conditions such as the Internet. Stellar [64] proposes a new Byzantine agreement protocol that works in a federated model, but it includes a more complicated trust relationships among participants since it studies a federation of multiple Byzantine fault-tolerant systems.

Other substrates. Several works [11, 91] propose using secure hardware for trusted data injection into blockchain systems. VOLT can naturally incorporate these ideas. Virtual chain [73] proposes a layer atop blockchains (like Caesar consensus), but only provides fork* consistency. Many works [16, 21, 31, 84] propose techniques to reduce equivocation by entities that generate hash chains to the security of a public blockchain. This mechanism can replace Caesar consensus, but it requires members to trust public blockchains. We can however leverage it to limit the BSP's misbehavior when member nodes are offline.

Finally, CoSi [83] enables a group of entities to create succinct signatures on a given statement. It can be used in VOLT to compress heartbeat messages. However, it requires increased coordination between member nodes and the BSP to generate a signature. Furthermore, the BSP must know how many signatures it has to collect and what policies member nodes follow to commit transactions. All of these are oblivious to the BSP in our current design. Thus the use of CoSi increases the complexity of the BSP.

8 Discussion

Flexibility. Our prototype implements the BSP abstraction in the cloud, but it can also be deployed in a decentralized manner on the infrastructure of member nodes. The BSP is still untrusted (by design), so it reduces the amount of trusted code in a blockchain system compared to systems such as Hyperledger [5] and Tendermint [24]. Additionally, while we describe an instantiation of Caesar consensus that provides guarantees assuming Byzantine members, our approach generalizes to other failure models (e.g., crashes) and consistency semantics (e.g., eventual consistency [82]). This is possible since VOLT records all heartbeats in a ledger (§3.4).

Evolving the BSP, membership, and state machines. Member nodes in VOLT only rely BSP's interface, so the BSP can change the underlying implementation anytime, without any approval from member nodes.

For each network, VOLT assumes a static membership list and a blockchain state machine. In principle, VOLT can allow member nodes to update them: we can treat them as additional state agreed-upon in a network. We posit that member nodes can reach consensus on such state updates using a protocol similar to Caesar consensus, without having to significantly change the BSP.[14]

Fairness. A key concern in blockchain systems is *fairness*, which requires that malicious entities should not be able to censor transactions of certain users. Permissionless

[14]Member nodes can use policies to prevent malicious participants from misusing such state updates e.g., adding a new member node requires approval from a quorum of existing members.

blockchains use incentives and transaction fees to achieve (weak) fairness properties. VOLT's BSP can employ secure enclaves for $\mathcal{T}$ so that it does not see transactions in plaintext, making it hard for the BSP to discriminate against certain transactions. Note however this is not a panacea: While we can try to hide the transaction data and some metadata (via enclaves or cryptographic methods [17, 54]), the BSP can always filter transactions based on any metatdata that can be inferred via collusion with malicious member nodes, a point made by Herlihy and Moir [49] who also propose a scheme to improve fairness in permissioned blockchains via accountability [46, 89]. VOLT can incorporate these ideas (§3.5).

Lock-in. While VOLT's implementation relies on a BSP (which may be operated by a cloud provider) for performance (§3), it is only an optimization. Indeed, the BSP's interface exposes a tamper-resistant ledger that can be validated in a decentralized manner (using code submitted by members of a blockchain network). Furthermore, the VOL itself is amenable to porting to a new BSP or any other permissioned blockchain system (§3.4).

9 Conclusion

This paper studies the problem of designing a permissioned blockchain for enterprise applications. We leverage the cryptographic properties of a tamper-resistant ledger to design a simple and resource-efficient consensus mechanism. The result is a permissioned blockchain system that includes the benefits of decentralization, while leveraging untrusted infrastructure.

References

[1] Bitcoin's scalability targets. `https://en.bitcoin.it/wiki/Scalability#Scalability_targets`.

[2] Dafny: A language and program verifier for functional correctness. `https://www.microsoft.com/en-us/research/project/dafny-a-language-and-program-verifier-for-functional-correctness/`.

[3] Ether historical prices (USD). `https://etherscan.io/chart/etherprice`.

