SOFIE - Secure Open Federation for Internet Everywhere

779984

DELIVERABLE D4.3

First Architecture and System Evaluation Report

Project title	SOFIE – Secure Open Federation for Internet Everywhere
Contract Number	H2020-IOT-2017-3 – 779984
Duration	1.1.2018 – 31.12.2020
Date of preparation	2.7.2019
Author(s)	Vasilios A. Siris, Spyros Voulgaris, Nikos Fotiou, Dimitrios Dimopoulos, Iakovos Pittaras, George C. Polyzos (AUEB-RC), Tommi Elo, Ektor Arzoglou, Dmitrij Lagutin (AALTO)
Responsible person	Vasilios A. Siris (AUEB-RC, vsiris@aueb.gr)
Target Dissemination Level	Public
Status of the Document	Completed
Version	1.00
Project web-site	https://www.sofie-iot.eu/

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 779984.

Table of Contents

1. Introduction

This is the third deliverable of WP4 (Evaluation), due at the end of June 2019, at the same time with D2.4 (Federation Architecture, 2nd version), D5.2 (Initial Platform Validation) and before D2.5 (Federation Framework, 2nd version — August 2019) and D3.3 (Business platforms, pilot release — September 2019). It is thus apparent that the design of the architecture, the business platforms, the pilots and the implementation of the federation framework components and the system(s) are in flux. Therefore, this deliverable focuses more on evaluating key aspects of the architecture and the system rather than providing an evaluation of a specific data point in the development and even more so of key mechanisms for achieving various functionalities enabling aspects of the architecture and system.

This First Architecture and System Evaluation Report, therefore, will mainly provide component results and also illustrate the type of evaluation being performed within the project. We expect to provide more concrete and integrated results in the next WP4 deliverable, D4.4 (Second Architecture and System Evaluation Report — April 2020). The work in WP4, up to June 2019 documented in this deliverable, follows the plan set in D4.1 and employs tools that have been set-up for the project and were presented in D4.2.

1.1 Goals of this deliverable

The goals for this first evaluation report are the following:

- to provide a first evaluation of the SOFIE approach, architecture, systems, and components, in order to promote the SOFIE approach and establish foundations for its impact on technology and business,
- to promote the evaluation approach and techniques within the project in order to provide guidelines for the pilot evaluation activities,
- to present the evaluation approach and techniques internally to the project to have them scrutinized to determine if they will be adequate, or further tools, methodologies and techniques will be necessary, for a convincing evaluation (and then the successful promotion) of SOFIE,
- to help in the selection of the most appropriate techniques for SOFIE and in particular the ledger and interledger technologies for the SOFIE use cases and pilots,
- to identify gaps to be addressed during further conceptual and technological design, development and evaluation (both regarding methodologies and their application).

1.2 Methodologies and approach

The methodologies employed for evaluation are many and diverse, from simple presentation of arguments and qualitative evaluation, through modelling, analytical evaluation and simulation, to implementation and measurements in real components and systems. Since pilots have a central position in the SOFIE project, an important evaluation direction will be undertaken using each pilot, considering the actual system and evaluating it in a specific application context. These evaluations will be performed towards the end of the pilots' lifetime and they will be integrated with other WP4 evaluation results in the final WP4 evaluation deliverable (D4.5). However, they also provide more concrete systems and applications in which we consider the SOFIE approach and evaluate it initially, starting from this deliverable. We also generalize from the specific choices made in the pilots to the use cases from which they have been inspired, evaluating many potential alternatives around them.

Thus, we are inspired and guided by the pilots and their use cases, as well as the software and solutions developed for them; however, WP4 aims to have a wider scope. Thus, it has chosen as one key approach for evaluation the emulation and/or simulation of the use cases considered in the pilots, but in a more general context, considering and evaluating various possible solutions

and their parameters, going beyond what is possible within the pilots. On the other hand, in order to achieve this breadth, it needs to model and abstract out various aspects of the pilots, as will be explicitly described below.

In addition to the methodologies and tools for evaluation, the questions to be answered (i.e., the targets of the evaluation) are diverse. They range from traditional performance metrics, which typically have limited generality, as they have to refer to fully specified systems, to more general questions such as security analysis, robustness, usability and even business analysis. It is therefore even more obvious that the tools to be used for evaluation must be diverse and applied at very different abstraction levels and under different assumptions.

Given the diverse goals and evaluation methodologies, the specific concepts and metrics for comparison and measurements are also diverse. A starting point is, of course, the Key Performance Indicators (KPIs) chosen in the first SOFIE architecture deliverable (D2.2, Annex 1). However, additional metrics and qualitative characterizations will be used in this deliverable and in the following ones for WP4.

1.3 Structure of this deliverable

Due to the diverse goals and evaluation methodologies involved, we decided to have the following structure for this deliverable. Section 2 is a high-level architecture evaluation. It focuses more on the style and desired properties of the architecture, rather than very specific architectural components and structure.

Section 3 is about an initial component evaluation. These are components chosen to be part of the SOFIE Federation Framework, which however has not yet been completed (the deliverable, D2.5, reporting on the Federation Framework implementation, is due on August 2019). Therefore, we decided to start here with the three components that we were more familiar with, having contributed to their definition and implementation, which were also in an advanced stage of definition and implementation.

Section 4 presents a detailed evaluation of IoT resource access and even though it relates to the components described and evaluated in Section 3, it goes to much more depth and considers many more alternatives and their tradeoffs than one would consider at the SOFIE component level. Moreover, because authorized access to IoT resources is necessary across different pilots, we decided to dedicate a separate section to this topic. Other sections do not intimately depend on it, but on the other hand it shows that much more detailed design and evaluation is possible and can uncover many useful alternatives in the IoT world.

Section 5, on Evaluation Scenarios, follows the SOFIE pilots, generalizes them into pilot inspired use cases by including alternatives that are not necessarily selected in the SOFIE pilots, abstracts them out to an appropriate degree and uses emulation and simulation to consider the various tradeoffs of many potential alternative design decisions, e.g., whether to use one blockchain overall, one blockchain per hand-off, a hierarchy of blockchains, blockchains of different types (public/private, or permissionless/permissioned, etc.) and the impact of these choices, evaluated from many viewpoints and using different methodologies, from descriptions and illustrative arguments to quantitative evaluation with specific metrics and numerical results.

Section 6 addresses business platforms evaluation (decentralized, build on SOFIE principles and, eventually, with SOFIE components). An illustration of a System Dynamics approach applied to the SOFIE architecture and pilot-based use cases is provided here, demonstrating the interactions among components (systems, players, parameters, etc.). It is preliminary work, qualitative only for the moment, but sets the stage and documents our efforts in this area.

Finally, the document concludes in Section 7 with a summary and outlook towards future WP4 work and its relationship with other SOFIE WPs and efforts in general.

2. Architecture evaluation

In this section we discuss and bring out key aspects of the SOFIE architecture that are critical for the SOFIE approach to IoT system federation and open business platform success.

2.1 Decentralization

Decentralization is the norm for large-scale real-world systems. Decentralization was the key concept in the development of packet switching, starting essentially with the ARPAnet (as opposed, for example, to the Telephone networks of the time). However, there is continuous cycling through centralization and decentralization in communications technology. For example, the original cable-based Ethernet was fully decentralized with a passive (and thus robust for the era) interconnection medium. However, as electronics became very reliable we moved towards (the less decentralized) switched Ethernet. Similarly, there is now a push towards centralizing large packet networks through SDN and powerful SDN controllers, allowing for better and tighter control of such networks. However, there are currently also efforts to fully decentralize the global Internet, even from second-order centralization, such as data and measurements and other information repositories that enforce common decisions, to independent agents or administrative domains. DLTs are the key tools in many such efforts; see e.g. Blockstack.[1]

For early IoT systems, isolation and centralization were the chosen paths for many reasons: necessity, due to a lack of appropriate standards or technologies, fast and cheap deployment, but most importantly, business reasons. Closed, centralized systems offer an edge against competitors and (some) peace of mind against attackers and even less exposure or “attack surface” for regulators and ethics enthusiasts. As the systems were increasing in size and geographical coverage, the relatively easier transition to Cloud systems was performed, still allowing a centralized model, but with the elasticity and scalability of the Cloud resources hiding the initial scalability and other concerns. This was also compatible with the closed business models and became the norm for IoT applications. That is, rather than a true (global) Internet of Things, many separate domains of Things separated by applications, business models, administrative domains, etc.

However, there are clear potential benefits (but also obvious challenges) for humanity and for the economy more specifically, in a global IoT, the more obvious one coming from the wide optimization potential of access to vast amounts of information (rather than its segregation into smaller domains). IoT scenarios involving a multitude of devices and platforms, many users, and many distinct administrative domains pose problems for centralized solutions, not only or primarily from a technological view point, but mostly from the business and governance perspectives. Designing an inherently decentralized architecture for multi-platform IoT ecosystems can reflect more closely the natural structure of data origins, avoiding the collection of data to a central repository, but allowing the interconnection of information under diverse rules and constraints.

Decentralization has certain intrinsic advantages over centralized approaches, mainly attributed to higher redundancy, availability, and tolerance to failures, as well as better load balancing.

2.2 Open business platforms

Further to decentralization, a key characteristic expected of future IoT and, more generally, business platforms, is openness. A business platform is a (software mainly) system where business transactions are undertaken with a high degree of automation. Maybe the best-known examples are the Apple App store and Google Play. In both these instances (and almost all such currently existing business platforms), Apple and Google, respectively, have a defining,

[1] https://blockstack.org

central, all powerful, and rulemaking position, deciding on who can “play” and also extracting a hefty fee out of the platform. Not only is the platform not decentralized, but it is also not open, i.e., not open to other players without the explicit and typically not automatic agreement of the defining player and, in particular, not open to competitors or game changers.

The SOFIE philosophy and statement to be proven is that open platforms are the future. They can support evolution and fast transformation and provide the correct incentives for players to participate and innovate and for society to benefit more and to better control the process through general rules applied equally to all. An investigation of decentralized open platforms is described in Section 6 of this document using the System Dynamics methodology and with ultimate goal to determine the conditions under which such platforms can emerge, grow, and prosper.

2.3 Multiple ledgers and interledger technology

The "one ledger to rule them all" approach is prone to failure, as it suffers a number of shortcomings. Public ledgers typically incur relatively high fees and longer latency to register a transaction, making them unsuitable for storing data very frequently. Private ledgers, on the other hand, can register transactions at a negligible cost and relatively faster than public ones, but their immutability guarantees are very low compared to those of public ledgers. A well designed interaction between ledgers of both types can bring significant benefits to a deployed framework, combining low-cost and fast data storage with high immutability guarantees. This is why a key component of the SOFIE approach is the use of interledger technologies. We provide an up-to-date and comprehensive survey, review and characterization/evaluation of interledger approaches in [Sir+19d].

2.4 Trust, security, transparency, availability, and accountability

These properties form the cornerstone of Distributed Ledgers, promoting transactions to a new level of trust and security that stems directly from cryptographic algorithms, rather than being imposed by a third party, institution or government. This is achieved by requiring consensus of a large number of nodes in validating transactions and, thereafter, storing them in an immutable structure, known as the blockchain. All nodes participating in a blockchain, be it permissioned or permissionless, essentially have access to the entire state and to all transactions recorded in it, which makes transparency an inherent property of blockchains. In the case of SOFIE, however, personal data, or data that are sensitive for certain entities, such as business secrets, should not be written in a blockchain in clear form. Instead, either an encrypted version of the information will be stored, or merely its hash.

Regarding availability, the fact that any participating node stores the entire history and state of a blockchain means that any interested entity can run one or more nodes in that blockchain, increasing availability. Finally, by using a blockchain for authorization and for acquiring the keys to access a service or a device, all accesses are recorded, providing for accountability. The aforementioned properties are particularly significant given the envisioned interactions among entities and businesses with diverse, and often conflicting interests.

Another relevant measure is usability, a qualitative measure that shows how easy it is for a system to be used, to be connected, etc. In this context it is important to also consider usability because DLTs might introduce significant complications in systems relying on them. (ISO 9241-11:2018 provides usability definitions and guidelines).

2.5 Architecture KPIs

In Deliverable D2.2, Annex 1, we defined a number of KPIs that will be used for the evaluation of the SOFIE architecture. In this section we further specify and extend these KPIs and report our progress on the architecture evaluation with respect to the defined KPIs.

