decode

First 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

Project no. 732546

DECODE

DEcentralised Citizens Owned Data Ecosystem

D1.4 First Version of the DECODE Architecture

Version Number: V1.0

Lead beneficiary: UCL

Due Date: October 2017

Authors: George Danezis, Shehar Bano, Mustafa Al Bassam, Alberto Sonnino (UCL)

Editors and reviewers:
Denis Rojo (Dyne); David Laniado, Pablo Aragón Asenjo (Eurecat); Oleguer Sagarra Pascual (IMI); Jaap-Henk Hoepman, Paulus Meessen (Radboud University); Jim Barritt, Priya Samuel (ThoughtWorks)

Dissemination level:		
PU	Public	✓
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)	

Approved by: Francesca Bria (Chief Technology and Digital Officer, Barcelona City Hall)

Date: 31/10/2017

Contents

Preface

Deliverable D1.4 describes the first iteration of the DECODE distributed architecture for decentralized and privacy friendly citizen applications. The description of work relating to D1.4 according to the revised consortium agreement defines its scope as:

"D1.4 : First version of DECODE architecture (11) - Initial distributed architecture based on the identified requirements and using existing IoT architectures and open source modules."

The work described in this deliverable took place over the first 11 months of the project, within the context of WP1, that acts as a point of technical coordination and integration for all technical efforts of the project. The effort consisted of individual partner's efforts, as well as a weekly technical coordination call, and an in-person meeting between all partners with a technical remit, and an in-person meeting in London on 20-21 September 2017, at the offices of Thoughtworks (TW).

Work has broadly progressed as foreseen in the proposal, and this document represents the current technical consensus among partners, on which DECODE plans to build pilots. In particular D1.4 captures key progresses in Task T1.1 relating to "Distributed Architecture Specification" and Task T1.3 on "Lean methodology, use cases and requirements". It also presents an embodiment of privacy preserving petitions, supporting Task T1.2 on "Privacy Design Strategies".

Deliverable D1.4 is structured in two main parts, representing the current DECODE architecture at two distinct levels of abstraction. Those also map to different, but interlinked, Tasks within WP1. Those parts are also natural, external facing, documents that we will as a consortium advertise the results of the project to the wider community.

- **The DECODE Architecture.** The first part of D1.4 contains a summary of the DECODE architecture. This is based on a 'living document' that the consortium maintains, under the editorial facilitation of Thoughtworks. As a living document, it evolves according to the technical conversations within the consortium, and changes reflect a deeper understanding of the pilots and requirements, as well as the best technical options available to implement them. We expect future deliverables to contain updates to this as the project progresses.

  In terms of public dissemination, the consortium will publish a white paper which will be widely available to all DECODE partners (beyond the technical ones that authored it) in late October or early November, and subsequent comments and discussions will shape its future contents.

- **The Chainspace shared smart-contracts platform (Technical Report).** The second part of D1.4 contains a technical report, describing the Chainspace platform that has been developed by DECODE research-focused partners, specifically for the needs of the project. It describes a high-integrity, high-availability decentralized platform to author and run smart-contracts. Furthermore, advanced techniques, such as sharding an custom atomic agreement and consensus protocols, are employed to guarantee correctness – even in the presence of some malicious DECODE nodes – as well as high scalability. Key applications for decode are described, such as polling for citizen participation, and in the IoT space supporting smart metering infrastructures.

The Chainspace technical report is already available publicly on the Arxiv pre-print service, and is now under peer-review in a leading scientific conference on distributed systems and security. The peer-review and scientific publication of novel security protocols is at the heart of the DECODE strategy for ensuring both wide dissemination of results, as well as to ensure the highest standard of quality and correctness through such external validation.

A considerable effort has already been invested, beyond preparing those documents, to experimenting with the underlying technologies. The appendix to D1.4 represents a fraction of this effort, by presenting in detail the smart contract underlying our prototype for a private polling smart contract. This contract both illustrates the flexibility of the platform, but also implements very advanced privacy features, explored in detail in Task T1.3, about how to implement both high-integrity, verifiable and privacy-preserving applications on DECODE.

Looking at the work ahead, D1.4 prepares the ground for concrete implementations of pilots, as well as further work on the underlying Chainspace platform. We expect deliverable D1.5 (Month 25) to present an updated view of the platform, informed by concrete implementations of pilots, as well as concrete performance and security measurements derived from those efforts.

DECODE Architecture overview

Denis Rojo, James Barritt, Jaap-Henk Hoepman, Mark de Villiers, Priya Samuel, George Danezis, Tom Demeyer, Shehar Bano, Oleguer Sagarra

2.1 Abbreviations

2FA - 2 Factor Authentication

3FA - 3 Factor Authentication

ABC - Attribute Based Credentials

ABE - Attribute Based Encryption

AI - Artificial Intelligence

AWS - Amazon Web Service

BFT - Byzantine Fault Tolerance

BLE - Bluetooth Low Energy

CRUD - Create Read Update Delete

DECODE - DEcentralised Citizen-owned Data Ecosystems

DRM - Digital rights management

EA - Early Access

EVM - Ethereum Virtual Machine

FST - Flying STone

GDPR - General Data Protection Regulation

GNU - GNU's not Unix

HMPO - Her Majesty's Passport office

ICT - Information and Communications Technology

IP - Internet Protocol

IPFS - InterPlanetary File System

IRMA - I Reveal My Attributes https://www.irmacard.org/irma/

IoT - Internet of Things

JSON - JavaScript Object Notation

JSON-LD - JavaScript Object Notation for Linked Data

JWT - Json Web Token

NGO - Non-governmental organization

OCR - Optical Character Recognition

OS - Operating System

OSHWA - Open Source Hardware Association

PbD - Privacy by Design

POSIX - Portable Operating System Interface

RBAC - Role Based Access Control

RFID - Radio-frequency IDentification

SDK - Software Development Kit

SNARK - Succinct Non-interactive ARgument of Knowledge

SSO - Single Sign On

URN - Uniform Resource Name

U2F - Universal 2nd Factor

XML - Extensible Markup Language

ZK - Zero Knowledge

2.2 Outline

This document describes the design of the DECODE (https://decodeproject.eu/) architecture.

It provides an overview of what DECODE is, why it is important and how it relates to the landscape of decentralised applications. It reviews the conceptual foundations on which it is built and how they combine to achieve its purpose. It describes at a high level the core components of the architecture and what role they play and finishes with illustrative examples of how it can be applied in the real world.

2.3 Introduction to DECODE

2.3.1 What is DECODE and why is it important?

DECODE (DEcentralised Citizen-owned Data Ecosystems) is an experimental project to enable practical alternatives to how we manage our personal data and interact on the internet. DECODE will develop technology that puts people in control of their personal data, giving them the ability to decide how it is shared.[1] The current models of data sharing enable service providers to collect citizens' data in exchange for services. Given the high value of this data to service providers, a vast number of them appropriate data to create value without providing people with a comparable compensation.

DECODE focuses research and development effort on novel notions of trust and privacy that can be operationalised in new governance frameworks, and innovative economic models based on digital commons. The digital commons are a form of commons involving the distribution and communal ownership of informational resources and technology.[2] Resources are typically designed to be used by the community by which they are created. In particular, the distinction between digital commons and other digital resources is that the community of people building them can intervene in the governing of their interaction processes and of their shared resources.[3]

In DECODE, Entitlements attached to private data will be searchable in the public domain but will grant access only to those parties that have the entitlement to access it. This novel concept of data rights and entitlements also applies to data being sent to or consumed by connected IoT objects in order to perform actions on the real world, allowing citizens to manage and control their devices and the data they generate.

DECODE enables participants to choose who they share their data with and what data is being shared with the recipient. The aim of DECODE is to provide state of the art tools to give people better control of their data on the internet. The project will contribute to enabling a free and open digital commons into which people can share their data in a secure and anonymous way. This will mean that all of society can benefit from the insights that can be gained from data.

The project will develop and test a free, open source, distributed, privacy aware, and trusted technology architecture for decentralised data governance and identity management.[4]

[1] Tom Symons and Theo Bass (Nesta), "Me, My Data and I: The Future of the Personal Data Economy," last modified 2017, https://decodeproject.eu/publications/me-my-data-and-ithe-future-personal-data-economy.

[2] Mayo Morell, "Governance of Online Creation Communities for the Building of Digital Commons: Viewed Through the Framework of the Institutional Analysis and Development," September 2014.

[3] Juan Carlos De Martin (Politecnico di Torino) Eleonora Bassi (Politecnico di Torino) Marco Ciurcina (Politecnico di Torino, "D1.8 Legal Frameworks for Digital Commons Decode Os and Legal Guidelines," 2017.

[4] Tom Symons (Nesta), "DECODE Project Dissemination Strategy and Communication Plan - Initial," 2017.

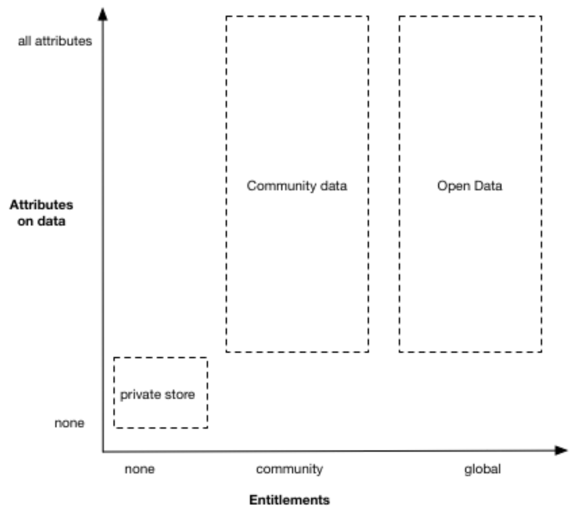


Figure 2.1: Digital commons

The goals of DECODE:

We identify the following key goals for DECODE:

- allow *participants* to manage access to their *private data*, by granting and revoke access through *entitlements*
- allow *operators* to write *smart rules* sign them and get the authorisation to run them on DECODE
- allow *smart rules* to access *private data* based on *entitlements* and matched *attributes*
- allow everyone to record *entitlements* on a *distributed ledger* whose integrity is resilient and verifiable

2.3.2 History and current state of the art

Decentralised technology ecosystems have been around for decades, and more recently there has been an explosion of decentralised platforms powering the rise of cryptocurrencies. In 2008-09, Satoshi Nakamoto implemented the first decentralised currency combining advancements in public key cryptography with a consensus algorithm widely known as "proof of work". This was a breakthrough in building a completely decentralised ecosystem that was centred around the transfer of 'assets' from one individual to another. Since then, there have been several advancements in using blockchain for creating both fungible and non-fungible assets.

Ethereum[5] provides a blockchain implementation that can be used to encode arbitrary state transitions, enabling it to be used as a decentralised 'back-end' for applications. Ethereum takes the very specific distributed application of bitcoin and generalises it to provide a massively decentralised computing infrastructure. Nodes in the network execute decentralised applications via "Smart Contracts" and record the state transitions on a public blockchain. These smart contracts are executed through a computing virtual machine called the "Ethereum Virtual Machine" (EVM).

Monax (published on the hyperledger project as Burrow) is an open platform for developers to build, ship, and run blockchain-based applications. It provides an abstraction over the underlying ledger techology and by default includes the Tendermint ledger engine and the Ethereum Virtual Machine. Tendermint have also

[5]Vitalik Buterin, "Ethereum White Paper," https://github.com/ethereum/wiki/wiki/White-Paper#history.

released a combination of these called Ethermint which effectively allows ethereum applications to run on Tendermint. Public ledgers such as Ethereum are both anonymous and permissionless systems, i.e any user can run validating nodes, and the inherent nature of the cryptography that is the basis for mining determines that an attacker would need to control more than 50% of the network. The decentralised and distributed nature of the network combined with the cost of proof of work, forms the basis of its trust model.

Identity on the internet has evolved from being implemented as centralised silos to federated identity models. Federated identity enables single sign on available across several large service provider platforms. Service providers continue to be data controllers, in both centralised and federated models. Sovrin[6] is a blockchain based identity platform, providing the user control, security, and portability.

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 being composed of 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

[6] Jason Law & Daniel Hardman Drummond Reed, "The Technical Foundations of Sovrin," last modified 2016, https://sovrin.org/wp-content/uploads/2017/04/The-Technical-Foundations-of-Sovrin.pdf.

2.4 Architectural principles

DECODE is composed of both hardware and software components - each component will adhere to the core architectural principles described below. The underlying philosophy adopted is that of the Unix philosophy[7] following key principles of modularity, clarity, composition, separation, simplicity, parsimony, transparency, robustness, representation and least surprise.

2.4.1 Free and open source

All work produced by DECODE will be published as free and open source according to licenses approved by the Free Software Foundation Europe and emerging open hardware standards. The openness of the platform will enable innovation and citizen participation.

DECODE adopts free software, however free software projects cannot be entirely considered as digital commons. As a matter of fact, writing a code and publishing it with a free license are not sufficient conditions in order to realize a free software. There are other necessary conditions, among them:

- the reputation inside the community
- the adoption of the good practices diffused in the community (for instance public repository, -continuous free upgrading, an efficient system of bugtrack and feedback the quality of the code, including its documentation to allow understanding of the code;
- the software's coverage, as the presence of automatic tests for evaluating the absence of bug on high percentages in the written code.[8]

2.4.2 Modularity and interoperability

Adopting the key unix principle of modularity (simple parts connected by clean interface), enables building independent components which can be reused and combined to form a flexible eco-system of software products. DECODE will develop modular privacy-aware tools and libraries that integrate with the operating system backed by a state of the art blockchain infrastructure supporting smart contracts and privacy protections.

DECODE will adopts a layered architecture, with components that build on top of each other. As opposed to building privacy aware applications solely in the application layer (layer 7) of the Operating System, privacy will be built into the lower layers as well, such as transport, network and data-link layers.

2.4.3 Reuse don't re-invent

DECODE aims to be built upon the solid foundations of existing well proven software wherever appropriate. For example, "DECODE OS" is based on the well known and solid Debian OS.

2.4.4 Decentralisation and federation

The current era in technology has seen a shift from large monolithic systems to distributed decentralised systems, this is to meet the requirements of system - scaling, resilience and fault tolerance but also provides

[7] Eric S. Raymond, *The Art of Unix Programming* (Pearson Education, 2003).
[8] Eleonora Bassi (Politecnico di Torino), "D1.8 Legal Frameworks for Digital Commons Decode Os and Legal Guidelines."

for decentralised governance models. DECODE builds upon decentralised models for data and identity management.

2.4.5 Privacy by design

DECODE aims to develop a privacy preserving data distribution platform to foster commons-based sharing economy models, where citizens own and control their data. This asks for a privacy by design based approach, for which the concept of privacy design strategies have recently been developed.[9]

The General Data Protection Regulation (GDPR), as well as other data protection or privacy protection laws and regulations, define data protection in legal terms. These terms are soft, open to interpretation, and highly dependent on context. Because of this inherent vagueness, engineers find such legal requirements hard to understand and interpret. The GDPR also mandates privacy by design, without describing clearly what this means exactly, let alone giving concrete guidelines on how to go about implementing privacy by design when actually designing a system. Intuitively, privacy design means addressing privacy concerns throughout the system development lifecycle, from the conception of a system, through its design and implementation, proceeding through its deployment all the way to the decommissioning of the system many years later. In terms of software engineering, privacy is a quality attribute, like security, or performance. To make privacy by design concrete, the soft legal norms need to be translated into more concrete design requirements that engineers understand. This is achieved using privacy design strategies.

Software can however enable or hinder an organisation in achieving GDPR compliance. As DECODE is designed with privacy in mind from the ground up it naturally affords a good foundation DECODE will provide transparency for participants about exactly where their data is and with whom it has been shared which will also enable GDPR compliance.

Further, many of the privacy by design principles will correlate with needs of GDPR compliance, for example right to be forgotten.

Privacy design strategies

As described in[10] a privacy design strategy specifies a distinct architectural goal in privacy by design to achieve a certain level of privacy protection. It is noted that this is different from what is understood to be an architectural strategy within the software engineering domain. Instead our strategies can be seen as goals of the privacy protection quality attribute (where a quality attribute is a term from software engineering describing non-functional requirements like performance, security, and also privacy). In the description of privacy design strategies we frequently refer to processing of personal data. Engineers should be aware that the legal concept of processing is broader than what a typical engineer understands processing to mean. In what follows we use the legal interpretation of processing, which includes creating, collecting, storing, sharing and deleting personal data.

The eight PbD principles proposed for DECODE are:

(1) *Minimise*: Limit the processing of personal data as much as possible.

[9]Eleonora Bassi Jaap-Henk Hoepman Shehar Bano, "D1.2 Privacy Design Strategies for the Decode Architecture," 2017.
[10]Michael Colesky, Jaap-Henk Hoepman, and Christiaan Hillen, "A Critical Analysis of Privacy Design Strategies," in *2016 IEEE Security and Privacy Workshops, SP Workshops 2016, San Jose, ca, Usa, May 22-26, 2016*, 2016, 33–40, https://doi.org/10.1109/SPW.2016.23.

(2) *Separate*: Prevent correlation of personal data by separating the processing logically or physically.

(3) *Abstract*: Limit as much as possible the amount of detail of personal data being processed.

(4) *Hide*: protect personal data, or make them unlinkable or unobservable. Prevent personal data becoming public. Prevent exposure of personal data by restricting access, or hiding its very existence.

(5) *Inform*: provide data subjects with adequate information about which personal data is processed, how it is processed, and for what purpose.

(6) *Control*: provide data subjects mechanisms to control the processing of their personal data.

