Chainspace: A Sharded Smart Contracts Platform

Mustafa Al-Bassam*, Alberto Sonnino*, Shehar Bano*, Dave Hrycyszyn† and George Danezis*
* University College London, United Kingdom
† constructiveproof.com

***Abstract*—Chainspace is a decentralized infrastructure, known as a distributed ledger, that supports user defined smart contracts and executes user-supplied transactions on their objects. The correct execution of smart contract transactions is verifiable by all. The system is scalable, by sharding state and the execution of transactions, and using $\mathcal{S}$-BAC, a distributed commit protocol, to guarantee consistency. Chainspace is secure against subsets of nodes trying to compromise its integrity or availability properties through Byzantine Fault Tolerance (BFT), and extremely high-auditability, non-repudiation and 'blockchain' techniques. Even when BFT fails, auditing mechanisms are in place to trace malicious participants. We present the design, rationale, and details of Chainspace; we argue through evaluating an implementation of the system about its scaling and other features; we illustrate a number of privacy-friendly smart contracts for smart metering, polling and banking and measure their performance.**

I. Introduction

Chainspace is a distributed ledger platform for high-integrity and transparent processing of transactions within a decentralized system. Unlike application specific distributed ledgers, such as Bitcoin [Nak08] for a currency, or certificate transparency [LLK13] for certificate verification, Chainspace offers extensibility though smart contracts, like Ethereum [Woo14]. However, users expose to Chainspace enough information about contracts and transaction semantics, to provide higher scalability through sharding across infrastructure nodes: our modest testbed of 60 cores achieves 350 transactions per second, as compared with a peak rate of less than 7 transactions per second for Bitcoin over 6K full nodes. Etherium currently processes 4 transactions per second, out of theoretical maximum of 25. Furthermore, our platform is agnostic as to the smart contract language, or identity infrastructure, and supports privacy features through modern zero-knowledge techniques [BCCG16, DGFK14].

Unlike other scalable but 'permissioned' smart contract platforms, such as Hyperledger Fabric [Cac16] or BigchainDB [MMM+16], Chainspace aims to be an 'open' system: it allows anyone to author a smart contract, anyone to provide infrastructure on which smart contract code and state runs, and any user to access calls to smart contracts. Further, it provides ecosystem features, by allowing composition of smart contracts from different authors. We integrate a value system, named CSCoin, as a system smart contract to allow for accounting between those parties.

However, the security model of Chainspace, is different from traditional unpermissioned blockchains, that rely on proof-of-work and global replication of state, such as Ethereum. In Chainspace smart contract authors designate the parts of the infrastructure that are trusted to maintain the integrity of their contract—and only depend on their correctness, as well as the correctness of contract sub-calls. This provides fine grained control of which part of the infrastructure need to be trusted on a per-contract basis, and also allows for horizontal scalability.

This paper makes the following contributions:

- It presents Chainspace, a system that can scale arbitrarily as the number of nodes increase, tolerates byzantine failures, and can be fully and publicly audited.
- It presents a novel distributed atomic commit protocol, called $\mathcal{S}$-BAC, for sharding generic smart contract transactions across multiple byzantine nodes, and correctly coordinating those nodes to ensure safety, liveness and security properties.
- It introduces a distinction between parts of the smart contract that execute a computation, and those that check the computation and discusses how that distinction is key to supporting privacy-friendly smart-contracts.
- It provides a full implementation and evaluates the performance of the byzantine distributed commit protocol, $\mathcal{S}$-BAC, on a real distributed set of nodes and under varying transaction loads.
- It presents a number of key system and application smart contracts and evaluates their performance. The contracts for privacy-friendly smart-metering and privacy-friendly polls illustrate and validate support for high-integrity and high-privacy applications.

Outline: Section II presents an overview of Chainspace; Section III presents the client-facing application interface; Section IV presents the design of internal data structures guaranteeing integrity, the distributed architecture, the byzantine commit protocols, and smart contract definition and composition. Section V argues the correctness and security; specific smart contracts and their evaluations are presented in Section VI; Section VII presents an evaluation of the core protocols and smart contract performance; Section VIII presents limitation and Section IX a comparison with related work; and Section X concludes.

II. System Overview

Chainspace allows applications developers to implement distributed ledger applications by defining and calling procedures of smart contracts operating on controlled objects, and abstracts the details of how the ledger works and scales. In this section, we first describe data model of Chainspace, followed by an overview of the system design, its threat model and security properties.

A. Data Model: Objects, Contracts, Transactions.

Chainspace applies aggressively the end-to-end principle [SRC84] in relying on untrusted end-user applications to build transactions to be checked and executed. We describe below key concepts within the Chainspace data model, that developers need to grasp to use the system.

Objects are atoms that hold state in the Chainspace system. We usually refer to an object through the letter o, and a set of objects as $o \in O$. All objects have a cryptographically derived unique identifier used to unambiguously refer to the object, that we denote $\mathsf{id}(o)$. Objects also have a type, denoted as $\mathsf{type}(o)$, that determines the unique identifier of the smart contract that defines them, and a type name. In Chainspace object state is immutable. Objects may be in two meta-states, either *active* or *inactive*. Active objects are available to be operated on through smart contract procedures, while inactive ones are retained for the purposes of audit only.

Contracts are special types of objects, that contain executable information on how other objects of types defined by the contract may be manipulated. They define a set of initial objects that are created when the contract is first created within Chainspace. A contract c defines a *namespace* within which *types* (denoted as $\mathsf{types}(c)$) and a *checker* v for *procedures* (denoted as $\mathsf{proc}(c)$) are defined.

A *procedure*, p, defines the logic by which a number of objects, that may be *inputs* or *references*, are processed by some logic and *local parameters* and *local return values* (denoted as lpar and lret), to generate a number of object *outputs*. Notionally, input objects, denoted as a vector $\vec{w}$, represent state that is invalidated by the procedure; references, denoted as $\vec{r}$ represent state that is only read; and outputs are objects, or $\vec{x}$ are created by the procedure. Some of the local parameters or local returns may be secrets, and require confidentiality. We denote those as spar and sret respectively.

We denote the execution of such a procedure as:

$$c.p(\vec{w}, \vec{r}, \mathsf{lpar}, \mathsf{spar}) \rightarrow \vec{x}, \mathsf{lret}, \mathsf{sret} \tag{1}$$

for $\vec{w}, \vec{r}, \vec{x} \in O$ and $p \in \mathsf{proc}(c)$. We restrict the type of all objects (inputs $\vec{w}$, outputs $\vec{x}$ and references $\vec{r}$) to have types defined by the same contract c as the procedure p (formally: $\forall o \in \vec{w} \cup \vec{x} \cup \vec{r}.\mathsf{type}(o) \in \mathsf{types}(c)$). However, public locals (both lpar and lret) may refer to objects that are from different contracts through their identifiers. We further require a procedure that outputs an non empty set of objects $\vec{x}$, to also take as parameters a non-empty set of input objects $\vec{w}$. Transactions that create no outputs are allowed to just take locals and references $\vec{r}$.

Associated with each smart contract c, we define a *checker* denoted as v. Those checkers are pure functions (ie. deterministic, and have no side-effects), and return a Boolean value. A checker v is defined by a contract, and takes as parameters a procedure p, as well as inputs, outputs, references and locals.

$$c.v(p, \vec{w}, \vec{r}, \mathsf{lpar}, \vec{x}, \mathsf{lret}, \mathsf{dep}) \rightarrow \{\mathsf{true}, \mathsf{false}\} \tag{2}$$

Note that checkers do not take any secret local parameters (spar or sret). A checker for a smart contract returns true only if there exist some secret parameters spar or sret, such that an execution of the contract procedure p, with the parameters passed to the checker alongside spar or sret, is possible as defined in Equation (1). The variable dep represent the context in which the procedure is called: namely information about other procedure executions. This supports composition, as we discuss in detail in the next section.

We note that procedures, unlike checkers, do not have to be pure functions, and may be randomized, keep state or have side effects. A smart contract defines explicitly the checker $c.v$, but does not have to define procedures *per se*. The Chainspace system is oblivious to procedures, and relies merely on checkers. Yet, applications may use procedures to create valid transactions. The distinction between procedures and checkers—that do not take secrets—is key to implementing privacy-friendly contracts.

Transactions represent the atomic application of one or more valid procedures to active input objects, and possibly some referenced objects, to create a number of new active output objects. The design of Chainspace is user-centric, in that a user client executes all the computations necessary to determine the outputs of one or more procedures forming a transaction, and provides enough evidence to the system to check the validity of the execution and the new objects.

Once a transaction is accepted in the system it 'consumes' the input objects, that become inactive, and brings to life all new output objects that start their life by being active. References on the other hand must be active for the transaction to succeed, and remain active once a transaction has been successfully committed.

An client packages enough information about the execution of those procedures to allow Chainspace to safely *serialize* its execution, and *atomically* commit it only if all transactions are valid according to relevant smart contract checkers.

B. System Design, Threat Model and Security Properties

We provide an overview of the system design, illustrated in Figure 1. Chainspace is comprised of a network of infrastructure *nodes* that manage valid objects, and ensure that only valid transactions are committed. A key design goal is to achieve scalability in terms of high transaction throughput and low latency. To this end, nodes are organized into shards that manage the state of objects, keep track of their validity, and record transactions aborted or committed. Within each shard all honest nodes ensure they consistently agree whether to accept or reject a transaction: whether an object is active or inactive at any point, and whether traces from contracts they know check. Across shards, nodes must ensure that transactions are *committed* if all shards are willing to commit the transaction, and rejected (or *aborted*) if any shards decide to abort the transaction—due to checkers returning false or objects being inactive. To satisfy

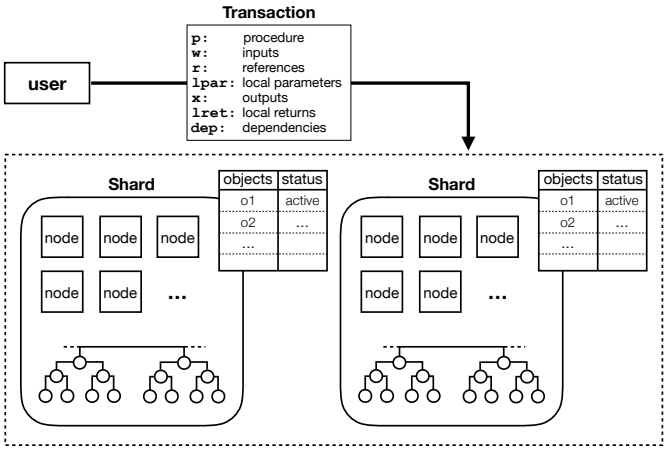

Fig. 1. Design overview of Chainspace system.

these requirements, Chainspace implements $\mathcal{S}$-BAC—a protocol that composes existing Byzantine agreement and atomic commit primitives in a novel way. Consensus on committing (or aborting) transactions takes place in parallel across different shards. For transparency and auditability, nodes in each shard periodically publish a signed hash chain of *checkpoints*: shards add a block (Merkle tree) of evidence including transactions processed in the current epoch, and signed promises from other nodes, to the hash chain.