[4] Ethereum average gasprice chart. `https://etherscan.io/chart/gasprice`.

[5] Hyperledger – Blockchain for Business. `https://www.hyperledger.org/`.

[6] IBM Blockchain. `https://www.ibm.com/blockchain/`.

[7] Microsoft Azure Blockchain as a Service (BaaS). `https://azure.microsoft.com/en-us/solutions/blockchain/`.

[8] Mining – Bitcoin Wiki. `https://en.bitcoin.it/wiki/Mining`.

[9] Sawtooth Lake. `https://intelledger.github.io/`.

[10] Solidity. `https://solidity.readthedocs.io/en/develop/`.

[11] The Cryptlet Fabric. `https://github.com/Azure/azure-blockchain-projects/blob/master/bletchley/CryptletsDeepDive.md`.

[12] Transaction – Bitcoin Wiki. `https://en.bitcoin.it/wiki/Transaction`.

[13] Trust, confidence and verifiable data audit. `https://deepmind.com/blog/trust-confidence-verifiable-data-audit/`.

[14] I. Abraham, D. Malkhi, K. Nayak, L. Ren, and A. Spiegelman. Solidus: An incentive-compatible cryptocurrency based on permissionless Byzantine consensus. *CoRR*, abs/1612.02916, 2016.

[15] A. S. Aiyer, L. Alvisi, A. Clement, M. Dahlin, J.-P. Martin, and C. Porth. BAR fault tolerance for cooperative services. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)*, pages 45–58, 2005.

[16] M. Ali, J. Nelson, R. Shea, and M. J. Freedman. Blockstack: a global naming and storage system secured by blockchains. In *Proceedings of the USENIX Annual Technical Conference (ATC)*, pages 181–194, 2016.

[17] E. Ben-Sasson, A. Chiesa, C. Garman, M. Green, I. Miers, E. Tromer, and M. Virza. Zerocash: Decentralized anonymous payments from Bitcoin. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 459–474, 2014.

[18] I. Bentov, A. Gabizon, and A. Mizrahi. Cryptocurrencies without proof of work. In *Proceedings of the International Financial Cryptography and Data Security Conference*, pages 142–157, 2016.

[19] I. Bentov, R. Pass, and E. Shi. Snow white: Provably secure proofs of stake. Cryptology ePrint Archive, Report 2016/919, 2016.

[20] D. J. Bernstein, N. Duif, T. Lange, P. Schwabe, and B.-Y. Yang. High-speed high-security signatures. *Journal of Cryptographic Engineering*, 2(2):77–89, 2012.

[21] J. Bonneau. EthIKS: Using Ethereum to audit a CONIKS key transparency log. In *Proceedings of the International Financial Cryptography and Data Security Conference*, pages 95–105, 2016.

[22] J. Bonneau, A. Miller, J. Clark, A. Narayanan, J. A. Kroll, and E. W. Felten. SoK: Research Perspectives and Challenges for Bitcoin and Cryptocurrencies. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 104–121, 2015.

[23] B. Braun, A. J. Feldman, Z. Ren, S. Setty, A. J. Blumberg, and M. Walfish. Verifying computations with state. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)*, pages 341–357, 2013.

[24] E. Buchman. Tendermint: Byzantine fault tolerance in the age of blockchains. Master's thesis, The University of Guelph, 2016.

[25] V. Buterin. Notes on scalable blockchain protocols. `https://github.com/vbuterin/scalability_paper/blob/master/scalability.pdf`, 2015.

[26] C. Cachin. Integrity and consistency for untrusted services. In *SOFSEM 2011: Theory and Practice of Computer Science*, pages 1–14, 2011.

[27] C. Cachin, I. Keidar, and A. Shraer. Fail-aware untrusted

storage. *SIAM Journal on Computing*, 40(2):493–533, Apr. 2011.

[28] M. Castro and B. Liskov. Practical Byzantine fault tolerance and proactive recovery. *ACM Transactions on Computer Systems (TOCS)*, 20(4):398–461, Nov. 2002.

[29] M. Castro, R. Rodrigues, and B. Liskov. Base: using abstraction to improve fault tolerance. *ACM Transactions on Computer Systems (TOCS)*, pages 236–269, 2003.