(7) *Enforce*: commit to a privacy friendly way of processing personal data, and enforce this.

(8) *Demonstrate*: provide evidence that you process personal data in a privacy friendly way.

2.4.6 User friendliness

Building user-friendly tools and applications for end-users, and app developers for easy adoption is a core principle for DECODE. Using an outside in lean approach, where requirements from users' are researched and analysed, and prototypes are tested on target community groups will allow DECODE to develop open, interactive and user friendly interfaces.

2.5 Conceptual foundations

This section describes the foundational concepts that are combined to achieve the purpose of DECODE.

DECODE is built upon several foundations:

- Decentralisation of trust
- A distributed ledger
- Zero knowledge proofs
- Attribute Based Cryptography
- Cryptographically verifiable entitlements
- A "Smart Rules" language to express governance of participants data
- A highly verifiable and controlled execution environment

A key concept behind DECODE is to build a system which aims towards a model of *decentralised trust*. This means that as much as possible, the control over the system should not be in the hands of a small number of entities over whom the participants of the system have no influence or recourse. For example, even though Bitcoin began life as a model for decentralisation, the current state is that the hash power is controlled by a few large mining pools. DECODE will seek to explore alternative, decentralised models and economic incentives (See section on Distributed Ledgers).

A distributed ledger with decentralised governance provides a public, resilient, tamper-resistant and censorship resistant record which allows any party to be able to verify some "fact" recorded within it. This verification is demonstrable through the use of cryptography.

The public nature of the ledger is in tension with a desire to maintain the privacy of the participants of the network. DECODE applies the concept of Zero knowledge proofs to allow the cryptographic proof of a transaction to be recorded in the ledger without needing to publicly record the data within the transaction itself.

In addition to the participant's transactional data, a key element of privacy is to allow a strong control over data which is directly related to *Identity*. In DECODE all data is represented as Attributes. DECODE takes an approach to identity which states that what a participant discloses about their identity should only be related to the minimum transfer of information required for a particular interaction. Further, this transfer of information should occur through a privacy preserving mechanism.

The cryptographic implementation of this mechanism is Attribute Based Credentials. This mechanism allows a participant to prove something about themselves without transferring any other identifying information to the *relying party* (The entity that requires proof). Underlying this selective disclosure mechanism is often a Zero knowledge protocol to allow for multi-show unlinkability. For example a participant can cryptographically prove their residency of a particular city without exposing sensitive information such as data of birth, national Id number, or the actual address. This mechanism can also be used to provide strong guarantees about authenticity of an interaction whilst preserving a level of individual anonymity, particularly relevant for scenarios of participatory democracy.

Building on its verifiable public record and privacy preserving cryptography and design, DECODE adds a mechanism for participants to declare and enforce agreements about how their data is consumed. DECODE refers to this mechanism as the entitlements a participant agrees to over their data. These entitlements are cryptographically verifiable and can be extended to be cryptographically enforceable through Attribute Based

Encryption.

We can extend this data sharing capability to datasets for wide sharing - for example we can consider individual contributions to an aggregate dataset. This is often called the Digital Commons.[11]

Based on the principle of User Friendliness, giving participants access to a means to express and understand these entitlements is a further aim of DECODE, which is expressed through the development of a Smart Rules language and user interface.

Finally, DECODE considers the full technology stack within which all of the above foundations executes. This includes the underlying hardware platforms Hub and the Operating System on which the software executes. DECODE terms this the *controlled execution environment* and explores ways to provide assurance, transparency and reproducibility of the execution environment.

Each of these conceptual building blocks are explored within the following sections which should provide the basics required to understand how the implementation functions. Each of the topics is a deep area of study in its own right so we provide references to allow further exploration.

A fundamental concept within DECODE is that *all* data is represented by **attributes**, described below. This allows us to build a conceptual model of how DECODE fulfils its purpose.

To describe how data (hereafter **attributes**) is produced and consumed we first specify the entities and roles within the DECODE *ecosystem*. That is, the set of systems, people and organisations that in combination fulfil the purpose of DECODE.

Real world entities that play a defined role in the DECODE ecosystem are:

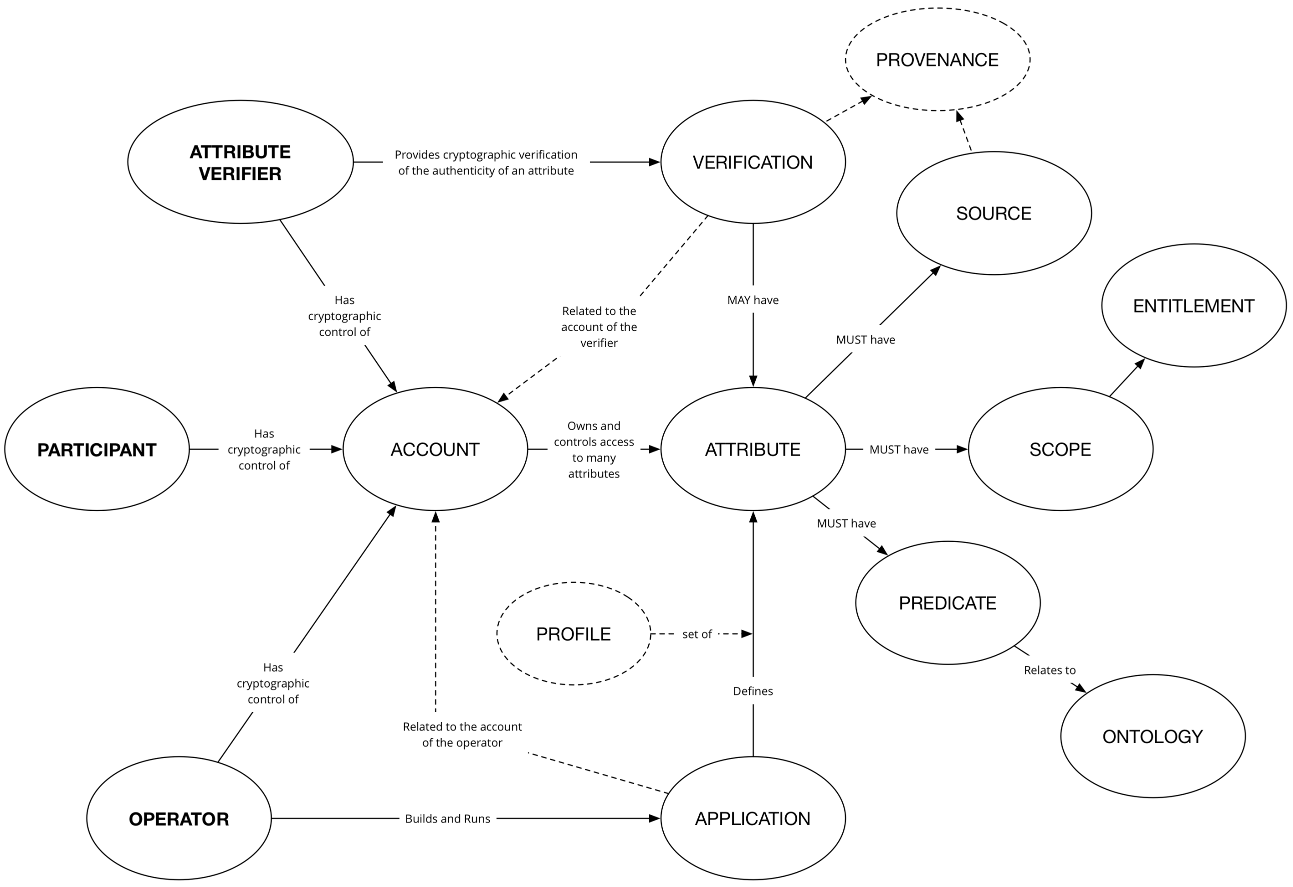
- Individual citizens
- Organisations
  - Public authorities
  - Businesses
  - NGO and civic society organizations

Within DECODE we define several *roles* that these individuals or organisations play within the ecosystem.

- Participant
- Operator
- Attribute Verifier
- Node Host

Participants represent the end consumers of DECODE - in this role an entity is storing data and interacting with *applications*. *Applications* are *operated* by *operators* who are entities that code and maintain applications that run in the DECODE ecosystem. *Attribute verifiers* are entities which have the ability to *verify claims* associated with a particular *attribute*, and provide cryptographic evidence of the claim. For example a public authority can verify that a participant is a resident of a particular city or country. *Node hosts* are members of the DECODE ecosystem who operate the underlying infrastructure of the DECODE network. Most commonly they will be hosting and running the nodes within the network, but may also run specialised services such as meta data services or online wallet services.

[11] Eleonora Bassi (Politecnico di Torino), "D1.8 Legal Frameworks for Digital Commons Decode Os and Legal Guidelines."



PROVENANCE
ATTRIBUTE VERIFIER
Provides cryptographic verification of the authenticity of an attribute
VERIFICATION
SOURCE
ENTITLEMENT
Has cryptographic control of
Related to the account of the verifier
MAY have
MUST have
PARTICIPANT
Has cryptographic control of
ACCOUNT
Owns and controls access to many attributes
ATTRIBUTE
MUST have
SCOPE
MUST have
PREDICATE
PROFILE
set of
Has cryptographic control of
Relates to
ONTOLOGY
Defines
Related to the account of the operator
OPERATOR
Builds and Runs
APPLICATION

Figure 2.2: Key conceptual terminology and relationships

All interactions within DECODE are cryptographically linked back to an *Account*. In its most basic form this can be thought of as a public / private key pair. In effect the controller of an account will own a private key and therefore there is some cryptographic evidence that this control can be demonstrated. This is common to all distributed ledger systems. As with these, it makes the security of the private keys a prime concern.

Applications within DECODE are subject themselves to a high degree of verification and transparency. All applications must be transparent about what *attributes* they wish to access / manipulate for a *Participant*. We refer to this set of attributes as a *Profile*.

A key concept to understand about DECODE is that attributes are strongly linked to applications. This provides a level of control and traceability over the system. It says that attributes can only be “created” (or “captured”) by applications and that there is a 1 to 1 relationship between attributes and applications. This means that e.g. even if two applications both capture a First Name, in DECODE these represent separate instances of attributes.

Creating a structured model around attributes provides a foundation allows us to build higher level constructs such as *Smart Rules* and *User Experience* which make it straightforward for *Application Developers* to produce high integrity, privacy aware *Applications* without needing access to highly specialised experts in the field of cryptography. It allows *Participants* to have a highly transparent view and control of their data also without the necessity to become cryptographic and privacy experts themselves. A key principle of DECODE is “User Friendliness” for both *Application developers* and *participants*.

2.5.1 Attributes

Data items alone provide little value, for example the string “Paris” could mean many things to many people. In order to be able to make a useful system that can process data, and in particular provide additional value for the purpose of data privacy and integrity, DECODE implements a conceptual model that attaches meta-data to an attribute. This model allows us to make **claims** about attributes. We begin with a basic structure of the form:

ATTRIBUTE = (SUBJECT PREDICATE OBJECT)

Where the SUBJECT is the entity to which the attribute relates (in DECODE terms, the account), the PREDICATE describes the relationship between the SUBJECT and the OBJECT and the OBJECT is the value of the attribute.

Thus we might say (e.g. in JSON):

```
locality : ["decode-account:543232", "schema:addressLocality", "Paris"]
    ^                  ^                         ^                 ^
    |                  |                         |                 |
ATTRIBUTE           SUBJECT                  PREDICATE          OBJECT
```

where `schema` and `decode-account` are URNs. `schema` expands to `https://schema.org/`.

This already provides a lot of value. However DECODE adds two further concepts to the model, PROVENANCE and SCOPE.

ATTRIBUTE = (SUBJECT PREDICATE OBJECT PROVENANCE SCOPE)

PROVENANCE provides traceable evidence to support the claim we are making. This evidence currently consists of two parts, the source of the attribute, by which we mean the application which captured it or generated it, and optionally one or more verification elements which are links to a cryptographic demonstration of that attribute. In our example this would be an Attribute Based Credential issued by the city authority of Paris associated with the private key of the account.

SCOPE relates to the agreement that is made between the application and the owner of the attribute (usually the Participant), in terms of entitlement. In DECODE terms, this is a link to an entitlement policy.

One element of attributes that is still under consideration and review is the *lifespan* of an attribute, or the time frame within which it exists. It is possible that attributes have an expiry and certain that attributes are dynamic with respect to time. For example I may move house and unavoidably become older with the passing of time. The thinking so far in these areas tends towards representing data as append only, immutable event streams. That is to say that you never “overwrite” or “mutate” an attribute, only add a new instance that represents the new value. The most common reference model to think about this is your bank account, consisting of a set of debits and credits. You can’t go back and “undo” a withdrawal, but you can make a compensating transaction and make a deposit of the same amount which in real terms ends at the same outcome.

Identity

Identity within DECODE inverts the current world position whereby participants know little about the operators of the services they are registered with but the services know everything about the identity of the participants. “Vendor relationship management” so to speak; where the vendors are DECODE-enabled applications. In DECODE, the focus is on strengthening the position of the participant in terms if understanding exactly what organisations are operating applications and what those applications are doing with the participants' data.

Participants The identity of the participants is irrelevant to the DECODE system, and also to the applications that run in the DECODE ecosystem. What is relevant are the attributes that are related to the participants. We would go as far as to say that identity is a concept not needed at all. In the real world we live and act in many different contexts, these activities and the relations in those contexts each define a perspective on who we "really" are. There is overlap, sure, but there is no context in which **all** aspects of us are relevant. So, what is our identity?

There are three options:

1. identity is what **we** think **we** are (i.e. self)
2. It is how the state defines us, typically through a number or code assigned at birth
3. It is the combination of all perspectives from all contexts combined

In the DECODE ecosystem we will keep the diverse and subtle ways of addressing aspects of our lives and selves in different (online, digital) contexts, and leverage the capacity of the medium to improve upon this in a privacy enhancing fashion. When thinking of identity in this ecosystem, option one, above, is irrelevant, and option three is fine in an abstract way, but fraught with privacy issues when it would be possible to address and use practically. That leaves the extremely narrow definition of the government assigned civic number. Apart from the issue that people exist without such numbers, this is just a single attribute of a person, at best a strictly formal (or legal) definition of identity, but missing out on just about everything we are.

Better to avoid discussion and confusion and **not** to use the word identity at all, and talk about different collections of attributes, relevant in different (online, or even DECODE-supported offline) contexts (or *applications*).

Let's call such a collection a **profile** for now.

In the end we are talking about physical people (AI's with civil rights are a ways off), even when assigning attributes that are purely abstract, or are transferable, these are about, or related to a person. This person is represented in the DECODE ecosystem as a profile, but **not uniquely**. One physical person will have control of the data related to multiple **profile**. These may overlap (in the values of certain attributes), or may not.

These profiles aren't entities in the DECODE system, they are a way of talking about *application-defined* collections of attributes. Profiles are the subject of **entitlements**, even when, for instance, the only attribute needed for the online alcohol-buying app is the age, that app would, in its use, **define** a profile with an age, and nothing else at all. For the sake of argument we leave out practicalities as payment, and the address to send the purchase to.

The connection to the real person in the real world is through a DECODE account that the person will authenticate against in order to interact with DECODE applications. This account is not part of the DECODE data that these applications have anything to do with, although the authentication app or apps (multiple means of authentication) could be seen as a special kind of DECODE enabled applications.

Authentication usually involves a participant providing various personally identifying facts to a system such as date of birth, passport id / driving licence number, potentially with additional offline checks and questions of the participant. For example in signing up to the UK's Gov.uk Verify you register with a federated identity provider (e.g. The Post Office). The post office has a mobile app that can capture images of your passport, OCR the details and confirm them against the HMPO (Her Majesty's Passport office) and then takes a photo using the phone camera in order to compare against the photo on the passport.

A participant demonstrates control of these **attributes** through some cryptographic means (essentially by holding a private key). This private key may be embedded on a physical device that the participant owns, such as a Ubikey or Smart Card issued by a civic authority. In the case of a device issued by an authority it may also contain attributes of interest to other DECODE applications, such as the fact that one lives in a particular city. These attributes, when stored, record the provenance and the semantic meaning of the relation in their URN, and can so be "officially verified" attributes that certain applications may require (such as voting in participatory budgeting applications, see below).

Taking this approach to "identity", also has the benefit of following a privacy by design principle of only providing the minimum set of information that is absolutely required for a particular interaction.

Attribute provenance

Accountability of systems Of course digital systems cannot be held accountable in any legal sense (yet), but we'll need to address the fact that consequences of using digital and 'autonomous' systems are no longer as pre-cooked as they once were.

No longer is it guaranteed that the mechanisms and processes though which technology interacts with the world are clear and understandable, even to the engineers who build the systems; the bias in the training data is invisibly and irretrievably encoded in the trained models. This means that technology needs to be able to be held accountable in itself, not only through the people running it or the engineers building it. In order to enable this, accountability needs to be designed into the systems, needs to be part of the systems, and thus, in a certain sense, systems need to be able to reflect on their actions; at least in response to queries.

Data transparency Taking stock of this *accountability design* challenge leads us to a couple of preconditions to a possible solution or implementation. One of the most important of these, especially in a DECODE context, is **data transparency**. What type of data was used, where, when and for what purpose was this data collected? In a system for data management (such as DECODE) the relevant metadata needs to be recorded and made available when needed, either directly to the user in response to queries, or to aggregation algorithms that produce data that itself needs to be able to provide an account of its provenance. The accountability mechanisms may not be part of DECODE, the data formats needed to make them possible **are**.

