decode

D1.5

Intermediate Version of DECODE Architecture

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement no. 732546

decode

Project no. 732546

DECODE

DEcentralised Citizens Owned Data Ecosystem

[D1.5] ["Intermediate Version of DECODE Architecture"]

Version Number: [V1.0]

Lead beneficiary: [UCL]

Due Date: [Jan 2019]

Author(s): Alberto Sonnino (UCL), Shehar Bano (UCL), George Danezis (UCL), Jim Barritt (TW)

Editors and reviewers: Jim Barritt (TW), Denis Jaromil Roio (DYNE)

Dissemination level:		
PU	Public	x
PP	Restricted to other programme participants (including the Commission Services)	
RE	Restricted to a group specified by the consortium (including the Commission Services)	
CO	Confidential, only for members of the consortium (including the Commission Services)	

To be approved by: Francesca Bria (Chief Technology and Digital Innovation Officer, Barcelona City Hall)

Date: [31/01/2019]

This report is currently awaiting approval from the EC and cannot be not considered to be a final

Contents

1 Introduction

DECODE is an evolution of the concept of decentralised systems which leverages state of the art cryptographic techniques such as Distributed Ledgers and Attribute Based Credentials to build a system that provides its Participants the capability to store data securely, give control and transparency over with whom and for what purpose data is shared and transact with other participants or organisations. At a high level we can describe DECODE as providing the following:

- A set of specifications for distributed ledgers to support decode
- A free and open source reference implementation of a distributed ledger
- A smart rule language that can be translated and graphically represented
- A GNU/Linux based operating system that can execute signed smart rule applications
- The documentation needed for operators to write and deploy smart rules that request access to private data
- An intuitive graphical interface for participants to allow smart rules to access their private data
- An ontology of attributes for private data that is aggregated by operators
- An attribute based cryptographic implementation that can grant access to data

The core technical components in the architecture that provide these features are the following:

- Decode OS - A linux distribution derived from and packaged to contain by default all necessary decode components[2]
- TorDam - A p2p discovery and networking component built on a private subnet over network and including a means for identifying and verifying peer nodes[3]
- Chainspace - A distributed ledger implementation[1]
- Zenroom - A restricted execution environment which is very lightweight and portable and can be run either on server or restricted compute devices (e.g. smartphones)[5]
- DECODE Wallet - A smartphone (Android and IOS) mobile application that allows particpants in the decode ecosystem to interact with decode applications (e.g. record their entry for a secure petition)[4]

Please refer to Section 3.1 “High Level Architecture” for a description of how these components interact, using an example of building a secure, verifiable but privacy preserving petition application.

As can be seen, DECODE is not a single application, rather a set of components that can be used together to build applications. The DECODE project has built the components and demonstrated for particular use cases how they might be used in combination. A core architectural philosophy has been to follow the linux approach of modularisation and combination of components. Therefore each of the components, excepting the wallet can be used independently and represent significant open source contributions in their

own right. The wallet is somewhat more specific to the use cases which are demonstrated but provides a template for subsequent implementors to build further systems.

From a technology perspective, a key goal of DECODE is to bring sophisticated cryptographic and opensource technology to a wider audience of developers, enabling future projects who are concerned with privacy and control of data to have access to these exciting and valuable technologies. The DECODE project, by creating these components and making them open source has already contributed significantly to the open source community in terms of working and reusable code. In the next stages it will bring these components into a field testing phase with the pilots.

Each of these components is described in detail in the various github repositories contained in the references. This paper focuses on the core component of the distributed ledger, called Chainspace. 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 [24] for a currency, or certificate transparency [19] for certificate verification, Chainspace offers extensibility though smart contracts, like Ethereum [32]. However, users expose to Chainspace enough information about contracts and transaction semantics, to provide higher scalability through sharding across infrastructure nodes. Ethereum 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 [8, 12].

Unlike other scalable but 'permissioned' smart contract platforms, such as Hyperledger Fabric [9] or BigchainDB [21], 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.

2 Backgrounds

We present backgrounds on cross-shard atomic commit protocols and coconut [30].

2.1 Cross-Shard Atomic Commit Protocols

The blockchain is maintained by computers (called nodes) that form a distributed network. Data on the blockchain cannot be deleted. Anyone can read data from the blockchain and verify its correctness. Only special node(s) can write to the blockchain by means of a consensus protocol, to ensure that the entire network agrees on new state of the blockchain as a result of the write operation. Earlier systems like Bitcoin [25] allowed a single node to be probabilistically elected and extend the blockchain. However, such systems have low consistency (forks can be created) and low performance (high latency and low throughput). Consequently, there has been a shift to committee-based designs [7] where a group of nodes collectively extends the blockchain typically via classical consensus protocols

such as BFT [10]. While these systems offer better performance, single-committee consensus is not scalable—as every nodes handles every transaction, adding more nodes to the committee decreases throughput.

This motivated the design of *sharded* systems, where multiple committees handle a subset of all the transactions allowing parallel execution of transactions. Every committee has its own blockchain and set of objects (or unspent transaction outputs, UTXO) that they manage—committees run an 'intra-shard' consensus protocol *e.g.,* BFT within themselves and extend their blockchains in parallel. Some transactions may operate on objects handled by different shards, effectively requiring the relevant shards to run another consensus protocol—*cross-shard protocol*—to enable agreement across the shards. If any shard rejects the transaction, all relevant shards should likewise reject the transaction.