Chainspace supports security properties against two distinct types of adversaries, both polynomial time bounded:

- **Honest Shards (HS).** The first adversary may create arbitrary contracts, and input arbitrary transactions into Chainspace, however they are bound to only control up to f faulty nodes in any shard. As a result, and to ensure the correctness and liveness properties of Byzantine consensus, each shard must have a size of at least $3f + 1$ nodes.
- **Dishonest Shards (DS).** The second adversary has, additionally to HS, managed to gain control of one or more shards, meaning that they control over f nodes in those shards. Thus, its correctness or liveness may not be guaranteed.

Faulty nodes in shards may behave arbitrarily, and collude to violate any of the security, safely or liveness properties of the system. They may emit incorrect or contradictory messages, as well as not respond to any or some requests.

Given this threat model, Chainspace supports the following security properties:

- **Transparency.** Chainspace ensures that anyone in possession of the identity of a valid object may authenticate the full history of transactions and objects that led to the creation of the object. No transactions may be inserted, modified or deleted from that causal chain or tree. Objects may be used to self-authenticate its full history—this holds under both the HS and DS threat models.
- **Integrity.** Subject to the HS threat model, when one or more transactions are submitted only a set of valid non-conflicting transactions will be executed within the system. This includes resolving conflicts—in terms of multiple transactions using the same objects—ensuring the validity of the transactions, and also making sure that all new objects are registered as active. Ultimately, Chainspace transactions are accepted, and the set of active objects changes, as if executed sequentially—however, unlike other systems such as Ethereum [Woo14], this is merely an abstraction and high levels of concurrency are supported.
- **Encapsulation.** The smart contract checking system of Chainspace enforces strict isolation between smart contracts and their state—thus prohibiting one smart contract from directly interfering with objects from other contracts. Under both the HS and DS threat models. However, cross-contract calls are supported but mediated by well defined interfaces providing encapsulation.
- **Non-repudiation.** In case conflicting or otherwise invalid transactions were to be accepted in honest shards (in the case of the DS threat model), then evidence exists to pinpoint the parties or shards in the system that allowed the inconsistency to occur. Thus, failures outside the HS threat model, are detectable; the guildy parties may be banned; and appropriate off-line recovery mechanisms could be deployed.

III. The Chainspace Application Interface

Smart Contract developers in Chainspace register a smart contract c into the distributed system managing Chainspace, by defining a checker for the contract and some initial objects. Users may then submit transactions to operate on those objects in ways allowed by the checkers. Transactions represent the execution of one or more procedures from one or more smart contracts. It is necessary for all inputs to all procedures within the transaction to be active for a transaction to be executed and produce any output objects.

Transactions are *atomic*: either all their procedures run, and produce outputs, or none of them do. Transactions are also *consistent*: in case two transactions are submitted to the system using the same active object inputs, at most one of them will eventually be executed to produce outputs. Other transactions, called *conflicting*, will be aborted.

Representation of Transactions. A transaction within Chainspace is represented by sequence of *traces* of the executions of the procedures that compose it, and their interdependencies. These are computed and packaged by end-user clients, and contain all the information a checker needs to establish its correctness. A Transaction is a data structure such that:

$$
\begin{aligned}
&\text{type } \textit{Transaction} : \textit{Trace} \text{ list}\\
&\text{type } \textit{Trace} : \text{Record } \{\\
&\quad c : \mathsf{id}(o), \quad p : \text{string},\\
&\quad \vec{w}, \vec{r}, \vec{x} : \mathsf{id}(o) \text{ list},\\
&\quad \mathsf{lpar}, \mathsf{lret} : \text{arbitrary data},\\
&\quad \mathsf{dep} : \textit{Trace} \text{ list}\}
\end{aligned}
$$

$$\frac{\alpha_0,\ \mathit{Valid}(t), \alpha' \qquad \alpha',\ \mathit{Valid}(T'), \alpha_1}{\alpha_0,\ \mathit{Valid}(T = t :: T'), \alpha_1}\ \text{(Sequence)}$$

$$\frac{\alpha_0,\ \mathit{Valid}(\mathsf{dep}), \alpha' \qquad \alpha', c.v(p, \vec{w}, \vec{r}, \mathsf{lpar}, \vec{x}, \mathsf{lret}, \mathsf{dep}), (\alpha' \setminus \vec{w}) \cup \vec{x} \qquad \begin{array}{r}\vec{w}, \vec{r} \in \alpha' \wedge \\ (\vec{x} \neq \emptyset) \rightarrow (\vec{w} \neq \emptyset) \wedge \\ \forall o \in \vec{w} \cup \vec{x} \cup \vec{r}.\mathsf{type}(o) \in \mathsf{types}(c)\end{array}}{\alpha_0,\ \mathit{Valid}(t = [c, p, \vec{w}, \vec{r}, \vec{x}, \mathsf{lpar}, \mathsf{lret}, \mathsf{dep}]), (\alpha' \setminus \vec{w}) \cup \vec{x}}\ \text{(Check)}$$

Fig. 2. The sequencing and checking validity rules for transactions.

To generate a set of traces composing the transaction, a *user executes on the client side all the smart contract procedures* required on the input objects, references and local parameters, and generates the output objects and local returns for every procedure—potentially also using secret parameters and returns. Thus the actual computation behind the transactions is performed by the user, and the traces forming the transaction already contain the output objects and return parameters, and sufficient information to check their validity through smart contract checkers. This design pattern is related to traditional *optimistic concurrency control*.

Only valid transactions are eventually committed into the Chainspace system, as specified by two validity rules *sequencing* and *checking* presented in Figure 2. Transactions are considered valid within a context of a set of active objects maintained by Chainspace, denoted with α. Valid transactions lead to a new context of active objects (eg. α'). We denote this through the triplet $(\alpha, \mathit{Valid}(T), \alpha')$, which is true if the execution of transaction T is valid within the context of active objects α and generates a new context of active objects α'. The two rules are as follows:

- (Sequence rule). A '*Trace* list' (within a '*Transaction*' or list of dependencies) is valid if each of the traces are valid in sequence (see Figure 2 rule for sequencing). Further, the active objects set is updated in sequence before considering the validity of each trace.
- (Check rule). A particular '*Trace*' is valid, if the sequence of its dependencies are valid, and then in the resulting active object context, the checker for it returns true. A further three side conditions must hold: (1) inputs and references must be active; (2) if the trace produces any output objects it must also contain some input objects; and (3) all objects passed to the checker must be of types defined by the smart contract of this checker (see Figure 2 rule for checking).

The ordering of active object sets in the validation rules result in a depth-first validation of all traces, which represents a depth-first execution and data flow dependency between them. It is also noteworthy that only the active set of objects needs to be tracked to determine the validity of new transactions, which is in the order of magnitude of active objects in the system. The much longer list of inactive objects, which grows to encompass the full history of every object in the system is not needed—which we leverage to enable better when validating transactions. It also results in a smaller amount of working memory to perform incremental audits.

A valid transaction is executed in a serialized manner, and committed or aborted atomically. If it is committed, the new set of active objects replaces the previous set; if not the set of active objects does not change. Determining whether a transaction may commit involves ensuring all the input objects are active, and all are consumed as a result of the transaction executing, as well as all new objects becoming available for processing (references however remain active). Chainspace ensures this through the distributed atomic commit protocol, $\mathcal{S}$-BAC.

Smart contract composition. A contract procedure may call a transaction of another smart contract, with specific parameters and rely upon returned values. This is achieved through passing the dep variable to a smart contract checker, a validated list of traces of all the sub-calls performed. The checker can ensure that the parameters and return values are as expected, and those dependencies are checked for validity by Chainspace.

Composition of smart contracts is a key feature of a transparent and auditable computation platform. It allows the creation of a library of smart contracts that act as utilities for other higher-level contracts: for example, a simple contract can implement a cryptographic currency, and other contracts—for e-commerce for example—can use this currency as part of their logic. Furthermore, we compose smart contracts, in order to build some of the functionality of Chainspace itself as a set of 'system' smart contracts, including management of shards mapping to nodes, key management of shard nodes, and governance.

Chainspace also supports the atomic batch execution of multiple procedures for efficiency, that are not dependent on each other.

Reads. Besides executing transactions, Chainspace clients, need to read the state of objects, if anything, to correctly form transactions. Reads, by themselves, cannot lead to inconsistent state being accepted into the system, even if they are used as inputs or references to transactions. This is a result of the system checking the validity rules before accepting a transaction, which will reject any stale state.

Thus, any mechanism may be used to expose the state of objects to clients, including traditional relational databases, or 'no-SQL' alternatives. Additionally, any indexing mechanism may be used to allow clients to retrieve objects with specific characteristics faster. Decentralized, read-only stores have been extensively studied, so we do not address the question of reads further in this work.

Privacy by design. Defining smart contract logic as checkers allows Chainspace to support privacy friendly-contracts by design. In such contracts some information in objects is not in the clear, but instead either encrypted using a public key,

or committed using a secure commitment scheme as [P+91]. The transaction only contains a valid proof that the logic or invariants of the smart contract procedure were applied correctly or hold respectively, and can take the form of a zero-knowledge proof, or a Succinct Argument of Knowledge (SNARK). Then, generalizing the approach of [MGGR13], the checker runs the verifier part of the proof or SNARK that validates the invariants of the transactions, without revealing the secrets within the objects to the verifiers.

IV. The Chainspace System Design

In Chainspace a network of infrastructure *nodes* manages valid objects, and ensure key invariants: namely that only valid transactions are committed. We discuss the data structures nodes use collectively and locally to ensure high integrity; and the distributed protocols they employ to reach consensus on the accepted transactions.

A. High-Integrity Data Structures

Chainspace employs a number of high-integrity data structures. They enable those in possession of a valid object or its identifier to verify all operations that lead to its creation; they are also used to support *non-equivocation*—preventing Chainspace nodes from providing a split view of the state they hold without detection.