KPI	Goal	Description	Metric	Method of verification
1	IoT operability	Prove operability of the implementation with IoT silos	Number of IoT silos	Detection of data flow in silos during implementation use case

The objective of this KPI is to prove the applicability of the SOFIE federation architecture and its components to existing IoT silos. The corresponding metric is the number of IoT silos where the architecture has been applied. The current version of the architecture and a subset of its components have been applied and evaluated to the following scenarios/silos:

- IoT resource access
- Food supply chain
- Electric vehicle energy marketplace
- Smart meters
- Mobile gaming

Each scenario utilizes different features of the architecture and its components, such as authentication and authorization, recording of data or hashes and execution of smart contracts in private/permissioned and public DLTs.

KPI	Goal	Description	Metric	Method of verification
2	IoT inter-operability	Prove interoperability across multiple IoT silos of the reference architecture	Number of IoT silo pairs	Implementation use case accesses data or actuates operations in different IoT silos

This KPI focuses on the application of the architecture and components to allow communication between different silos. For example, these silos can involve the platforms of different actors of a scenario, such as the transportation and the storage and distribution centre platforms in the food chain scenario. The corresponding metric that represents this KPI (and which we would like to maximize) is the number of IoT silo pairs that exchange data through the SOFIE architecture. The semantic interoperability will utilize W3C's Web of Things (WoT) things description model.

KPI	Goal	Description	Metric	Method of verification
3	Ledger use	Validate SOFIE implementation capability with multiple ledgers	Number of distributed ledgers[2]	Ledgers have detectable data passing through SOFIE implementations

A key goal of the SOFIE platform is to utilize different DLTs with different performance tradeoffs (execution cost and transaction time) and features, such as privacy, transparency, and trust.

[2] Across significantly different ledger technologies, e.g. Ethereum and Ethereum Classic are not considered different ledgers, as their differences are small enough to allow applications developed on Ethereum to be deployed on Ethereum Classic with only minor changes.

The DLTs that have been used in the evaluation experiments reported in this deliverable and the pilots defined in WP5 include the following ledgers:

- Public Ethereum, including the Rinkeby and Ropsten public Ethereum test networks
- Private Ethereum network
- Hyperledger Fabric
- Hyperledger Indy
- KSI blockchain

KPI	Goal	Description	Metric	Method of verification
4	Interledger use	Validate SOFIE implementation operating across multiple ledgers	Number of distributed ledger pairs	Implementation use case shown to result in operations across multiple ledgers

A primary goal of SOFIE is to enable the interoperation and exchange of information across different DLTs with different performance tradeoffs and features. The evaluation experiments reported in the current deliverable consider the interoperation of a private Ethereum network with the Rinkeby and Ropsten public Ethereum testnets.

Experiments with interledger smart contracts for the Identification, Authentication, Authorization (IAA) component have been conducted and are reported in Section 4.1. In addition to the interoperation of public/permissioned ledgers, we have experimented with Decentralized Identifiers (DIDs) recorded on Hyperledger Indy, which are reported in Section 3.3.

Ongoing evaluation work is investigating the interaction between the following pairs of DLTs:

- Hyperledger Fabric - Public Ethereum
- Private Ethereum - KSI blockchain

KPI	Goal	Description	Metric	Method of verification
5	Ledger independence	Demonstrate capability of developing applications using ledgers, where a sufficient abstraction can be provided to applications to allow them to be targeted simultaneously to multiple ledger technologies	Number of Business Platforms (BP) samples classified into *success* or *partial success*	Demonstrate that a BP sample can be deployed on two ledgers with only configuration changes, and the BP sample users are able to use either one with only configuration item changes

The goal of this KPI is to demonstrate the capability of developing applications using sufficient abstractions that allow the applications to run over multiple ledger technologies.

The experiments reported in this deliverable involve distributed apps implemented mainly using the Solidity language, which compiles them into Ethereum Virtual Machine (EVM) bytecode. The EVM bytecode can be executed in private Ethereum networks and public Ethereum (and the testnets). However, EVM can also be deployed in a "Blockchain as a service" type of Ethereum network (e.g., networks provided by Microsoft Azure[3] and Amazon's Managed Blockchain[4]), as well as in other blockchain systems (such as Hyperledger Fabric[5] and

[3] https://azure.microsoft.com/en-us/blog/ethereum-blockchain-as-a-service-now-on-azure/
[4] https://aws.amazon.com/managed-blockchain/
[5] https://www.hyperledger.org/blog/2018/10/26/hyperledger-fabric-now-supports-ethereum

Hyperledger Sawtooth[6]). Secondly, SOFIE's "Privacy and Data sovereignty" component (see section 3.2) and "Identification, authentication, authorization" component (see section 3.3) are based on generic enough identification systems and can be configured to be used with various types of ledgers, including Hyperledger Indy, as well as traditional ones (LDAP).

KPI	Goal	Description	Metric	Method of verification
6	Privacy designed in as a fundamental requirement	Demonstrate GDPR compliance where relevant	Number of operational GDPR features referenced and supported.[7]	Final specifications have clear references to features implementing named GDPR requirements. Relevant pilot specifications also refer to the needed features

This KPI concerns the compliance of the SOFIE architecture with the GDPR and its metric is the number of operational GDPR features referenced and supported.

The SOFIE architecture has the following features related to privacy and GDPR. Firstly, as discussed and investigated in section 5, the SOFIE architecture does not record personal data to immutable ledgers. Instead, the immutability of data recorded in local databases is ensured by recording hashes of the data in public ledgers. Secondly, in the various scenarios only the minimum set of data is stored in a public ledger, in order to ensure the correct operation and functionality that pertains to the specific scenario. Finally, SOFIE's applications do not process data without having permissions granted by users, e.g., in the mobile gaming scenario when the user installs the app, a pop-up screen is displayed asking for Storage, Access location, etc., permission.

KPI	Goal	Description	Metric	Method of verification
7	Device owner payments across ledgers	Ability of silo owners to send and receive payments or other value transfers	Number of ledger pairs supporting value transfer	Observation of value transfer as part of a use case in an implementation

Whereas KPI 4 on "Interledger use" focuses on the interoperability, in general between different ledgers, this KPI concerns the transfer or, more accurately, the exchange of value, between different ledgers. An example of such an exchange of value is discussed in detail in Section 4.1 and involves the exchange of a payment token, stored in a public Ethereum blockchain, with an access token stored in a private Ethereum blockchain. This exchange is performed using functionality of the interledger component.

[6] https://sawtooth.hyperledger.org/docs/seth/releases/latest/introduction.html
[7] The number of GDPR articles which lead to operational goals is generally thought to be about 10. See e.g. https://iapp.org/resources/article/top-10-operational-impacts-of-the-gdpr/

KPI	Goal	Description	Metric	Method of verification
8	Data sovereignty	Ability of data owners to reject or allow access, possibly for a specific time interval, to their data Each datum has an accompanying authorization list, which the data owner can modify	Number of pilot use cases utilizing data owner data sovereignty features and data owner is from a different silo than the storage silo	Count the number of use cases

This KPI is related to the ability of data owners to reject or allow access, possibly for a specific time interval, to their data. This KPI can be verified with the number of pilot use cases utilizing data owner data sovereignty features, where the owner can be in a different silo than the storage silo.

All scenarios presented in Section 5 can leverage the "Privacy and Data sovereignty" component of the SOFIE architecture to achieve data sovereignty. Solutions related to data access with specific functionality and features are investigated in more detail in Section 4.

KPI	Goal	Description	Metric	Method of verification
9	User responsiveness	Apparent responsiveness of system for end users	Number of seconds user gets response for an action initiated by the user	Measuring from the onset of user action until the user gets a response by the system (to the user interface he or she is using)

The user responsiveness KPI measures the number of seconds required for a user to receive a response for an action he/she initiated. We have performed a number of experiments related to this KPI for various types of user actions, mostly related to user authorization. These experiments identify two sources of delay a) delay introduced due to complex cryptographic computations and b) delays due to the block mining time. In Sections 3.2 and 3.3 we measure the time required for performing various cryptographic operations related to the components of the SOFIE architecture, whereas in Section 4.1 we measure the delay introduced by the public Ethereum blockchain during the user authorization process.

KPI	Goal	Description	Metric	Method of verification
10	System performance	Overall system performance reflecting the diverse needs and requirements of different use cases	Acceptable system performance for users and pilots	Qualitative evaluation of system metrics.

This KPI reflects the overall system performance. Different use cases can have diverse needs and requirements. The system metrics that we consider in the evaluation studies are discussed below.

Execution cost (GAS consumption): When using an Ethereum blockchain the gas consumption measures the execution cost that captures both the amount of processing and the storage used by a particular smart contract or smart contract function. The execution cost is

used in the performance evaluation studies reported in this report to quantify the cost for executing smart contracts and storing data on the Ethereum blockchain.

Delay: When the operations of a scenario include transactions on a blockchain such as Ethereum that implements Proof-of-Work consensus, the delay is due mainly to the block confirmation time. Similar to the execution cost, the delay involving blockchain transactions quantifies the performance cost for obtaining the advantages of public blockchains, in terms of decentralized trust, immutability, and availability.

In addition to the above traditional metrics, in some scenarios we identify quantitative metrics related to features provided by DLTs, such as immutability. Specifically, in scenarios where we periodically record hashes on a public ledger, the immutability of data that is produced from the time the last hash is recorded on a public ledger until the time the next hash is recorded cannot be ensured. The impact that the time interval or the amount of data produced between the consecutive recording of hashes on a public ledger is application specific.

The following Table 1 summarizes the project's current minimum achievements regarding the aforementioned KPIs.

Table 1. Current status of the SOFIE architecture KPIs

KPI	Metric	Current min. number
1 IoT operability	Number of IoT silos	5
2 IoT interoperability	Number of IoT silo pairs	-
3 Ledger use	Number of distributed ledgers	4
4 Interledger use	Number of distributed ledger pairs	1
5 Ledger independence	Number of BP samples classified into success or partial success	-
6 Privacy designed in as a fundamental requirement	Number of operational GDPR features referenced and supported	-
7 Device owner payments across ledgers	Number of ledger pairs supporting value transfer	1
8 Data sovereignty	Number of pilot use cases utilizing data owner data sovereignty features and data owner is from a different silo than the storage silo	4
9 User responsiveness	Number of seconds user gets response for an action initiated by the user	Various measurements have been performed
10 System performance	Acceptable system performance for users and pilots	Various measurements have been performed

3. Initial component evaluation

3.1 Interledger

The main purpose of the SOFIE interledger component is to enable transactions between actors and devices belonging to different (isolated) IoT platforms or silos. Each IoT silo either utilizes or is connected to one or more DLTs. The interledger component then enables interaction between these DLTs.

The interledger component can utilize different mechanisms depending on the specific scenario and its requirements. For example, interactions between a public and a permissioned ledger can use hashed time-lock contracts to cryptographically link transactions and events on the two ledgers. In such a scenario, the public ledger can record payments while the permissioned ledger can record authorization transactions and events. Alternatively, hashes of records stored on the permissioned ledger can be periodically recorded on the public ledger in order to provide a timestamped anchoring point, exploiting the wide-scale decentralized trust provided by the public ledger. Finally, interactions between a public or permissioned ledger and a ledger storing DID documents can focus on the resolution of DIDs to DID documents. The interledger functionality can be implemented in different entities, which include the entities that are interacting, a third party, or multiple third parties. In the latter case, some coordination between the entities may be necessary. A detailed survey of interledger approaches is contained in [Sir+19d].

Below we provide more details on the hash-lock and time-lock mechanisms, which are utilized in Section 4.1.

A **hash-lock** is a cryptographic lock that can be unlocked by revealing a secret whose hash is equal to the lock's value, *h*. Unlocking a hash-lock can be one of the conditions for performing a transaction or for executing a smart contract function. On a single blockchain, a hash-lock can be linked to an off-chain capability, e.g., message decryption, if the hash-lock secret is the secret key that can decrypt the message.

Hash-locks can be used on two or more blockchains, which support the same hash function, to link a transaction on one chain to a transaction on the other chain: if the two transactions have hash-locks with the same value, then unlocking one hash-lock would reveal the secret that unlocks the other; hence, the two transactions are cryptographically linked through a dependence relation. More generally, hash-locks combined with AND/OR logic operators can implement elaborate dependencies involving transactions on multiple chains.