[30] B.-G. Chun, P. Maniatis, S. Shenker, and J. Kubiatowicz. Attested append-only memory: Making adversaries stick to their word. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)*, pages 189–204, 2007.

[31] J. Clark and A. Essex. Commitcoin: Carbon dating commitments with bitcoin. In *Proceedings of the International Financial Cryptography and Data Security Conference*, pages 390–398, 2012.

[32] A. Clement, M. Kapritsos, S. Lee, Y. Wang, L. Alvisi, M. Dahlin, and T. Riche. UpRight cluster services. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)*, pages 277–290, 2009.

[33] A. Clement, E. Wong, L. Alvisi, M. Dahlin, and M. Marchetti. Making Byzantine fault tolerant systems tolerate Byzantine faults. In *Proceedings of the USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, pages 153–168, 2009.

[34] Cognizant. Blockchain's smart contracts: Driving the next wave of innovation across manufacturing value chains. `https://www.cognizant.com/whitepapers/blockchains-smart-contracts-driving-the-next-wave-of-innovation-across-manufacturing-value-chains-codex2113.pdf`, June 2016.

[35] D. Creer, R. Crook, M. Hornsby, N. G. Avalis, M. Simpson, N. Weisfeld, B. Wyeth, and I. Zielinski. Proving Ethereum for the clearing use case. Technical report, Royal Bank of Scotland, 2016.

[36] K. Croman, C. Decker, I. Eyal, A. E. Gencer, A. Juels, A. M. Ahmed Kosba, P. Saxena, E. Shi, E. Gün Sirer, D. Song, and R. Wattenhofer. On scaling decentralized blockchains (a position paper). In *Proceedings of the Workshop on Bitcoin Research (BITCOIN)*, 2016.

[37] G. Danezis and S. Meiklejohn. Centrally banked cryptocurrencies. In *Proceedings of the Network and Distributed System Security Symposium (NDSS)*, 2016.

[38] I. Eyal. The miner's dilemma. In *Proceedings of the IEEE Symposium on Security and Privacy*, 2015.

[39] I. Eyal, A. E. Gencer, E. Gün Sirer, and R. van Renesse. Bitcoin-NG: A scalable blockchain protocol. In *Proceedings of the USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2016.

[40] I. Eyal and E. Gün Sirer. Majority is not enough: Bitcoin mining is vulnerable. In *Proceedings of the International Financial Cryptography Conference*, 2014.

[41] A. J. Feldman, W. P. Zeller, M. J. Freedman, and E. W. Felten. SPORC: Group collaboration using untrusted cloud resources. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 337–350, 2010.

[42] M. J. Fischer, N. A. Lynch, and M. S. Paterson. Impossibility of distributed consensus with one faulty process. In *Proceedings of the Symposium on Principles of Database Systems*, pages 1–7, 1983.

[43] J. Garay, A. Kiayias, and N. Leonardos. The Bitcoin backbone protocol: Analysis and applications. In *Proceedings of the International Conference on the Theory and Applications of Cryptographic Techniques (EUROCRYPT)*, 2015.

[44] V. Gramoli. On the danger of private blockchains (when PoW can be harmful to applications with termination requirements). In *Proceedings of the Workshop on Distributed Cryptocurrencies and Consensus Ledgers (DCCL)*, July 2016.

[45] A. Haeberlen. A case for the accountable cloud. In *Proceedings of the Workshop on Large-Scale Distributed Systems and Middleware (LADIS)*, 2009.

[46] A. Haeberlen, P. Kouznetsov, and P. Druschel. PeerReview: practical accountability for distributed systems. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)*, pages 175–188, 2007.

[47] J. D. Halamka, A. Lippman, and A. Ekblaw. The potential for blockchain to transform electronic health records. Technical report, Harvard business review, 2017.

[48] J. Hendricks, S. Sinnamohideen, G. R. Ganger, and M. K. Reiter. Zzyzx: Scalable fault tolerance through byzantine locking. In *International Conference on Dependable Systems and Networks (DSN)*, pages 363–372, 2010.

[49] M. Herlihy and M. Moir. Enhancing accountability and trust in distributed ledgers. *CoRR*, abs/1606.07490, 2016.