Hash-DAG structure. Objects and transactions naturally form a directed acyclic graph (DAG): given an initial state of active objects a number of transactions render their inputs invalid, and create a new set of outputs as active objects. These may be represented as a directed graph between objects, transactions and new objects and so on. Each object may only be created by a single transaction trace, thus cycles between future transactions and previous objects never occur. We prove that output object identifiers resulting from valid transactions are fresh (see Security Theorem 1). Hence, the graph of objects inputs, transactions and objects outputs form a DAG, that may be indexed by their identifiers.

We leverage this DAG structure, and augment it to provide a high-integrity data structure. Our principal aim is to ensure that given an object, and its identifier, it is possible to unambiguously and unequivocally check all transactions and previous (now inactive) objects and references that contribute to the existence of the object. To achieve this we define as an identifier for all objects and transactions a cryptographic hash that directly or indirectly depends on the identifiers of all state that contributed to the creation of the object.

Specifically, we define a function $\mathsf{id}(\mathit{Trace})$ as the identifier of a trace contained in transaction T. The identifier of a trace is a cryptographic hash function over the name of contract and the procedure producing the trace; as well as serialization of the input object identifiers, the reference object identifiers, and all local state of the transaction (but not the secret state of the procedures); the identifiers of the trace's dependencies are also included. Thus all information contributing to defining the Trace is included in the identifier, except the output object identifiers.

We also define the $\mathsf{id}(o)$ as the identifier of an object o. We derive this identifier through the application of a cryptographic hash function, to the identifier of the trace that created the object o, as well as a unique name assigned by the procedures creating the trace, to this output object. (Unique in the context of the outputs of this procedure call, not globally, such as a local counter.)

An object identifier $\mathsf{id}(o)$ is a high-integrity handle that may be used to authenticate the full history that led to the existence of the object o. Due to the collision resistance properties of secure cryptographic hash functions an adversary is not able to forge a past set of objects or transactions that leads to an object with the same identifier. Thus, given $\mathsf{id}(o)$ anyone can verify the authenticity of a trace that led to the existence of o.

A very important property of object identifiers is that future transactions cannot re-create an object that has already become inactive. Thus checking object validity only requires maintaining a list of active objects, and not a list of past inactive objects:

Security Theorem 1. *No sequence of valid transactions, by a polynomial time constrained adversary, may re-create an object with the same identifier with an object that has already been active in the system.*

Proof: We argue this property by induction on the serialized application of valid transactions, and for each transaction by structural induction on the two validity rules. Assuming a history of $n-1$ transactions for which this property holds we consider transaction n. Within transaction n we sequence all traces and their dependencies, and follow the data flow of the creation of new objects by the 'check' rule. For two objects to have the same $\mathsf{id}(o)$ there need to be two invocations of the check rule with the same contract, procedure, inputs and references. However, this leads to a contradiction: once the first trace is checked and considered valid the active input objects are removed from the active set, and the second invocation becomes invalid. Thus, as long as object creation procedures have at least one input (which is ensured by the side condition) the theorem holds, unless an adversary can produce a hash collision. The inductive base case involves assuming that no initial objects start with the same identifier – which we can ensure axiomatically. ■

We call this directed acyclic graph with identifiers derived using cryptographic functions a Hash-DAG, and we make extensive use of the identifiers of objects and their properties in Chainspace.

Node Hash-Chains. Each node in Chainspace, that is entrusted with preserving integrity, associates with its shard a hash chain. Periodically, peers within a shard consistently agree to seal a *checkpoint*, as a block of transactions into their hash chains. They each form a Merkle tree containing all transactions that have been accepted or rejected in sequence by the shard since the last checkpoint was sealed. Then, they extend their hash chain by hashing the root of this Merkle tree and a block sequence number, with the head hash of the chain so far, to create the new head of the hash chain. Each peer signs the new head of their chain, and shares it with all other peers in the shard, and anyone who requests it. For strong auditability additional information, besides committed or aborted transactions, has to be included in the Merkle tree: node should log any promise to either commit or abort a transaction from any other peer in any shard (the prepared(T,*) statements explained in the next sections).

All honest nodes within a shard independently create the

same chain for a checkpoint, and a signature on it—as long as the consensus protocols within the shards are correct. We say that a checkpoint represents the decision of a shard, for a specific sequence number, if at least $f + 1$ signatures of shard nodes sign it. On the basis of these hash chains we define a *partial audit* and a *full audit* of the Chainspace system.

In a *partial audit* a client is provided evidence that a transaction has been either committed or aborted by a shard. A client performing the partial audit may request from any node of the shard evidence for a transaction T. The shard peer will present a block representing the decision of the shard, with $f + 1$ signatures, and a proof of inclusion of a commit or abort for the transaction, or a signed statement the transaction is unknown. A partial audit provides evidence to a client of the fate of their transaction, and may be used to detect past of future violations of integrity. A partial audit is an efficient operation since the evidence has size $O(s + \log N)$ in N the number of transactions in the checkpoint and s the size of the shard—thanks to the efficiency of proving inclusion in a Merkle tree, and checking signatures.

A *full audit* involves replaying all transactions processed by the shard, and ensuring that (1) all transactions were valid according to the checkers the shard executed; (2) the objects input or references of all committed transactions were all active (see rules in Figure 2); and (3) the evidence received from other shards supports committing or aborting the transactions. To do so an auditor downloads the full hash-chain representing the decisions of the shard from the beginning of time, and re-executes all the transactions in sequence. This is possible, since—besides their secret signing keys—peers in shards have no secrets, and their execution is deterministic once the sequence of transactions is defined. Thus, an auditor can re-execute all transactions in sequence, and check that their decision to commit or abort them is consistent with the decision of the shard. Doing this, requires any inter-shard communication (namely the promises from other shards to commit or abort transactions) to be logged in the hash-chain, and used by the auditor to guide the re-execution of the transactions. A full audit needs to re-execute all transactions and requires evidence of size $O(N)$ in the number N of transactions. This is costly, but may be done incrementally as new blocks of shard decisions are created.

B. Distributed Architecture & Consensus

A network of *nodes* manages the state of Chainspace objects, keeps track of their validity, and record transactions that are seen or that are accepted as being committed.

Chainspace uses sharding strategies to ensure scalability: a public function $\phi(o)$ maps each object o to a set of nodes, we call a *shard*. These nodes collectively are entrusted to manage the state of the object, keep track of its validity, record transactions that involve the object, and eventually commit at most one transaction consuming the object as input and rendering it inactive. However, nodes must only record such a transaction as committed if they have certainty that all other nodes have, or will in the future, record the same transaction as consuming the object. We call this distributed algorithm the *consensus* algorithm within the shard.

For a transaction T we define a set of *concerned nodes*, $\Phi(T)$ for a transaction structure T. We first denote as ζ the set of all objects identifiers that are input into or referenced by any trace contained in T. We also denote as ξ the set of all objects that are output by any trace in T. The function $\Phi(T)$ represents the set of nodes that are managing objects that should exist, and be active, in the system for T to succeed. More mathematically, $\Phi(T) = \bigcup\{\phi(o_i)|o_i \in \zeta \setminus \xi\}$, where $\zeta \setminus \xi$ represents the set of objects input but not output by the transaction itself (its free variables). The set of concerned peers thus includes all shard nodes managing objects that already exist in Chainspace that the transaction uses as references or inputs.

An important property of this set of nodes holds, that ensures that all smart contracts involved in a transaction will be mapped to some concerned nodes that manage state from this contract:

Security Theorem 2. *If a contract c appears in any trace within a transaction T, then the concerned nodes set $\Phi(T)$ will contain nodes in a shard managing an object o of a type from contract c. I.e. $\exists o.\mathsf{type}(o) \in \mathsf{types}(c) \wedge \phi(o) \cap \Phi(T) \neq \emptyset$.*

Proof: Consider any trace t within T, from contract c. If the inputs or references to this trace are not in ξ—the set of objects that were created within T—then their shards will be included within $\Phi(T)$. Since those are of types within c the theorem holds. If on the other hand the inputs or references are in ξ, it means that there exists another trace within T from the same contract c that generated those outputs. We then recursively apply the case above to this trace from the same c. The process will terminate with some objects of types in c and shard managing them within the concerned nodes set—and this is guarantee to terminate due to the Hash-DAG structure of the transactions (that may have no loops). ■

Security Theorem 2 ensures that the set of concerned nodes, includes nodes that manage objects from all contracts represented in a transaction. Chainspace leverages this to distribute the process of rule validation across peers in two ways:

- For any existing object o in the system, used as a reference or input within a transaction T, only the shard nodes managing it, namely in $\phi(o)$, need to check that it is active (as part of the 'check' rule in Figure 2).
- For any trace t from contract c within a transaction T, only shards of concerned nodes that manage objects of types within c need to run the checker of that contract to validate the trace (again as part of the 'check' rule), and that all input, output and reference objects are of types within c.

However, all shards containing concerned nodes for T need to ensure that all others have performed the necessary checks before committing the transaction, and creating new objects.

There are many options for ensuring that concerned nodes in each shards do not reach an inconsistent state for the accepted transactions, such as Nakamoto consensus through proof-of-work [Nak08], two-phase commit protocols [LL94], and classical consensus protocols like Paxos [L$^+$01], PBFT [CL$^+$99], or xPaxos [LCQV15]. However, these approaches lack in performance, scalability, and/or security. We design an open,

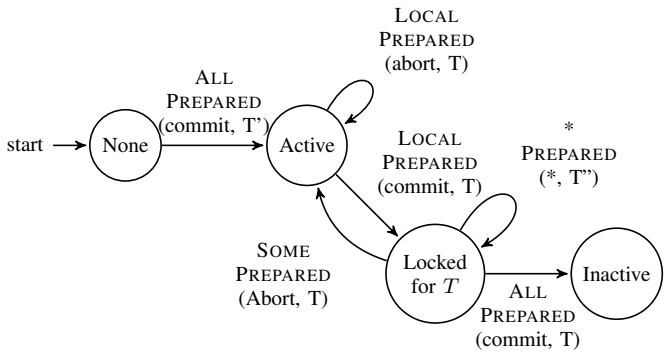

Fig. 3. The state machine representing the active, locked and inactive states for any object within Chainspace. Each node in a shard replicates the state of the object, and participates in a consensus protocol that allows it to derive the invariants "Local prepared", "All prepared", and "Some prepared" to update the state of an object.

scalable and decentralized mechanism to perform *Sharded Byzantine Atomic Commit* or $\mathcal{S}$-BAC.

C. Sharded Byzantine Atomic Commit ($\mathcal{S}$-BAC).

Chainspace implements the previously described intra-shard consensus algorithm for transaction processing in the *byzantine* and *asynchronous* setting, through the *Sharded Byzantine Atomic Commit* ($\mathcal{S}$-BAC) protocol, that combines two primitive protocols: *Byzantine Agreement* and *atomic commit*.