In DECODE the provenance metadata is provided through the *application*. When a participant stores some data in a DECODE-enabled system, the participant **always** does this through an *application*. Similarly, the application may generate data on behalf of the participant and store it (or a link to it) through DECODE. The data recorded or stored always has the relationship recorded that it comes from this particular *application* (The **source**).

Attribute verification with ABC

In order to provide a stronger assertion about the **provenance** of attributes in DECODE, we incorporate the cryptographic mechanism of Attribute Based Credentials (ABC)

An attribute in this case is any indivisible piece of personal information that can be described by a

> bit-string, such as an identifier, a qualification or a property of an entity.[12]
>
> Informally, an Attribute-Based Credential (ABC) is a cryptographic container of attributes that can provide security assurances for all participants in the system.[13]

For example, when selling a bottle of wine, a vendor has to verify that their customer is over the age of 18. The customer shows their credential; an identity card issued by the government, to convey the information 'date of birth' to the shop owner, in order to prove that they have the attribute 'being over the age of 18'. ABCs provide a cryptographic way to authenticate using selectively disclosed personal attributes. This means that in the above example, we can use an ABC credential to convey just the property of 'being over the age of 18', without revealing any of the other attributes in our credential, and even without linking this event to previous interactions.

A different example, proves the usefulness when attempting to digitise a membership system.

Classically, a library card allows a member to borrow a book. This credential reveals no more information about the borrower than their membership of the library. When building a digital library card system, it is difficult to verify active membership without cross-referencing the card with a list of authorised members. ABCs allow the showing of a membership credential, without having to check the list of authorised members. In addition it can provide multi-show un-linkability of the credential showings, which prevents them from leaking information about the specific member through logging of the interactions. There are three parties involved in the use of ABCs: the *Issuer* of the credential, the user or *Owner* of the credential (in DECODE terms, the *Participant*, and the party that wishes to verify a credential (*Relying Party*).

The proposed model for ABCs in DECODE is based on Idemix,[14][15] since the DECODE implementation requires multiple verifications of non-identifying credentials to be unlinkable. A tested implementation of ABCs is IRMA by the Privacy by Design Foundation (https://privacybydesign.foundation/en/, https://credentials.github.io/). Credentials can be part of a claim.

So how are claims actually verified in the first place? In the example, the entity responsible for verification would be the city of Barcelona. This could be a physical process, or could be done online (as in the Dutch DigiD mechanism), and involves some exchange between the city and the individual. The result of this exchange would be a cryptographic token, signed by the city, which, invoked with a specific smart rule would result in an attribute with verified provenance and value being set in the DECODE platform. This whole process could take place through a website which is run by the city of Barcelona (and thus is a DECODE enabled application). Required will be a mechanism by which the DECODE network can **trust** the public key of the city of Barcelona, i.e. there will need to be a registration protocol to establish this trust.

ABCs could provide both the trust mechanism for externally verified attributes, and the initial cryptographic tokens can be produced through a process of proving (possibly in zero-knowledge) the control of a credential.

The User Journey for this interaction would involve the person authenticating this website and then creating a "city_of_residence: Barcelona" credential signed with the city's private key. In this example there would be a

[12] Gergely Alpar, *Attribute-Based Identity Management: Bridging the Cryptographic Design of Abcs with the Real World* (Uitgever niet vastgesteld, 2015).

[13] Ibid.

[14] Jan Camenisch and Anna Lysyanskaya, "An Efficient System for Non-Transferable Anonymous Credentials with Optional Anonymity Revocation," in *Advances in Cryptology - EUROCRYPT 2001, International Conference on the Theory and Application of Cryptographic Techniques, Innsbruck, Austria, May 6-10, 2001, Proceeding*, 2001, 93–118, https://doi.org/10.1007/3-540-44987-6_7.

[15] IBM Research, "Identity Mixer," https://www.zurich.ibm.com/identity_mixer/.

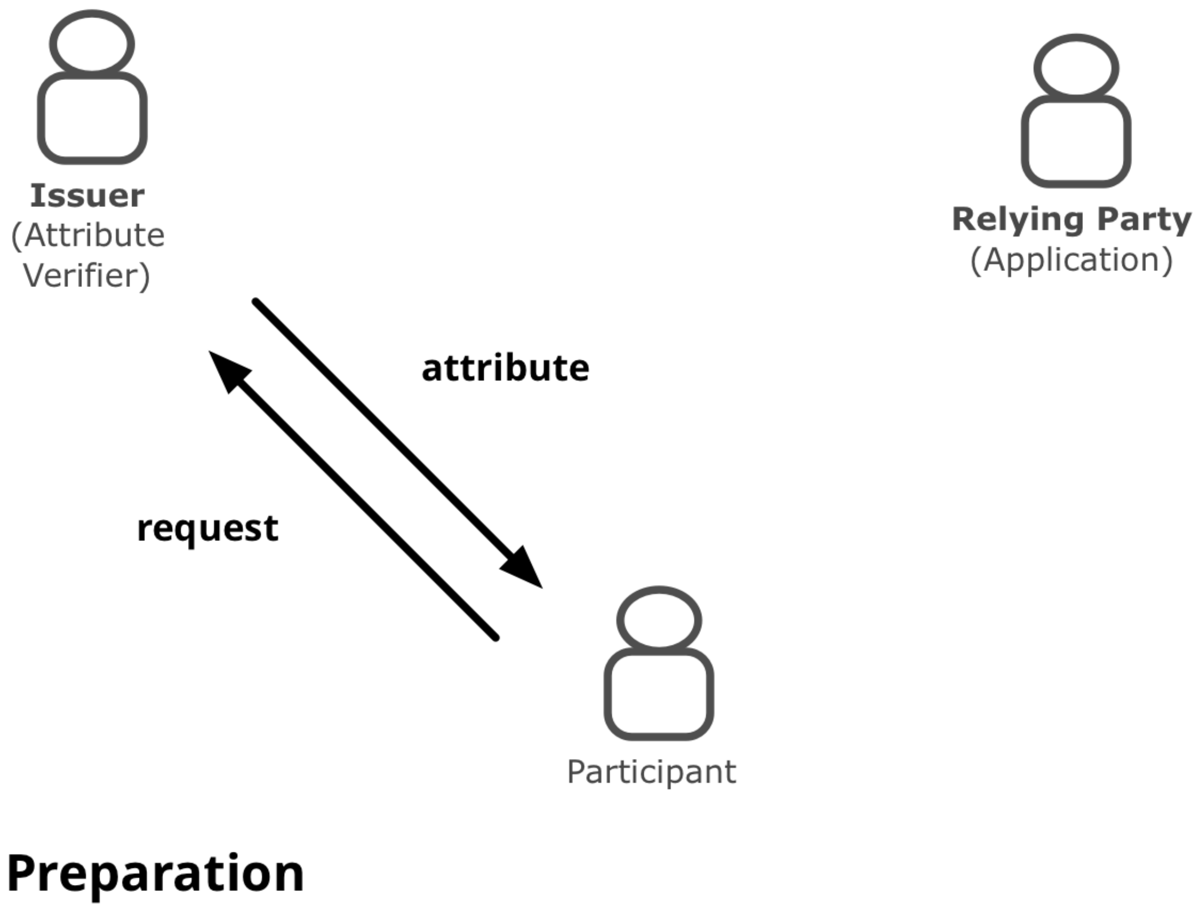


Figure 2.3: User requests a credential

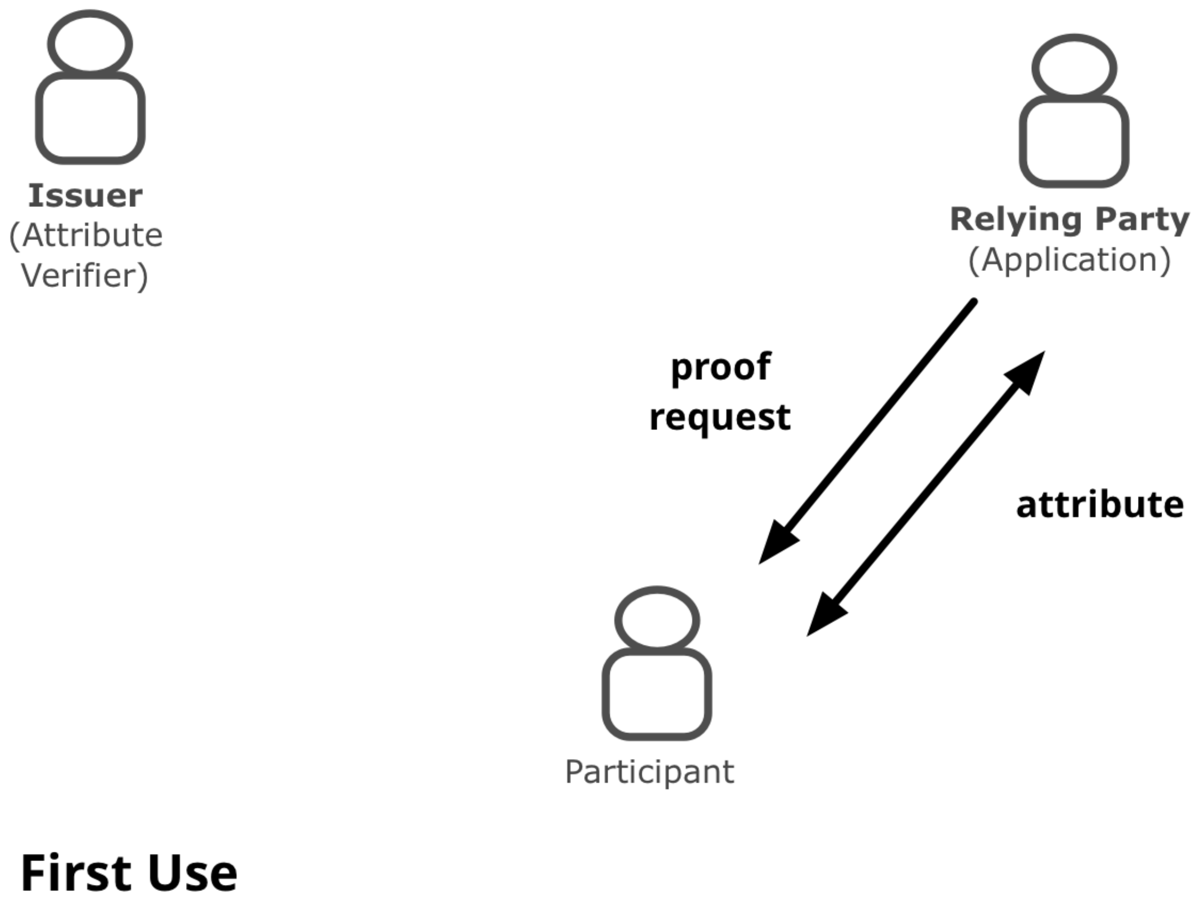


Figure 2.4: User presents credential

validity time limit on the attribute, a month, perhaps, within the credential. People move. Because the choice of using applications that accepts this credential is in the hands of the participant they have strong control of how this link is used. In a P2P sharing application a different proof-of-residency attribute may be good enough, for instance.

In order to make it straightforward for developers to build DECODE applications, the mechanisms for interacting with and validating external or “official” claims will be a core part of the language that is used to express Smart Rules.

2.5.2 Entitlements

We define two parties in any given data exchange, the **data owner** and the **data consumer**. An **entitlement** is an agreement of disclosure controlled by the **data owner**. A **data entitlement** concerns the sharing of data. In DECODE an **entitlement** is defined in a **policy** *and* implemented with the application of cryptography.

Entitlement policies

The DECODE architecture is inherently distributed and as such the management of entitlement policies will need to respond to some well understood challenges :

Challenge of embedded decisions The entitlement policy for a piece of data would need to be consistent wherever that data is stored - the data may be sharded or replicated or both.

Challenge of lack of overview Distributed entitlement policies make it difficult to gather and understand policies governing the data.

Challenge of identity integration A data entitlement system within the context of a distributed system may need to interface with one of many identity systems.

Challenge of expression A formal expression of an entitlement policy should have a rich model of expression.

There are 4 key elements to an entitlement policy :

- What **attributes** are being shared
- With **whom** is the data owner sharing data
- For what **purpose**
- Under what **conditions** will the data consumer use the data (e.g. https://opendatacommons.org/licenses/pddl/)
- What is the **timeframe** within which this entitlement is valid (Expiry date)
- How does the entitlement respond to changes to the value of the attribute

This last point is a subject of further research and exploration - how should an entitlement be considered when the underlying attribute is superseded. For example, if a participant provides a new address to the application, does the entitlement automatically apply to the new address? It may be possible to express such “rules” using the Smart Rules language.

Rather than attempting to build a hierarchical entitlements system by classifying certain attributes into privacy groups, such as “sensitive, personal, public” DECODE specifies all entitlements at the granularity of individual attributes.

DECODE defines three possible access levels:

Access level	Description
`owner-only`	Subject can see neither the existence of this attribute, or its value
`can-discover`	Subject can see that the data item has a value for this attribute, but not what it is
`can-access`	Subject can both see that the data item has a value and read that value

In most cases, the participants in the system will not be creating the entitlements directly, they will be interacting with DECODE applications. These applications will have the ability to declare what entitlements they require and the participants can agree to them, in much the same way that users can accept authorisation grants using OAuth.

Implementation (access control)

Defining and declaring entitlements is a matter of describing access rules. In order for these to be useful we require a mechanism to enforce them. In a traditional system we would simply "trust" that the system has been coded to take account of the entitlement declaration - for example we might install an authorisation server product to define and store entitlements and rely on the developers of the system to code appropriate controls into the system that communicate with the authorisation server.

DECODE moves towards a more decentralised model and ultimately will explore ways in which both the data and the mechanisms of access control can be decentralised, including the issuing and verification of credentials.

An intermediate position which provides for low investment integration to existing systems is to leverage the core foundations of a distributed ledger and attribute credentials to improve on the more "traditional" access control methods.

The following sections describe both of these scenarios, starting with leveraging cryptographic credentials in a "standard" data access scenario and then expanding on this to move towards a more decentralised data storage and optimisation of encryption through Attribute Based Encryption.

The possibility of having completely decentralised credential issuers is currently a research topic and will be explored as the project progresses. It is imagined that the first scenario will integrate with the existing applications sooner.

Using ABC as an authorization mechanism In this scenario, we leverage a "traditional" architecture whereby a central entity stores data of many people and protects it by means of access control implementations. DECODE allows for the possibility to be integrated with such systems as the source of authorisation information. By referring to entitlement policies stored within DECODE, the application can

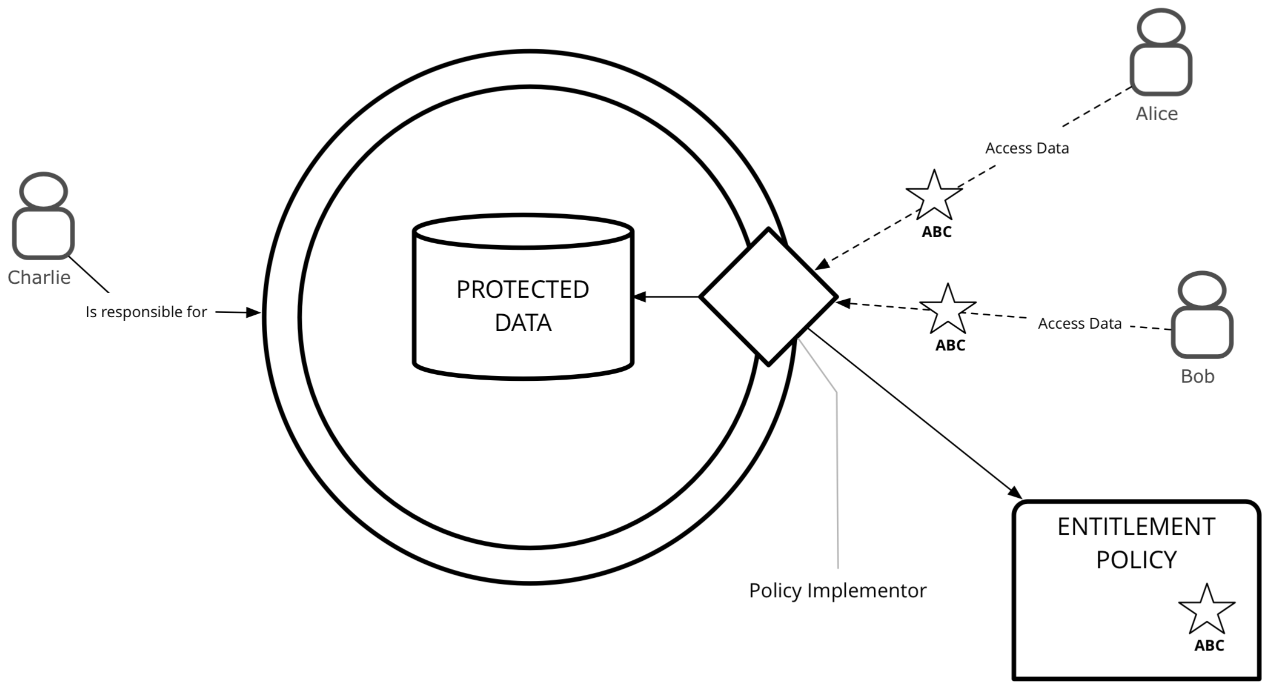


Figure 2.5: 'Data vault' access control

then accept Attribute Based Credentials presented by users and validate them against the policy, in order to make an authorisation decision. This model allows for DECODE to be easily integrated to existing systems whilst still providing state of the art cryptographic security.