These properties are achieved by running a cross-shard consensus protocol across the relevant shards such as the two-phase atomic commit protocol. This protocol has two phases which are run by a coordinator. In the first *voting* phase, the nodes tentatively write changes locally and report their status to the coordinator. If the coordinator does not receive status message from a node (*e.g.,* because the node crashed or the status message was lost), it assumes that the node's local write failed and sends a rollback message to all the nodes to ensure any local changes are reversed. If the coordinator receives status messages from all the nodes, it initiates the second *commit* phase and sends a commit message to all the nodes so they can permanently write the changes. In the context of sharded blockchains, the atomic commit protocol operates on shards (which make the local changes associated with the voting phase via an intra-shard consensus protocol like BFT), rather than nodes. Another important consideration in the context of sharded blockchains is who will assume the role of the coordinator. There currently exist two key approaches [7]; either (*i*) the client act as coordinator, or (*ii*) the shards collectively act as coordinator.

2.2 Coconut: Threshold Issuance Selective Disclosure Credentials

Selective disclosure credentials allow the issuance of a credential to a user, and the subsequent unlinkable revelation (or 'showing') of some of the attributes it encodes to a verifier for the purposes of authentication, authorisation or to implement electronic cash. While a number of schemes have been proposed, these have limitations, particularly when it comes to issuing fully functional selective disclosure credentials without sacrificing desirable distributed trust assumptions. Some entrust a single issuer with the credential signature key, allowing a malicious issuer to forge any credential or electronic coin. Other schemes do not provide the necessary re-randomisation or blind issuing properties necessary to implement modern selective disclosure credentials. No existing scheme provides all of threshold distributed issuance, private attributes, re-randomisation, and unlinkable multi-show selective disclosure.

Coconut a novel scheme that supports distributed threshold issuance, public and private attributes, re-randomization, and multiple unlinkable selective attribute revelations. Coconut allows a subset of decentralised mutually distrustful authorities to jointly issue credentials, on public or private attributes. These credentials cannot be forged by users, or any small subset of potentially corrupt authorities. Credentials can be re-randomised before selected attributes being shown to a verifier, protecting privacy even in the case all authorities and verifiers collude.

The lack of full-featured selective disclosure credentials impacts platforms that support 'smart contracts', such as Ethereum, Hyperledger and Chainspace. They all share the limitation that verifiable smart contracts may only perform operations recorded on a

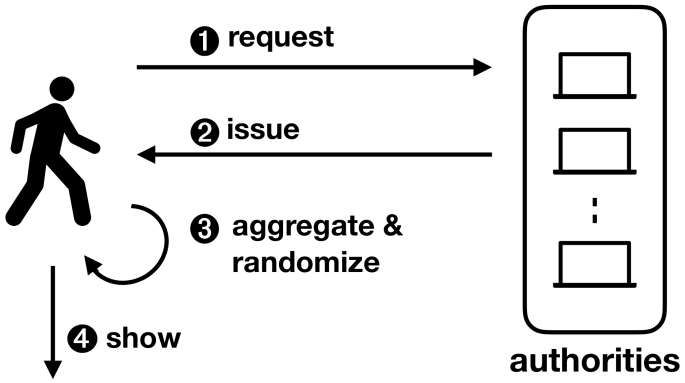

Figure 1: A high-level overview of Coconut architecture.

public blockchain. Moreover, the security models of these systems generally assume that integrity should hold in the presence of a threshold number of dishonest or faulty nodes (Byzantine fault tolerance). It is desirable for similar assumptions to hold for multiple credential issuers (threshold aggregability). Issuing credentials through smart contracts would be very useful. A smart contract could conditionally issue user credentials depending on the state of the blockchain, or attest some claim about a user operating through the contract—such as their identity, attributes, or even the balance of their wallet. As Coconut is based on a threshold issuance signature scheme, that allows partial claims to be aggregated into a single credential, it allows collections of authorities in charge of maintaining a blockchain, or a side chain based on a federated peg, to jointly issue selective disclosure credentials.

Coconut is a fully featured selective disclosure credential system, supporting threshold credential issuance of public and private attributes, re-randomisation of credentials to support multiple unlikable revelations, and the ability to selectively disclose a subset of attributes. It is embedded into a smart contract library, that can be called from other contracts to issue credentials. The Coconut architecture is illustrated in Figure 1. Any Coconut user may send a Coconut request command to a set of Coconut signing authorities; this command specifies a set of public or encrypted private attributes to be certified into the credential (❶). Then, each authority answers with an issue command delivering a partial credentials (❷). Any user can collect a threshold number of shares, aggregate them to form a consolidated credential, and re-randomize it (❸). The use of the credential for authentication is however restricted to a user who knows the private attributes embedded in the credential—such as a private key. The user who owns the credentials can then execute the show protocol to selectively disclose attributes or statements about them (❹). The showing protocol is publicly verifiable, and may be publicly recorded.

3 System Overview

3.1 High Level Architecture

Figure 2 shows a high level view of how the DECODE components work together to create a network of validating nodes to which transactions can be submitted to chainspace. The DECODE P2P Network is formed by the Tor Dam nodes [3] communicating with each other. These run on DECODE OS which can be run on hardware “hubs” as explored in deliverable X. This forms the foundation of the network between DECODE validating nodes. We call these validating nodes because they are running Chainspace nodes and therefore constitue the distributed ledger described within this document.