- *Byzantine agreement* ensures that all honest members of a shard of size $3f + 1$, agree on a specific common sequence of actions, despite some f malicious nodes within the shard. It also guarantees that when agreement is sought, a decision or sequence will eventually be agreed upon. The agreement protocol is executed within each shard to coordinate all nodes. We use MODSMART [SB12] implementation of PBFT for state machine replication that provides an optimal number of communications steps (similar to PBFT [CL$^+$99]). This is achieved by replacing reliable broadcast with a special leader-driven Byzantine consensus primitive called Validated and Provable Consensus (VP-Consensus).
- *Atomic commit* is ran across all shards managing objects relied upon by a transaction. It ensures that each shard needs to accept to commit a transaction, for the transaction to be committed; even if a single shard rejects the transaction, then all agree it is rejected. We propose the use of a simple two-phase commit protocol [BHG87], composed with an agreement protocol to achieve this—loosely inspired by Lamport and Gray [GL06]. This protocol was the first to reconcile the needs for distributed commit, and replicated consensus (but only in the non-byzantine setting).

$\mathcal{S}$-BAC composes the above primitives in a novel way to ensure that shards process safely and consistently all transactions. Figure 4 illustrates a simple example of the $\mathcal{S}$-BAC protocol to commit a single transaction with two inputs and one output that we may use as an example. The corresponding object state transitions have been illustrated in Figure 3. The combined protocol has been described below. For ease of understanding, in our description we state that all messages are sent and processed by shards. In reality, some of these are handled by a designated node in each shard—the BFT-Initiator —as we discuss at the end of this section.

Initial Broadcast (Prepare). A user acts as a transaction initiator, and sends 'prepare(T)' to at least one honest concerned node for transaction T. To ensure at least one honest node receives it, the user may send the message to $f + 1$ nodes of a single shard, or $f + 1$ nodes in each concerned shard.

Sequence Prepare. Upon a message 'prepare(T)' being received, nodes in each shard interpret it as the initiation of a two-phase commit protocol performed across the concerned shards. The shard locally sequences 'prepare(T)' message through the Byzantine consensus protocol.

Process Prepare. Upon the first action 'prepare(T)' being sequenced through BFT consensus in a shard, nodes of the shard implicitly decide whether it should be committed or aborted. Since all honest nodes in the shard have a consistent replica of the full sequence of actions, they will all decide the same consistent action following 'prepare(T)'.

Transaction T is to be committed if it is valid according to the usual rules (see Figure 2), in brief: (1) the objects input or referenced by T in the shard are active, (2) there is no other instance of the two-phase commit protocol on-going concerning any of those objects (no locks held) and (3) if T is valid according to the validity rules, and the smart contract checkers in the shard. Only the checkers for types of objects held by the shard are checked by the shard.

If the decision is to commit, the shard broadcasts to all concerned nodes 'prepared(T,commit)', otherwise it broadcasts 'prepared(T, abort)'—along with sufficient signatures to convince any party of the collective shard decision (we denote this action as LOCALPREPARED(*, T)). The objects used or referenced by T are 'locked' (Figure 3) in case of a 'prepared commit' until an 'accept' decision on the transaction is reached, and subsequent transactions concerning them will be aborted by the shard. Any subsequent 'prepare(T'')' actions in the sequence are ignored, until a matching accept(T, abort) is reached to release locks, or forever if the transaction is committed.

Process Prepared (accept or abort). Depending on the decision of 'prepare(T)', the shard sequences 'accept(T,commit)' or 'accept(T,abort)' through the atomic commit protocol across all the concerned shards—along with all messages and signatures of the bundle of 'prepared' messages relating to T proving to other shards that the decision should be 'accept(T,commit)' or 'accept(T,abort)' according to its local consensus. If it receives even a single 'LOCALPREPARED(T,abort)' from another shard it instead will move to reach consensus on 'accept(T, abort)' (denoted as SOMEPREPARED(abort,T)). Otherwise, if all the shards respond with 'LOCALPREPARED(T,commit)' it will reach a consensus on ALLPREPARED(commit,T). The final decision is sent to the user, along with all messages and signatures of the bundle of 'accept' messages relating to T proving that the final decision should be to commit or abort according to responses from all concerned shards.

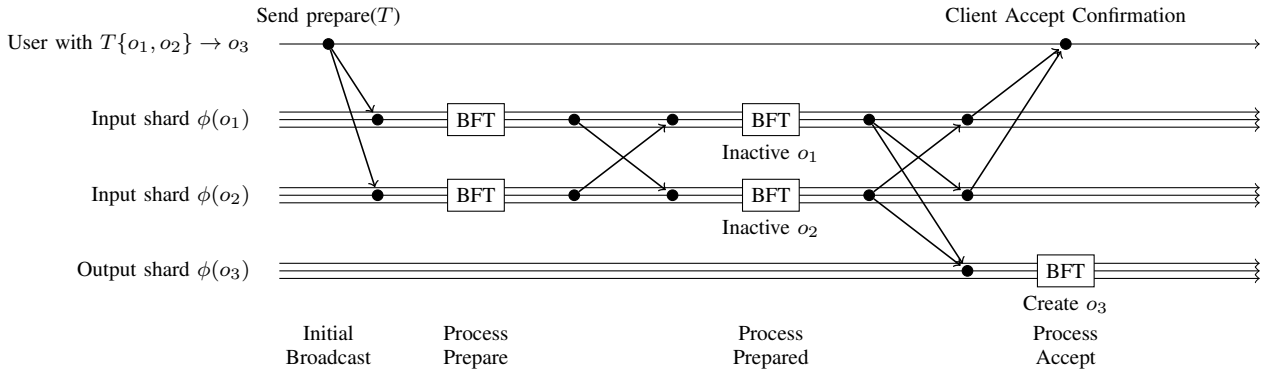

Fig. 4. $\mathcal{S}$-BAC for a transaction T with two inputs (o_1, o_2) and one output object (o_3). The user sends the transaction to all nodes in shards managing o_1 and o_2. The BFT-Initiator takes the lead in sequencing T, and emits 'prepared(accept, T)' or 'prepared(abort, T)' to all nodes within the shard. Next the BFT-Initiator of each shard assesses whether overall 'All proposed(accept, T)' or 'Some proposed(abort, T)' holds across shards, sequences the accept(T,*), and sends the decision to the user. All cross-shard arrows represent a multicast of all nodes in one shard to all nodes in another.

It is possible, that a shard hears a prepared message for T before a prepare message, due to unreliability, asynchrony or a malicious user. In that case the shard assumes that a 'prepare(T)' message is implicit, and sequences it.

Process Accept. When a shard sequences an 'accept(T, commit)' decision, it sets all objects that are inputs to the transaction T as being inactive (Figure 3). It also creates any output objects from T via BFT consensus that are to be managed by the shard. If the output objects are not managed by the shard, the shard sends requests to the concerned shards to create the objects. On the other hand if the shard decision is 'accept(T, abort)', all nodes release locks held on inputs or references of transaction T. Thus those objects remain active and may be used by other transactions.

As previously mentioned, some of the messages in $\mathcal{S}$-BAC are handled by a designated node in each shard called the BFT-Initiator . Specifically, the BFT-Initiator drives the composed $\mathcal{S}$-BAC protocol by sending 'prepare(T)' and then 'accept(T, *)' messages to reach BFT consensus within and across shards. It is also responsible for broadcasting consensus decisions to relevant parties. The protocol supports a two-phase process to recover from a malicious BFT-Initiator that suppresses transactions. As nodes in a shard hear all messages, they wait for the BFT-Initiator to act on it until they time out. They first send a reminder to the BFT-Initiator along with the original message to account for network losses. Next they proceed to wait; if they time out again, other nodes perform the action of BFT-Initiator which is idempotent.

D. Concurrency & Scalability

Each transaction T involves a fixed number of *concerned nodes* $\Phi(T)$ within Chainspace, corresponding to the shards managing its inputs and references. If two transactions T_0 and T_1 have disjoint sets of concerned nodes ($\Phi(T_0) \cap \Phi(T_1) = \emptyset$) they cannot conflict, and are executed in parallel or in any arbitrary order. If however, two transactions have common input objects, only one of them is accepted by all nodes. This is achieved through the $\mathcal{S}$-BAC protocol. It is local, in that it concerns only nodes managing the conflicting transactions, and does not require a global consensus.

From the point of view of scalability, Chainspace capacity grows linearly as more shards are added, subject to transactions having on average a constant, or sub-linear, number of inputs and references (see Figure 6). Furthermore, those inputs must be managed by different nodes within the system to ensure that load of accepting transactions is distributed across them.

V. Security and Correctness

A. Security & Correctness of $\mathcal{S}$-BAC

The $\mathcal{S}$-BAC protocol guarantees a number of key properties, on which rest the security of Chainspace, namely *liveness consistency*, and *validity*. Before proceeding with stating those properties in details, and proving them we note three key invariants, that nodes may decide:

- LocalPrepared(commit / abort, T): A node considers that LocalPrepared(commit / abort, T) for a shard holds, if it receives at least $f + 1$ distinct signed messages from nodes in the shard, stating 'prepared(commit, T)' or 'prepared(abort, T)' respectively. As a special case a node automatically concludes LocalPrepared(commit / abort, T) for a shard it is a member of, if all the preconditions necessary to provide that answer are present when an 'prepare(T)' is sequenced.
- AllPrepared(commit, T): A node considers that 'AllPrepared(commit, T)' holds if it believes that 'LocalPrepared(commit, T)' holds for all shards with concerned nodes for T. Note this may only be decided after reaching a conclusion (e.g. through receiving signed messages) about all shards.
- SomePrepared(abort, T): A node considers that 'SomePrepared(abort, T)' holds if it believes that 'LocalPrepared(abort, T)' holds for at least one shard with concerned nodes for T. This may be concluded after only reaching a conclusion for a single shard, including the shard the node may be part of.

Liveness ensures that transactions make progress once proposed by a user, and no locks are held indefinitely on objects, preventing other transactions from making progress.