The implementation of this will be based around the Json Web Token (JWT) specification.

Using ABE as an access control mechanism Evolving the model towards a more decentralised approach, DECODE proposes that data can be encrypted in order that it can be stored in the open (for example on a massively distributed file sharing system such as IPFS. A baseline implementation of this would be simply to encrypt the data with the public key of every user who has the entitlement. DECODE will explore the use of Attribute Based Encryption (ABE) a state of the art cryptographic mechanism which is an evolving topic but that has the potential to significantly reduce the overhead of allowing groups of people cryptographic access to an encrypted data source.

At a high level, it relates cryptographically an attribute that the participants possess to the encryption of the data. In one way this can be seen as each participant owning a key which is unique to them, where the encrypted data can be decrypted by any of the keys. For example a participant "Charlie" may want to share their monthly energy usage with other members of their block of flats. Charlie can use ABE to encrypt using the public key of the attribute "member of block of flats" such that anyone else in the block can decrypt it. In this model, DECODE enables the owner of the data to enforce the access control as opposed to relying on a third party as in the previous scenario.

ABE allows users to encode more complex access conditions. For example you can encrypt a party invite only for your neighbours: `( NOT ('being under the age of 18')AND 'being a resident of my block')` The data can be encrypted in such a way that only individual agents with those particular set of attributes in their wallet can decrypt the message.

The main models for ABE come from previous work on 'Attribute-based Encryption for Fine-grained Access

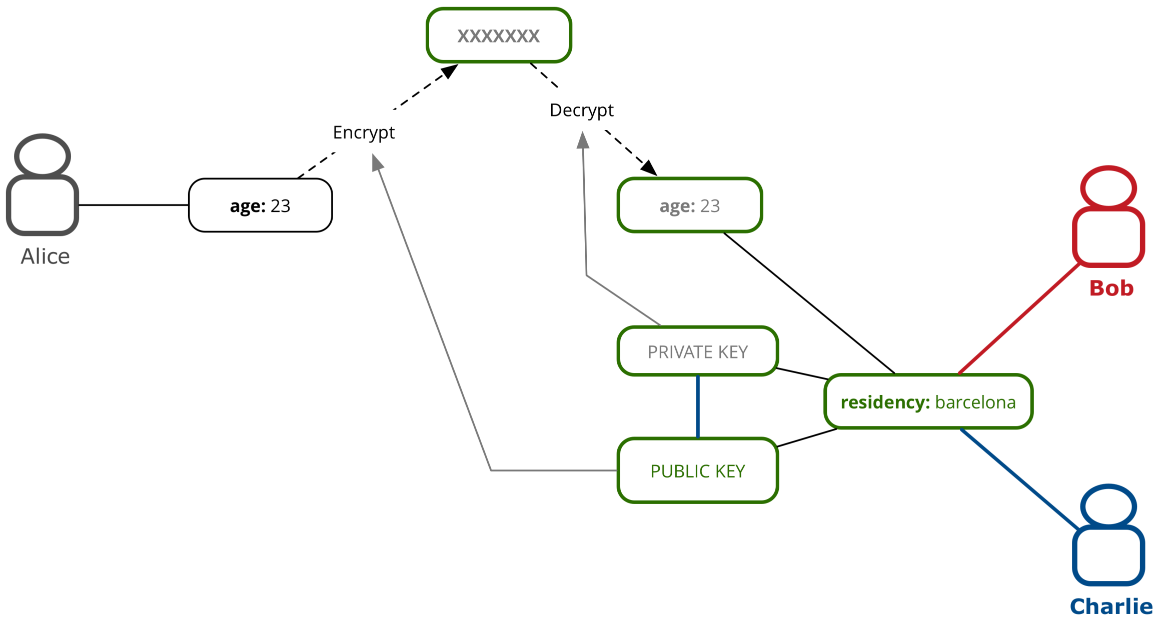


Figure 2.6: Attribute based encryption

Control of Encrypted Data'[16] and 'Ciphertext-Policy Attribute-Based Encryption'[17]. Reference implementations are available, but at the time of writing none have been selected to be supported by DECODE.

The design of DECODE will try to support this method of user controlled access policies.

One of the complexities with ABE is the scheme by which keys are issued. Many existing schemes rely on a central party to issue and manage keys. DECODE will explore more decentralised models in this space.

However, if the participants wished to broadcast their data to a larger, but still restricted, set of users; this method of access control will turn into a key-distribution problem; which is outside of the scope of the main functionality of DECODE at this stage.

Controlling access to large datasets or streams of data

We require a mechanism for controlling access to either large datasets or streams of data. Perhaps a participant wishes to publish a dataset including all their movement data from their phone for the last two months and yet control access to certain attributes.

This is also an ongoing research topic within DECODE.

Potential options are:

- Encrypt each data item in the list as above
- Separate the data into "columns" ie. each data attribute is becomes an array of values and these are then encrypted using ABE

[16] Vipul Goyal et al., "Attribute-Based Encryption for Fine-Grained Access Control of Encrypted Data," in *Proceedings of the 13th Acm Conference on Computer and Communications Security*, CCS '06 (New York, NY, USA: ACM, 2006), 89–98, http://doi.acm.org/10.1145/1180405.1180418.

[17] John Bethencourt, Amit Sahai, and Brent Waters, "Ciphertext-Policy Attribute-Based Encryption," in *Proceedings of the 2007 Ieee Symposium on Security and Privacy*, SP '07 (Washington, DC, USA: IEEE Computer Society, 2007), 321–334, http://dx.doi.org/10.1109/SP.2007.11.

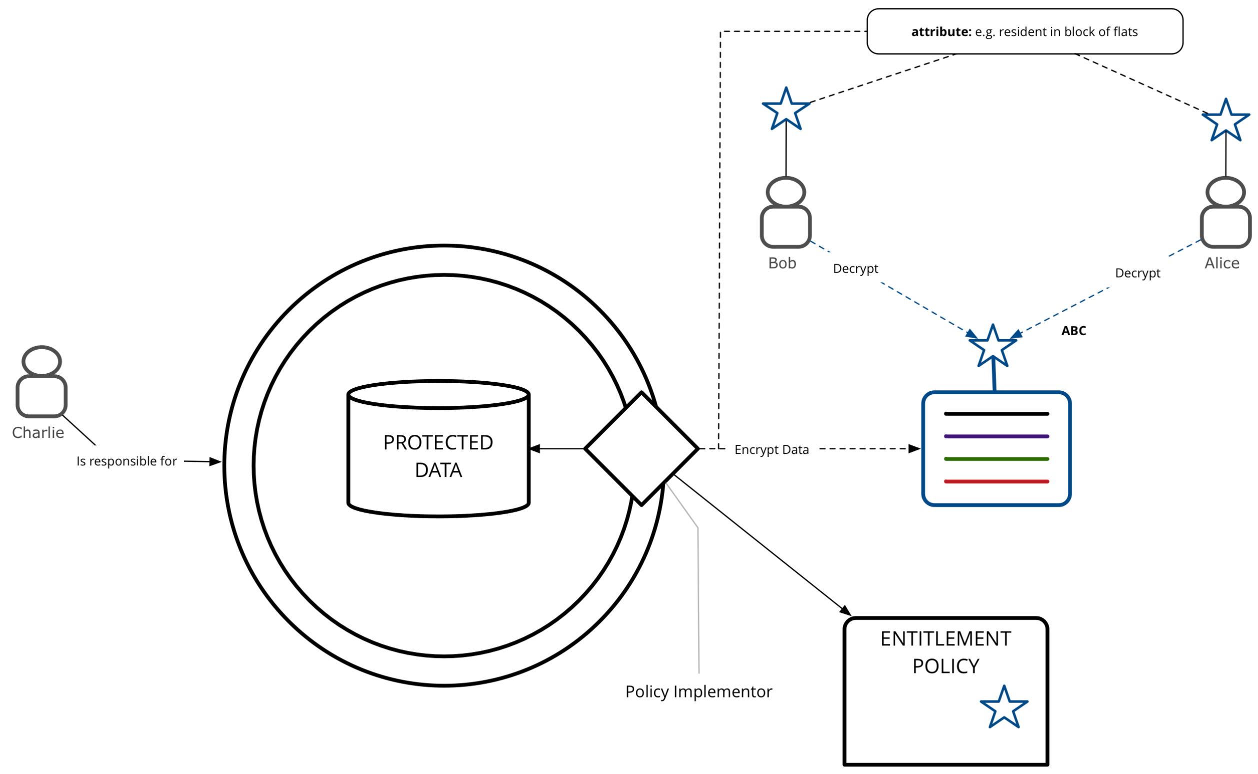


Figure 2.7: Attribute based encryption access control

- Investigate DRM tech for encrypting large streams (e.g. video). Can similar approaches be applied to user data?

2.5.3 Distributed ledger

Within DECODE, the ledger provides for two characteristics, Integrity and Availability. This is an important distinction within DECODE. Privacy (or Confidentiality) is not a core function of the ledger but is rather enabled by its features and design.

In combination with ABC and ABE, DECODE achieves privacy through the core principle that *no private data must be stored on the ledger*. Privacy is one of the classic issues faced by decentralised systems (the other being scalability). For example while Bitcoin provides some level of anonymity, all nodes have access to all transactions and all the data within them. The same is true for Ethereum, although it is moving in this direction by integrating features from ZCash to allow zk-SNARK computations from solidity, in the upcoming metropolis release. Other ledger implementations are exploring different designs by which privacy can be obtained, for example R3 Corda.

DECODE puts this principal at the centre of its Privacy by Design strategy. The cryptographic mechanism by which this is achieved is through Zero Knowledge proofs.

In short these allow a participant to perform an action (transaction) and record a verifiable proof of that transaction which has no relation to the data used within the transaction. The proof can be stored quite safely on any number of nodes since it contains no private data. It is also possible for multiple nodes to validate the transaction and build up a level of confidence in it (Integrity) and enable redundancy of the proof (Availability).

In practical terms for example, in combination with attribute based cryptography, this allows for anonymous yet verifiable petitions (see examples section). The resulting petition has a high degree of integrity because the ledger provides a Byzantine Fault tolerant replication mechanism and a high degree of Availability because it is decentralised and therefore not under the control of a single party or system. This makes it extremely resistant to many forms of failure or attack.

In summary the key requirements of a distributed ledger for the purposed of DECODE are:

- Byzantine Fault Tolerance
- Decentralised network
- Ability to implement contracts that transaction execution from verification via ZK Proofs
- An environment that allows a higher order language to be created (See Smart Rules)
- Ability to scale horizontally
- Open source

Querying the ledger

Having a robust and verifiable source of truth for transactions is of little value if applications cannot make use of the record. To this end, DECODE will provide a query interface over the ledger which will be a separate capability. This area is still under development and updates will be provided via. the whitepaper as it evolves.

2.5.4 Authentication

DECODE recognises two scenarios involving Authentication.

a) Authentication to DECODE wallets and nodes

This is the means by which the participant protects access to their wallet. In its most basic form this will be the standard option of a password credential.

DECODE also supports and will explore the concepts of more sophisticated authentication, for example external hardware security devices and leveraging biometric capabilities of devices.

b) The use of decode as a federated authentication provider

It is possible for **operators** to provide a “Login with DECODE” option. In this scenario the operator would enable an integration whereby the participant would be redirected to their DECODE wallet, authenticate there as above and then an exchange of application specific cryptographic credentials would be passed back to the website, allowing them to be authenticated.

A key principle at work with this scenario is that the operators must themselves be transparent to the participants. This means in practice that in order to allow login with DECODE the operator must first register with DECODE and itself be cryptographically auditable in any actions it takes in the DECODE system.

This raises questions around the governance of the DECODE ecosystem which will be explored as it is field tested and evolved.

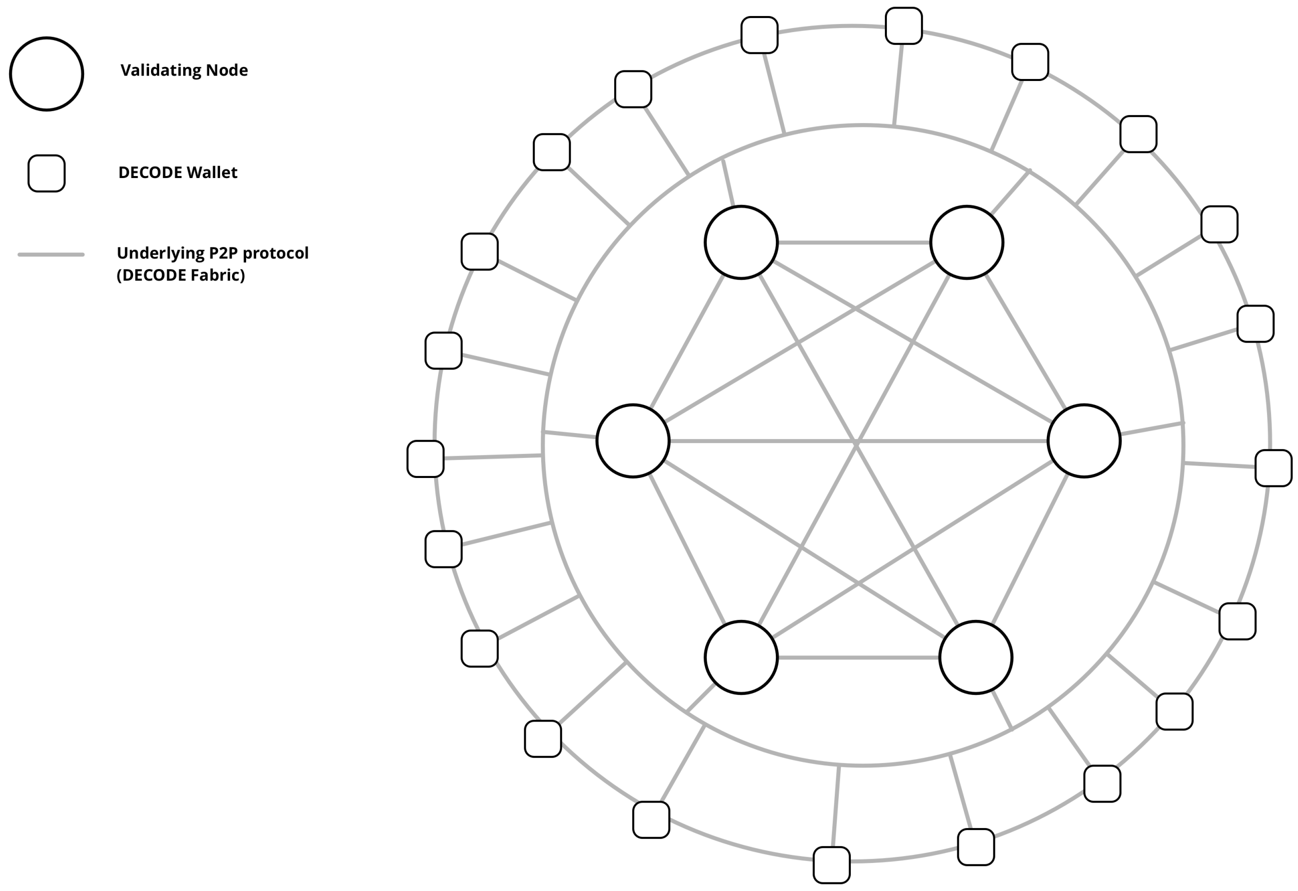


Figure 2.8: Decode Network Architecture

2.6 DECODE architecture

DECODE is at the core a distributed P2P network of "nodes" that together provide data privacy and integrity services to application developers.

The overall architecture is similar to other distributed ledgers and we have followed familiar terminology, such as "Wallet". We describe here the roles and responsibilities of each of the components and how they work together to achieve our goals.

2.6.1 Data storage

There are three significant *types* of data which are stored within the DECODE system:

1. Attributes
2. Ledger Transactions
3. IoT Data streams

Attributes are stored in the **Attribute Store**. In the simplest case the implementation is to store in encrypted form locally to the wallet. This will be the starting point for the implementation of DECODE. It is however possible that a **distributed storage** mechanism could be provided using a P2P protocol such as IPFS. This capability would be exposed via the attribute store interface. The advantage of a distributed store are:

- Resilience / backup
- Access same data on multiple devices
- Wallet itself need not have storage capabilities (can act purely as a crypto engine / key manager)

A distributed store could take advantage of the existing P2P network of DECODE validating nodes or be formed of a separate network, either an existing network or one that is formed by DECODE participants.

Ledger Transactions will be stored in the ledger node system, dependant on the implementation of the ledger. Our privacy by design principles ensure that no *private* data will be stored on the ledger. It is possible that encrypted data could be stored on the ledger.

IOT Data streams IOT data represents a special case of data in that it is likely to involve larger volumes of time series data. DECODE will continue to explore this space as it moves into implementation. A key question will be how to leverage existing IOT data stores / aggregators such as the AWS IOT. Following our principle of "Reuse don't Re-Invent", one option is to provide tools that allow decode to be integrated as an entitlements and access control mechanism over such existing aggregators and data stores. A more involved option is for DECODE to provide a custom store (based on open source stack such as Cassandra or Elastic Search). Elastic search for example already provides a mechanism for Role Based Access Control (RBAC) which may provide a starting point.

The theme of data storage will continue to evolve and will be published via the whitepaper.

2.6.2 Validating Nodes

The integrity and resilience of the network is provided through a distributed network of Validating nodes. One of the key architectural features of DECODE is that it separates *execution* of logic (*contracts*) from the *verification* of that logic, which allows for privacy aware execution.