Time-locks are blockchain locks that can be unlocked only after an interval has elapsed. This interval can be measured in absolute time or in the number of blocks mined after a specific block. One usage of time-locks are refunds: a user (payer) can make a deposit to a smart contract address. The smart contract can have a function, which typically also includes a hash-lock, for a second user to transfer the deposit to another account (the payee's account). However, if the second user never calls this function, then the first user's deposit could be locked indefinitely in the smart contract's account. To avoid this, the smart contract can also include a refund function that allows the first user to transfer the amount he/she deposited back to his/her account; however, this function can be called only after some time interval, which is the interval in which the second user must transfer the deposit from the smart contract account to the payee's account.

For the evaluation of our proof of concept we deployed an instance of the component that implements the interledger functionality and is connected to a local node (running Go-Ethereum) and to the Rinkeby or Ropsten public Ethereum testnet. We evaluated an authorization use case where some information has to be processed by a smart contract (more details are

provided in section 4.1). We measured the gas consumption and the delay in two cases: (a) when a single smart contract is used, and (b) when the whole functionality is split in two smart contracts, located in different blockchains, interacting through the interledger component. In the latter case, the first smart contract was responsible for authorizing users, whereas the second smart contract was responsible for handling payments. SOFIE's interledger component utilizing hash-locks and time-locks is responsible for ensuring the *atomicity* of the joint transaction.

Figures 1 and 2 below illustrate the obtained results. In the case where two blockchains are used, we measure the gas consumption and the delay of the public blockchain only. The execution cost (gas) shown in Figure 1 in the case of two blockchains considers only the cost for the public blockchain, since the other blockchain is a permissioned blockchain. The results show that utilizing two blockchains can reduce the total execution cost.

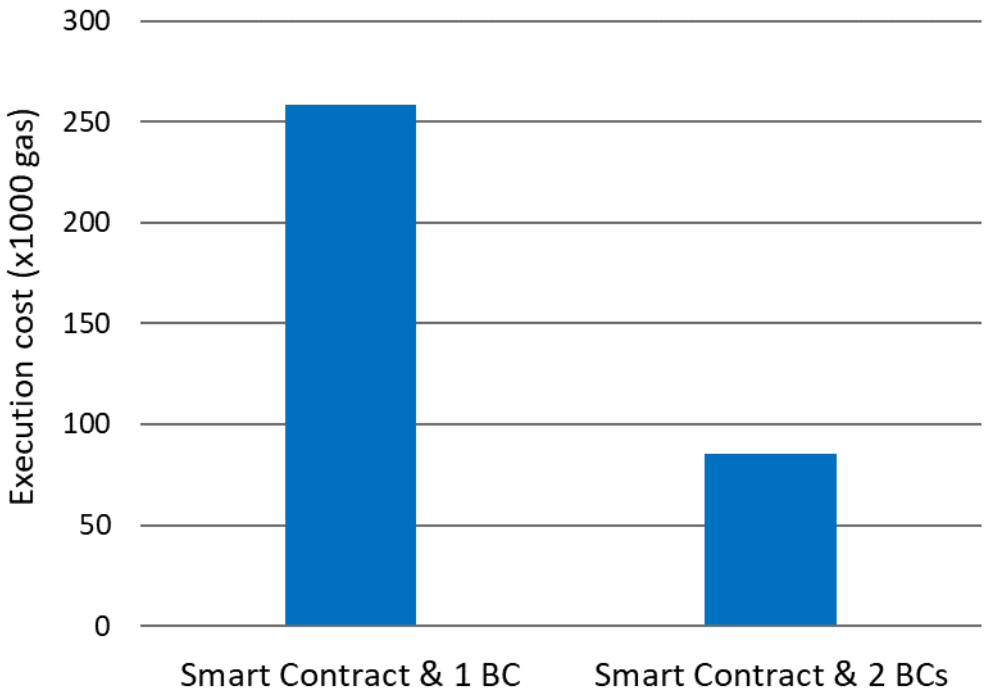


Figure 1. Gas cost when a smart contract and one blockchain are used, and when a smart contract and two blockchains are used

The delay shown in Figure 2 depends mainly on the block confirmation time. In the case of two blockchains, only the transactions on the public blockchain incur a high delay, due to the block confirmation time on such blockchains. For the specific scenario considered, which is discussed in more detail in Section 4.1, there are four transaction on the public blockchain in the case a single blockchain is used, whereas there are three transactions on the public blockchain when two blockchains are used. The results in Figure 2 illustrate the gains in terms of reduced delay when two blockchains are used.

Additional results when two blockchains are utilized, a public blockchain for recording only hashes of data and a private blockchain for storing data and/or executing smart contracts, are presented in Section 5.

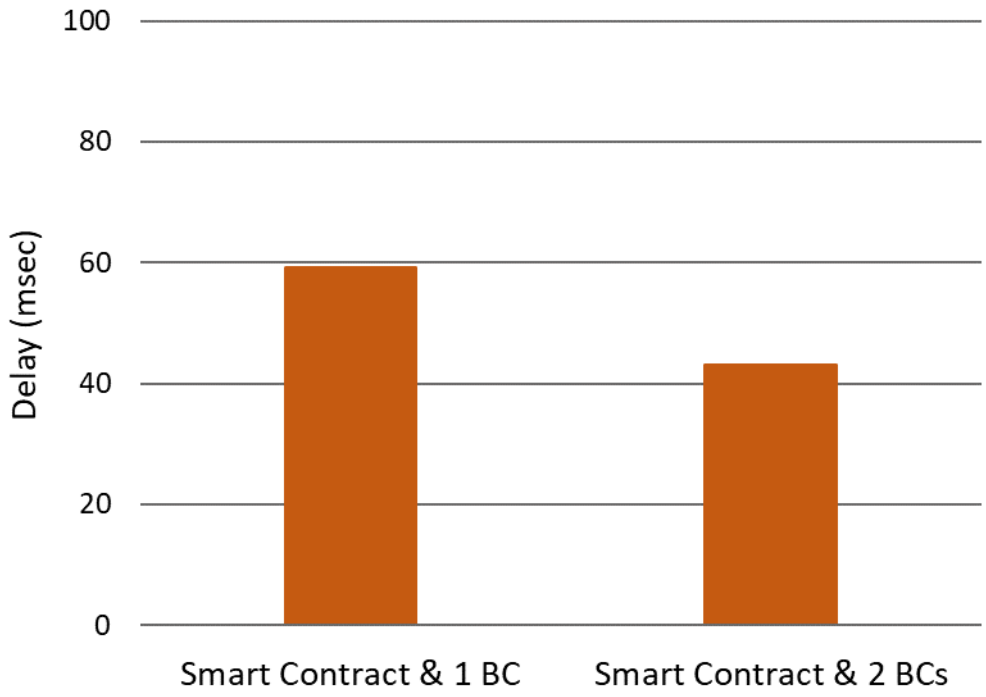


Figure 2. Delay when a smart contract and one blockchain are used and when a smart contract and two blockchains are used

3.2 Privacy and data sovereignty

User privacy is protected by using verifiable credentials and zero-knowledge proofs during the authorization process. Our privacy and data sovereignty component leverages Hyperledger Indy[8] and its SDK for this purpose. An Indy-based authorization process involves the following phases [Lag+19]:

Network setup. During this phase, which is executed only once, a "pool" of Indy nodes is created. The configuration file of this pool includes the "decentralized identifier" (DID) of a "Steward" node that is responsible for writing information to the Indy ledger. This DID is considered to be well-known. Stewards are the only entities that are allowed to record information in the ledger.

Trust Anchor setup. During this phase a DID known as the "Trust Anchor" is generated. A Trust Anchor is used for signing requests sent to the Steward, i.e., a Trust Anchor is responsible for managing DIDs and verifiable credentials (VCs) and it is the only entity that can communicate with a Steward. For each VC type, an anchor creates the "credential definition scheme" (i.e., a JSON-encoded description of the VC) and publishes it to the Hyperledger Indy ledger (through the Steward). These actions are executed once per VC type.

Client VC generation. Each client that has some trust relationship with an anchor may create a DID and send a "credential request" to the Anchor. The Anchor then responds with the corresponding credential, publishing at the same time (through the Steward) information to the ledger that can be used for verifying the issued credentials.

Client authorization. A client authorization by an authorization server (AS) is implemented using the following procedure:

[8] https://www.hyperledger.org/projects/hyperledger-indy

1) The client makes an HTTP request to the AS specifying a "grant type=DID" in the URL.
2) The AS generates a proof request, asking the client to prove that he holds the necessary VCs, issued by the appropriate Trust Anchor. This request includes, among other fields, a nonce.
3) The client generates a proof based on the credential and repeats the request by including now the proof in the payload.

Utilizing DIDs and VCs for authorization offers several advantages over how traditional X.509 certificates and Public Key Infrastructures (PKIs) are currently used. (In principle, all the improvements could also be implemented with them as well, but due to e.g. the significantly higher cost of X.509 certification and the number of certificates required, that would be highly impractical, if not impossible.) Traditional certificates are designed to be semi-permanent and human-readable: the user receives their certificate once and uses it in several situations. The certificate usually also contains much (unnecessary) information about the user, including their real identity, and the user reveals all attributes of the certificate when using it. This leads to a high cost of issuing the certificates (e.g., the user's real identity must be verified, usually by manual means) and serious privacy issues, since the user's activities can be easily tracked from service to service by multiple parties when using certificates.

The DIDs and VCs are designed to allow more fine-grained, machine-readable, and short-lived credentials, thus improving privacy and reducing the costs of issuing the credentials. While there are also proposals to allow X.509 certificates to support zero knowledge proofs [Del+16], Hyperledger Indy contains built-in support for zero knowledge proofs, which in turn further improve privacy by allowing the users to prove properties about themselves without disclosing their credentials.

Currently there exist multiple federated identity management solutions such as single sign-on systems and eduroam. However, they usually rely on their own non-standard identity management solutions and allow only certain members to participate in the first place (it is not easy to interconnect public organizations located in different countries, or for non-educational institutions to join eduroam). DIDs and VCs are open standards, allowing easy deployment and adoption, thereby allowing any pair of organizations to co-operate with each other with a low barrier of entry.

Data sovereignty is protected using access control. This SOFIE component supports access control delegation to an *Authentication Server*. This functionality is implemented through a smart contract that implements a hash lock and it can either simply relay messages or it can verify the relationship between a resource (and its owner) and the Authentication Server. In particular, the component implements three protocols [FSP18]:

Straw man protocol: This protocol is based on a smart contract that provides the following methods:

- request: Invoked by a client wishing to access a resource. It accepts as input a deposit and the resource identifier. The contract examines potential access control rules and determines if the client deposit suffices for accessing the requested resource. If all checks succeed, the contract generates the appropriate event, which is received by the corresponding authorization server.
- authorize: Invoked by an Authentication Server upon successful client authorization. It transfers the deposit that the client made (when she invoked the request method) to the service provider. Then, it creates an appropriate event used for notifying the client that authorization has been granted.

With this protocol, initially a client requests a protected resource from a resource server and the server responds with a token and the URI of a smart contract that protects the requested

resource. Then, the user invokes the request method of the smart contract. An event is broadcast and received by the appropriate AS which examines if the user can be authorized to access the requested resource. If this is true, the AS generates a session key *sk* (using the process described in [Fot+16]), encrypts it using the public key of the client, and invokes the authorize method of the smart contract. The smart contract examines if the AS that invoked the authorize method is allowed to do so. This check is implemented by simply examining if the public key of the entity that invoked that method is equal to the public key of the legitimate AS. The drawback of the straw man protocol is that the payment to the provider takes place without any checks. Note that with the solution described in [Fot+16], the user is able to perform certain verifications after trying to use the received *sk*. However, with the straw man protocol, these verifications can only be used for dispute resolution.