$\mathcal{S}$-BAC Theorem 1. *Liveness: Under the 'honest shards' threat model, a transaction T that is proposed to at least one honest concerned node, will eventually result in either being committed or aborted, namely all parties deciding accept(commit, T) or accept(abort, T).*

Proof: We rely on the liveness properties of the byzantine agreement (shards with only f nodes will reach a consensus on a sequence), and the broadcast from nodes of shards to all other nodes of shards, including the shards that manage transaction outputs. Assuming prepare(T) has been given to an honest node, it will be sequenced withing an honest shard BFT sequence, and thus a prepared(commit, T) or prepared(abort, T) will be sent from the $2f + 1$ honest nodes of this shard, to the $2f + 1$ nodes of the other concerned shards. Upon receiving these messages the honest nodes from other shards will schedule a prepare(T) message within their shards, and the BFT will eventually sequence it. Thus the user and all other honest concerned nodes will receive enough 'prepared' messages to decide whether to proceed with 'ALLPREPARED(commit, T)' or 'SOMEPREPARED(abort, T)' and proceed with sequencing them through BFT. Eventually, each shard will sequence those, and decide on the appropriate 'accept'. ■

The second key property ensures that the execution of all transactions could be serialized, and thus is correct.

$\mathcal{S}$-BAC Theorem 2. *Consistency: Under the 'honest shards' threat model, no two conflicting transactions, namely transactions sharing the same input will be committed. Furthermore, a sequential executions for all transactions exists.*

Proof: A transaction is committed only if some nodes conclude that 'ALLPREPARED(commit, T)', which presupposes all shards have provided enough evidence to conclude 'LOCALPREPARED(commit, T)' for each of them. Two conflicting transaction, sharing an input or reference, must share a shard of at least $3f + 1$ concerned nodes for the common object—with at most f of them being malicious. Without loss of generality upon receiving the prepare(T) message for the first transaction, this shard will sequence it, and the honest nodes will emit messages for all to conclude 'ALLPREPARED(commit, T)'—and will lock this object until the two phase protocol concludes. Any subsequent attempt to prepare(T') for a conflicting T' will result in a LOCALPREPARED(abort, T') and cannot yield a commit, if all other shards are honest majority too. After completion of the first 'accept(commit, T)' the shard removes the object from the active set, and thus subsequent T' would also lead to SOMEPREPARED(abort, T'). Thus there is no path in the chain of possible interleavings of the executions of two conflicting transactions that leads to them both being committed. ■

$\mathcal{S}$-BAC Theorem 3. *Validity: Under the 'honest shards' threat model, a transaction may only be committed if it is valid according to the smart contract checkers matching the traces of the procedures it executes.*

Proof: A transaction is committed only if some nodes conclude that 'ALLPREPARED(commit, T)', which presupposes all shards have provided enough evidence to conclude 'LOCALPREPARED(commit, T)' for each of them. The concerned nodes include at least one shard per input or reference object for the transaction; for any contract c represented in the transaction, at least one of those shards will be managing object from that contract. Each shard checks the validity rules for the objects they manage (ensuring they are active, and not locked) and the contracts those objects are part of (ensuring the calls to c pass its checker) in order to LOCALPREPARED(accept, T). Thus if all shards say LOCALPREPARED(accept, T) to conclude that 'ALLPREPARED(commit, T)', all object have been checked as active, and all the contract calls within the transaction have been checked by at least one shard—whose decision is honest due to at most f faulty nodes. If even a single object is inactive or locked, or a single trace for a contract fails to check, then the honest nodes in the shard will emit 'prepared(abort, T)' upon sequencing 'prepare(T)', and the final decision will be 'SOMEPREPARED(abort, T)'. ■

B. Auditability

In the previous sections we show that if each shard contains at most f faulty nodes (honest shard model), the $\mathcal{S}$-BAC protocol guarantees consistency and validity. In this section we argue that if this assumption is violated, i.e. one or more shards contain more than f byzantine nodes each, then honest shards can detect faulty shards. Namely, enough auditing information is maintained by honest nodes in Chainspace to detect inconsistencies and attribute them to specific shards (or nodes within them).

The rules for transaction validity are summarized in Figure 2. Those rules are checked in a distributed manner: each shard keeps and checks the active or inactive state of objects assigned to it; and also only the contract checkers corresponding to the type of those objects. An honest shard emits a proposed(T, commit) for a transaction T only if those checks pass, and proposed(T, abort) otherwise or if there is a lock on a relevant object. A dishonest shard may emit proposed(T, *) messages arbitrarily without checking the validity rules. By definition, an invalid transaction is one that does not pass one or more of the checks defined in Figure 2 at a shared, for which the shard has erroneously emitted a proposed(T, commit) message.

Security Theorem 3. *Auditability: A malicious shard (with more than f faulty nodes) that attempts to introduce an invalid transaction or object into the state of one or more honest shards, can be detected by an auditor performing a full audit of the Chainspace system.*

Proof: We consider two hash-chains from two distinct shards. We define the pair of them as being valid if (1) they are each valid under full audit, meaning that a re-execution of all their transactions under the messages received yields the same decisions to commit or abort all transactions; and (2) if all prepared(T,*) messages in one chain are compatible with all messages seen in the other chain. In this context 'compatible' means that all prepared(T,*) statements received in one shard from the other represent the 'correct' decision to commit or abort the transaction T in the other shard. An example of incompatible message would result in observing a proposed(T, commit) message being emitted from the first shard to the second, when in fact the first shard should have aborted the transaction, due to the checker showing it is invalid or an input being inactive.

Due to the property of digital signatures (unforgeability and non-repudiation), if two hash-chains are found to be 'incompatible', one belonging to an honest shard and one belonging to a dishonest shard, it is possible for everyone to determine which shard is the dishonest one. To do so it suffices to isolate all statements that are signed by each shard (or a peer in the shard)—all of which should be self-consistent. It is then possible to show that within those statements there is an inconsistency—unambiguously implicating one of the two shards in the cheating. Thus, given two hash-chains it is possible to either establish their consistency, under a full audit, or determine which belongs to a malicious shard. ■

Note that the mechanism underlying tracing dishonest shards is an instance of the age-old double-entry book keeping[1]: shards

[1]The first reported use is 1340AD [LW94].

keep records of their operations as a non-repudiable signed hash-chain of checkpoints—with a view to prove the correctness of their operations. They also provide non-repudiable statements about their decisions in the form of signed proposed(T,*) statements to other shards. The two forms of evidence must be both correct and consistent—otherwise their misbehaviour is detected.

VI. System and Applications Smart Contracts

A. System Contracts

The operation of a Chainspace distributed ledger itself requires the maintenance of a number of high-integrity high-availability data structures. Instead of employing an ad-hoc mechanism, Chainspace employs a number of *system smart contracts* to implement those. Effectively, instantiation of Chainspace is the combination of nodes running the basic $\mathcal{S}$-BAC protocol, as well as a set of system smart contracts providing flexible policies about managing shards, smart contract creation, auditing and accounting. This section provides an overview of system smart contracts.

Shard management. The discussion of Chainspace so far, has assumed a function $\phi(o)$ mapping an object o to nodes forming a shard. However, how those shards are constituted has been abstracted. A smart contract ManageShards is responsible for mapping nodes to shards. ManageShards initializes a singleton object of type MS.Token and provides three procedures: MS.create takes as input a singleton object, and a list of node descriptors (names, network addresses and public verification keys), and creates a new singleton object and a MS.Shard object representing a new shard; MS.update takes an existing shard object, a new list of nodes, and $2f + 1$ signatures from nodes in the shard, and creates a new shard object representing the updated shard. Finally, the MS.object procedure takes a shard object, and a non-repudiable record of malpractice from one of the nodes in the shard, and creates a new shard object omitting the malicious shard node—after validating the misbehaviour. Note that Chainspace is 'open' in the sense that any nodes may form a shard; and anyone may object to a malicious node and exclude it from a shard.

Smart-contract management. Chainspace is also 'open' in the sense that anyone may create a new smart contract, and this process is implemented using the ManageContracts smart contract. ManageContracts implements three types: MC.Token, MC.Mapping and MC.Contract. It also implements at least one procedure, MC.create that takes a binary representing a checker for the contract, an initialization procedure name that creates initial objects for the contract, and the singleton token object. It then creates a number of outputs: one object of type MC.Token for use to create further contracts; an object of type MC.Contract representing the contract, and containing the checker code, and a mapping object MC.mapping encoding the mapping between objects of the contract and shards within the system. Furthermore, the procedure MC.create calls the initialization function of the contract, with the contract itself as reference, and the singleton token, and creates the initial objects for the contract.

Note that this simple implementation for ManageContracts does not allow for updating contracts. The semantics of such an update are delicate, particularly in relation to governance and backwards compatibility with existing objects. We leave the definitions of more complex, but correct, contracts for managing contracts as future work. In our first implementation we have hardcoded ManageShards and ManageContracts.

Payments for processing transactions. Chainspace is an open system, and requires protection againt abuse resulting from overuse. To achieve this we implement a method for tracking value through a contract called CSCoin.

The CSCoin contract creates a fixed initial supply of coins—a set of objects of type The CSCoin.Account that may only be accessed by a user producing a signature verified by a public key denoted in the object. A CSCoin.transfer procedure allows a user to input a number of accounts, and transfer value between them, by producing the appropriate signature from incoming accounts. It produces a new version of each account object with updated balances. This contract has been implemented in Python with approximately 200 lines of code.

The CSCoin contract is designed to be composed with other procedures, to enable payments for processing transactions. The transfer procedure outputs a number of local returns with information about the value flows, that may be used in calling contracts to perform actions conditionally on those flows. Shards may advertise that they will only consider actions valid if some value of CSCoin is transferred to their constituent nodes. This may apply to system contracts and application contracts.

B. Application level smart contracts

This section describes some examples of privacy-friendly smart contracts and showcases how smart contract creators may use Chainspace to implement advanced privacy mechanisms.

Smart-Meter Private Billing.

We implement a basic private smart-meter billing mechanism [JJK11, RD12] using the contract SMet: it implements three types SMet.Token, SMet.Meter and SMet.Bill; and three procedures, SMet.createMeter, SMet.AddReading, and SMet.computeBill. The procedure SMet.createMeter takes as input the singletone token and a public key and signature as local parameters, and it outputs a SMet.Meter object tied to this meter public key if the signature matches. SMet.Meter objects represent a collection of readings and some meta-data about the meter. Subsequently, the meter may invoke SMet.addReading on a SMet.Meter with a set of cryptographic commitments readings and a period identifier as local parameters, and a valid signature on them. A signature is also included and checked to ensure authenticity from the meter. A new object SMet.Meter is output appending the list of new readings to the previous ones. Finally, a procedure SMet.computeBill is invoked with a SMet.Meter and local parameters a period identifier, a set of tariffs for each reading in the period, and a zero-knowledge proof of correctness of the bill computation. The procedure outputs a SMet.Bill object, representing the final bill in plain text and the meter and period information.