The validating nodes are key to providing the integrity and availability of the DECODE network. Therefore we build them from the ground up with a strong emphasis on verifiability by basing them on the DECODE OS. Each node will also contain the distributed ledger node and any other libraries and software that is required to participate in the DECODE network. This is likely to include cryptographic functionality and P2P networking capabilities to allow dynamic and evolving P2P networks to be formed. It is possible that the P2P networking capabilities can come from the underlying ledger implementation, or be augmented with custom protocols.

It is important to note that DECODE does not mandate that every participant host a validating node. The minimum software you need access to in order to participate is a DECODE Wallet.

A key research question for DECODE is how to cover the cost of computation for these validating nodes? Who operates them? What are their motivations? The bottom line is that it incurs a cost to purchase hardware and consume electricity. For example, Ethereum has created a self sustaining network by offering a financial incentive to those nodes who execute the transactions, which is a straightforward economic model.

DECODE seeks to explore alternative incentive models that do not involve direct financial reward and this will be an ongoing initiative for the project. It looks to models that work for the common good of citizens and will be exploring scenarios that move towards an "economy of the commons" as opposed to the centralised concentration of wealth within a few large organisations, which is the current dominant force in the data economy.

2.6.3 Wallets

The wallet is the minimum component a person requires to interact with DECODE. Every **participant** will have their own wallet. The wallet has several core functions:

- Store securely cryptographic material (e.g. private keys)
- Securely store Attribute based credentials, linked to private keys
- Execute DECODE transactions (via Smart Rules) and submit them to the Ledger for verification
- Store, encrypted the participant's **attributes**
- Define and publish **entitlement policies** based on interactions with **applications**
- Provide the participant with a graphical user interface that allows them to manage their attributes, entitlements and applications.

Optionally a participant can push the cryptographic functions of the wallet onto a hardware device, similar to Trezor, Ledger Nano from the blockchain world. A discussion of these options is to be found in the hardware section.

We have identified two options for participants to obtain a wallet and begin interacting with DECODE. This is a topic of research for the DECODE system and is likely to evolve as we move into field testing with real communities of participants.

User experience

The wallet will be one of the primary interfaces between DECODE and the participant, the other being the *applications* themselves. The wallet interface will be explored through a user centric design process which will aim to provide a state of the art experience for participants focused on transparency of who they have shared their data with. The Wallet will also be the point at which a participant interacts with smart rules.

a) Download a Wallet

The wallet will be available as a standalone application that a participant can download to their device (either laptop or mobile device). In combination with a hardware security device this will form the maximum level of protection that a participant can achieve.

Whenever the participant must perform a cryptographically sensitive activity such as signing a transaction, they will be redirected to their wallet to perform this.

a) Wallets Hosted by Operators

We recognise however that *requiring* participants to download software and engage with a registration process *may* be a barrier to adoption. We plan to explore this through user research and field experiments. With this in mind, the architecture of DECODE also supports the concept of an ***hosted wallet***.

DECODE intends to provide tools and documentation to allow **operators**[18] to host wallets *on behalf* of their existing users. This is a low barrier to entry for an operator, involving minimal integration whereby the operator maintains any existing authentication mechanisms they have in place and links existing accounts to a DECODE wallet. It also preserves any existing authentication structures the application may have, allowing DECODE functionality to be added in a decoupled and safe manner.

[18] It might be possible for 3rd Parties to create an online wallet hosting service, however we have yet to explore the demand for this scenario.

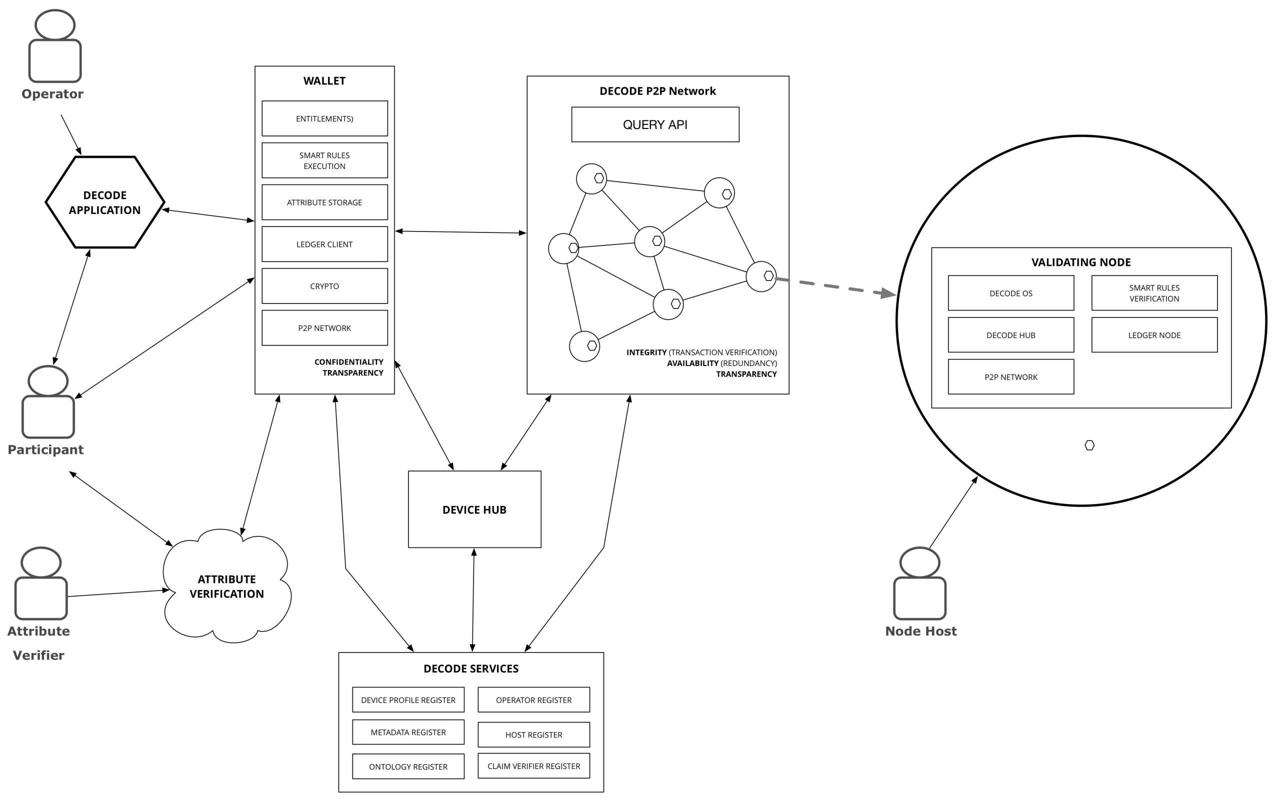


Figure 2.9: Decode Overview

To maintain the core values of DECODE however, a constraint of this scenario is that the user *must be aware and consciously agree* to the connection between their account and DECODE. Whilst the process itself maybe "1 click" for the user, the user journey must include a "connect my account to DECODE" phase. The exact design guidelines for how this is achieved will be developed on an ongoing basis.

There is a tradeoff with this scenario in that while it allows for lower barrier to adoption, it also diminishes the true value of DECODE as a decentralised system. In this scenario, the trust relationship has not been decentralised, i.e. the participant is still trusting the operator as an organisation with all their data.

2.6.4 The ledger implementation

Chainspace

As part of the mission of DECODE, we present a distributed ledger implementation **Chainspace** (`http://chainspace.io`) which has been designed deliberately with privacy and scalability in mind and is fully aligned to the goals and principles of DECODE. The full technical details of this implementation can be found within the Chainspace whitepaper.[19]

In summary, Chainspace provides a highly scalable, BFT fault tolerance ledger which separates transaction *execution* from *verification*. In implementation it provides for this in an entirely technology neutral and decoupled manner.

Chainspace contracts can be written in any language and are composed of two asymmetric but cryptographically related components. These are the *contract* and the *checker*. The *contract* is responsible for

[19] Alberto Sonnino George Danezis Mustafa Al-Bassam, "Chainspace: A Sharded Smart Contracts Platform," 2017.

executing the transaction, defining the constraints that are required. The result of the execution of a *contract* is a *proof* which has no data from the transaction but which can be cryptographically verified by the *checker*. The network of Chainspace nodes are responsible for verifying transactions and publishing the verifications as a blockchain. In implementation, Chainspace creates multiple blockchains, please see the Chainspace whitepaper[20] for more details.

Alternatives

The architecture of DECODE, following the guiding principles of being modular and reusing code, is not restricted to the implementation of Chainspace to provide the underlying ledger capability. It is possible for example that with the cryptographic advances in Solidity already mentioned, that it would be possible to build DECODE using any ledger system that also incorporated the solidity VM, either Ethereum itself or for example, the Hyperledger Burrow project from Monax.

As the project evolves alternatives will be explored and tested as they may solve for different tradeoffs, these will be documented in the public whitepaper.

2.6.5 Smart rules engine

What we call "Smart Rules" in DECODE are a computable[21] sociolect[22] that can be parsed into a semantic model referred to a finite ontology and executed by a distributed computing cluster. It is of central importance to grant participants the access to such a language and clear understanding of what it expresses and of the consequences of its execution.

The open nature of the smart rules is extremely important when compared to the popularization of "sharing economies" that apply mostly unknown rules that are opaque to the participants and undemocratically adjusted by third parties who are not participating in the economy, but in most cases just profiting from it.

The DECODE project plans the development of a language for "smart rules" that is not conceived to stay behind the scenes, but to be understood and modified: this a different approach to data management rather than CRUD-type interaction, an approach that is also necessitated by the distributed and write once immutable nature of blockchain technologies.

The underpinnings of the smart rules implementation are explained in-depth in the forthcoming deliverable D3.3 "Data Privacy and Smart Language requirements, its initial set of smart rules and related ontology".

2.6.6 Operating system architecture

The DECODE OS is the base operating system running all software designed, developed and deployed for the DECODE project. This operating system is based on the Devuan GNU+Linux distribution, a fork of the now 20 years old Debian distribution, maintained by the Dyne.org foundation and an open community of volunteers. Devuan forked Debian to preserve the simplicity and minimalism of the SystemV tradition in UNIX systems, still running modern software applications and inheriting the security patches from Debian.

The primary goal of the DECODE OS can be explained in brief by defining it as a "controlled execution environment" where, from the making of its base to the execution of every single application, all steps are

[20]Ibid.
[21]Elliott Sober, "Computability and Cognition," *Synthese* (1978).
[22]Max M Louwerse, "Semantic Variation in Idiolect and Sociolect," *Computers and the Humanities* (2004).

recorded on a ledger of events that can be saved, analysed and shipped along with every instance of the operating system. A secondary goal of this development is that of making the results of such a recorded sequence of operations reproducible.

DECODE's implementation of a distributed computational system aims to be solid and fit for mission critical purposes by leveraging well established standard practices in the UNIX world. Contrary to the monolithic applications implementing blockchain functionalities in a single runtime environment running in application space, our implementation of a DECODE Node[23] is a controlled execution environment unit for Smart Rules grafted on the classic concept of a UNIX-like operating system, keeping POSIX.1b and SystemV compatibility.

The DECODE OS is explained in depth in the project deliverable "First Release of the DECODE OS."[24]

2.6.7 Hardware Hubs

We use the term "Hub" to refer to any underlying compute infrastructure that can execute either the wallet software or a validating node. In this sense a hub can be any one of the following:

- Physical server or PC
- A virtual machine in a public or private cloud infrastructure (e.g. AWS, Google Cloud, Azure)
- A single board computer for e.g. OLinuxXIno Lime.
- A smart card running limited cryptographic code
- A mobile device such as a phone or tablet

This section provides a high level description of the requirements of the hardware, and a section specifically around the use of hardware for improving security.

In general, a participant who has more control over the full stack execution environment also has more control over the DECODE platform. Ultimately this is down to the hardware level and DECODE aims to provide details of how a participant or an operator can operate DECODE nodes that are entirely open source and auditable.

DECODE will support and explore a wide range of hardware software configurations, following the principle of being open and modular. This will allow participants to customise the level of security and control they adopt. There is a tradeoff between investment on the part of the participant to purchase and configure hardware devices vs the extra level of control and reduce need for trust of a third party.

A key principle is that DECODE should not *mandate* the use of custom hardware by participants, in order to allow for greater adoption. It remains to be seen if participants "vote with their feet" and choose hardware devices, once DECODE is operational.

Requirements

We list five key requirements of the hardware hubs here:

- Ability to run DECODE OS
- Transparency

[23]Jim Barritt Jill Irving Jen Hughes, "D1.1 Decode Scenarios and Requirements Definition Report," last modified 2017, https://decodeproject.eu/publications/pilot-scenarios-and-requirements.

[24]Ivan Jelinčić Dr.Denis Jaromil Roio, "D4.4 First Release of Decode Os," 2017.

- Deployability and availability
- Hardware security
- Connectivity

Ability to run DECODE OS The processor of the Hub must be able to run the DECODE OS. To assure compatibility of a specific processor, it must be made a target of the DECODE OS SDK.[25] The DECODE OS SDK is modelled after the Devuan SDK and supports all its architecture targets.

Additionally the following capabilities are required:

- Network connectivity (either wireless or ethernet)
- Processing power and memory to support:
  - cryptographic operations
  - embedded web server operations
  - execution of smart rules language
- Local storage in the initial phases at least enough to store attributes

The exact requirements of these will be determined as the project moves forward. Different configurations will have different requirements - for example there will be significantly more minimum compute power required for a validating node than just running a single wallet, which is should be possible to run on a mobile device.

Transparency One of the goals of the DECODE platform is to create a level playing field that enables developers from all backgrounds to contribute to society by implementing innovative applications and opening up new economical, technological and social values based on the new infrastructure that DECODE will provide. To facilitate the participation of these developers, all the elements of the DECODE architecture should be open source. For that reason any device specifically designed to operate as a DECODE Hub should be open source and compliant with the Open Source Hardware Association (OSHWA). Schematics, design files and documentation should be available for designers to build upon. DECODE will encourage hardware designers to join their efforts in creating a more secure and open hardware.

Deployability and availability Deployability is key to ensure the adoption and success of DECODE. Therefore, the Hub must be easily available. The most available potential Hub is any commercial computer. According to Barcelona data-sheet 20171 the Household ICT penetration is 88.3 in 2015 (% on population 16 to 74 years with a computer at home). Despite most of commercial computers not being open source this is a viable alternative for promoting an early adoption. For scenarios where an embedded device with a small form factor is required, low cost open source single board computers (SBCs) are a good alternative for the DECODE Hub.

Hardware security The need for privacy and confidentiality differ according to the use cases of the platform. For instance, noise quality data requires less security than biometric health-centered data that's being sent to a physician. This requires encryption on many different levels. To build trust in the platform, the DECODE hub needs to respond to the level of security for each separate use case. Software-based security

[25] Ivan Jelinčić Dr.Denis Jaromil Roio Dr.Vincenzo Katolaz Nicosia, "D4.1 Decode Os Software Development Kit (Sdk)," 2017.

means that access conditions can be hacked and logs can be tampered with. In addition, the data itself cannot be considered secure if left unprotected on a regular PC.

For this reason the DECODE offers Participants the option to adopt hardware-based security when a higher level of confidentiality is required. The interaction with these security measures should be easy and effortless for the end-user.

Potential hardware security features for the Hub:

- **Multiple factor authentication:** The security protection provided by a single authentication method, e.g. a password, have proved to be weak. To make access conditions more secure, the system can ask for a multiple factor authentication. This could be the combination of a password with some token that the user have (like an RFID or BLE device) or biometric data like fingerprint readers or voice recognition.
- **Secure processors:** Secure microcontrollers with built-in cryptographic engines and secure boot loader can guard against threats such as cryptanalysis intrusions, physical tampering, and reverse engineering. These secure microcontrollers are equipped with silicon-level anti-tampering features that allow to make them tamper resistant and to provide tamper evidence. The security keys, used to run cryptographic algorithms, need to be stored in a secure memory managed by a secure microcontroller and should only be accessible by the secure microcontroller and not from the outside of the memory.
- **Tamper avoidance:** Anti-intrusion sensors can be incorporated into the electrical design to ensure someone tampering physically with the device would not have access to any sensitive data.
- **Side channel attacks protection:** Side channel attacks consist in attacks based on information gained from the physical implementation of a system, rather than brute force or weaknesses in the algorithms. For example, timing information, power consumption, electromagnetic leaks or even sound can provide an extra source of information, which can be exploited to break the system. The first step to protect against this types of threats is the reduction of the electromagnetic and sound radiations. Other measures include random calculations and delays introduced between normal operations and try to balance the power consumption of different data values.
- **Access attempt detection:** Any attempt to access the system data remotely is detected. If the access is illegitimate (unauthenticated, coming from an unknown IP address, etc.), access is denied and in some cases data can be erased.

Connectivity Decentralized transactions will be continuously being carried by the Node. This will result in the Node using the network resources in an extensive manner. To reduce the latency in the communication and to avoid bandwidth bottlenecks, a high speed Ethernet port should be part of the specifications.

2.6.8 Keys and smart cards

The DECODE architecture will provide an authentication method to interact with the system, as described, the main entry point will be via the Wallet.

Traditional authentication mechanisms consist in only one factor authentication like a pin code or password. But password based security has proven to be insufficient to prevent unauthorized access to websites, networks and hardware devices. For this reason, users are now requiring multiple factor authentication, usually called 2FA for Two-factor authentication, 3FA for Three-factor authentication, etc. By combining several authentication factors, the authentication process is made more secure. Authentication factors include:

- Knowledge factors: The users needs to prove knowledge of a secret that only themselves know. Typical secrets include passwords, PIN codes, answer to secret question etc.
- Possession factors: An object that the user owns is used as a key to access the system. Typically, objects used in authentication include passive or active tokens, smart card chips packaged in a variety of form factors (key, token, ring, badge, etc.).
- Inherent factors: something associated with the user, such as biometrics, are used as a key to access the system. Examples include fingerprint readers, retina scanners, voice recognition etc.