First protocol. The first protocol is an improvement to the straw man approach which allows an AS to verify that a client is communicating with a legitimate server. In order to achieve this goal, we extend the request method of the smart contract to include an additional field, i.e., the Hash based Message Authentication Code (HMAC) of the token generated using the session key *sk*. The value for this field is provided by the server, in its response to a client request. Now an AS, after generating the *sk*, calculates the same HMAC, and checks if the value of the latter calculation is equal to the value provided by the server. If this is true, then the server is considered legitimate

Second protocol. The second protocol extends the previous construction by enabling smart contracts to verify the relationship between a server and an AS. This functionality is achieved by having the client "challenge" the server during her request. The challenge used is a random number, which the server should obfuscate in a way that only an AS that shares a secret key with the server could read. The smart contract should therefore learn the challenge from the client and should expect the response from the AS. In order to "hide" the challenge we use a hash-lock (i.e., the client provides the hash of the challenge and the AS has to provide the corresponding pre-image).

A preliminary version of this component has been implemented using Ethereum smart contracts. This technology has some limitations that led us to certain design choices. In particular, although each user in Ethereum owns a public/private key pair, a smart contract has access only to each user's "address," i.e., the last 20 bytes of the hash of her public key. This means that users have to explicitly include their public keys with every smart contract function invocation; in our implementation, we added an additional field in each function which is used for storing the callee's public key. Furthermore, Ethereum keys are constructed using the secp256k1 elliptic curve; encrypting content using this curve can be cumbersome since specialized constructions, such as the elliptic curve integrated encryption scheme [Sho11], are required. For these reasons, we selected to not use Ethereum's keys in our constructions, using instead keys based on the Curve25519 elliptic curve [Ber06]. Curve25519 is a well-supported, fast curve which is ideal for key establishment, as it allows a user A to generate a symmetric encryption key that can be used for communicating with a user B, using only B's public key.

The main constructions of our smart contract, which is deployed in SOFIE's local testbed, are implemented in five functions: *requestS()*, *request1()*, *request2()*, each implementing the *request()* method for our three protocols (straw man, first construction, and second construction), and *authorize1()* and *authorize2()*, that implement the *authorize()* method for the first two protocols and for the last protocol, respectively. Table 2 below illustrates the cost, measured in Ethereum "gas," for invoking each function.

Table 2. Cost for invoking smart contract functions in the privacy and data sovereignty component

Function	**Cost** (measured in gas)
requestS()	123186
request1()	128218
request2()	253488
authorize1()	57950
authorize2()	63746

The endpoints for these measurements are implemented using JavaScript. Interactions with the Ethereum blockchain are implemented using the Ethereum JavaScript API, whereas cryptographic operations are implemented using the TweetNaCl library.[9]

Furthermore, this SOFIE component supports crypto token-based access control [Fot+19]. Many legacy access control mechanisms implement access control using "tokens" that indicate the capabilities of a client over a resource. However, token management, security, and semantics interpretation cannot be trivially implemented, especially in the context of the IoT. For this reason, in this component we leverage the capability of the Ethereum blockchain to support custom tokens and we implement an access control mechanism.

Ethereum has specified a "token standard" called ERC20 [VV15]. This standard defines some functions that a smart contract should implement in order to be treated as supporting a token (i.e., a new type of coin). Many popular Ethereum wallets can handle ERC20-based tokens. The core of this access control mechanism is built using two of these functions, namely balanceOf and transfer. The first function returns the token balance of a user. The second function can be invoked by a user A in order to transfer some tokens (he owns) to another user B.

The smart contract of this component provides access control as follows. Initially a resource owner that owns the smart contract assigns all tokens to himself. We refer to this user as the "owner." The owner then transfers at least one token to each authorized client. As a matter of fact, the number of tokens a client owns can be used as an indication of his role: the more tokens he owns, the more privileged his role. The contract owner can protect an operation by specifying the roles (i.e., the balance in custom tokens) of the authorized clients. Therefore, in the simplest case, an operation can be protected simply by having the smart contract function checking if the client that invokes it owns the necessary number of tokens (this check is trivially implemented using the balanceOf function).

SOFIE's token-based access control has some intriguing security properties. Firstly, tokens can only be used by their owners and token owners cannot transfer them to other users. Even if the blockchain keys of a user are compromised, our construction prevents token transfer (of course the stolen keys can be used for issuing transactions on behalf of the victim users). This is a significant advantage compared to traditional token-based access control mechanisms where, not only the corresponding tokens have to be secured, but also a token recipient should be able to verify the binding between the token and the user who sent it (i.e., additional mechanisms for detecting stolen tokens should be in place). In other words, the responsibility (and security) of the binding of tokens to token owners is performed by the blockchain, rather than being the responsibility of each user (which opens security issues). Furthermore, blockchains are

[9] https://tweetnacl.cr.yp.to. TweetNaCl is promoted as the world's first auditable high-security cryptographic library.

indelible, append-only, and tamper-proof logs, hence, in case of a security incident or in case of a dispute, they can provide undeniable auditing information. Moreover, our construction offers secure and effective revocation. Ethereum's mechanisms guarantee that only an owner can revoke tokens (provided, of course, that the owner's private key is secured), and that token revocation has immediate effect. Finally, since our construction is based on an established Ethereum standard, libraries and wallets that support it can be used for implementing client applications.

3.3 Identification, authentication, authorization

This SOFIE component supports the following Identification/Authentication mechanisms: URIs (e.g., Web of Things URIs) for identification coupled with digital certificates for authentication, usernames for identification bound to secret passwords for authentication, and *decentralized identifiers* associated with a *DID document*, stored in a blockchain, and used for authentication.

DIDs are a new identification mechanism; their properties have not been thoroughly studied, yet. In the following we discuss the feasibility of deploying DIDs in IoT devices [Kor+19].

In order to utilize distributed identifiers (and verifiable credentials) the IoT device should have:

- sufficient *performance* for cryptographic operations,
- a sufficient amount of *energy* to perform the required operations,
- non-volatile *storage* space to store the code and cryptographic keys, and
- a sufficient *entropy* source to generate random cryptographic keys.

From a *performance* point of view, the most limiting factor is the performance of public key cryptographic operations, namely key generation, signature generation, and signature verification. Presently, most DID solutions utilize elliptic curve cryptography (ECC)—as opposed to e.g. RSA—due to its significantly smaller key size and the fact that all three operations are relatively fast and take roughly a similar amount of time (with RSA, key generation can take orders of magnitude longer than signature generation or verification operations). Lately, there has been much research about the performance of ECC on constrained devices. Past research [HS13] shows that operations with the common Ed25519 [Ber+12] signature scheme using a standard public domain NaCl[10] library on a popular 8-bit AVR microcontroller take about 23 million and 32 million cycles for signature generation and verification, respectively. Newer optimizations [Dül+15] reduce the cost of the elliptic curve point multiplication on the comparable Curve25519 from 23 million to 14 million cycles on a 8-bit device, while on a 32-bit low cost ARM Cortex-M0 core, the point multiplication uses only about 3.6 million cycles. Therefore, Cortex-M0 devices, which are available for less than half a dollar in large quantities and run at up to 48MHz, can perform up to 13 ECC operations per second. Since modern 8-bit microcontrollers usually run at 16-32MHz, even such extremely constrained devices are able to perform all the necessary cryptographic operations for DID usage within a few seconds, which is acceptable performance for most IoT use cases. In cases where the device is even more constrained, a hardware accelerator for cryptographic functions may be used.

Finally, while the DID itself is just a simple string and easy to process as such, the related technology of Verifiable Credentials (VC) are usually expressed in JSON format. There might be cases where the device includes a cryptographic accelerator, but is otherwise extremely constrained and, therefore, unable to parse JSON. In that case, VCs can be encoded in a more machine-friendly binary format such as BSON[11], as the VC specifications do not mandate usage

[10] NaCl: Networking and Cryptography library. Available at: https://nacl.cr.yp.to/
[11] "BSON (Binary JSON) Serialization" Available at: http://bsonspec.org/

of any specific encoding format. And since DIDs only utilize the ledgers for a few operations, network performance is normally not an issue even with constrained devices.

IoT devices often have only limited *energy* available, which has to be taken into account when designing security and privacy solutions. An optimized ECC implementation running on a Cortex-M0 using the slightly weaker 233-bit sect233k1 curve uses only 20-34*µJ* of energy for the elliptic curve point multiplication [Cle+14]. Such energy consumption is very low compared to the energy consumption of the wireless transmission or the overall consumption of the IoT device, which can easily consume hundreds of *µWs or more.*[12] Even with very simple 8-bit devices, the energy consumption of ECC operations is reasonable, around 20*mJ* per point multiplication, while an optimized hardware accelerator for cryptographic operations provides even lower energy consumption, in the order of *µJ*s per ECC point multiplication [AJM14]. So, while there are some cases where an extremely constrained (say) sensing device that only sends data very infrequently is unable to utilize public-key cryptography due to energy consumption concerns, in most IoT applications energy consumption does not prevent usage of public-key cryptography and therefore DIDs. In many IoT applications, such as vending machines, devices may even be constrained in terms of e.g. processing power, while having plenty of energy available.

The *storage* of cryptographic keys should also, in most cases, not be an issue, as long as non-volatile storage is available. ECC offers compact keys and signatures, with sizes of 256 bits and 512 bits respectively for the security level equivalent to 128-bit symmetric encryption. Hence, a public/private key pair would use only 64 bytes of space, so even a few kilobytes of storage space is sufficient to store multiple keys or credentials. However, in some applications, storing the keys on the device can present an unacceptable security risk of key leakage, unless the device utilizes e.g. a trusted platform module (TPM). In such situations, using, e.g., a proxy solution that can act as a guardian for the keys may be a safer solution.

Finally, generating secure cryptographic keys requires a sufficient *entropy* source. This can be a challenge in the IoT environment, where the devices often have a limited amount of input sources available for entropy. In that case, the entropy can be provided by a hardware-based random number generator (RNG) that is embedded in the device's processor [SMS07]. If a hardware-based RNG is not feasible, there are several alternatives. The device's private key can be generated by another party, e.g. by the manufacturer already at the factory or by the device owner when the device is taken into use. Having the manufacturer generate the keys for all devices of that type obviously poses a security risk, thus having the owners generate the keys is a better solution. An even better solution is if the owner (or, more specifically, the owner's app used to initialize the device) can act as an additional source of entropy during the initialization process, thus letting only the device be aware of the actual key generated.

As a proof of concept that even quite constrained devices have sufficient performance to deploy DIDs, the current uPort implementation was tested on a first-generation Raspberry Pi (700 MHz BCM2835 CPU, 512 MB of RAM, released in 2012), running a Raspbian GNU/Linux 9 distribution and using the latest code of the Node.js 'ethr-did'[13] package of 26 September 2018. 100 operations were running and timed in each test and the tests were repeated three times. The key pair and signature generation both took 126 ms when run separately, while generating the key pair and using it immediately to generate the signature took 230 ms overall as summarized in Table 3, below.

[12] The total average power consumption of a simple wireless sensor utilizing Bluetooth Low Energy protocol is around 200-1000*µW.*

[13] uPort, Ethr-DID Library, available at: https://github.com/uport-project/ethr-did

Table 3. Performance of uPort Node.js implementation on first-generation Raspberry Pi

Function	**Time** (ms)
Key pair generation	126
Signature generation	126
Key pair + signature generation	230

It is worth mentioning that the original Raspberry Pi is a very slow device by modern standards. It contains a single-core ARM CPU utilizing the old ARMv6 instruction set. Newer ARMv7 or ARMv8 devices would offer significantly higher per-clock performance (along with more cores and higher clock speed), and devices with hardware acceleration for cryptographic operations would perform orders of magnitude faster. Also, the JavaScript and Node.js environment used by the current uPort implementation are relatively slow solutions. A DID implementation written in C or other lower-level language would perform faster and require significantly less memory and storage space. Therefore, the setup where the tests were run can be considered a worst-case scenario. Despite all of these limitations, already the current uPort Node.js implementation runs on the first-generation Raspberry Pi with an acceptable performance, as the signature generation takes 126 ms, so the whole process of e.g. parsing and verifying[14] a credential should take well below a second.

Overall, most devices with a 32-bit CPU can utilize DIDs with the currently available software, while more constrained devices (e.g., 8-bit microcontrollers) would be able to use DIDs with an optimized software implementation.

The primary authorization mechanism used by this component is OAuth 2.0. This component supports two models which involve a different level of OAuth 2.0 integration with blockchains and smart contracts, supporting different tradeoffs in terms of privacy, delay, and cost [Sir+19b]:

- Linking authorization grants to blockchain payments and recording authorization information on the blockchain.
- Using a smart contract to handle authorization requests and encode authorization policies.