This proof of correctness is provided to the checker—rather than the secret readings—which proves that the readings matching the available commitments and the tariffs provided

yield the bill object. The role of the checker, which checks public data, in both those cases is very different from the role of the procedure that is passed secrets not available to the checkers to protect privacy. This contracts has been implemented in about 200 lines of Python code and is evaluated in section Section VII.

A Platform for Decision Making. An additional example of Chainspace's privacy-friendly application is a smart voting system. We implement the contract SVote with three types, SVote.Token, SVote.Vote and SVote.Tally; and three procedures.

SVote.createElection, consumes a singleton token and takes as local parameters the options, a list of all voter's public key, the tally's public key, and a signature on them from the tally. It outputs a fresh SVote.Vote object, representing the initial stage of the election (all candidates having a score of zero) along with a zero-knowledge proof asserting the correctness of the initial stage.

SVote.addVote, is called on a SVote.Vote object and takes as local parameters a new vote to add, homomorphically encrypted and signed by the voter. In addition, the voter provides a zero-knowledge proof certifying that her vote is a binary value and that she voted for exactly one option. The voter's public key is then removed from the list of participants to ensure that she cannot vote more than once. If all proofs are verified by the checker and the voter's public key appears in the list, a new SVote.Vote object is created as the homomorphic addition of the previous votes with the new one. Note that the checker does not need to know the clear value of the votes to assert their correctness since it only has to verify the associated signatures and zero-knowledge proofs.

Finally, the procedure SVote.tally is called to threshold decrypt the aggregated votes and provide a SVote.Tally object representing the final election's result in plain text, along with a proof of correct decryption from the tally. The SVote contract's size is approximately 400 lines.

VII. Implementation & Evaluation

We implemented a prototype of Chainspace in ∼10K lines of Python and Java code. The implementation consists of two components: a Python contracts environment and a Java node. We have released the code as an open-source project on GitHub.[2]

Python Contract Environment. The Python contracts environment allows developers to write, deploy and test smart contracts. These are deployed on each node by running the Python script for the contract, which starts a local web service for the contract's checker. The contract's checker is then called though the web service. The environment provides a framework to allow developers to write smart contracts with little worry about the underlying implementation, and provides an auto-generated checker for simple contracts.

Java Node Implementation. The Java node implements a shard replica that accepts incoming transactions from clients and initiates, and executes, the $\mathcal{S}$-BAC protocol. For BFT consensus, we use the BFT-SMaRt [BSA14] Java library—one of the very few maintained open source libraries implementing byzantine consensus.

[2]URL omitted for double-blind review.

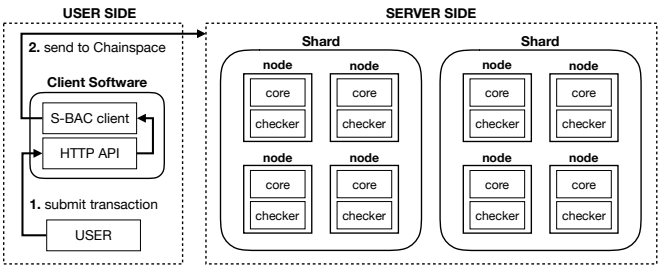

Fig. 5. Diagram illustrating the implementation of a Chainspace system with two shards managing four nodes each. The user submits the transaction to its local $\mathcal{S}$-BAC client through a built-in HTTP API (arrow 1). Then, this $\mathcal{S}$-BAC client sends the transaction to Chainspace (arrow 2).

To communicate with Chainspace, end users also run an $\mathcal{S}$-BAC–enabled client. First, she creates a transaction through the Python environments using one or many existing smart contracts. She then submits the transaction to its $\mathcal{S}$-BAC client through the HTTP API as indicated in Figure 5, that sends the transaction to Chainspace according to the BFT-SMaRt protocol.

A node is composed of a server divided in two parts: the core and the checker. To communicate with other nodes, each node also contains an $\mathcal{S}$-BAC client. When a transaction is received, the core is in charge of verifying that the input objects and references are active (neither locked nor inactive). Then, the node runs the checker associated with the contract, in an isolated container. (The checker is provided by the contract's creator when the node starts up, and interfaces with the node through an HTTP API.) When the client submits a transaction with dependencies, the core recursively checks each dependent transaction first, and the top-level transaction at last (similar to depth-first search algorithm).

Performance Measurements. We evaluated the performances and scalability of our implementation of Chainspace, through deployments on Amazon EC2 containers. We launched up to 96 nodes on *t2.medium* virtual machines, each containing 8 GB of RAM on 2 virtual CPUs and running GNU/Linux Debian 8.1. We sent transactions to the network from a Chainspace client running on a t2.xlarge virtual machine, containing 16 GB of RAM and 4 virtual CPUs, also running GNU/Linux Debian 8.1. In our tests, we map objects to shards randomly using the mapping function $\phi(o) = id(o) \mod K$ where K is a constant representing the number of shards and $id(o)$ is the SHA256 hash of the object.

We first measure the effect of the number of shards on transaction throughput (Figure 6). The transaction throughput of Chainspace scales linearly with the number of shards: with 4 nodes per shard, the number of transactions per second (t/s) increases on average by 22 for 1-input transactions for each shard added. This is because as inputs are randomly assigned to shards based on their hashes, the transaction processing load is spread out over a larger number of shards.

Next we investigate the effect of shard size (the number of nodes per shard) on transaction throughput (Figure 9). We fix the number of shards to 2, and increase the number of

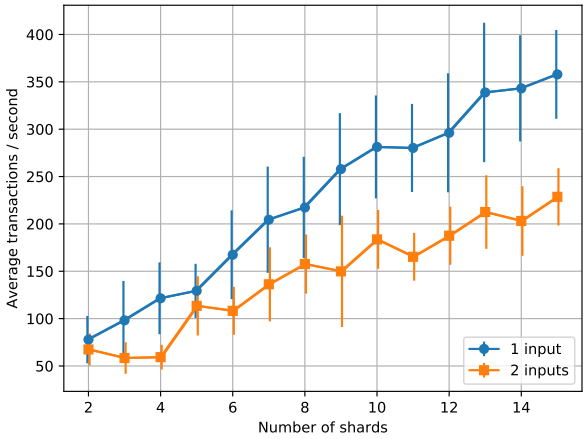

Fig. 6. The effect of the number of shards on transaction throughput. (Shards: 2, nodes per shard: 4, input-to-shard mapping: random. Repeats: 20.)

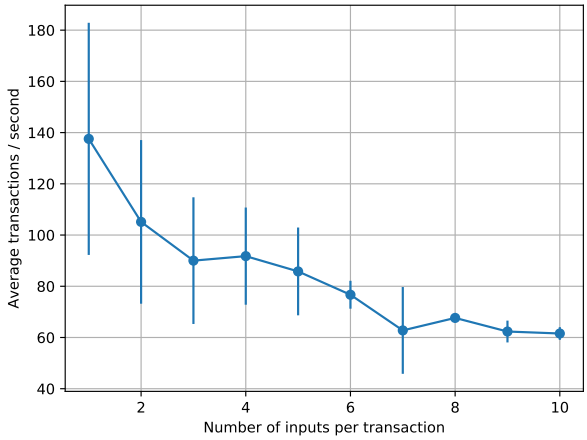

Fig. 7. The effect of the number of inputs per transaction on transaction throughput. (Shards: 2, nodes per shard: 4, input-to-shard mapping: random. Repeats: 20.)

nodes per shard from 2 to 48. With BFT-SMaRt configured for $3f + 1$ fault tolerance, we observe an expected gracious decrease in transaction throughput: for each node added, the throughput reduces on average by 1.6 transactions per second. This is because in order for a BFT-SMaRt node to realise consensus for a message, it must receive a result from at least $f + 1$ nodes. Thus, the bottleneck is the latency of the $f + 1$th node with the highest response time.

Another factor that can potentially affect transaction throughput is the number of inputs per transaction: the more shards touched by the transaction inputs, the longer it will take to run $\mathcal{S}$-BAC among all the concerned shards. In Figure 7, we study how the number of inputs per transaction affects transaction throughput. We measure this for 5 shards, varying the number of inputs per transaction from 1 to 10, and the inputs are randomly mapped to shards as previously stated. The transaction throughput decreases asymptotically until it becomes stable at around 40 transactions per second. This is because $\mathcal{S}$-BAC's maximum time in processing transactions is capped at the time it takes to process transactions that touch all the 5 shards. Increasing the number of inputs does not further deteriorate the transaction throughput.

Finally, we measure the client-perceived latency—the time from when a client submits a transaction until it receives a decision about whether the transaction has been committed—under varying system loads expressed in terms of transactions received per second. Figure 8 shows the effect of transactions received by the system per second (all 1-input transactions) on client-perceived latency for 2 shards, each having 4 nodes. Recall from Figure 6 that the average throughput for a Chainspace system with similar configuration is 75 1-input transactions per second. Consequently, we observe in Figure 9 that the increase in latency with varying system loads is smaller for 20 t/s–60 t/s (average 69 ms), but the values start to get bigger after 60 t/s (average 210 ms). This is when the system reaches its maximum transaction throughput, causing a backlog of transactions to be processed.

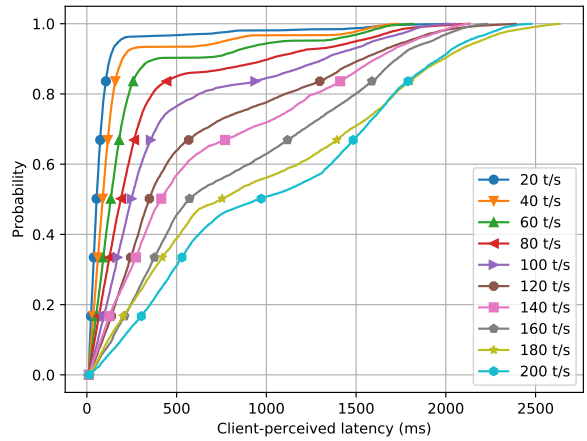

Fig. 8. The cumulative distribution function of delay for the client to receive a final commit or abort response, for varying system load. (Shards: 2, nodes per shard: 4, inputs per transaction: 1, input-to-shard mapping: random. Repeats: 5.)

Smart Contract Benchmarks. We evaluate the cost and performance of some smart contracts described in Section VI-A. We compute the mean (μ) and standard deviation (σ) of the execution of each procedure (denoted as [g]) and checker (denoted as [c]) in the contracts. Each figure is the result of 10,000 measured on a dual-core Apple MacBook Pro 4.1, 2.7GHz Intel Core i7. The last column indicates the transaction's size resulting from executing the procedure. All cryptographic operations as digital signatures and zero-knowledge proofs have been implemented using the Python library petlib [pet17], wrapping OpenSSL.