In order to make access more secure DECODE will optionally support hardware based authentication with login tokens that provides 2FA via the U2F standard. The login token can safeguard and manage digital keys for strong authentication and provide crypto processing. These modules traditionally come in the form of a plug-in card or an external device that can be attached directly to a DECODE Hub.

Open source examples for hardware security modules and login tokens:

Authentication keys:

- **Nitrokey:** Nitrokey is an USB key that enables high security encryption, signing of data and login to the Web, networks and computers. Both Hardware and software are Open Sourced.
- **U2F zero:**U2F Zero is a USB key that works with all services that support U2F. It works for 2 factor authentication and sometimes password replacement. There is a button on the key that the users press to authenticate themselves.
- **FST-01 (Flying stone):** This is a small 32 bit computer that comes in the shape of a USB key

Hardware secure modules:

- **CrypTech Alpha:** "The CrypTech Alpha is an open source standalone prototype key-storage and hardware cryptography platform."
- **Pitchfork:** Project PITCHFORK is a small dedicated computer for handling your cryptographic operations and keys.

2.6.9 IoT connectivity

A core use-case for the DECODE project is to provide a set of tools to enable personal IOT data to be securely managed.

Integrating with the plethora of IoT devices on the market is a non-trivial problem and one that the DECODE project considers solving by :

Normalisation Data from devices range from high level representations in JSON and XML to a series of bytes. Normalisation is the process of transforming into a common, open format. JSON-LD (https://json-ld.org/) is an example of one such format.

Semantic Understanding Once data is normalised it is important to understand the meaning of the data. To add this understanding you could use an ontology such as the M3-Lite (http://ontology.fiesta-iot.eu/ontologyDocs/m3-lite.owl#).

An example of a tool that provides this approach is the device-hub software offered by Thingful. The tool is available at `https://github.com/thingful/device-hub`

2.7 DECODE applications

2.7.1 What is a DECODE application?

A DECODE application is any software application capable of interacting with the DECODE nodes, constituting the main endpoint for human interaction with DECODE.

A DECODE application is composed of several parts:

- Systems external to DECODE such as websites, databases and mobile apps
- Smart rules created by the *application developer* and deployed to the DECODE network. These represent core contracts of the application
- A DECODE account cryptographically controlled by the *Operator* of the application which is linked to the application
- A *Profile* which declares which attributes an application requires and their related metadata

The systems which are external to DECODE will interact with DECODE via a Wallet that is connected to the account of the operator. This will provide the ability to deploy contracts and execute transactions.

As previously mentioned, applications will be able to query the ledger to obtain information about transactions (see section "Distributed Ledgers"). Although at time of writing this is still an evolving topic.

An important aspect of applications is that they should provide a high degree of transparency as to who *operates* them, what data they will require / generate and for what purpose. To this end, DECODE will require all applications to be registered in an *application register* which will provide references to evidence concerning the operator. For example, in the UK there is a public record of companies which can be referenced e.g. `https://beta.companieshouse.gov.uk/company/04091535`.

DECODE will continue to explore and evolve the concepts around operators and their trust relationship with participants. Please refer to the decode project website for the latest news on how DECODE is being applied and tested in the field.

The range of applications for DECODE is wide. In order to bring together the concepts presented here and illustrate how they work together in a more concrete way, we consider here several applications as illustrative examples of the way in which the various DECODE components interact to provide a service.

These are:

- Participatory Democracy
- Participatory Citizen Sensing (IoT)
- Peer to peer verification of credentials
- Digital commons and Open data

2.7.2 Example - participatory democracy through petitions

In this example a City Authority would like to allow citizens to participate more directly in decision making. One way to do this is to provide support for Citizens to support proposed initiatives through signing a petition. This example of course can be generalised to any context in which a group of people wish to create a petition and is closely related to an example where participants vote between choices, either as a Poll or a vote.

It is important to note that actual Legal voting is somewhat beyond the scope of DECODE at this stage as it requires many more legal constraints, however it can be seen as a step in that direction that exemplifies the use of cryptography and ledgers.

There are several important invariants for this system:

(1) It should not be possible for any entity to be able to connect the real world identity of a signatory of the petition to their signature
(2) Any entity should be able to cryptographically verify the results of the petition (how many signatories there were)
(3) It should not be possible for a participant to sign the petition twice, or should be possible to identify that two signatures are from the same account
(4) A signatory must demonstrate cryptographic evidence that they are a legal resident of the city
(5) It must be possible for a signatory to withdraw their support
(6) It should be possible to gather demographic information (e.g. Age group) about the signatories in a privacy preserving manner and only with their consent and awareness
(7) The petition mechanism should be decentralised to avoid interference or subversion by any single entity

The foundation for implementing this scenario in DECODE is that of Attribute Based Credentials (ABC). As we have described previously, this mechanism allows that a participant can obtain a credential issued by the city authority and cryptographically linked to the participants private key, but *without the participant being required to share their private key with the council.* This means that whilst the citizen must engage with the authority in an exchange in order to obtain the credential, the council has no contact with the *account* of the participant and so cannot later link the real world identity to the signature satisfying (1).

It is also possible when issuing the credential to provide a unique token that can be used to satisfy (3).

The integrity and tamper resistance is provided by utilising a distributed ledger to verify the results. The application of zero knowledge proofs provides us with a mechanism to ensure that we do not need to store the credentials themselves (private data) on the ledger.

The core *petition mechanism* is an example of a “distributed application” this is in DECODE terms a more tightly defined notion than for example in Ethereum. In DECODE it is a “distributed application with levels of guarantee about privacy by design built in”.

The core of the implementation of the petition in DECODE is the Smart Rules which describe the logic of the *contract* that implements the invariants above.

The execution of these smart rules are as described previously made of two cryptographically bound parts, the *contract* and the *verifier*. The contract will execute in the wallet and will cryptographically access the credential, confirm that this user has not voted twice and that the credential is indeed provided by the relevant city authority (it is cryptographically linked to a well-known and trusted public key of the city). And if all the invariants are met, will construct a proof which can be stored on the ledger, linked to the contract instance (the particular petition the participant is signing) to which it belongs. At this stage it increments a counter in the contract to indicate the number of votes.

This proof is the “signature” of the participant. Given that the ledger has the list of proofs, it is now possible for anyone to be able to cryptographically verify that those proofs were in fact generated by someone in possession of the credentials indicating their residency, and that they have executed the invariants in the contract which prevent them from voting twice (2, 3).

We can also extend the contract such that at the same time as incrementing a counter for the number of signatures, we also increment various counters for aggregate statistics, for example age range. Because the *contract* is executed within the control of the participant (inside their wallet), they choose whether or not to provide this data. Depending on the circumstances, it may or may not be mandatory to also increment the age range counter. In any case, this needs to be clearly communicated to the participant during the process of signing (as discussed in the user experience section of wallets). Another important privacy by design principle here is that only the information required is transmitted. For example an alternate implementation would be to send (or worse record) the participant's data of birth to be used to calculate the age range.

By employing the combination of ABC, a privacy preserving ledger (does not contain any private data) and zero knowledge proofs, DECODE is able to satisfy the key invariants outlined and provide a robust and provable engine that will give authority to any petition that is recorded, robust enough to meet the stringent demands of a city authority engaged in participatory democracy. As noted, to be able to conduct a formal legal *vote* is beyond this scope as it requires considerably more controls around physical voting devices for example, however, its possible to see that with the application of hardware described previously DECODE takes a step in that direction.

Signing petitions or voting is not limited to participatory democracy. It is in fact a very common activity amongst any group and will have wide-ranging utility to community movements and organisations. For example UNICEF have successfully developed a community project called Ureport which does exactly this and enables communities to have a voice.

2.7.3 Example - participatory citizen sensing (IoT entitlements)

Noise pollution is an issue for a number of citizens living in particularly noisy areas, such as, large public squares where activities take place. In these areas there is higher than average participation in citizens setting up IoT devices that measure noise. In this example, an involved citizen is interested in the correlation between health data (sleep patterns) and the levels of noise pollution in the area.

It is important to note that this example, integrates entitlements across public and sensitive personal data. In this context, private, personal and sensitive personal data can be defined as follows

- Private Data: In information science, this refers to data that is considered private, but may not may not be related to a physical person.
- Personal Data: Any information related to a natural person or 'Data Subject', that can be used to directly or indirectly identify the person.[26]
- Sensitive personal: Personal data consisting of information on the racial or ethnic origin of the data subject, political opinions, religious beliefs etc.[27]

Several cities include IoT devices that measure pollution levels (including noise) and publish this data in an open an accessible format. Sentilo is an example of an open source sensor and actuator platform designed to fit in the City architecture, in several cities such data is freely and openly accessible.

Individual citizens regularly measure and gather data from wearable devices such as activity, sleep, blood pressure, and heart rate etc. Terrabytes of IoT data from personal wearable devices currently live in closed

[26]EU, "GDPR Glossary of Terms," http://www.eugdpr.org/glossary-of-terms.html.
[27]Ibid.

data silos which are not amenable to contributing to a Digital Commons dataset. Combining both private and public data in this context, can lead to new insights about citizens habits and correlations. DECODE provides a privacy aware solution to establishing this union of data sets, giving people the ability to control the visibility of their personal data.

In this example, DECODE will be used as a smart rule based data entitlements engine for IoT data streams. Within DECODE, a complete and normalised ontology for the IoT data will be provided by the IoT connector. This maps all data coming from external sensor devices to an ontology known to DECODE. With privacy in mind, a user's preferences on data sharing policies are captured and translated to a smart rule language.

The smart rules enable setting data entitlements below (as a prototype)

- owner-only - the data is not discoverable or accessible and is essentially private.
- can-discover - the data is discoverable and will be made available for search. The data is not accessible.
- can-access - the data is accessible and by default discoverable.

To qualify the terms used above: - discoverable - the existence of the data can be found. - accessible - the value of the data can be viewed. For the data to be accessible it is of course discoverable.

Based on the user's preferences and the ontology of the IoT data being processed by DECODE, entitlements rules to access this data are added to the attributes on the data. DECODE stores three types of data in this scenario

- Meta-data about the devices connected to DECODE
- Entitlements store
- Transactional data on the ownership of the data
- Device registry (a registry of devices that are known to DECODE and interact with the system)

Within DECODE, this continues to be a topic of research and evolution. IoT data is currently both fragmented and siloed, with the application of user interfaces that easily and transparently allow users to add/remove/change entitlements on their IoT data, DECODE will enable IoT data to be used to share IoT data in privacy preserving way.

2.7.4 Example - peer to peer identity and reputation verification

There has been rapid growth in the sharing economy in the last decade. The presence of several large online platforms such as Airbnb and has enabled more people to engage in sharing resources available to them. A core driving factor in this fast growing economy is *trust*, and is one of the biggest concerns of using sharing economy platforms.

Sharing economy companies are beginning to understand the importance of that trust. In a peer-to-peer marketplace, verifying user identity increases trust, and from there users begins to build their online reputations. Identity verification is currently implemented via third parties undertaking the process of conducting identity and background checks on the users of the platform. Concerns have been raised that the Verified ID model disrupts privacy. Criticism is also spread to the reliance on social networks, which opens up issues of surveillance, identity theft and fake identity use. This presents a use case for a peer to peer identity and reputation verification. In this example, DECODE is used as a decentralised platform that enables users to anonymously and securely verify the identity of other users in their community.

The core implementation of P2P verification in DECODE relies on a group of people acting as a decentralised Issuer within the context of ABC.

- An unverified user submits a 'claim' for an identity verification into the system, for example, this could be a claim that 'User X is a resident of block 10, Nicholson street, Edinburgh'.
- A smart contract is created for this claim, and this is submitted into DECODE.
- Existing verified users (or nominated administrators) are notified of this claim, and they can anonymously verify the claim, based on their location or association with the claimant via other applications.
- When a consensus is reached on the results of the claim contract, the transaction is deemed complete and the user making the claim is then promoted to a verified user.
- The claimant is issued with attribute based credential certificate to denote that they are a verified user.

There is potential for using various DECODE enabled applications for the verification of separate identity attributes and reputation credentials. This moves away from using a centralised issuing party for credential checking, this provides users the privacy and control of their data.

This is an area of research both within DECODE and among the blockchain community. Findings in 'Designing To Facilitate Genuine Accommodation Sharing: Identity and Reputation Verification'[28] particularly in the area of accommodation sharing have shown that relying only on software based solutions is not enough. There was a strong demand for community based design decisions, such as approaching networks of friends or neighborhoods for identity and reputation verification.

2.7.5 Example - digital commons and open data

The latest years have seen a rise of movements demanding transparency of data in general, and specially in the domain of public administrations. Citizens and companies are increasingly asking for datasets to be released in the form of Open Data, so they can be shared and reused. These datasets, however, usually provide a general broad picture but should be combined with other crowd-sourced pools of data and also with private data in order to fully leverage their potential.

DECODE in this sense can help, by providing a platform where an application to view public data and generate custom visualizations of it, in relation to each user's private information, can be developed. Also, it can help in providing a platform allowing a controlled crowd-sourcing of data, thus enabling the idea of Digital Commons to go beyond the simple concept of Open Data. The earlier examples of IoT entitlements or of a petition system that allows to disclose general information about the voters are two concrete examples of this.

A detailed example could be a dashboard that mixes public data sources as well as crowd-sources one. There, each user can generate visualizations that put their profiles in contrast with the general "public trends", thus obtaining unique information of their private information in a controlled way (which would be loaded on the client side). The configuration of each dashboard could be shared with peers so each person gets their own vision, and even the private data in it could also selectively be shared with the proper entitlements. Such an application would on the one hand effectively use a mix of public, private and crowd-sourced data for the common good, and on the other hand exemplify and induce the need to users to "donate" their data to generate Digital Commons that allow to tackle societal problems such as pollution and other citizen concerns.

[28]Indre Leonaviciute, "Designing to Facilitate Genuine Accommodation Sharing: Identity and Reputation Verification," 2016.

2.8 Conclusion

This paper has presented the architecture for the DECODE project, and innovative proposal for a decentralised application framework that focuses specifically on data privacy and entitlements. It combines state of the art cryptographic techniques and evolves the concepts found in distributed ledgers to contribute to the growing need for people to have more control of who has access to their data and more transparency over who is using it and for what purpose.

We have demonstrated the core components of the architecture and the principles and foundational concepts on which it is built, completing the picture with illustrative examples of how DECODE can be applied to real world scenarios.

The DECODE project is specifically target to applying the latest innovations in technology to the good of citizens and furthering a socially beneficial agenda as opposed to the centralisation and exploitation of individuals through their data that is rife in the modern world.

The ongoing evolution of the architecture will be documented via a set of public whitepapers that will be published on the DECODE website and notifications will be made via twitter (https://twitter.com/decodeproject (https://twitter.com/decodeproject)).

2.9 DECODE Glossary

Account An account is the software construct within decode that provides control of attributes and represents either the Participant, Operator or Attribute verifier in any decode transaction. The account will ultimately relate to some cryptographic construct such as private / public key pair. Attributes will be related to and controlled by an account. An account will be able to submit transactions to the ledger.

Attribute All data which is in DECODE are attributes. All attributes have the potential to demonstrate Provenance.

Attribute provenance Meta-data related to an attribute that relates it to an ontology. All attributes must specify a link to an ontology. We may also consider optionally adding other meta-data to attributes that for example describes the source of the attribute data.

Attribute Provenance Verification A cryptographic demonstration of the validity of the provenance of an attribute. A key property of provenance is that it should be possible to demonstrate provenance without revealing any connection back to the decode account and hence all the other attributes related to it, thus preserving privacy of the participant. The assumed mechanism for this is via some implementation of Zero Knowledge Proofs. All provenance verification must be cryptographically relatable back to an account within DECODE.

Attribute vault Process running in the DECODE OS. Allows to control access to the attribute data.

Attribute verifier An entity which has the credentials / authority to verify the provenance of an attribute. The expression of this verification will be through cryptographic form which will then be associated with the attribute owned by a participant. e.g. Participant Alice has the attribute “resident of city X” which is related to some cryptographic evidence that this claim is true, traceable back to some authoritative source of city X. An attribute verifier will require an account in order to be able to create this verification. The entity who is the attribute verifier may act also act as an operator at the same time. Further, attribute verification and the subsequent use of that verification may happen within the same application. The authenticity of the verifier may itself be verified by its own attributes.

Attribute Verification Synonym: Attribute Provenance Verification

Blockchain A type of distributed ledger that records transactions in a sequence of “blocks”

Operator Organisation or individual who is responsible for the build, deployment and operation of an application. Operators may also own and control attributes via an account within decode. Operators may also create new attributes that are then associated with participants. The minimum requirements of eligibility for an operator to deploy an application to the decode platform require further discussion as the platform evolves. Taking into account a general desire for greater transparency to the participant of the organisations with whom they are interacting and sharing data.

DECODE Architecture The internal architecture of DECODE, node anatomy, communications between nodes, components that are used to build DECODE.

DECODE Ecosystem The environment in which DECODE operates, involving other systems, pilots, organisations, open source communities. We can consider various specific ecosystems for example the "Sharing economy" ecosystem which may have different characteristics.