With the first model, the initial communication between the client and the Authorization Server (AS) occurs as in the normal OAuth 2.0 framework. However, instead of the AS providing the client with authorization credentials after consent is given by the resource owner, the authorization credentials are disclosed after the payment for resource access is recorded on the blockchain. Hence, the resource owner does not need to be online to provide consent, as in the case of the normal OAuth 2.0 procedure.

In the second model, a smart contract is used to transparently record prices and other authorization policies defined by the resource owner, who is also the owner of the smart contract. Examples of such policies include permitting resource access to specific clients, determined by their public keys on the blockchain, and dependence of access authorization on IoT events that are recorded on the blockchain. Whereas in the previous model the client and the AS communicated directly, in this model the interaction is through the smart contract. The smart contract code is executed by all blockchain nodes, providing a secure and reliable execution environment; this provides higher protection against DoS attacks, compared to the first model where resource access requests are sent directly to the AS. An additional advantage

[14] Which in ECC is 2-3 times slower compared to signature generation, depending on the implementation.

of allowing a smart contract to handle resource authorization requests is that the smart contract can securely bind the protected resource with the AS responsible for handling authorization requests.

A preliminary version of this component has been deployed in a local Ethereum node running Go-Ethereum, which is connected to the Rinkeby public Ethereum testnet. The local node runs on a computer with a 4 core CPU at 3.40 GHz, 16 GB RAM, and 64 bit Ubuntu. Smart contracts are written in Solidity with the Remix Web-based editor. The authorization server is based on a PHP implementation of the OAuth 2.0 framework. The client uses *Web3.js* to interact with the Rinkeby blockchain. The following table shows that the second model requires more than three times the amount of gas, hence more than three times the amount of EVM (Ethereum Virtual Machine) resources, compared to the first model; this quantifies the tradeoff between the advantages of the second model and its higher cost. Regarding the delay, the first model involves three blockchain transactions whereas the second model four. Since the total delay is expected to depend mainly on the block mining time, the second model is expected to have a 33% higher delay for responding to authorization requests.

Table 4. Gas cost for each OAuth2.0 integration model

Model	**Cost** (measured in gas)
Model 1	102476
Model 2	366277

4. IoT resource access detailed evaluation

In this section we present three solutions for IoT resource access that combine OAuth2.0 with blockchains. This section goes to much more depth and considers many more alternatives and their tradeoffs than what we have seen in Section 3 that focuses on SOFIE components. Other sections of this deliverable do not intimately depend on it, but it shows that much more detailed design and evaluation is possible and can uncover many useful alternatives in the IoT world.

OAuth 2.0 is a framework for delegating authorization to access a protected resource [Har+12]. It enables a third-party application (client) to obtain access with specific permissions to a resource, with the consent of the resource owner. Access to the resource is achieved through access tokens, created by an authorization server. The specific format of the access tokens is opaque to the clients and to OAuth 2.0. The authorization consent by the resource owner is provided after the owner is authenticated; however, the authentication procedure is not part of OAuth 2.0. Authorization is provided for different levels of access, such as read and write/modify, which are termed scopes, and for a specific time interval. The OAuth 2.0 authorization flows can involve intermediate messages exchanged before the access token is provided by the authorization server. The details of the authorization flow do not impact the general approach of the proposed models, hence in our discussion we only consider the initial client request and the authorization server's response containing the access token.

One type of access token is the bearer token. Bearer tokens allow the holder (bearer) of the token, independently of its identity, to access the protected resource. OAuth 2.0 assumes secure communication between the different entities. Moreover, it assumes that the protected resource is always connected to the Internet, hence it can communicate with the authorization server to check the validity and scope of the access tokens presented by clients requesting resource access. Both these requirements are not always achievable in constrained environments [Sei+16].

JSON Web Token (JWT) is an open standard that defines a compact format to transmit claims between parties as a JSON object [JBS15a]. JWTs can use the JSON Web Signature (JWS) structure to digitally sign or integrity protect claims with a Message Authentication Code (MAC) [JBS15b]. Hence, unlike simple bearer tokens, JWT/JWS tokens are self-contained, i.e., they include all the necessary information for the protected resource to verify their integrity without communicating with the authorization server. Of course, this requires that during its initialization phase the protected resource is cryptographically bound with the authorization server.

In constrained environments, in addition to intermittent or no connectivity, the communication between the client and the protected resource is not always secure, hence transmitting bearer tokens or even self-contained JWTs over such (insecure) links can allow other parties to obtain them through eavesdropping. For this reason, in constrained environments Proof-of-Possession (PoP) tokens are used [Sei+19]. PoP tokens include a normal access token, such as a JWT/JWS, and a PoP key [JBT16]. Access to the protected resource is not possible solely with the access token; the PoP key is also necessary. Hence, the PoP key must be kept secret and not transmitted in cleartext over insecure links. Finally, a more efficient encoding of access tokens based on CBOR (Concise Binary Object Representation) is proposed to reduce the amount of data transferred [Sei+19].

The advantages from combining authorization based on frameworks such as OAuth 2.0 with blockchains and smart contracts are the following:

- Blockchains can immutably record hashes of the information exchanged during authorization and cryptographically link authorization grants to payments and other IoT events recorded on the blockchain. These records serve as indisputable receipts in the case of disagreement.

- Smart contracts can encode authorization policies in an immutable and transparent manner. Policies can depend on payments as well as on other IoT events that are recorded on the same or on different blockchains.
- Smart contracts run on all nodes of a blockchain. Hence, sending resource access requests to smart contracts can protect against DoS attacks that involve a very high resource request rate, since requests are not handled by one node, which would be a single point of failure.

The solutions presented below leverage the hash-lock and time-lock mechanisms presented in Section 3.1.

Contracts that include both hash and time-locks are referred to as hashed time-lock contracts (HTLCs).[15] HTLCs have been used for atomic cross-chain trading (atomic swaps),[16] [But16] and for off-chain transactions between trustless parties [PD16]. HTLCs can be implemented in blockchains with simple scripting capabilities, such as the Bitcoin blockchain, without requiring the advanced functionality of smart contracts. Smart contracts do not increase the capabilities of interledger mechanisms based on hash and time-locks, but increase the intra-ledger functionality. We investigate these features for decentralized authorization to constrained IoT devices.

4.1 Interledger and decentralized authorization

In this section, we present four models that we have developed [Sir+19a] and that allow different tradeoffs in terms of cost, delay, complexity, and privacy:

(1) Linking authorization grants to blockchain payments
(2) Smart contract handling of authorization requests
(3) Smart contract and two blockchains for authorization and payment with interledger mechanisms
(4) Decentralized authorization with multiple Authorization Servers

The first two models are our baseline scenarios: in the first, only hashes of authorization information are immutably recorded on the blockchain and smart contracts are not used, whereas the second model utilizes a smart contract, but on a single (public) blockchain. The third model exploits two blockchains whose transactions are securely linked using interledger mechanisms and quantifies the significant cost reduction that can be achieved by moving smart contract authorization functionality to a permissioned or private blockchain. The fourth model focuses on decentralized authorization for constrained IoT devices utilizing two blockchains with interledger mechanisms.

In all the models presented below, the client sends a resource access request to the URL of the AS (model 1) or to the address of the smart contract responsible for handling access to the IoT device (models 2, 3, and 4). The URL or smart contract address can be obtained by the client sending a query to the IoT device or, e.g., reading a QR code on it. However, this approach cannot ensure that the legitimate URL or smart contract address is provided by the IoT device. This can be ensured if the client uses a registry service that resides on the blockchain and contains a binding between the IoT device's URI and the URL of the AS or the smart contract address handling authorization, or by including this information in Decentralized Identifier (DID) documents [Ree19].

Finally, in all models we assume that the client, the resource owner, and the ASes have an account (public/private key pair) on the blockchain (on both the authorization and the payment blockchains for models 3 and 4).

[15] Hash Time-Lock Contracts (HTLC). Available at: https://en.bitcoin.it/wiki/Hashed_Timelock_Contracts
[16] Atomic cross-chain trading. Available at: https://en.bitcoin.it/wiki/Atomic_cross-chain_trading

A. Model #1: Linking authorization grants to payments and recording authorization information on the blockchain

With this model the initial communication between the client and the Authorization Server (AS) follows the normal authorization message exchange, such as OAuth 2.0 (Figure 3). Specifically, in step 1 the client requests resource access from the AS. The AS generates a random PoP key which it sends to the client[17] together with its encryption with the secret key[18] K_{Thing} shared by the Thing (IoT device) and the AS; the client will later use the PoP key to establish a secure communication link with the Thing. Also, the AS sends to the client the access token encrypted with a secret *s*, i.e., $E_s(token)$, the hash $h = Hash(s)$ of the secret *s*, and the price for the requested level of resource access. The secret *s* is a secret randomly generated by the AS and is required for the client to decrypt $E_s(token)$ and obtain the access token; the AS will reveal the secret *s* once it confirms that the payment for resource access has been committed on the blockchain. Communicating the price from the AS to the client allows different levels of resource access, encoded in the access token's scopes, to correspond to different prices.

In step 3, two hashes are submitted to the blockchain: the first is the hash of the token that the AS will reveal to the client once payment has been confirmed. The second is the hash of three items: $E_{KThing}(PoP)$, the PoP key, and $E_s(token)$; the second hash serves as proof of the information that is communicated using OAuth between the AS and the client. Note that the above authorization exchange does not ensure that the access token the client obtains from the AS indeed allows access to the Thing.

Also in step 3, a hashed time-lock payment is initiated on the blockchain, which allows the client to deposit an amount equal to the requested price (step 4). This amount will be transferred to the resource owner's account if the secret *s* (hash-lock) is submitted to the contract by the AS (step 5) within some time interval. If the time interval is exceeded, then the client can request a refund of the amount it deposited. Once the secret *s* is revealed, the client can get *s* from the blockchain (step 6) and decrypt $E_s(token)$ and thus obtain the access token. At this point, the client has all the necessary information to request access from the Thing, using normal OAuth 2.0 with the modifications from the ACE framework.[19]

[17] The communication link between the client and the AS is secured, hence the PoP key cannot be obtained through eavesdropping.
[18] The secret key that the Thing and AS share is established during the provisioning (or commissioning) phase, when the Thing is bound to the AS.
[19] https://datatracker.ietf.org/wg/ace

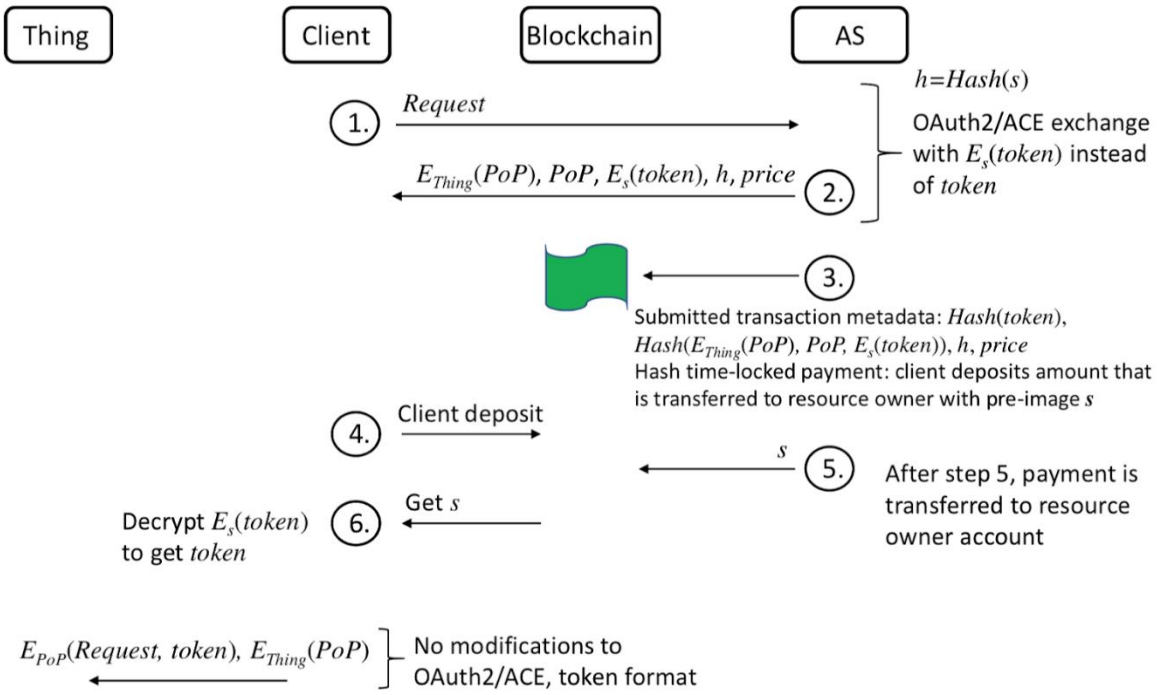


Figure 3. Model 1: Authorization grants are linked to blockchain payments and the hashes of the authorization information exchanged are recorded on the blockchain for verification in case of disputes

B. Model #2: Smart contract handling authorization requests

In the second model, a smart contract is used to transparently record prices and other authorization policies defined by the resource owner, who is also the owner of the smart contract. Examples of such policies include permitting resource access to specific clients, determined by their public/private key pairs on the blockchain, and adding a dependency of the access authorization on IoT events that are recorded on the blockchain.