[50] S. J. Jason Teutsch and P. Saxena. When cryptocurrencies mine their own business. In *Proceedings of the International Financial Cryptography Conference*, 2016.

[51] J.P. Morgan and Oliver Wyman. Unlocking economic advantage with blockchain: A guide for asset managers. `http://www.oliverwyman.com/our-expertise/insights/2016/jul/unlocking-economic-advantage-with-blockchain.html`, July 2016.

[52] S. King and S. Nadal. PPCoin: Peer-to-peer crypto-currency with proof-of-stake. `http://peerco.in/assets/paper/peercoin-paper.pdf`, 2012.

[53] 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 *Proceedings of the USENIX Security Symposium*, 2016.

[54] A. Kosba, A. Miller, E. Shi, Z. Wen, and C. Papamanthou. Hawk: The blockchain model of cryptography and privacy-preserving smart contracts. In *Proceedings of the IEEE Symposium on Security and Privacy*, 2016.

[55] R. Kotla, L. Alvisi, M. Dahlin, A. Clement, and E. Wong. Zyzzyva: Speculative Byzantine fault tolerance. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)*, pages 45–58, 2007.

[56] R. Kotla and M. Dahlin. High throughput Byzantine fault tolerance. In *International Conference on Dependable*

Systems and Networks (DSN), pages 575–584, 2004.

[57] K. R. M. Leino. Dafny: An automatic program verifier for functional correctness. In *International Conference on Logic for Programming, Artificial Intelligence, and Reasoning*, pages 348–370, 2010.

[58] D. Levin, J. R. Douceur, J. R. Lorch, and T. Moscibroda. Trinc: Small trusted hardware for large distributed systems. In *Proceedings of the USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, pages 1–14, 2009.

[59] J. Li, M. Krohn, D. Mazières, and D. Shasha. Secure untrusted data repository (SUNDR). In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2004.

[60] J. Li and D. Maziéres. Beyond one-third faulty replicas in Byzantine fault tolerant systems. In *Proceedings of the USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2007.

[61] L. Luu, J. Teutsch, R. Kulkarni, and P. Saxena. Demystifying incentives in the consensus computer. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, 2015.

[62] P. Mahajan, S. Setty, S. Lee, A. Clement, L. Alvisi, M. Dahlin, and M. Walfish. Depot: Cloud storage with minimal trust. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 307–322, 2010.

[63] J.-P. Martin, L. Alvisi, and M. Dahlin. Small Byzantine quorum systems. In *International Conference on Dependable Systems and Networks (DSN)*, pages 374–383, 2002.

[64] D. Mazières. The Stellar consensus protocol: A federated model for Internet-level consensus. Technical report, Stellar Devlopment Foundation, Apr. 2015.

[65] D. Mazières and D. Shasha. Building secure file systems out of Byzantine storage. In *Proceedings of the ACM Conference on Principles of Distributed Computing (PODC)*, pages 108–117, 2002.

[66] S. Micali. ALGORAND: the efficient and democratic ledger. *CoRR*, abs/1607.01341, 2016.

[67] A. Miller, Y. Xia, K. Croman, E. Shi, and D. Song. The Honey Badger of BFT Protocols. Cryptology ePrint Archive, Report 2016/199, 2016.

[68] M. Moradi, F. Qian, Q. Xu, Z. M. Mao, D. Bethea, and M. K. Reiter. Caesar: High-Speed and Memory-Efficient Forwarding Engine for Future Internet Architecture. In *Symposium on Architectures for Networking and Communications Systems*, pages 171–182, 2015.

[69] S. Nakamoto. Bitcoin: A peer-to-peer electronic cash system. Oct. 2008.

[70] A. Narayanan, J. Bonneau, E. Felten, A. Miller, and S. Goldfeder. *Bitcoin and Cryptocurrency Technologies: A Comprehensive Introduction*. Princeton University Press, 2016.

[71] C. Natoli and V. Gramoli. The blockchain anomaly. *CoRR*, abs/1605.05438, 2016.

[72] K. Nayak, S. Kumar, A. Miller, and E. Shi. Stubborn mining: Generalizing selfish mining and combining with an eclipse attack. In *Proceedings of the IEEE European Symposium on Security and Privacy*, 2016.