DECODE Platform Software with a high degree of privacy by design that provides the core functionality of DECODE. For example, DECODE OS, Distributed Ledger, device metadata.[29][30]

DECODE Application A DECODE application is a domain specific software application which will leverage the DECODE platform. Some part of this application may be a website and it is anticipated that some element of the application will involve definition and deployment of Smart Rules, Profile definitions and Ontology meta data.

DECODE SDK The SDK is a development kit for modules developed and distributed by other consortium partners. It can be used to build, test and profile individual software applications on top of the DECODE OS, both locally and remotely on the continous integration infrastructure.[31]

DECODE Node Controlled execution environment where the DECODE Hub runs. Decentralized transactions will be continuously being carried by the Node.[32]

DECODE Hub The DECODE Hub is the hardware component of the DECODE architecture. The Hub is any device on which the DECODE OS is installed. It provides connectivity to IoT devices, connects to other DECODE nodes and supports the DECODE OS. As any other component of the DECODE architecture, the Hub needs to follow the values of openness, security, scalability, deployability and flexibility.[33]

DECODE OS The DECODE OS is the base operating system running all software designed, developed and deployed for the DECODE project.[34] The DECODE OS is a blend of Devuan OS, with DECODE specific packages included within it.

DECODE Core The DECODE Core is a process that acts as a co-ordinator for different components of DECODE (Application Smart Rule engines, IOT Connector, Attribute Vault, Ledger Connector).

Distributed Ledger A consensus of replicated, shared, and synchronized digital data geographically spread across multiple sites, countries, or institutions. There is no central administrator or centralised data storage.

Entitlement It describe which Participant's attribute can view or have access to another attribute.

[29] Camilla Serri Colombo Priya Samuel Jim Barritt, "D4.2 Online Test Infrastructure," 2017.
[30] Jill Irving, "D1.1 Decode Scenarios and Requirements Definition Report."
[31] Dr.Denis Jaromil Roio, "D4.1 Decode Os Software Development Kit (Sdk)."
[32] Kristoffer Engdahl Ernesto E. Lopez, "D4.3 Provisional Hardware Platform," last modified 2017, https://decodeproject.eu/publications/decodes-provisional-hardware-platform.
[33] Ibid.
[34] Dr.Denis Jaromil Roio, "D4.4 First Release of Decode Os."

IOT Connector Previous Term: Device Hub. Maps IoT data coming from external sensor devices to an ontology known to DECODE.

Ledger Connector Allows the processes part of DECODE Core to interface with the Distributed Ledger.

Ledger Node This term is derived from the Bitcoin concept of a node. Bitcoin is a widely known cryptocurrency based on blockchain that organizes nodes in a peer-to-peer (P2P) network; any node can join and become part of the network. If a node receives new information, it broadcasts it to rest of the network. While all nodes listen to and broadcast information, only special nodes can append information to the blockchain.[35]

Metadata service The metadata service is responsible for maintaining an index of discoverable data and their locations, displaying any data that is accessible, allow a user to make an entitlement request to access data that is not yet accessible.[36]

Node Host This refers to an (online) service provider hosting nodes on behalf of participants.

Ontology In computer science and information science, an ontology is a formal naming and definition of the types, properties, and interrelationships of the entities that really or fundamentally exist for a particular domain of discourse. It is thus a practical application of philosophical ontology, with a taxonomy. Specifically in information science terminology, an ontology is a referenceable "schema" which describes the attribute. As raised, we need to be careful this does not become too restrictive. There is a wider discussion around this from the device hub work. While all attributes require a definition, registering changes to ontologies is an open process. Some governance / structure will be needed (to be discussed later). We will also need to consider evolution of ontologies over time.

Participant An individual who digitally participates in the DECODE ecosystem. Participation occurs by interacting with an application. participants own and control attributes via their account.

Profile Relates to the set of attributes which are required application to run. There are as many profiles as there are Applications. It is application specific as to whether the attributes need to be verified. A Profile may also declare attributes that it will generate and then be associated with an account, for example browsing behaviour, favorites, reccommendations.

Smart Contract A computer protocol intended to facilitate, verify, or enforce the negotiation or performance of a contract.

Smart Rule engine Process running in the DECODE OS. It is part of the Application and runs the Application Smart Rules.

[35] George Danezis (University College London) Mustafa Al-Bassam (University College London) Shehar Bano (University College London), "D3.1 Survey of Technologies for Abc, Entitlements and Blockchains," 2017.

[36] Mark deVilliers, "Data Access and Transaction Management Module," last modified 2017, `https://decodeproject.eu/publications/decodes-data-access-and-transaction-management-module`.

Smart Rule They are Application specific functions that take attributes (or objects) and create entitlements.

SSO (Single Sign On) It is a property of access control of multiple related, yet independent, software systems. With this property, a user logs in with a single ID and password to gain access to a connected system or systems without using different usernames or passwords.

Transaction Transactions are the application of one or more valid procedures - according to type - to active input objects, and possibly some referenced objects, to create a number of new active output objects.[37]

[37] George Danezis, "Chainspace."

Bibliography

Alpar, Gergely. *Attribute-Based Identity Management: Bridging the Cryptographic Design of Abcs with the Real World*. Uitgever niet vastgesteld, 2015.

Bethencourt, John, Amit Sahai, and Brent Waters. "Ciphertext-Policy Attribute-Based Encryption." In *Proceedings of the 2007 Ieee Symposium on Security and Privacy*, 321–334. SP '07. Washington, DC, USA: IEEE Computer Society, 2007. http://dx.doi.org/10.1109/SP.2007.11.

Buterin, Vitalik. "Ethereum White Paper."
https://github.com/ethereum/wiki/wiki/White-Paper#history.

Camenisch, Jan, and Anna Lysyanskaya. "An Efficient System for Non-Transferable Anonymous Credentials with Optional Anonymity Revocation." In *Advances in Cryptology - EUROCRYPT 2001, International Conference on the Theory and Application of Cryptographic Techniques, Innsbruck, Austria, May 6-10, 2001, Proceeding*, 93–118, 2001. https://doi.org/10.1007/3-540-44987-6_7.

Colesky, Michael, Jaap-Henk Hoepman, and Christiaan Hillen. "A Critical Analysis of Privacy Design Strategies." In *2016 IEEE Security and Privacy Workshops, SP Workshops 2016, San Jose, ca, Usa, May 22-26, 2016*, 33–40, 2016. https://doi.org/10.1109/SPW.2016.23.

deVilliers, Mark. "Data Access and Transaction Management Module." Last modified 2017.
https://decodeproject.eu/publications/decodes-data-access-and-transaction-management-module.

Dr.Denis Jaromil Roio, Ivan Jelinčić. "D4.4 First Release of Decode Os," 2017.

Dr.Denis Jaromil Roio, Ivan Jelinčić, Dr.Vincenzo Katolaz Nicosia. "D4.1 Decode Os Software Development Kit (Sdk)," 2017.

Drummond Reed, Jason Law & Daniel Hardman. "The Technical Foundations of Sovrin." Last modified 2016.
https://sovrin.org/wp-content/uploads/2017/04/The-Technical-Foundations-of-Sovrin.pdf.

Eleonora Bassi (Politecnico di Torino), Juan Carlos De Martin (Politecnico di Torino), Marco Ciurcina (Politecnico di Torino. "D1.8 Legal Frameworks for Digital Commons Decode Os and Legal Guidelines," 2017.

Ernesto E. Lopez, Kristoffer Engdahl. "D4.3 Provisional Hardware Platform." Last modified 2017.
https://decodeproject.eu/publications/decodes-provisional-hardware-platform.

EU. "GDPR Glossary of Terms." http://www.eugdpr.org/glossary-of-terms.html.

George Danezis, Alberto Sonnino, Mustafa Al-Bassam. "Chainspace: A Sharded Smart Contracts Platform," 2017.

Goyal, Vipul, Omkant Pandey, Amit Sahai, and Brent Waters. "Attribute-Based Encryption for Fine-Grained Access Control of Encrypted Data." In *Proceedings of the 13th Acm Conference on Computer and Communications Security*, 89–98. CCS '06. New York, NY, USA: ACM, 2006.
http://doi.acm.org/10.1145/1180405.1180418.

IBM Research. "Identity Mixer." https://www.zurich.ibm.com/identity_mixer/.

Jaap-Henk Hoepman, Eleonora Bassi, Shehar Bano. "D1.2 Privacy Design Strategies for the Decode Architecture," 2017.

Jill Irving, Jim Barritt, Jen Hughes. "D1.1 Decode Scenarios and Requirements Definition Report." Last modified 2017. https://decodeproject.eu/publications/pilot-scenarios-and-requirements.

Leonaviciute, Indre. “Designing to Facilitate Genuine Accommodation Sharing: Identity and Reputation Verification,” 2016.

Louwerse, Max M. “Semantic Variation in Idiolect and Sociolect.” *Computers and the Humanities* (2004).

Morell, Mayo. “Governance of Online Creation Communities for the Building of Digital Commons: Viewed Through the Framework of the Institutional Analysis and Development,” September 2014.

Mustafa Al-Bassam (University College London), George Danezis (University College London), Shehar Bano (University College London). “D3.1 Survey of Technologies for Abc, Entitlements and Blockchains,” 2017.

(Nesta), Tom Symons. “DECODE Project Dissemination Strategy and Communication Plan - Initial,” 2017.

Priya Samuel, Camilla Serri Colombo, Jim Barritt. “D4.2 Online Test Infrastructure,” 2017.

Raymond, Eric S. *The Art of Unix Programming*. Pearson Education, 2003.

Sober, Elliott. “Computability and Cognition.” *Synthese* (1978).

Symons, Tom, and Theo Bass (Nesta). “Me, My Data and I: The Future of the Personal Data Economy.” Last modified 2017.
https://decodeproject.eu/publications/me-my-data-and-ithe-future-personal-data-economy.

The Chainspace Distributed Ledger

3.1 Introduction

This second part of th deliverable contains the technical report describing the Chainspace platform that has been developed by DECODE. Chainspace has been specifically designed for the needs of the project: it consists in a high-integrity, high-availability decentralized platform to create and execute smart-contracts. Examples of these smart contracts, presenting in detail the implementation of a smart metering system and an e-voting platform can be found in the appendices.

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. This section presents 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. The full paper is available at [**?**].

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 section achieves the following goals:

- 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.

3.2 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.

3.2.1 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:

$$c.p(\vec{w}, \vec{r}, \mathsf{lpar}, \mathsf{spar}) \to \vec{x}, \mathsf{lret}, \mathsf{sret} \tag{3.1}$$

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.

$$c.v(p, \vec{w}, \vec{r}, \mathsf{lpar}, \vec{x}, \mathsf{lret}, \mathsf{dep}) \to \{\mathsf{true}, \mathsf{false}\} \tag{3.2}$$

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 (3.1). 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

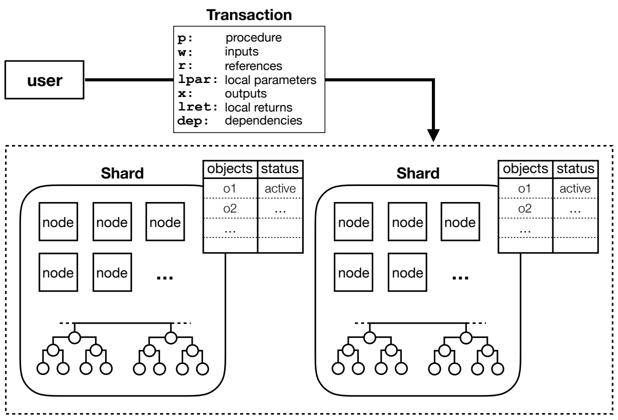


Figure 3.1: Design overview of Chainspace system.

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.

3.2.2 System Design, Threat Model and Security Properties

We provide an overview of the system design, illustrated in Figure 3.1. 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 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.

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

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

Figure 3.2: The sequencing and checking validity rules for transactions.

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.

3.3 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:

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

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 3.2. 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 3.2 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 3.2 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.

3.4 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.

3.4.1 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 3.2); 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.

3.4.2 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 3.2).
- 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.

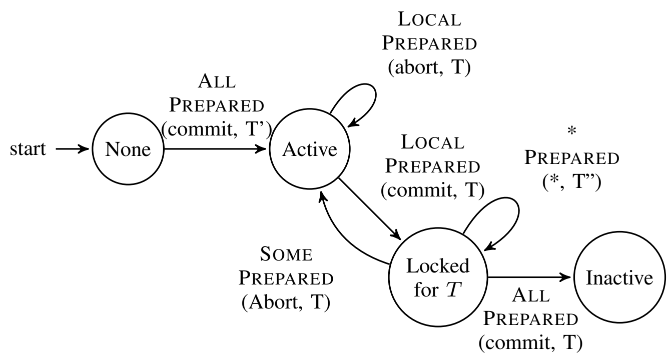


Figure 3.3: The state machine representing the active, locked and inactive states for any object within Chainspace. Each node in a shard replicates the state of the object, and participates in a consensus protocol that allows it to derive the invariants "Local prepared", "All prepared", and "Some prepared" to update the state of an object.

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

3.4.3 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 Mod-SMaRt [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 3.4 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 3.3. The combined protocol has been described below. For ease of

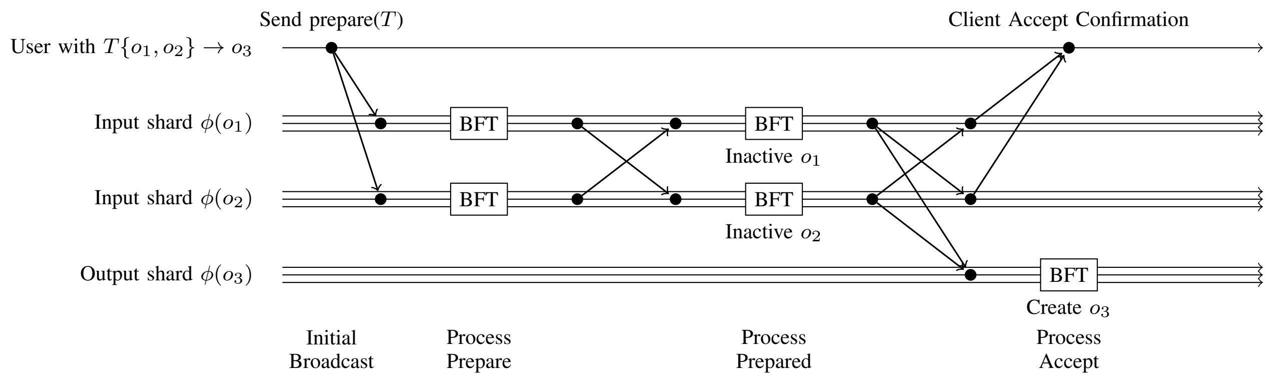


Figure 3.4: $\mathcal{S}$-BAC for a transaction T with two inputs (o_1, o_2) and one output object (o_3). The user sends the transaction to all nodes in shards managing o_1 and o_2. The BFT-Initiator takes the lead in sequencing T, and emits 'prepared(accept, T)' or 'prepared(abort, T)' to all nodes 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.

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 3.2), 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 3.3) 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.

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 3.3). 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.

3.4.4 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. Furthermore, those inputs must be managed by different nodes within the system to ensure that load of accepting transactions is distributed across them.

3.5 Security and Correctness

3.5.1 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)'. □

3.5.2 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 3.2. 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 3.2 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 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.

3.6 System and Applications Smart Contracts

3.6.1 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:

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

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.

3.6.2 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 ??.

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.

3.7 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.

3.8 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.

3.9 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.

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.

[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.

[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.