CSCoin—Contract size: ∼200 lines

Operation		μ [ms]	σ [ms]	size [B]
createAccount	[g]	4.845	± 0.683	512
	[c]	0.022	± 0.005	-
authTransfer	[g]	4.986	± 0.684	1114
	[c]	5.750	± 0.474	-

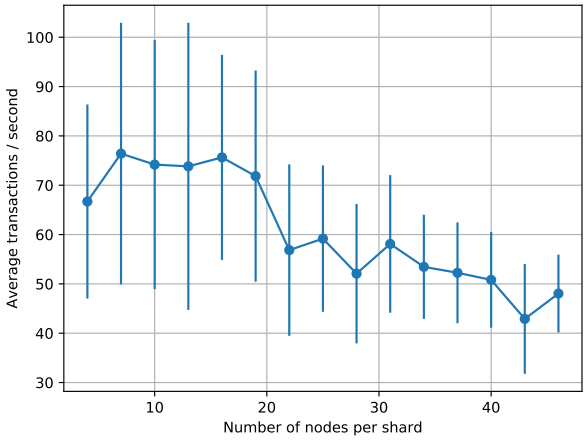

Fig. 9. The effect of the number of nodes per shard on transaction throughput. (Shards: 2, inputs per transaction: 1, input-to-shard mapping: random. Repeats: 20.)

The user needs to generate a signing key pair to create an account in the CSCoin contract, which takes about 5 ms. However, verifying the account creation only requires to check the transaction's format, and it is therefore very fast. Transferring money is a little more expensive due to the need to sign the amount transferred and the beneficiary, and verifying the signature in the checker.

SMet—Contract size: ∼200 lines

Operation	μ [ms]	σ [ms]	size [B]
createMeter [g]	4.786	± 0.480	∼600
[c]	0.060	± 0.003	-
addReading [g]	5.286	± 0.506	∼1100
[c]	5.965	± 0.697	-
computeBill [g]	5.043	± 0.513	∼1100
[c]	5.870	± 0.603	-

Similarly to CSCoin, creating a meter requires generating a cryptographic key pair which takes about 5 ms, while verifying the meter's creation is faster and only requires checking the transaction's format. Adding new readings takes about 5 ms, as the user needs to create a signed commitment of the readings which requires elliptic curve operations and an ECDSA signature. Computing the bill takes slightly longer (5.8 ms), and involves homomorphic additions, and verifying the bill involves checking a zero-knowledge proof of the billing calculation.

SVote—Contract size: ∼400 lines

Operation	μ [ms]	σ [ms]	size [B]
createElection [g]	11.733	± 1.028	∼1227
[c]	11.327	± 0.782	-
addVote [g]	14.086	± 1.043	∼2758
[c]	28.178	± 1.433	-
tally [g]	253.286	± 7.793	∼1264
[c]	11.589	± 0.937	-

The SVote contract is more expensive than the others since it extensively uses zero-knowledge proofs and more advanced cryptography. For simplicity, this smart contract has been tested with three voters and two options. First of all, creating a new election event requires building a signed homomorphic encryption of the initial value for each option, and a zero-knowledge proof asserting that the encrypted value is zero; this takes roughly 11 ms to generate the transaction and to run the checker. Next, each time a vote is added, the user proves two zero-knowledge statements—one asserting that she votes for exactly one option and one proving that her vote is a binary value—and computes an ECDSA signature on her vote, which takes about 11 ms and generates a transaction of about 2.7 kB. Verifying the signature and the two zero-knowledge proofs are slower and takes about 30 ms. Finally, tallying is the slowest operation since it requires to decrypt the homomorphic encryption of the votes' sum.

VIII. Limitations

Chainspace has a number of limitations, that are beyond the scope of this work to tackle, and deferred to future work.

The integrity properties of Chainspace rely on all shards managing objects being honest, namely containing at most f fault nodes each. We have chosen to let any set of nodes can create a shard. However, this means that the function $\phi(o)$ mapping objects to shards must avoid dishonest shards. Our isolation properties ensure that a dishonest shard can at worse affect state from contracts that have objects mapped to it. Thus, in Chainspace, we opt to allow the contract creator to designate which shards manage objects from their contract. This embodies specific trust assumptions where users have to trust the contract creator both for the code (which is auditable) and also for the choice of shards to involve in transactions—which is also public.

In case one or more shards are malicious, we provide an auditing mechanism for honest nodes in honest shards to detect the inconsistency and to trace the malicious shard. Through the Hash-DAG structure it is also possible to fully audit the histories of two objects, and to ensure that the validity rules hold jointly—in particular the double-use rules. However, it is not clear how to automatically recover from detecting such an inconsistency. Options include: forcing a fork into one or many consistent worlds; applying a rule to collectively agree the canonical version; patching past transactions to recover consistency; or agree on a minimal common consistent state. Which of those options is viable or best is left as future work.

Checkers involved in validating transactions can be costly. For this reason we allow peers in a shard to accept transactions subject to a SCCoin payment to the peers. However, this 'flat' fee is not dependent on the cost or complexity of running the checker which might be more or less expensive. Etherium [Woo14] instead charges 'gas' according to the cost of executing the contract procedure—at the cost of implementing their own virtual machine and language.

Finally, the $\mathcal{S}$-BAC protocol ensures correctness in all cases. However, under high contention for the same object the rate of aborted transactions rises. This is expected, since the $\mathcal{S}$-BAC protocol in effect implements a variant of optimistic concurrency control, that is known to result in aborts under high contention. There are strategies for dealing with this in the distributed systems literature, such as locking objects in some conventional order—however none is immediately applicable to the byzantine setting.

IX. Comparisons with Related Work

Bitcoin's underlying blockchain technology suffers from scalability issues: with a current block size of 1MB and 10 minute inter-block interval, throughput is capped at about 7 transactions per second, and a client that creates a transaction has to wait for about 10 minutes to confirm. In contrast, mainstream payment processing companies like Visa confirm transactions within a few seconds, and have high throughput of 2000 transactions per second on average, peaking up to 56,000 transactions per second [vis]. Reparametrization of Bitcoin—such as Bitcoin-NG—can improve this to a limited extent up to 27 transactions per second and 12 second latency, respectively [CDE+16]. More significant improvement requires a fundamental redesign of the blockchain paradigm.

The most comparable system to Chainspace is OmniLedger [KJG+17]—that was developed concurrently—and provides a scalable distributed ledger for a cryptocurrency, and cannot support generic smart contracts. OmniLedger assigns nodes (selected using a Sybil-attack resistant mechanism) into shards among which state, representing coins, is split. The node-to-shard assignment is done every epoch using a bias-resistant decentralized randomness protocol [SJK+16] to prevent an adversary from compromising individual shards. A block-DAG (Directed Acyclic Graph) structure is maintained in each shard rather than a single blockchain, effectively creating multiple blockchains in which consensus of transactions can take place in parallel. Nodes within shards reach consensus through the Practical Byzantine Fault Tolerant (PBFT) protocol [CL+99] with ByzCoin [KJG+16]'s modifications that enable $O(n)$ messaging complexity. In contrast, Chainspace uses BFT-SMaRt 's PBFT implementation [SB12] as a black box, and inherits its $O(n^2)$ messaging complexity—however, BFT-SMaRt can be replaced with any improved PBFT variant without breaking any security assumptions.

Similar to Chainspace, OmniLedger uses an atomic commit protocol to process transactions across shards. However, it uses a different, client-driven approach to achieve it. To commit a transaction, the client first sends the transaction to the network. The leader of each shard that is responsible for the transaction inputs (input shard) validates the transaction and returns a proof-of-acceptance (or proof-of-rejection) to the client, and inputs are locked. To unlock those inputs, the client sends proof-of-accepts to the output shards, whose leaders add the transaction to the next block to be appended to the blockchain. In case the transaction fails the validation test, the client can send proof-of-rejection to the input shards to roll back the transaction and unlock the inputs. To avoid denial-of-service, the protocol assumes that clients are incentivized to proceed to the Unlock phase. Such incentives may exist in a cryptocurrency application, where coin owners only can spend them, but do not hold for a generalized platform like Chainspace where objects may have shared ownership. Hence, Chainspace's atomic commit protocol has the entire shard—rather than a single untrusted client—act as a coordinator. Other related works include improvements to Byzantine consensus for reduced latency and decentralization [Buc16, Maz15, SYB14], but these do not support sharding.

Elastico [LNZ+16] scales by partitioning nodes in the network into a hierarchy of committees, where each committee is responsible for managing a subset (shard) of transactions consistently through PBFT. A final committee collates sets of transactions received from committees into a final block and then broadcasts it. At the end of each epoch, nodes are reassigned to committees through proof-of-work. The block throughput scales up almost linear to the size of the network. However, Elastico cannot process multi-shard transactions.

RSCoin [DM16] is a permissioned blockchain. The central bank controls all monetary supply, while mintettes (nodes authorized by the bank) manage subsets of transactions and coins. Like OmniLedger, communication between mintettes takes place indirectly, through the client—and also relies on the client to ensure completion of transactions. RSCoin has low communication overhead, and the transaction throughput scales linearly with the number of mintettes, but cannot support generic smart contracts.

Some systems improve transaction latency by replacing its probabilistic guarantees with strong consistency. ByzCoin [KJG+16] extends Bitcoin-NG for high transaction throughput. A consensus group is organized into a communication tree where the most recent miner (the leader) is at the root. The leader runs an $O(n)$ variant of PBFT (using CoSI) to get all members to agree on the next microblock. The outcome is a collective signature that proves that at least two-thirds of the consensus group members witnessed and attested the microblock. A node in the network can verify in $O(1)$ that a microblock has been validated by the consensus group. PeerConsensus [DSW16] achieves strong consistency by allowing previous miners to vote on blocks. A *Chain Agreement* tracks the membership of identities in the system that can vote on new blocks. Algorand [Mic16] replaces proof-of-work with strong consistency by proposing a faster *graded* Byzantine fault tolerance protocol, that allows for a set of nodes to decide on the next block. A key aspect of Algorand is that these nodes are selected randomly using algorithimic randomness based on input from previously generated blocks. However, none of those systems are designed to support generic smart contracts.