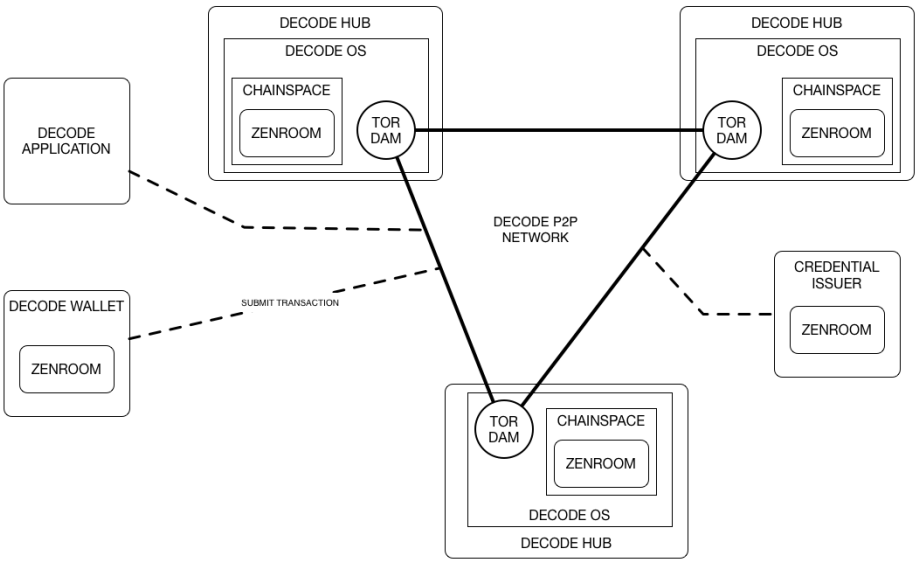

Figure 2: A high-level overview of DECODE architecture.

As describe in Section 4, Chainspace provides an extremely flexible model for creating smart contracts. This has allowed us to create an implementation of a chainspace checker that executes Zenroom scripts. This allows us to write cryptographically advanced contracts in zenroom and execute them on the chainspace network. One such contract is based on the coconut protocol and allows us to implement a privacy preserving petition, as described in Section 5.1. We can see in the figure that the DECODE Wallet also executes Zenroom. The wallet is a react-native mobile application which integrates directly with zenroom on the mobile device and thus allows the participant to maintain complete control of their cryptographic key materials on their device.

Being able to submit and verify transactions is only part of the application ecosystem. In order for the information to be interesting and accessible we will require some entry point. We call this the “Decode Application” and will usually be a website which provides a user experience for the particular application. In the example of a privacy preserving petition, this would be the website that hosts the petitions and publicises them and also eventually displays the results. The participant will visit the website and be provided a link (either directly if browsing from their mobile device or via a QR code if not) which opens in the wallet and passes key information about the petition which allows them to sign it. Note that the application itself never has access to any private material of the wallet, and no such information is shared directly to the application, only via the ledger. Finally in the case of the petition example, we require a system to allow the user to generate a credential (perhaps providing evidence of their residency in a particular city). This component is known as the “Credential Issuer”. The credential issuer will implement the issuing part of the coconut protocol, also utlising Zenroom to execute the cryptographic functions.

3.2 Production Readiness

The delivery goal of DECODE is to develop technology components and field test them in real world scenarios, working with community organisations to test the technology. These are called the "Pilots".

It is expected that different components will reach different levels of maturity during the project. The deliverable *D4.16 DECODE architecture stability usability* due at the end of the project will use this model as the basis to report to what level it has been possible to develop the technology, what limitations it has (the *operating envelope* and what future work could be done to mitigate the limitations. We define three phases of delivery evolution for the technology over the course of the project which are summarised in Table 1.

Phase	Description
PHASE-0	Pilot 'alpha' (initial, controlled early testing phase
PHASE-1	Pilot 'beta' (expected full scale of pilots during the time of the DECODE project funding
PHASE-2	Wider community adoption (post DECODE project funding

Table 1: Phases for considering production readiness

At each phase, we consider what is required of the technology and assess the components against these. This results in a definition of the "operating envelope" of the technology so that it can be seen what the limitations of each component are and what would be required to meet the next level. For each pilot we will define a set of key metrics which can then be compared to the *operating envelope* of the components in order to make an assessment of their suitability. At each phase we consider the following dimensions, which are essentially the "Non functional Requirement" (NFR) categories of the systems (sumarised in Table **??**).

NFR Category	Description
Scalability	Ability of the system to meet the end-user *availability* and *perceived latency* expectations for a given number of users of the system
Integrity	Ability of the system to preserve and evidence (audit) the integrity of data stored within it mapped against a defined *failure model*
Confidentiality	Ability of the system to preserve the confidentiality (privacy) of the data stored within it mapped against a defined *threat model*
Usability	Ability of the system to enable end-users to achieve their *user goals* whilst meeting their expectations around *ease of use* and *time taken to reach a goal*
Operability	Ability of the system to enable operators to perform *administrative functions* and provide *diagnostic metrics* to operators to aid touble-shooting and maintain the expected *quality of service* as defined by the *NFR metrics*
Evolvability	Ability of the system to be *evolved* through making *software changes* meeting the expectations of delivery teams an developers in terms of *developer experience*

Table 2: Phases for assessing production readiness

For each category we define specific metrics which can be measured and assessed against the expected conditions at each phase of delivery.