Appendices

Example Chainspace Contracts

A.1 Voting

```
"""A smart contract that implements a voting system bank."""

####################################################################
# imports
####################################################################
# general
from hashlib import sha256
from json    import dumps, loads
# chainspace
from chainspacecontract import ChainspaceContract
# crypto
from petlib.bn    import Bn
from petlib.ec    import EcGroup
from petlib.ecdsa import do_ecdsa_sign, do_ecdsa_verify
from chainspacecontract.examples.utils import setup, key_gen, pack, unpack, add, add_side
from chainspacecontract.examples.utils import binencrypt, make_table, dec
from chainspacecontract.examples.utils import provezero, verifyzero, provebin, verifybin, proveone,
    verifyone

## contract name
contract = ChainspaceContract('vote')


####################################################################
# methods
####################################################################
# ------------------------------------------------------------------
# init
# ------------------------------------------------------------------
@contract.method('init')
def init():
    # return
    return {
        'outputs': (dumps({'type' : 'VoteToken'}),)
    }

# ------------------------------------------------------------------
# create vote event
# NOTE:
#   - only 'inputs', 'reference_inputs' and 'parameters' are used to the framework
#   - if there are more than 3 param, the checker has to be implemented by hand
# ------------------------------------------------------------------
@contract.method('create_vote')
def create_vote(inputs, reference_inputs, parameters, options, participants, tally_priv, tally_pub):

    # genrate param
    params = setup()
    pub = unpack(tally_pub)

    # encrypt initial score
    (a, b, k) = binencrypt(params, pub, 0)   # encryption of a zero
    c = (a, b)
    scores = [pack(c) for _ in loads(options)]

    # new vote object
    new_vote = {
        'type'          : 'VoteObject',
        'options'       : loads(options),
        'scores'        : scores,
        'participants'  : loads(participants),
        'tally_pub'     : tally_pub
    }

    # proof that all init values are zero
    proof_init = provezero(params, pub, c, unpack(tally_priv))

    # return
    return {
        'outputs': (inputs[0], dumps(new_vote)),
        'extra_parameters' :
            (pack(proof_init),)
    }

# ------------------------------------------------------------------
# add vote
# NOTE:
#   - only 'inputs', 'reference_inputs' and 'parameters' are used to the framework
#   - if there are more than 3 param, the checker has to be implemented by hand
# ------------------------------------------------------------------
@contract.method('add_vote')
def add_vote(inputs, reference_inputs, parameters, added_vote, voter_priv, voter_pub):


    # retrieve old vote & init new vote object
    old_vote = loads(inputs[0])
    new_vote = loads(inputs[0])
    added_vote = loads(added_vote)

    # generate params & retrieve tally's public key
    params = setup()
    tally_pub = unpack(old_vote['tally_pub'])

    # encrypt votes & proofs to build
    enc_added_votes = []  # encrypted votes
    proof_bin       = []  # votes are binary, well-formed, and the prover know the vote's value
    sum_a, sum_b, sum_k = (0, 0, 0)  # sum of votes equals 1

    # loop over votes
    for i in range(0,len(added_vote)):
        # encrypt added vote
        (a, b, k) = binencrypt(params, tally_pub, added_vote[i])
        c = (a, b)
        enc_added_votes.append(pack(c))

        # update new scores
        new_c = add(unpack(old_vote['scores'][i]), c)
        new_vote['scores'][i] = pack(new_c)

        # construct proof of binary
        tmp1 = provebin(params, tally_pub, (a,b), k, added_vote[i])
        proof_bin.append(pack(tmp1))

        # update sum of votes
        if i == 0:
            sum_a, sum_b, sum_k = (a, b, k)
        else:
            sum_c = (sum_a, sum_b)
            sum_a, sum_b, sum_k = add_side(sum_c, c, sum_k, k)

    # build proof that sum of votes equals 1
    sum_c = (sum_a, sum_b)
    proof_sum = proveone(params, tally_pub, sum_c, sum_k)

    # remove voter from participants
    new_vote['participants'].remove(voter_pub)

    # compute signature
    (G, _, _, _) = params
    hasher = sha256()
    hasher.update(dumps(old_vote).encode('utf8'))
    hasher.update(dumps(enc_added_votes).encode('utf8'))
    sig = do_ecdsa_sign(G, unpack(voter_priv), hasher.digest())

    # return
    return {
        'outputs': (dumps(new_vote),),
        'extra_parameters' : (
            dumps(enc_added_votes),
            pack(sig),
            voter_pub, # already packed
            dumps(proof_bin),
            pack(proof_sum)
        )
    }

# ------------------------------------------------------------------
# tally
# NOTE:
#   - only 'inputs', 'reference_inputs' and 'parameters' are used to the framework
#   - if there are more than 3 param, the checker has to be implemented by hand
# ------------------------------------------------------------------
@contract.method('tally')
```

```
def tally(inputs, reference_inputs, parameters, tally_priv, tally_pub):

    # retrieve last vote
    vote = loads(inputs[0])

    # generate params & retrieve tally's public key
    params = setup()
    table  = make_table(params)
    (G, _, (h0, _, _, _), _) = params

    # decrypt aggregated results
    outcome = []
    for item in vote['scores']:
        outcome.append(dec(params, table, unpack(tally_priv), unpack(item)))

    # proof of decryption
    proof_dec = []
    for i in range(0, len(vote['scores'])):
        a, b = unpack(vote['scores'][i])
        ciphertext = (a, b - outcome[i] * h0)
        tmp = provezero(params, unpack(tally_pub), ciphertext, unpack(tally_priv))
        proof_dec.append(pack(tmp))

    # signature
    hasher = sha256()
    hasher.update(dumps(vote).encode('utf8'))
    hasher.update(dumps(outcome).encode('utf8'))
    sig = do_ecdsa_sign(G, unpack(tally_priv), hasher.digest())

    # pack result
    result = {
        'type'      : 'VoteResult',
        'outcome'   : outcome
    }

    # return
    return {
        'outputs': (dumps(result),),
        'extra_parameters' : (
            dumps(proof_dec),
            pack(sig)
        )
    }

# ------------------------------------------------------------------
# read
# ------------------------------------------------------------------
@contract.method('read')
def read(inputs, reference_inputs, parameters):

    # return
    return {
        'returns' : (reference_inputs[0],),
    }
```

```
####################################################################
# checker
####################################################################
# ------------------------------------------------------------------
# check vote's creation
# ------------------------------------------------------------------
@contract.checker('create_vote')
def create_vote_checker(inputs, reference_inputs, parameters, outputs, returns, dependencies):
    try:

        # retrieve vote
        vote  = loads(outputs[1])
        num_votes = len(vote['options'])

        # check format
        if len(inputs) != 1 or len(reference_inputs) != 0 or len(outputs) != 2 or len(returns) != 0:
            return False
        if num_votes < 1 or num_votes != len(vote['scores']):
            return False
        if vote['participants'] == None:
            return False

        # check tokens
        if loads(inputs[0])['type'] != 'VoteToken' or loads(outputs[0])['type'] != 'VoteToken':
            return False
        if vote['type'] != 'VoteObject':
            return False

        # check proof
        params = setup()
        proof_init = unpack(parameters[0])
        tally_pub  = unpack(vote['tally_pub'])
        for value in vote['scores']:
            if not verifyzero(params, tally_pub, unpack(value), proof_init):
                return False

        # otherwise
        return True

    except (KeyError, Exception):
        return False

# ------------------------------------------------------------------
# check add vote
# ------------------------------------------------------------------
@contract.checker('add_vote')
def add_vote_checker(inputs, reference_inputs, parameters, outputs, returns, dependencies):
    try:

        # retrieve vote
        old_vote = loads(inputs[0])
        new_vote = loads(outputs[0])
        num_votes = len(old_vote['options'])

        # check format
        if len(inputs) != 1 or len(reference_inputs) != 0 or len(outputs) != 1 or len(returns) != 0:
            return False
        if num_votes != len(new_vote['scores']) or num_votes != len(new_vote['scores']):
            return False
        if new_vote['participants'] == None:
            return False
        if old_vote['tally_pub'] != new_vote['tally_pub']:
            return False

        # check tokens
        if new_vote['type'] != 'VoteObject':
            return False

        # check that voter has been removed from participants
        if not parameters[2] in old_vote['participants']:
            return False
        if parameters[2] in new_vote['participants']:
            return False
        if len(old_vote['participants']) != len(new_vote['participants']) + 1:
            return False

        # generate params, retrieve tally's public key and the parameters
        params = setup()
        tally_pub  = unpack(old_vote['tally_pub'])
        added_vote = loads(parameters[0])
        sig        = unpack(parameters[1])
        voter_pub  = unpack(parameters[2])
        proof_bin  = loads(parameters[3])
        proof_sum  = unpack(parameters[4])

        # verify signature
        (G, _, _, _) = params
        hasher = sha256()
        hasher.update(dumps(old_vote).encode('utf8'))
        hasher.update(dumps(added_vote).encode('utf8'))
        if not do_ecdsa_verify(G, voter_pub, sig, hasher.digest()):
            return False

        # verify proofs of binary (votes have to be bin values)
        for i in range(0, num_votes):
            if not verifybin(params, tally_pub, unpack(added_vote[i]), unpack(proof_bin[i])):
                return False

        # verify proof of sum of votes (sum of votes has to be 1)
        sum_a, sum_b = unpack(added_vote[-1])
        sum_c = (sum_a, sum_b)
        for i in range(0, num_votes-1):
            sum_c = add(sum_c, unpack(added_vote[i]))
        if not verifyone(params, tally_pub, sum_c, proof_sum):
            return False

        # verify that output == input + vote
        for i in range(0, num_votes):
            tmp_c = add(unpack(old_vote['scores'][i]), unpack(added_vote[i]))
            if not new_vote['scores'][i] == pack(tmp_c):
                return False

        # otherwise
        return True
```

```
    except (KeyError, Exception):
        return False

# ------------------------------------------------------------------
# check tally
# ------------------------------------------------------------------
@contract.checker('tally')
def tally_checker(inputs, reference_inputs, parameters, outputs, returns, dependencies):
    try:

        # retrieve vote
        vote   = loads(inputs[0])
        result = loads(outputs[0])

        # check format
        if len(inputs) != 1 or len(reference_inputs) != 0 or len(outputs) != 1 or len(returns) != 0:
            return False
        if len(vote['options']) != len(result['outcome']):
            return False

        # check tokens
        if result['type'] != 'VoteResult':
            return False

        # generate params, retrieve tally's public key and the parameters
        params = setup()
        (G, _, (h0, _, _, _), _) = params
        tally_pub  = unpack(vote['tally_pub'])
        proof_dec  = loads(parameters[0])
        sig        = unpack(parameters[1])
        outcome    = result['outcome']

        # verify proof of correct decryption
        for i in range(0, len(vote['scores'])):
            a, b = unpack(vote['scores'][i])
            ciphertext = (a, b - outcome[i] * h0)
            if not verifyzero(params, tally_pub, ciphertext, unpack(proof_dec[i])):
                return False

        # verify signature
        hasher = sha256()
        hasher.update(dumps(vote).encode('utf8'))
        hasher.update(dumps(result['outcome']).encode('utf8'))
        if not do_ecdsa_verify(G, tally_pub, sig, hasher.digest()):
            return False

        # otherwise
        return True

    except (KeyError, Exception):
        return False

# ------------------------------------------------------------------
# check read
# ------------------------------------------------------------------
@contract.checker('read')
```

```
def read_checker(inputs, reference_inputs, parameters, outputs, returns, dependencies):
    try:

        # check format
        if len(inputs) != 0 or len(reference_inputs) != 1 or len(outputs) != 0 or len(returns) != 1:
            return False

        # check values
        if reference_inputs[0] != returns[0]:
            return False

        # otherwise
        return True

    except (KeyError, Exception):
        return False


####################################################################
# main
####################################################################
if __name__ == '__main__':
    contract.run()



####################################################################
```

A.2 Smart Metering

```

"""A smart contract that implements a smart meter."""

####################################################################
# imports
####################################################################
# general
from hashlib import sha256
from json    import dumps, loads
# chainspace
from chainspacecontract import ChainspaceContract
# crypto
from petlib.ecdsa import do_ecdsa_sign, do_ecdsa_verify
from chainspacecontract.examples.utils import setup, key_gen, pack, unpack

## contract name
contract = ChainspaceContract('smart_meter')


####################################################################
# methods
####################################################################
# ------------------------------------------------------------------
# init
# ------------------------------------------------------------------
```

```
@contract.method('init')
def init():

    # return
    return {
        'outputs': (dumps({'type' : 'SMToken'}),),
    }

# ------------------------------------------------------------------
# create meter
# NOTE:
#   - only 'inputs', 'reference_inputs' and 'parameters' are used to the framework
#   - if there are more than 3 param, the checker has to be implemented by hand
# ------------------------------------------------------------------
@contract.method('create_meter')
def create_meter(inputs, reference_inputs, parameters, pub, info, tariffs, billing_period):

    # new meter
    new_meter = {
        'type'           : 'SMMeter',
        'pub'            : pub,
        'info'           : info,
        'readings'       : [],
        'billing_period' : loads(billing_period),
        'tariffs'        : loads(tariffs)
    }

    # return
    return {
        'outputs': (inputs[0], dumps(new_meter))
    }

# ------------------------------------------------------------------
# add_reading
# NOTE:
#   - only 'inputs', 'reference_inputs' and 'parameters' are used to the framework
#   - if there are more than 3 param, the checker has to be implemented by hand
# ------------------------------------------------------------------
@contract.method('add_reading')
def add_reading(inputs, reference_inputs, parameters, meter_priv, reading, opening):

    # compute output
    old_meter = loads(inputs[0])
    new_meter = loads(inputs[0])

    # create commitement to the reading
    (G, g, (h0, _, _, _), _) = setup()
    commitment = loads(reading) * g + unpack(opening) * h0

    # update readings
    new_meter['readings'].append(pack(commitment))

    # hash message to sign
    hasher = sha256()
    hasher.update(dumps(old_meter).encode('utf8'))
    hasher.update(dumps(pack(commitment)).encode('utf8'))
    # sign message
    sig = do_ecdsa_sign(G, unpack(meter_priv), hasher.digest())

    # return
    return {
        'outputs': (dumps(new_meter),),
        'extra_parameters' : (
            pack(commitment),
            pack(sig)
        )
    }

# ------------------------------------------------------------------
# compute_bill
# NOTE:
#   - only 'inputs', 'reference_inputs' and 'parameters' are used to the framework
#   - if there are more than 3 param, the checker has to be implemented by hand
# ------------------------------------------------------------------
@contract.method('compute_bill')
def compute_bill(inputs, reference_inputs, parameters, readings, openings, tariffs):

    # get meter
    meter = loads(inputs[0])

    # compute total bill
    G = setup()[0]
    total_bill   = sum(r*t for r,t in zip(loads(readings), loads(tariffs)))
    sum_openings = sum(o*t for o,t in zip(unpack(openings), loads(tariffs))) % G.order()

    # new bill
    bill = {
        'type'           : 'SMBill',
        'info'           : meter['info'],
        'total_bill'     : total_bill,
        'billing_period' : meter['billing_period'],
        'tariffs'        : meter['tariffs']
    }

    # return
    return {
        'outputs': (dumps(bill),),
        'extra_parameters' : (
            dumps(total_bill),
            pack(sum_openings),
        )
    }

# ------------------------------------------------------------------
# read_bill
# ------------------------------------------------------------------
@contract.method('read')
def read(inputs, reference_inputs, parameters):

    # return
    return {
        'returns' : (reference_inputs[0],),
    }
```

```
####################################################################
# checker
####################################################################
# ------------------------------------------------------------------
# check meter's creation
# ------------------------------------------------------------------
@contract.checker('create_meter')
def create_meter_checker(inputs, reference_inputs, parameters, outputs, returns, dependencies):
    try:

        # loads data
        meter = loads(outputs[1])

        # check format
        if len(inputs) != 1 or len(reference_inputs) != 0 or len(outputs) != 2 or len(returns) != 0:
            return False
        if meter['pub'] == None or meter['info'] == None or meter['billing_period'] == None:
            return False
        if meter['readings'] == None or meter['tariffs'] == None:
            return False

        # check tokens
        if loads(inputs[0])['type'] != 'SMToken' or loads(outputs[0])['type'] != 'SMToken':
            return False
        if meter['type'] != 'SMMeter':
            return False

        # otherwise
        return True

    except (KeyError, Exception):
        return False

# ------------------------------------------------------------------
# check add reading
# ------------------------------------------------------------------
@contract.checker('add_reading')
def add_reading_checker(inputs, reference_inputs, parameters, outputs, returns, dependencies):
    try:

        # get objects
        old_meter = loads(inputs[0])
        new_meter = loads(outputs[0])

        # check format
        if len(inputs) != 1 or len(reference_inputs) != 0 or len(outputs) != 1 or len(returns) != 0:
            return False
        if old_meter['pub'] != new_meter['pub'] or old_meter['info'] != new_meter['info']:
            return False
        if old_meter['tariffs'] != new_meter['tariffs'] or old_meter['billing_period'] !=
            new_meter['billing_period']:
            return False

        # check tokens
        if old_meter['type'] != new_meter['type']:
            return False

        # check readings' consistency
        if new_meter['readings'] != old_meter['readings'] + [parameters[0]]:
            return False

        # hash message to sign
        hasher = sha256()
        hasher.update(dumps(old_meter).encode('utf8'))
        hasher.update(dumps(parameters[0]).encode('utf8'))

        # verify signature
        G = setup()[0]
        pub = unpack(old_meter['pub'])
        sig = unpack(parameters[1])
        if not do_ecdsa_verify(G, pub, sig, hasher.digest()):
            return False

        # otherwise
        return True

    except (KeyError, Exception):
        return False

# ------------------------------------------------------------------
# check compute bill
# ------------------------------------------------------------------
@contract.checker('compute_bill')
def compute_bill_checker(inputs, reference_inputs, parameters, outputs, returns, dependencies):
    try:

        # get objects
        meter = loads(inputs[0])
        bill  = loads(outputs[0])

        # check format
        if len(inputs) != 1 or len(reference_inputs) != 0 or len(outputs) != 1 or len(returns) != 0:
            return False
        if meter['billing_period'] != bill['billing_period'] or meter['info'] != bill['info']:
            return False
        if meter['tariffs'] != bill['tariffs']:
            return False
        if bill['total_bill'] != loads(parameters[0]):
            return False

        # check tokens
        if bill['type'] != 'SMBill':
            return False

        # get objects
        tariffs      = bill['tariffs']
        commitements = meter['readings']
        total_bill   = loads(parameters[0])
        sum_openings = unpack(parameters[1])

        # verify bill
        (G, g, (h0, _, _, _), _) = setup()
        bill_commitment = G.infinite()
        for i in range(0, len(commitements)):
            bill_commitment = bill_commitment + tariffs[i] * unpack(commitements[i])

        if bill_commitment - sum_openings * h0 != total_bill * g:
            return False

        # otherwise
        return True

    except (KeyError, Exception):
        return False

# ------------------------------------------------------------------
# check read bill
# ------------------------------------------------------------------
@contract.checker('read')
def read_checker(inputs, reference_inputs, parameters, outputs, returns, dependencies):
    try:

        # check format
        if len(inputs) != 0 or len(reference_inputs) != 1 or len(outputs) != 0 or len(returns) != 1:
            return False

        # check values

        if reference_inputs[0] != returns[0]:
            return False

        # otherwise
        return True

    except (KeyError, Exception):
        return False


####################################################################
# main
####################################################################
if __name__ == '__main__':
    contract.run()



####################################################################
```