[73] J. Nelson, M. Ali, R. Shea, and M. J. Freedman. Extending existing blockchains with virtualchain. In *Proceedings of the Workshop on Distributed Cryptocurrencies and Consensus Ledgers (DCCL)*, July 2016.

[74] R. Pass, L. Seeman, and abhi shelat. Analysis of the blockchain protocol in asynchronous networks. Cryptology ePrint Archive, Report 2016/454, 2016.

[75] R. Pass and E. Shi. The sleepy model of consensus. Cryptology ePrint Archive, Report 2016/918, 2016.

[76] R. A. Popa, J. R. Lorch, D. Molnar, H. J. Wang, and L. Zhuang. Enabling security in cloud storage SLAs with CloudProof. In *Proceedings of the USENIX Annual Technical Conference (ATC)*, 2011.

[77] J. Schneider, A. Blostein, B. Lee, S. Kent, I. Groer, and E. Beardsley. Blockchain: Putting theory to practice. In *Goldman Sachs' profiles of innovation*, May 2016.

[78] F. Schuster, M. Costa, C. Fournet, C. Gkantsidis, M. Peinado, G. Mainar-Ruiz, and M. Russinovich. VC3: Trustworthy data analytics in the cloud using SGX. In *Proceedings of the IEEE Symposium on Security and Privacy*, 2015.

[79] D. Schwartz, N. Youngs, and A. Britto. The Ripple protocol consensus algorithm. Technical report, Ripple Labs Inc, 2014.

[80] P. L. Seijas, S. Thompson, and D. McAdams. Scripting smart contracts for distributed ledger technology. Cryptology ePrint Archive, Report 2016/1156, 2016.

[81] A. Shraer, C. Cachin, A. Cidon, I. Keidar, Y. Michalevsky, and D. Shaket. Venus: Verification for untrusted cloud storage. In *Proceedings of the Cloud Computing Security Workshop (CCSW)*, pages 19–30, 2010.

[82] A. Singh, P. Fonseca, P. Kuznetsov, R. Rodrigues, and P. Maniatis. Zeno: Eventually consistent Byzantine-fault tolerance. In *Proceedings of the USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, pages 169–184, 2009.

[83] 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 IEEE Symposium on Security and Privacy*, 2016.

[84] A. Tomescu and S. Devadas. Catena: Efficient non-equivocation via Bitcoin. Cryptology ePrint Archive, Report 2016/1062, 2016.

[85] G. Wood. Ethereum: A secure decentralised generalised transaction ledger homestead revision. `http://gavwood.com/paper.pdf`.

[86] World Economic Forum. The future of financial infrastructure: An ambitious look at how blockchain can reshape financial services. `https://www.weforum.org/reports/the-future-of-financial-infrastructure-an-ambitious-look-at-how-blockchain-can-reshape-financial-services`, Aug. 2016.

[87] J. Yin, J.-P. Martin, A. Venkataramani, L. Alvisi, and M. Dahlin. Byzantine fault-tolerant confidentiality. In *Proceedings of the International Workshop on Future*

Directions in Distributed Computing, pages 12–15, 2002.

[88] J. Yin, J.-P. Martin, A. Venkataramani, L. Alvisi, and M. Dahlin. Separating agreement from execution for byzantine fault tolerant services. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)*, pages 253–267, 2003.

[89] A. R. Yumerefendi and J. S. Chase. Strong accountability for network storage. In *Proceedings of the USENIX Conference on File and Storage Technologies (FAST)*, 2007.

[90] V. Zamfir. Introducing Casper "The Friendly Ghost". `https://blog.ethereum.org/2015/08/01/introducing-casper-friendly-ghost`, 2015.

[91] F. Zhang, E. Cecchetti, K. Croman, A. Juels, and E. Shi. Town Crier: an authenticated data feed for smart contracts. Cryptology ePrint Archive, Report 2016/168, 2016.

[92] W. Zheng, A. Dave, J. G. Beekman, R. A. Popa, J. E. Gonzalez, and I. Stoica. Opaque: An oblivious and encrypted distributed analytics platform. In *Proceedings of the USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2017.