Scalability

The system should support a defined set of criteria for scalability. It is important to note that setting the constraints for these parameters can significantly alter the cost of an implementation. We do not need to achieve infinite scalability, it is more important to understand the limits of the systems we are building (operating envelope) and understand if that is appropriate for the context in which they operate. Key metrics we can use to define this envelope are:

- Client-side latency (i.e. how long does a tx take end to end on the client device, say the wallet)
- Latency of all network calls in the system (e.g. between nodes or to a database, the checker)
- Number of transactions per second (as measured on the endpoints of each network service)
- Memory consumption
- Storage consumption
- Cpu utilisation
- Time to recover from a fault

Integrity

The design of the system should define what happens in these case and how to recover. When considering integrity, we should also consider how one can verify or demonstrate the integrity of the data to an entity outside the system (e.g. via digital signatures, hash chains of proof). Integrity in terms of resilience should consider two fault types:

- Crash faults
- Byzantine faults

Confidentiality

- What data is stored where
- Encrpytion mechanisms
- Threat model - what threats are mitigated, what are the limitations?
- Trust model (limitations on rquiring trusted partners)
- Practices around data management
- Penetration testing
- Secure coding practices (owasp to 10, validation of vulnerabilities against CVE database of libraries)

Usability

- Measure time taken to complete a task (e.g. from signing in to signing a petitoin
- User feedback forms (ratings / feedback etc)
- User testing and observation
- Internationalisation
- Operations staff and developers are also users of the system!

Operability

- Provide visibility of NFR metrics via a User Experience
- Installation documentation and tools
- Packaging of software for installation
- Compatibility with target operating systems and libraries

Evolvability

- Developer documentation
- Modularisation
- Test coverage
- Infrastructure as code [23]
- Continuous Delivery practices[15]

3.3 Threat Modelling

Threat Modelling is an integral part of application planning and review. For this project, we have followed the best practices recommended by OWASP [27]. It is not possible to conduct a threat modelling excercise in the abstract, and therefore for each of the pilot use cases, we conducted a separate threat modelling excercise. The detailed results of these are available separately from this document, included with documentation of the particular pilots. The exercise consists of analysing attack vectors and their motivations by any possible actors at any point in the architecture or user flow. The exercise concludes with recommended mitigations and raising of risks to accompany development and deployment decisions. The process of Threat Modelling consists of the following steps.

- Review application architecture
- Document attackers, motivations, and attack scenarios, using STRIDE [26]
- Organise vectors into attack trees and prioritise
- Decide on mitigation strategy for identified threats

4 Distributed Ledger

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.

4.1 Data Model: Objects, Contracts, Transactions.

Chainspace applies aggressively the end-to-end principle [29] 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

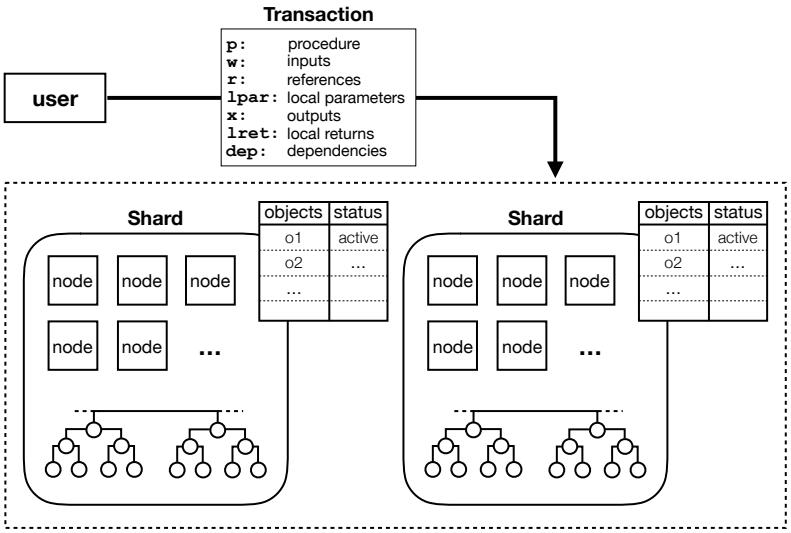

Figure 3: Design overview of Chainspace system, showing the interaction between users, transactions, objects and nodes in shards.

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. A 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.

4.2 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:

type *Transaction* : *Trace* list
type *Trace* : Record $\{$
c : id(o), p : string,
$\vec{w}, \vec{r}, \vec{x}$: id(o) list,
lpar, lret : arbitrary data,
dep : *Trace* list$\}$

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 4. 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 (α, *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 4 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 4 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).

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 [28]. 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 [22], 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.

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.

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

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

Figure 4: The sequencing and checking validity rules for transactions.

4.3 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 $\mathsf{id}(Trace)$ 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 recreate 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 $\mathrm{id}(o)$ 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 4); 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.

4.4 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 *shard*(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.* $\exists o.\mathsf{type}(o) \in \mathsf{types}(c) \wedge \mathsf{shard}(o) \cap \Phi(T) \neq \emptyset$.

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 *shard*(o), need to check that it is active (as part of the 'check' rule in Figure 4).
- 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 [24], two-phase commit protocols [18], and classical consensus protocols like Paxos [17], PBFT [11], or xPaxos [20]. However, these approaches lack in performance, scalability, and/or security. We design an open, scalable and decentralized mechanism to perform *Sharded Byzantine Atomic Commit*.