Whereas in the previous model the client and the AS interacted directly, in this model the interaction is through the smart contract; this is similar to the model shown in Figure 4, but using a single blockchain for both authorization and payment. The smart contract code is executed by all blockchain nodes, providing a secure and reliable execution environment; this provides higher protection against DoS attacks.

C. Model #3: Smart contract and two blockchains with interledger mechanisms

In this model the smart contract handling authorization requests and encoding policies is located on an authorization blockchain, while payments for access are performed on a payment blockchain, as shown in Figure 4. Depending on whether the authorization chain is public or a permissioned blockchain, different tradeoffs in transaction cost, delay, and privacy can be realized.

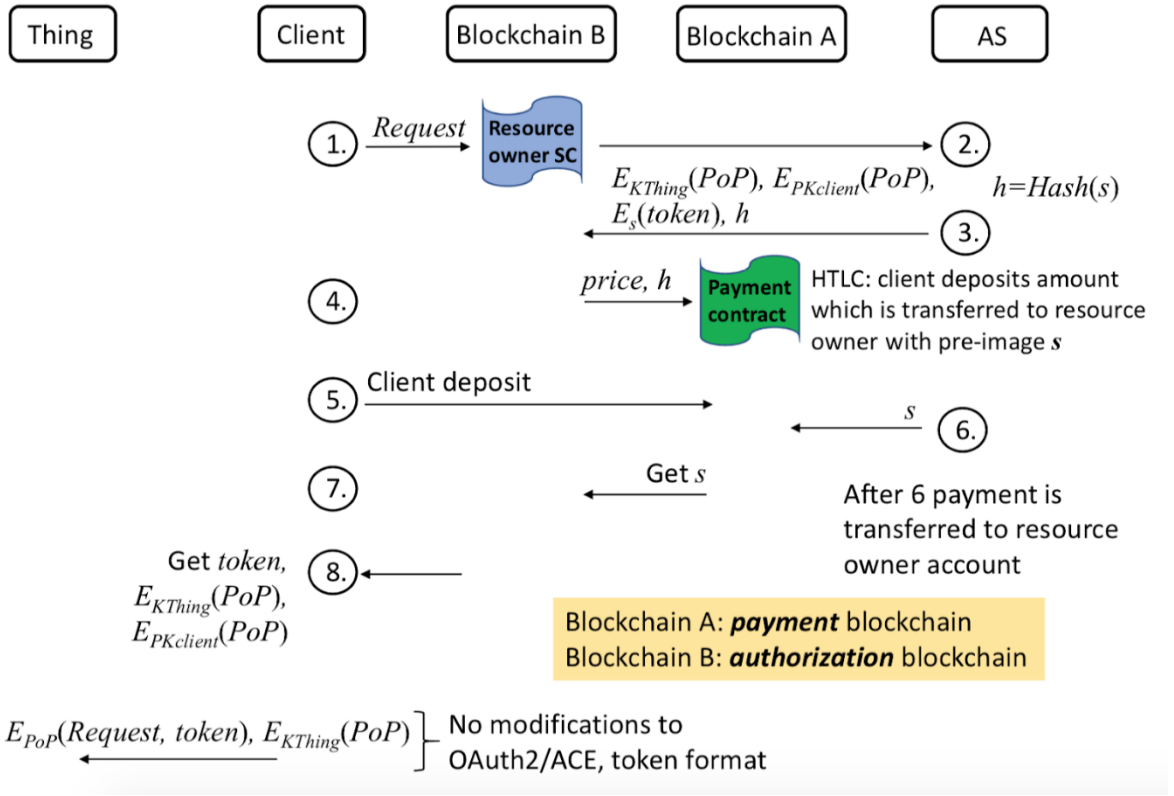


Figure 4. Model 3: Smart contract and two blockchains with interledger mechanisms

A hashed time-lock payment is initiated on the blockchain, where the client can deposit an amount corresponding to the resource access price. The amount will be transferred to the resource owner's account if the secret *s* is revealed. Once revealed, the secret *s* can be submitted to the smart contract on the authorization blockchain, which serves as a record on this blockchain that the payment was successfully performed. The client can obtain the secret *s* from the authorization blockchain together with the other necessary authorization information to access the protected resource.

One issue with the above model is how the payment contract on the payment chain is triggered by the resource owner's smart contract residing on the authorization chain. One option is to have an interledger gateway read the price and hash *h* from the authorization chain and submit it to the payment chain to initiate a payment (Step 4 in Figure 4), and later read the secret s submitted by the AS on the payment chain and record it on the authorization chain (Step 7); the interledger gateway can receive a fee for performing this function. Another alternative is to have this function performed by the AS, or the client.

D. Model #4: Decentralized authorization with multiple ASes

The authorization functionality cannot all be moved onto the blockchain since it involves processing secret information: keys are needed to produce token signatures and keys are shared with the Thing. Performing the authorization functions redundantly in the nodes of a private blockchain would provide a higher level of resilience to node failures compared to a single AS, but that would result in reduced security since compromising a single blockchain node would lead to secret keys being disclosed.

Rather than moving all the authorization functionality to the blockchain, we propose an alternative approach for decentralized authorization that ensures security and provides fault tolerance if a number of ASes are faulty or misbehave. Let *n* be the number of ASes that are collectively responsible for providing authorization. Each AS *i* shares a different secret key,

K_{Thingi}, with the protected resource (Thing). Authorized access to the Thing requires tokens from *m* out of *n* servers. The policy specifying the required number of ASes is defined in the smart contract and is also known to the Thing. Fault tolerance is provided by having *n* ASes which can respond to requests, but requiring only $m < n$ for authorization to proceed. Compared to having a single AS, the proposed scheme provides higher security since *m* ASes need to agree in order for the client to access the protected resource.

There are two alternatives for how *m* servers are selected to provide authorization. In the first, the smart contract selects *m* specific servers; this requires that the smart contract maintains a list of ASes. The list can be updated with information such as the time each AS last responded to an authorization request. Such information allows the smart contract to prioritize ASes in order to select those that respond quickly, hence avoiding ASes that have a high delay or are faulty.[20] Our evaluation considers this first alternative.

In the second alternative, the smart contract simply allows all ASes to respond to the authorization request, and selects the first *m* ASes that respond. With this approach the smart contract does not need to maintain the list of ASes. However, there is a possibility that the smart contract receives more than *m* responses. This depends on the duration for mining a block on the blockchain (in the case of public blockchains with Proof-of-Work consensus), or for obtaining consensus to add it to the blockchain (in the case of permissioned blockchains). In public blockchains, these responses can incur a gas cost independent of whether the ASes that gave the response were among the *m* ASes to provide decentralized authorization.

In response to the client's authorization request, each AS sends a different PoP key PoP_i, encrypted with the Thing's secret key and the client's public key, and an access token with a MAC tag to ensure its integrity (Figure 5). The client thus obtains *m* different PoP keys, which it XORs to obtain the secret PoP key that will be used to establish a secure communication link with the Thing. These *m* PoP keys, encrypted with the Thing's key K_{Thingi} that it shares with each of the *m* ASes, are also sent to the Thing. Hence, if the Thing performs the same XOR function on the m PoP keys, it will obtain the same PoP key as the client.

In order to reduce the amount of data transmitted to constrained devices we propose two schemes for reducing the authorization information the client sends to the Thing: (a) aggregate MAC tags and (b) transmission of common token fields once. With aggregate MAC tags [KL08], the client does not send to the Thing the token payloads received from the *m* ASes, but only one aggregate MAC tag that is computed by taking the XOR of the *m* MAC tags the client receives from the *m* ASes. With the second optimization, the client sends the token fields that are common to all ASes only once (these correspond to $token_1, \ldots token_m$ in Figure 5). The common token fields include the subject (Thing) the token refers to, the scope of access, the token creation time, the token validity time, and the token type. The fields which are different include the AS and token ID fields.

[20] Detection of misbehaving ASes that generate incorrect tokens is also possible.

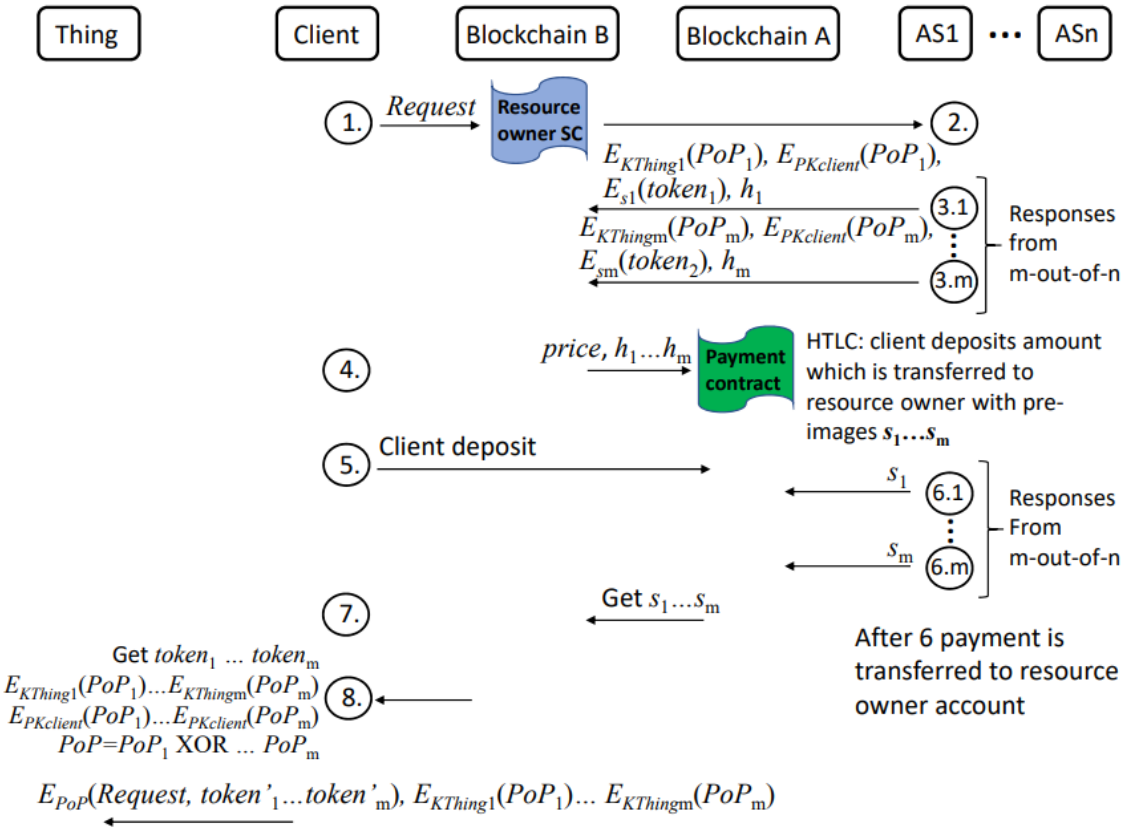


Figure 5. Model 4: Decentralized authorization; each authorization grant requires m out of n AS responses

4.1.1 Evaluation

For evaluation of these four models we deployed a local node running Go-Ethereum connected to the Rinkeby public Ethereum testnet[21] and a node running Parity connected to the Ropsten testnet.[22] The AS was based on a PHP implementation of the OAuth 2.0 framework,[23] extended to support CWT's CBOR encoding.[24] The client used Web3.js to interact with the blockchain.