Some recent systems provide a transparent platform based on blockchains for smart contracts. Hyperledger Fabric [Cac16] is a permissioned blockchain to setup private infrastructures for smart contracts. It is designed around the idea of a 'consortium' blockchain, where a specific set of nodes are designated to validate transactions, rather than random nodes in a decentralized network. Each smart contract (called *chaincode*) has its own set of *endorsers* that re-execute submitted transactions to validate them. A *consensus service* then orders transactions and filters out those endorsed by too few. It uses *modular consensus*, which is replaceable depending on the requirements (e.g., Apache Kafka or SBFT).

Ethereum [Woo14] provides a decentralized Turing-complete virtual machine, called EVM, able to execute smart-contracts. Its main scalability limitation results from every node having to process every transaction, as Bitcoin. On the other hand, Chainspace's sharded architecture allows for a ledger linearly scalable since only the nodes concerned by the transaction—that is, managing the transaction's inputs or references—have to process it. Ethereum plans to improve scalability through sharding techniques [BCWD15], but their work is still theoretical and does not provide any implementation or measurements. One major difference with Chainspace is that Ethereum's smart contract are executed by the node,

contrarily to the user providing the outputs of each transaction. Chainspace also supports smart contracts written in any kind of language as long as checkers are pure functions, and there are no limitations for the code creating transactions. Some industrial systems [tez17, roo17, cor17] implement similar functionalities as Chainspace, but without any empirical performance evaluation.

In terms of security policy, Chainspace system implements a platform that enforces high-integrity by embodying a variant of the Clark-Wilson [CW87], proposed before smart contracts were heard of.

X. Conclusions

We presented the design of Chainspace—an open, distributed ledger platform for high-integrity and transparent processing of transactions. Chainspace offers extensibility though privacy-friendly smart contracts. We presented an instantiation of Chainspace by parameterizing it with a number of 'system' and 'application' contracts, along with their evaluation. However, unlike existing smart-contract based systems such as Ethereum [Woo14], it offers high scalability through sharding across nodes using a novel distributed atomic commit protocol called $\mathcal{S}$-BAC, while offering high auditability. We presented implementation and evaluation of $\mathcal{S}$-BAC on a real cloud-based testbed under varying transaction loads and showed that Chainspace's transaction throughput scales linearly with the number of shards by up to 22 transactions per second for each shard added, handling up to 350 transactions per second with 15 shards. As such it offers a competitive alternative to both centralized and permissioned systems, as well as fully peer-to-peer, but unscalable systems like Ethereum.

Acknowledgements. George Danezis, Shehar Bano and Alberto Sonnino are supported in part by EPSRC Grant EP/M013286/1 and the EU H2020 DECODE project under grant agreement number 732546. Mustafa Al-Bassam is supported by a scholarship from The Alan Turing Institute. Many thanks to Daren McGuinness and Ramsey Khoury for discussions about the Chainspace design.

References

[BCCG16] Jonathan Bootle, Andrea Cerulli, Pyrros Chaidos, and Jens Groth. Efficient zero-knowledge proof systems. In *Foundations of Security Analysis and Design VIII*, pages 1–31. Springer, 2016.

[BCWD15] Vitalik Buterin, Jeff Coleman, and Matthew Wampler-Doty. Notes on scalable blockchain protocols (verson 0.3.2), 2015.

[BHG87] Philip A Bernstein, Vassos Hadzilacos, and Nathan Goodman. *CONCURRENCY CONTROL AND RECOVERY IN DATABASE SYSTEMS*. Addison- Wesley, 1987.

[BSA14] Alysson Bessani, João Sousa, and Eduardo E. P. Alchieri. State machine replication for the masses with bft-smart. In *Proceedings of the 2014 44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, DSN '14, pages 355–362, Washington, DC, USA, 2014. IEEE Computer Society.

[Buc16] Ethan Buchman. Tendermint: Byzantine fault tolerance in the age of blockchains. http://atrium.lib.uoguelph.ca/xmlui/bitstream/handle/10214/9769/Buchman_Ethan_201606_MAsc.pdf, Jun 2016. Accessed: 2017-02-06.

[Cac16] Christian Cachin. Architecture of the hyperledger blockchain fabric. In *Workshop on Distributed Cryptocurrencies and Consensus Ledgers*, 2016.

[CDE+16] Kyle Croman, Christian Decker, Ittay Eyal, Adem Efe Gencer, Ari Juels, Ahmed Kosba, Andrew Miller, Prateek Saxena, Elaine Shi, and Emin Gün. On scaling decentralized blockchains. In *3rd Workshop on Bitcoin and Blockchain Research, Financial Cryptography 16*, 2016.

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

[cor17] Corda: A distributed ledger. https://docs.corda.net/_static/corda-technical-whitepaper.pdf, 2017 (visited August 9, 2017).

[CW87] David D Clark and David R Wilson. A comparison of commercial and military computer security policies. In *Security and Privacy, 1987 IEEE Symposium on*, pages 184–184. IEEE, 1987.

[DGFK14] George Danezis, Jens Groth, C Fournet, and Markulf Kohlweiss. Square span programs with applications to succinct nizk arguments. Springer Berlin Heidelberg, 2014.

[DM16] George Danezis and Sarah Meiklejohn. Centrally banked cryptocurrencies. In *Network and Distributed System Security*. The Internet Society, 2016.

[DSW16] Christian Decker, Jochen Seidel, and Roger Wattenhofer. Bitcoin meets strong consistency. In *Proceedings of the 17th International Conference on Distributed Computing and Networking*, ICDCN '16, pages 13:1–13:10, New York, NY, USA, 2016. ACM.

[GL06] Jim Gray and Leslie Lamport. Consensus on transaction commit. *ACM Transactions on Database Systems (TODS)*, 31(1):133–160, 2006.

[JJK11] Marek Jawurek, Martin Johns, and Florian Kerschbaum. Plug-in privacy for smart metering billing. In *Privacy Enhancing Technologies - 11th International Symposium, PETS 2011, Waterloo, ON, Canada, July 27-29, 2011. Proceedings*, pages 192–210, 2011.

[KJG+16] Eleftherios Kokoris Kogias, Philipp Jovanovic, Nicolas Gailly, Ismail Khoffi, Linus Gasser, and Bryan Ford. Enhancing bitcoin security and performance with strong consistency via collective signing. In *25th USENIX Security Symposium (USENIX Security 16)*, pages 279–296, Austin, TX, 2016. USENIX Association.

[KJG+17] Eleftherios Kokoris-Kogias, Philipp Jovanovic, Linus Gasser, Nicolas Gailly, and Bryan Ford. Omniledger: A secure, scale-out, decentralized ledger. *IACR Cryptology ePrint Archive*, 2017:406, 2017.

[L+01] Leslie Lamport et al. Paxos made simple. *ACM Sigact News*, 32(4):18–25, 2001.

[LCQV15] Shengyun Liu, Christian Cachin, Vivien Quéma, and Marko Vukolic. Xft: practical fault tolerance beyond crashes. *CoRR, abs/1502.05831*, 2015.

[LL94] Butler Lampson and David B Lomet. Distributed transaction processing using two-phase commit protocol with presumed-commit without log force, August 2 1994. US Patent 5,335,343.

[LLK13] Ben Laurie, Adam Langley, and Emilia Kasper. Certificate transparency. Technical report, 2013.

[LNZ+16] Loi Luu, Viswesh Narayanan, Chaodong Zheng, Kunal Baweja, Seth Gilbert, and Prateek Saxena. A secure sharding protocol for open blockchains. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, CCS '16, pages 17–30, New York, NY, USA, 2016. ACM.

[LW94] Luc Lauwers and Marleen Willekens. Five hundred years of bookkeeping: a portrait of luca pacioli. *Tijdschrift voor Economie en Management*, 39(3):289–304, 1994.

[Maz15] David Mazieres. The stellar consensus protocol: A federated model for internet-level consensus. https://www.stellar.org/papers/stellar-consensus-protocol.pdf, 2015. Accessed: 2016-08-01.

[MGGR13] Ian Miers, Christina Garman, Matthew Green, and Aviel D Rubin. Zerocoin: Anonymous distributed e-cash from bitcoin. In *Security and Privacy (SP), 2013 IEEE Symposium on*, pages 397–411. IEEE, 2013.

[Mic16] Silvio Micali. Algorand: The efficient and democratic ledger. http://arxiv.org/abs/1607.01341, 2016. Accessed: 2017-02-09.

[MMM+16] Trent McConaghy, Rodolphe Marques, Andreas Müller, Dimitri De Jonghe, Troy McConaghy, Greg McMullen, Ryan Henderson, Sylvain Bellemare, and Alberto Granzotto. Bigchaindb: a scalable blockchain database. *white paper, BigChainDB*, 2016.

[Nak08] Satoshi Nakamoto. Bitcoin: A peer-to-peer electronic cash system. 2008.

[P+91] Torben P Pedersen et al. Non-interactive and information-theoretic secure verifiable secret sharing. In *Crypto*, volume 91, pages 129–140. Springer, 1991.

[pet17] petlib. https://github.com/gdanezis/petlib, 2017 (version July 20, 2017).

[RD12] Alfredo Rial and George Danezis. Privacy-preserving smart

metering. In *ISSE 2012 - Securing Electronic Business Processes, Highlights of the Information Security Solutions Europe 2012 Conference, Brussels, Belgium, October 23-24, 2012*, pages 105–115, 2012.

[roo17] Rsk. http://www.rsk.co, 2017 (visited August 9, 2017).

[SB12] João Sousa and Alysson Bessani. From byzantine consensus to bft state machine replication: A latency-optimal transformation. In *Proceedings of the 2012 Ninth European Dependable Computing Conference*, EDCC '12, pages 37–48, Washington, DC, USA, 2012. IEEE Computer Society.

[SJK+16] Ewa Syta, Philipp Jovanovic, Eleftherios Kokoris-Kogias, Nicolas Gailly, Linus Gasser, Ismail Khoffi, Michael J. Fischer, and Bryan Ford. Scalable bias-resistant distributed randomness. *IACR Cryptology ePrint Archive*, 2016:1067, 2016.

[SRC84] Jerome H Saltzer, David P Reed, and David D Clark. End-to-end arguments in system design. *ACM Transactions on Computer Systems (TOCS)*, 2(4):277–288, 1984.

[SYB14] David Schwartz, Noah Youngs, and Arthur Britto. The ripple protocol consensus algorithm. https://ripple.com/files/ripple_consensus_whitepaper.pdf, 2014. Accessed: 2016-08-08.

[tez17] Tezos – a self-amending crypto-ledger. https://www.tezos.com/static/papers/position_paper.pdf, 2017 (visited August 9, 2017).

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

[Woo14] Gavin Wood. Ethereum: A secure decentralised generalised transaction ledger. *Ethereum Project Yellow Paper*, 151, 2014.