4.5 Leaderful Sharded Byzantine Atomic Commit

Atomic commit protocols such as two-phase commit [14] have long been used in distributed systems to allow a transaction to be committed atomically. The goal is for the system to have a consistent state by ensuring that *all* the resource managers corresponding to the transaction accept it, or reject it. Effectively, a transaction rejected by a single resource manager will lead to all the other resource managers rejecting the transaction.

Recently, atomic commit protocols have been adapted to achieve consistency in sharded distributed ledgers [7] based on blockchains. A blockchain is a transparent and publicly verifiable distributed ledger. Every data item (or block) in a blockchain verifies all the previous blocks, thus offering transparency and integrity. A blockchain is maintained by a group of nodes that have to agree (or reach consensus) on whether or not to add a block made of transactions submitted by client to a blockchain. A key limitation of blockchains has been poor performance [7]. As every node handles every transaction, the system performance degrades under high transaction load. Counter-intuitively, adding more nodes to the system leads to further performance deterioration due to the communication complexity of reaching consensus among a larger set of nodes.

To address blockchain scalability issues, a number of recent systems have moved to sharded system designs. The key idea is to create groups (or shards) of nodes that handle only a subset of all the transactions. These systems achieve optimal performance and scalability because: (i) non-conflicting transactions can be processed in parallel by multiple shards, and (ii) the system can scale up via creation of new shards. The separation of transaction handling across shards is not perfectly 'clean'—a transaction might rely on data managed by multiple shards. In such cases, all the concerned shards process the transaction. This implies that consensus has to be reached not only within a shard (intra-shard consensus), but also across all the concerned shards (cross-shard consensus). Intra-shard consensus is typically achieved via Byzantine Fault Tolerant protocols like PBFT. For cross-shard consensus, typically atomic commit protocol is run across all the concerned shards (with each shard acting as a single resource manager) to ensure that the transaction is accepted by all or none of the concerned shards.

L-SBAC builds on top of S-BAC [6] and integrates design features from Atomix [16]. In S-BAC, all input shards communicate with all other input shards, which creates a communication complexity of $O(n^2)$ where n is the number of input shards. L-SBAC allocates

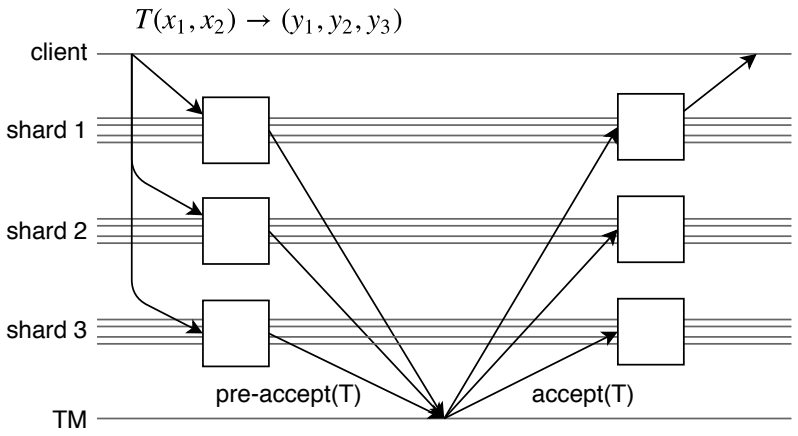

Figure 5: An example execution of L-SBAC for a valid transaction $T(x_1, x_2) \rightarrow (y_1, y_2, y_3)$ with two inputs (x_1 and x_2, both are active) and three outputs (y_1, y_2, y_3), where the final decision is accept(T).

a leader, called Transaction Manager (TM), that coordinates the protocols to reduce costs communication to $O(n)$ in the happy case, similarly to Atomix. L-SBAC also looks at how to fall back in case such a leader fails.We illustrate L-SBAC taking the example of a transaction $T(x_1, x_2) \rightarrow (y_1, y_2, y_3)$ with two inputs, x_1 managed by *shard* 1 and x_2 managed by *shard* 2; and three outputs, y_1 managed by *shard* 1, y_2 managed by *shard* 2, and y_3 managed by *shard* 3.

L-SBAC Design Figure 5 illustrates the L-SBAC protocol; the client first sends the transaction to all input and output shards. Contrarily to S-BAC, shards create t *dummy* objects upon configuration. If a shard is involved in a transaction but only handles output objects, the transaction consumes one of its dummy inputs instead; and creates a new dummy object upon completion. These dummy objects are handled by S-BAC similarly to how Chainspace handles token, and are necessary to mitigate replay attacks.

Shards sequence the transaction and send pre-accept(T) or pre-abort(T) messages to the TM; the TM then aggregates and reflects these messages to all shards involved in the transaction. They then sequence the transaction and issue an accept(T) or abort(T) back to the client. The result of L-SBAC is that, as in Atomix and traditional two phase commit protocols, the communication complexity is only $O(n)$ in the number of shards.

Transaction Manager Atomix uses this approach with the TM being the client. However the TM can be a shard, in which case the input shards contact in turn each node of the TM shard until they reach one honest node. We show how L-SBAC guarantees liveness under different threat models.