[21] https://www.rinkeby.io/
[22] https://www.ropsten.etherscan.io/
[23] https://github.com/bshaffer/oauth2-server-php
[24] https://github.com/2tvenom/CBOREncode

Table 5. Execution cost (gas) and delay (Rinkeby) of decentralized authorization models

Model	Gas	Delay (s) (and 95% conf. int.)
Hashes of auth. Inform.-Fig. 3	102489	43.2 (43.2, 44.1)
SC & 1 BC	258166	59.3 (57.6, 61.1)
SC & 2 BCs-Fig. 4	85682	43.0 (39.8, 46.2)
Dec-Auth 2-of-4 & 1 BC	1440540	60.5 (54.4, 67.3)
Dec-Auth 2-of-4 & 2 BCs-Fig. 5	332569	42.1 (39.4, 44.8)
Dec-Auth 3-of-4 & 1 BC	2124249	63.7 (57.2, 70.2)
Dec-Auth 3-of-4 & 2 BCs-Fig.5	447940	44.7 (39.6, 49.9)

In Table 5, the smart contract & one blockchain and the decentralized & one blockchain models are equivalent to Figure 4 (smart contract & two blockchains) and Figure 5 (decentralized authorization & two blockchains) with one blockchain for both authorization and payment. For the results with two blockchains, we use the public blockchain (Rinkeby or Ropsten) as the payment chain and a private Ethereum network as the authorization chain. The results shown include the gas and delay due to transactions on the public blockchain only.

The graphical representation of the results when only hashes are recorded on the blockchain, when a smart contract and one blockchain are used to handle authorization requests, and when two blockchains are used, one for authorization and one for payments, are shown in Figure 6, below.

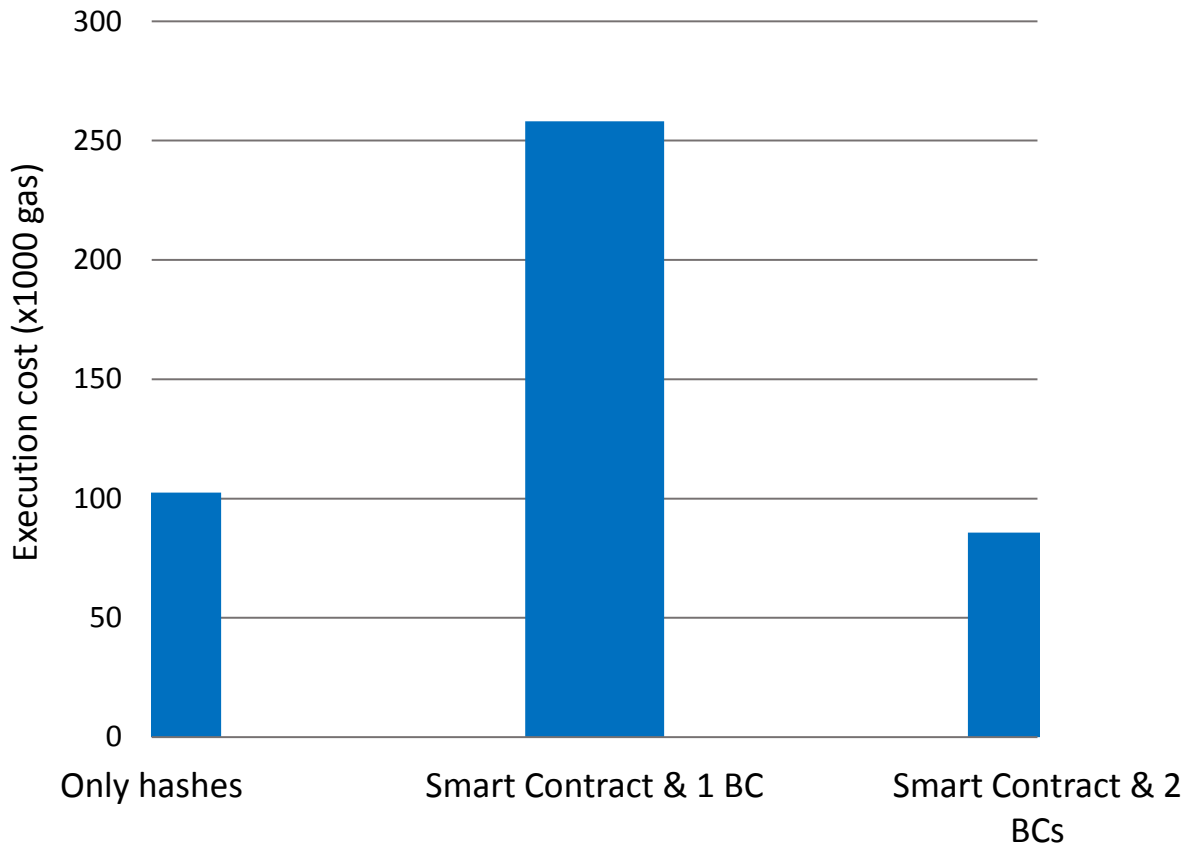


Figure 6. Execution cost when only hashes are recorded on the blockchain, when a smart contract and 1 blockchain are used and when a smart contract and 2 blockchains are used

The results would be similar if we used another technology, e.g., Hyperledger Fabric, as the authorization chain.

a) Gas: The second column in Table 5 shows that the execution cost of a smart contract on the Ethereum Virtual Machine (gas) on the public Rinkeby testnet is significantly higher compared to simply recording hashes (first line in Table 5). However, when the smart contract authorization functionality is moved to a private blockchain (models with 2 BCs in Table 5), then the execution cost is significantly reduced: For 1, 2, and 3 ASes the execution cost when two blockchains are used is 33.2%, 23.1%, and 21.1% of the execution cost when a single blockchain is used.

b) Delay: The delay is due mainly to the block mining time. The smart contract model with one blockchain has four transactions, while the model that records only hashes has three; hence, the delay for the smart contract model is expected to be 33% higher; this agrees with the results in the third column of Table 5, according to which the smart contract model with one blockchain has average delay 59.3 s, which is 37.3% higher than the delay when only hashes are recorded, 43.2 s (also shown is the confidence interval from 20 runs). Table 5 quantifies the reduced delay when a public chain is combined with a private chain: e.g., the 2 out of 4 decentralized model with two chains has average delay 42.1 s, which is 30.4% smaller than the delay with one chain, 60.5 s. Table 5 also shows that for both one and two blockchains, the average delay is not significantly influenced by the number of ASes. Also, for two blockchains the average delay is close to the delay when only hashes are recorded.

The graphical representation of the results for the delay when only hashes are recorded on the blockchain, when a smart contract and one blockchain are used to handle authorization requests, and when two blockchains are used, one for authorization and one for payments, are

shown in Figure 7, below. Note that the execution cost when only hashes are recorded is higher than the cost when a smart contract and two blockchains are used because in the second case the cost involves only the payment transaction and no data/hashes are recorded on the payment (public) blockchain.

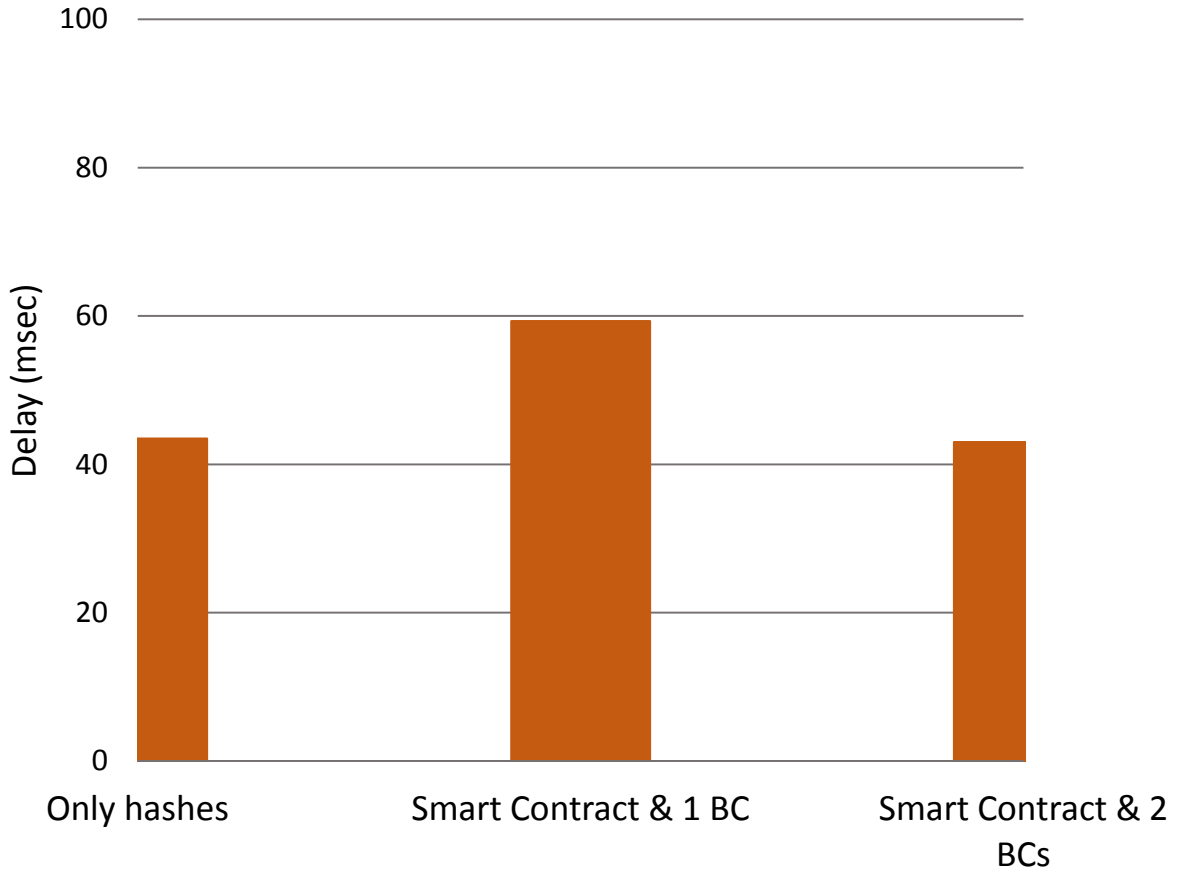


Figure 7. Delay when only hashes are recorded on the blockchain, when a smart contract and one blockchain are used, and when a smart contract and two blockchains are used

Table 6 shows that the delays and confidence intervals for the Ropsten testnet are higher than the Rinkeby testnet. We attribute this difference to the fact that Rinkeby uses the Proof-of-Authority (PoA) for distributed consensus, while Ropsten uses Proof-of-Work, as the Ethereum mainnet. For both the Rinkeby and Ropsten testnets, the delay depends on the gas price that is given when a transaction is submitted.

c) Reduction of the data the client sends to the Thing: Utilizing CWT encoding instead of JWT reduces the size of tokens from 310 to 122 bytes. For the smart contract with one blockchain model, the transaction cost is higher by approximately 17% compared to the cost shown in Table 5.

Table 6. Delay (Ropsten) of decentralized authorization models

Model	Delay (s) (and 95% conf. int.)
Hashes of auth. Inform.-Fig. 3	53.2 (40.3. 66.1)
SC & 1 BC	64.4 (52.3, 77.1)
SC & 2 BCs-Fig. 4	57.8 (46.0, 69.7)
Dec-Auth 2-of-4 & 1 BC	76.7 (61.5, 92.0)
Dec-Auth 2-of-4 & 2 BCs-Fig.5	49.1 (37.7, 60.5)
Dec-Auth 3-of-4 & 1 BC	77.5 (60.5, 94.6)
Dec-Auth 3-of-4 & 2 BCs-Fig.5	52.2 (42.6, 61.7)

The proposed optimizations further reduce the amount of data that the client needs to send to the Thing. For decentralized authorization with three ASes and without the optimizations, the client sends 366 bytes (3 × 122 bytes) for three tokens and 96 bytes (3 × 32 bytes) for three PoP keys, or a total of 462 bytes. With aggregate MACs, the client sends one aggregate MAC instead of three, i.e. 64 bytes less, hence a total of 398 bytes, which is a 13.9% reduction. The optimization where common token fields are sent once results in 84 bytes less, an 18.2% reduction, reducing the size to 314 bytes. The two optimizations together give a 32.0% reduction of the number of bytes the client needs to send to the Thing. The reduction for more ASes would be higher.