If the TM is a shard, under the honest shard assumption the shard that is in charge of being the Transaction Manager is live, and therefore progress is always made. However, any particular node of such a shard may not be honest. Therefore we need to send messages to at least $f + 1$ nodes to ensure at least one is honest. Both the client and other honest nodes may do this sequentially upon a timeout. Thus, as soon as the first honest node receives the message the protocol progresses.

Under dishonest shard assumption or if the TM is the client, the TM may act arbitrary, but anyone can make the protocol progress by taking over at any time the role of the TM since the TM does not act on the basis of any secrets. Therefore ensuring that nodes queried about the first or second phase of L-SBAC act in an idempotent manner anyone else can take over and complete the protocols. This "anyone" may be a honest node in a shard that wants to finally unlock a resource, upon a timeout. It may be other users that

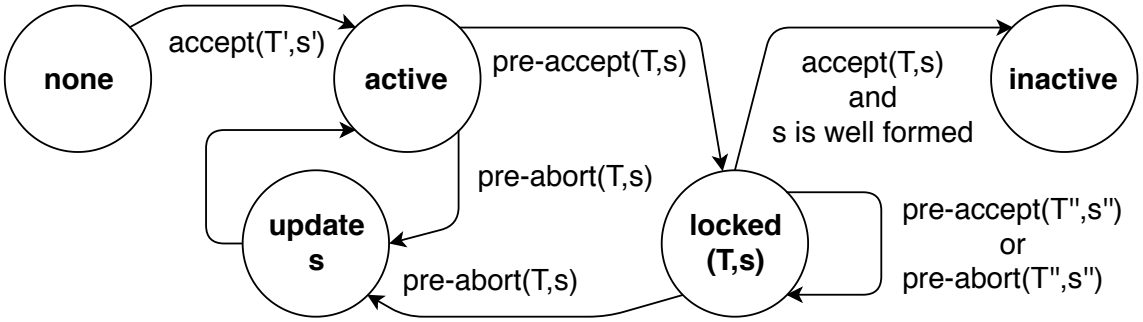

Figure 6: State machine representing a the life cycle of Chainspace objects.

wish to use a resource; or it may be an external service that has as a job to periodically close open L-SBAC instances. Therefore, L-SBAC guarantees liveness as long as there is at least one honest entity in the system; and thus holds also under dishonest shard assumption.

Object sequence numbers To prevent replay attacks, L-SBAC associates a sequence number s_{o_i} to each object (and dummy object) o_i; $s_{o_i} = 0$ when the object is created. The sequence number is intrinsically liked to the object (*i.e.,* when clients query shards to obtain an object o_i, they also receive s_{o_i}). Figure 6 shows the finite state machine describing the life cycle of objects.

Upon submitting a transaction $T(o_1, \ldots, o_k) \rightarrow (o_{k+1}, \ldots, o_n)$, the client sends along a sequence number s computed as below:

$$s = max\{s_{o_1}, \ldots, s_{o_k}\} \tag{3}$$

The sequence number s is the maximum of the sequence numbers s_{o_i} of each input object $o_i \in T$.

Upon reception of a new pair (T, s), each shard saves (T, s) in a local cache memory—the sequence number s acts as session identifier associated with the transaction T. Then, shards perform the first phase of S-BAC to decide whether the transaction is to accept or to abort; if the transaction is to accept, they emit pre-accept(T, s); otherwise, they send pre-abort(T, s).

Upon reception of any pre-accept(T, s) or pre-abort(T, s) messages, shards first verify that they previously cached the pair (T, s) associated with the message; otherwise they ignore it. All shards have now enough evidences to verify whether s is correctly computed (*i.e.,* if it is computed according to Equation 3). If it is the case and they received a pre-accept(T, s) message from each concerned shard, they emit accept(T). Otherwise, they emit abort(T) and update the sequence numbers of each input objects $(s_{o_1}, \ldots, s_{o_k})$ to $(s + 1)$; and delete (T, s) from the cache.

4.6 System Contracts

The operation of Chainspace 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 L-SBAC 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 *shard*(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.

5 Smart Contract Applications

We present two example of smart contract applications relevant to DECODE.

5.1 Privacy-preserving petition

We consider the scenario where several authorities managing the country *C* wish to issue some long-term credentials to its citizens to enable any third party to organize a privacy-preserving petition. All citizens of *C* are allowed to participate, but should remain anonymous and unlinkable across petitions. This application extends the work of Diaz *et al.* [13] which does not consider threshold issuance of credentials.

Our petition system is based on the Coconut library contract and a simple smart contract called “petition". There are three types of parties: a set of signing authorities representing *C*, a petition initiator, and the citizens of *C*. The signing authorities create an instance of the Coconut smart contract as described in [31]. As shown in Figure 7, the citizen provides a *proof of identity* to the authorities (❶). The authorities check the citizen’s identity, and issue a blind and long-term signature on her private key k. This signature, which the citizen needs to obtain only once, acts as her long term *credential* to sign any petition (❷).

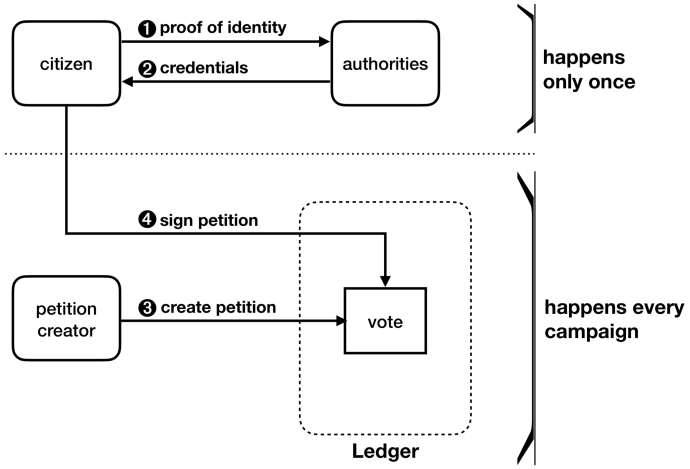

Figure 7: The petition application.

Any third party can *create a petition* by creating a new instance of the petition contract and become the “owner" of the petition. The petition instance specifies an identifier $g_s \in \mathbb{G}_1$ unique to the petition where its representation is unlinkable to the other points of the scheme[1], as well as the verification key of the authorities issuing the credentials and any application specific parameters (e.g., the options and current votes) (❸). In order to *sign* a petition, the citizens compute a value $\zeta = g_s^k$. They then adapt the zero-knowledge proof of the algorithm of [31] to show that ζ is built from the same attribute k in the credential; the petition contract checks the proofs and the credentials, and checks that the signature is fresh by vserifying that ζ is not part of a spent list. If all the checks pass, it adds the citizens’ signatures to a list of records and adds ζ to the spent list to prevents a citizen from signing the same petition multiple times (prevent double spending) (❹). Also, the zero-knowledge proof ensures that ζ has been built from a signed private key k; this means that the users correctly executed the callback to prove that they are citizens of *C*.

[1] This identifier can be generated through a hash function $\mathbb{F}_p \rightarrow \mathbb{G}_1 : \widetilde{H}(s) = g_s \mid s \in \mathbb{F}_p$.

Operation	vote	vote opposite
Yes; Male; Barcelona	$\mathcal{E}(0)$	$\mathcal{E}(1)$
No; Male; Barcelona	$\mathcal{E}(0)$	$\mathcal{E}(1)$
Yes; Female; Barcelona	$\mathcal{E}(1)$	$\mathcal{E}(0)$
No; Female; Barcelona	$\mathcal{E}(0)$	$\mathcal{E}(1)$
...	...	...

Table 3: Example of voting option for demographic decision-making contract.

Security consideration. Coconut's blindness property prevents the authorities from learning the citizen's secret key, and misusing it to sign petitions on behalf of the citizen. Another benefit is that it lets citizens sign petitions anonymously; citizens only have to go through the issuance phase once, and can then re-use credentials multiple times while staying anonymous and unlinkable across petitions. Coconut allows for distributed credentials issuance, removing a central authority and preventing a single entity from creating arbitrary credentials to sign petitions multiple times.

Javascript implementation. The petition contract described above has also been independently implemented in JavaScript [2]. The main motivation of a JavaScript implementation is for the client to be able to locally (in the browser) and trustlessly compute the necessary cryptographic constructs and operations without relying on any single 3rd party, and at the same time provide the users with a nice interface and user-experience [3].

5.2 Demographic decision-making smart contract

We extends the privacy-preserving petition application presented in Section 5.1 to a demographic decision-making contract capable to collect statistical information about the participants. Similarly to Section 5.1, we consider several authorities wishing to issue some long-term credentials to a set participants, enabling any third party to organize a decision-making event. All participants remain anonymous and unlinkable across events, but can participate only once per event.

Our application is based on the Coconut library smart contract and a simple smart contract called "demographic", extending the "petition" described in Section 5.1. The application considers three types of parties: a set of signing authorities, an event organizer, and the participants. The authorities create an instance of the Coconut smart contract as described in [30], and issue credentials to the participants as described in Section 5.1.

Any third party can create an event by running a new instance of the demographic contract, and specifying the options. Participants vote on the option corresponding to their demographic information by showing their credentials to the smart contract, and uploading encrypted votes for each possible option, as well as the opposite of each vote. Table 3 illustrates a simple example of demographic decision-making contract, collecting statistical information about the gender and city of the participant. Participants encrypt 1 to indicate their choice (*i.e.,* $\mathcal{E}(1)$); otherwise they encrypt 0 (*i.e.,* $\mathcal{E}(0)$). They also upload a zk-proof ensuring that the sum of their votes equals 1, that the sum of each row of the table also equals 1, and that votes (and their opposite) are binary values. Similarly to Section 5.1, to prevent participants to vote multiple times, the demographic contract instance specifies an identifier $g_s \in \mathbb{G}_1$ unique to the event where its representation is

[2] https://github.com/jadwahab/Coconut-petition

[3] https://www.benthamsgaze.org/2018/11/12/coconut-e-petition-implementation/

unlinkable to the other points of the scheme. It also specifies the verification key of the authorities issuing the credentials and any application specific parameters.

6 Conclusion

We presented the intermediary architecture of DECODE. As its heart, DECODE relies on a distributed ledger, Chainspace, which is 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 'application' contracts. However, unlike existing smart-contract based systems such as Ethereum [32], it offers high scalability through sharding across nodes using a novel distributed atomic commit protocol, while offering high auditability. 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.

References

[1] Decode chainspace. https://github.com/DECODEproject/chainspace. Accessed: 2019-01-21.

[2] Decode os. https://github.com/DECODEproject/decode-os. Accessed: 2019-01-21.

[3] Decode tordam. https://github.com/DECODEproject/tor-dam. Accessed: 2019-01-21.

[4] Decode wallet. https://github.com/DECODEproject/wallet. Accessed: 2019-01-21.

[5] Decode zenroom. https://github.com/DECODEproject/zenroom. Accessed: 2019-01-21.

[6] AL-BASSAM, M., SONNINO, A., BANO, S., HRYCYSZYN, D., AND DANEZIS, G. Chainspace: A Sharded Smart Contracts Platform. In *Proceedings of the Network and Distributed System Security Symposium (NDSS)* (2018).

[7] BANO, S., AL-BASSAM, M., AND DANEZIS, G. The Road to Scalable Blockchain Designs. *;login: The USENIX Magazine 42*, 4 (2017).

[8] BOOTLE, J., CERULLI, A., CHAIDOS, P., AND GROTH, J. Efficient zero-knowledge proof systems. In *Foundations of Security Analysis and Design VIII*. Springer, 2016, pp. 1–31.

[9] CACHIN, C. Architecture of the hyperledger blockchain fabric. In *Workshop on Distributed Cryptocurrencies and Consensus Ledgers* (2016).

[10] CASTRO, M., LISKOV, B., ET AL. Practical byzantine fault tolerance. In *OSDI* (1999), vol. 99, pp. 173–186.

[11] CASTRO, M., LISKOV, B., ET AL. Practical byzantine fault tolerance. In *OSDI* (1999), vol. 99, pp. 173–186.

[12] DANEZIS, G., GROTH, J., FOURNET, C., AND KOHLWEISS, M. Square span programs with applications to succinct nizk arguments. Springer Berlin Heidelberg.

[13] DIAZ, C., KOSTA, E., DEKEYSER, H., KOHLWEISS, M., AND NIGUSSE, G. Privacy preserving electronic petitions. *Identity in the Information Society 1*, 1 (2008), 203–219.

[14] GRAY, J. N. Notes on database operating systems. In *Operating Systems*. Springer, 1978, pp. 393–481.

[15] HUMBLE, J., AND FARLEY, D. *Continuous Delivery: Reliable Software Releases Through Build, Test, and Deployment Automation*, 1st ed. Addison-Wesley Professional, 2010.

[16] KOKORIS-KOGIAS, E., JOVANOVIC, P., GASSER, L., GAILLY, N., AND FORD, B. Omniledger: A secure, scale-out, decentralized ledger. *IACR Cryptology ePrint Archive 2017* (2017), 406.

[17] LAMPORT, L., ET AL. Paxos made simple. *ACM Sigact News 32*, 4 (2001), 18–25.

[18] LAMPSON, B., AND LOMET, D. B. Distributed transaction processing using two-phase commit protocol with presumed-commit without log force, Aug. 2 1994. US Patent 5,335,343.

[19] LAURIE, B., LANGLEY, A., AND KASPER, E. Certificate transparency. Tech. rep., 2013.

[20] LIU, S., CACHIN, C., QUÉMA, V., AND VUKOLIC, M. Xft: practical fault tolerance beyond crashes. *CoRR, abs/1502.05831* (2015).

[21] MCCONAGHY, T., MARQUES, R., MÜLLER, A., DE JONGHE, D., MCCONAGHY, T., MCMULLEN, G., HENDERSON, R., BELLEMARE, S., AND GRANZOTTO, A. Bigchaindb: a scalable blockchain database. *white paper, BigChainDB* (2016).

[22] MIERS, I., GARMAN, C., GREEN, M., AND RUBIN, A. D. Zerocoin: Anonymous distributed e-cash from bitcoin. In *Security and Privacy (SP), 2013 IEEE Symposium on* (2013), IEEE, pp. 397–411.

[23] MORRIS, K. *Infrastructure As Code: Managing Servers in the Cloud*, 1st ed. O'Reilly Media, Inc., 2016.

[24] NAKAMOTO, S. Bitcoin: A peer-to-peer electronic cash system, 2008.

[25] NAKAMOTO, S. Bitcoin: A peer-to-peer electronic cash system.

[26] OWASP. Owasp stride classification scheme. `https://update-wiki.owasp.org/index.php/Threat_Risk_Modeling#STRIDE`. Accessed: 2019-01-21.

[27] OWASP. Owasp threat modelling. `https://www.owasp.org/index.php/Application_Threat_Modeling`. Accessed: 2019-01-21.

[28] PEDERSEN, T. P., ET AL. Non-interactive and information-theoretic secure verifiable secret sharing. In *Crypto*, vol. 91.

[29] SALTZER, J. H., REED, D. P., AND CLARK, D. D. End-to-end arguments in system design. *ACM Transactions on Computer Systems (TOCS) 2*, 4.

[30] SONNINO, A., AL-BASSAM, M., BANO, S., AND DANEZIS, G. Coconut: Threshold issuance selective disclosure credentials with applications to distributed ledgers. *arXiv preprint arXiv:1802.07344* (2018).

[31] SONNINO, A., AL-BASSAM, M., BANO, S., AND DANEZIS, G. Coconut: Threshold issuance selective disclosure credentials with applications to distributed ledgers. *CoRR abs/1802.07344* (2018).

[32] WOOD, G. Ethereum: A secure decentralised generalised transaction ledger. *Ethereum Project Yellow Paper 151* (2014).