4.2 Authorization and TEE

We now present an IoT access model that leverages Trusted Execution Environments (TEEs). TEEs provide a secure environment for executing code and storing data. A TEE runs in isolation and in parallel to the normal (or “rich”) operating system, ensuring the confidentiality and integrity of code and data. However, because a TEE runs on a single device, it cannot provide high availability or decentralized trust. Combining blockchains and TEEs can combine the gains of both: decentralized trust and high availability from blockchains and privacy and trust when interacting with the real world through TEEs.

The high-level architecture of the proposed model is shown in Figure 8. Authorization for IoT resources is outsourced to an Authorization Server (AS), which can provide authorization for multiple IoT resources. Resource access is provided by a device with a TEE, which supports integrity and confidentiality. Depending on the specific type of TEE technology, different restrictions can exist. For example, ARM’s TrustZone supports a single secure enclave, whereas Intel’s SGX can support multiple enclaves.

Figure 8 shows that the client device and the AS interact with the blockchain, whereas the IoT resource does not have continuous network connectivity. The client accesses the IoT resource directly using device-to-device communication. Moreover, because the device-to-device link is insecure, the client and IoT resource need to establish a shared secret key to secure their link; this is achieved using PoP tokens. Note that remote attestation of the IoT resource can still be performed in periods where the IoT resource has network connectivity. Alternatively, remote attestation can be performed on-demand, using the client as an intermediate node, similar to

how the client is the intermediate node between the IoT resource and the AS for the authorization procedure described below.

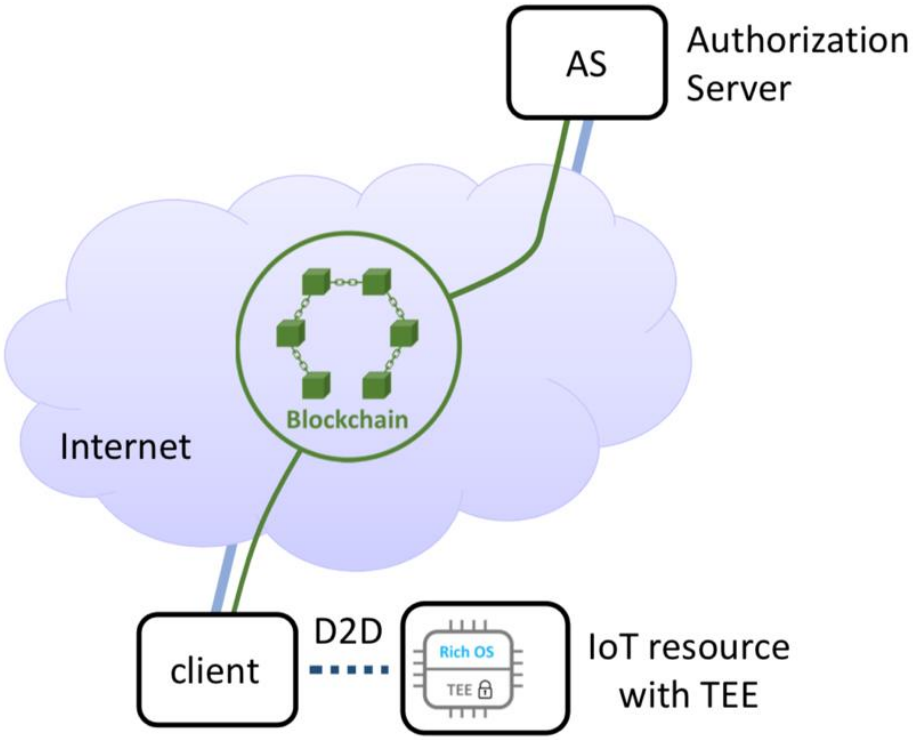


Figure 8. High-level architecture for delegated authorization exploiting an IoT resource's Trusted Execution Environment (TEE)

We assume that the client, the AS, and the resource owner, have an account (public/private key pair) on the blockchain. The client will use his account to pay for accessing the IoT resource. A client's deposit, assuming the authorization procedure is smoothly completed, will be transferred to the resource owner's account. Finally, the AS has an account to send transactions in order to set up a Hashed Time-Lock Contract (HTLC) as we discuss below.

Figure 9 shows the messages exchanged among the client, the AS, and the IoT resource. We assume that service discovery, during which the IoT resource is discovered, and the identification of the AS, which handles authorization requests for the IoT resource, has already occurred, so it is not shown in this figure. The AS that handles the authorization requests can be discovered by sending an initial unauthorized resource request message to the IoT resource or through a QR code located on the IoT resource. Steps 1 and 2 include the normal OAuth 2.0 message exchange between the client and the AS. According to OAuth 2.0, the communication between the client and the AS is secured using TLS, hence the information exchanged in Steps 1 and 2 is secured. After these two steps, the client has obtained the PoP key with which it can establish a secure link with the IoT resource. This is possible since the client also receives from the AS and forwards to the IoT device over the device-to-device connection the PoP key encrypted with the secret key $K_{Resource}$ that the IoT resource shares with the AS; sharing a secret key between the AS and the IoT resource can be achieved during the resource's initialization. In Step 3, the client forwards the encrypted PoP key to the IoT resource along with the access token. The above procedure is followed because, as shown in Figure 8, the IoT resource does not have continuous network connectivity, but only Device-to-Device (D2D) connectivity. On the other hand, if the IoT resource had continuous network connectivity, then it could instead obtain the PoP key directly from the AS. As shown in Figure 9, interaction of the client and the IoT

resource can still be performed using D2D communication, even if the IoT resource had continuous network connectivity.

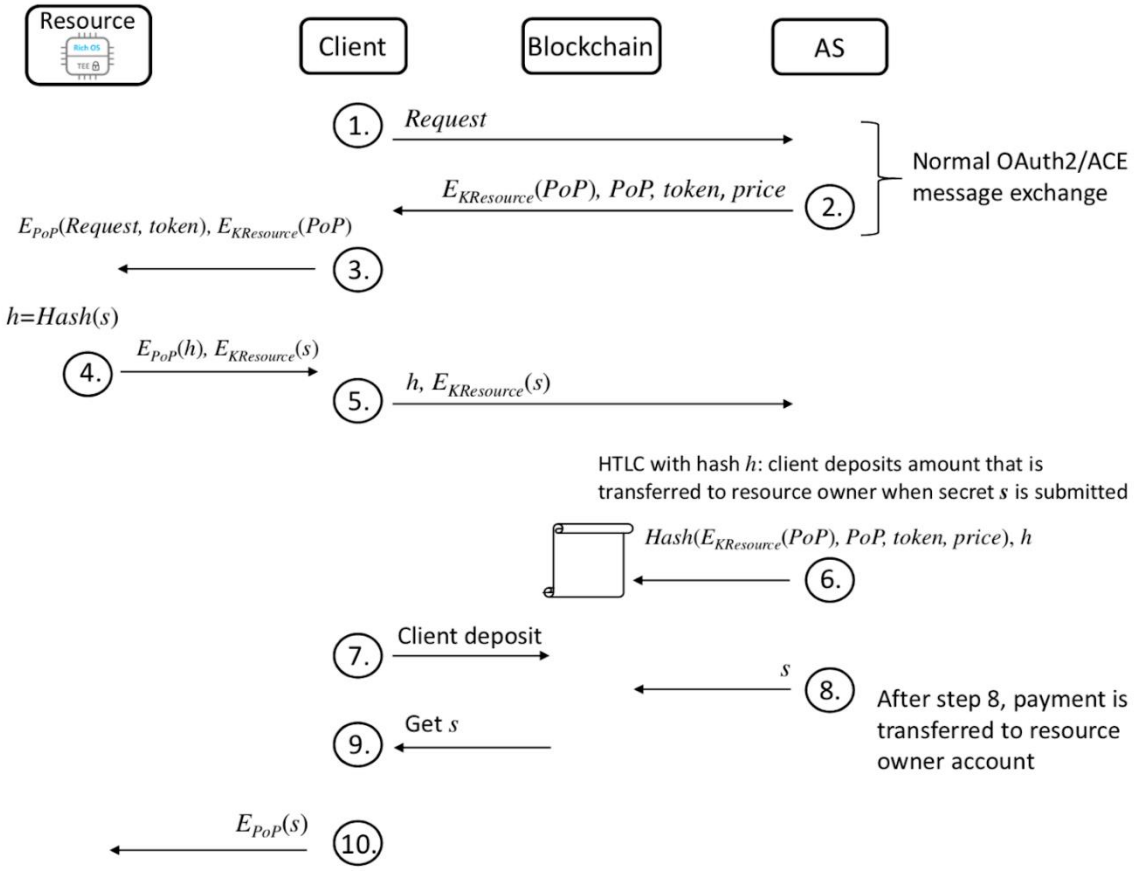


Figure 9. Message exchange for accessing the IoT resource with a TEE. The Trusted Application running in the TEE is responsible for generating the secret s, computing its hash h = Hash(s), and ensuring that the client provides the true secret to obtain resource access (Step 10).

When the IoT device receives the access token, it verifies its validity. If the access token is formatted as a signed JWT, then this verification involves checking the signature included in the JWT token. Alternatively, if the IoT resource had continuous network connectivity, then it could use introspection to communicate with the AS in order to verify the access token and the corresponding access rights that it allows [Sir+19c].

After verifying the access token, the IoT resource generates a secret *s* and computes its hash *h = Hash(s)*, which will be used in the hash-lock of the payment contract. In Step 4, over the secure communication channel it established with the client, the IoT resource sends the hash *h* and the secret *s* encrypted with the secret key $K_{Resource}$ that the IoT resource shares with the AS. After receiving the hash *h* and $E_{KResource}(s)$, the client forwards both to the AS in Step 5. In Step 6, the AS creates a hashed time-lock payment on the blockchain. The hash-lock is the same *h* that the AS received from the resource through the client, while the price is what the AS sent to the client in Step 2. Additionally, in Step 6 the hash of the authorization information the AS sent to the client in Step 2 is also submitted to the blockchain; in case of disputes, this hash is a non-repudiatable receipt of the information that the AS sent to the client.

Note that after Step 6, due to the transparency of transactions stored on the blockchain, the client can verify that the payment contract hash is the same as the value *h* that it had received from the IoT resource. Additionally, the payment contract also has a time-lock, hence the client's

deposit and disclosure of the secret *s* by the AS, which we discuss below, must occur within a maximum time interval; the time-lock allows either party (client or AS) to abort the procedure, if the other party has delayed taking the action required on its side. Hence, there is no need for trust between the client and the AS.

The client deposits the amount for resource access to the blockchain contract in Step 7. The deposit is not transferred directly to the resource owner's account, but to the payment contract's account which acts as an escrow service. The deposit is transferred to the resource owner's account only when the AS reveals to the contract the secret *s* that unlocks the hash-lock. Note that it is necessary that the hash function algorithm used at the IoT resource for computing the hash is the same as the hash function in the blockchain contract. If the AS does not reveal the secret *s* within the time defined by the contract's time-lock, then the client can submit a refund request for the deposit to be returned to the client's account; hence, by jointly using hash-locks and time-locks, the payment contract is a Hashed Time-Lock Contract (HTLC) and the two actions, client deposit and AS revealing the secret *s*, either both happen or neither of the two happens, i.e., they are atomic. If the above events occur smoothly, then the secret *s* would be revealed on the blockchain. Hence, the client can obtain the secret *s* in Step 9 and send it to the IoT resource in order to obtain access (Step 10). Note also that once the secret *s* is revealed on the blockchain, anyone can obtain it; however, the secret alone is not enough to gain access to the IoT resource since both the access token and the PoP key are required.

Since the secret and hash were produced in the IoT resource's TEE and access to the resource is also provided through the TEE, knowledge of the secret *s*, together with the access token and PoP key, ensure that the client can access the IoT resource according to the scope defined in the access token. From a high-level perspective, the TEE can be viewed as a trusted local ledger. The proposed model uses hash-locks and time-locks, which are interledger mechanisms that enable atomic cross-chain trading or atomic swaps, to cryptographically bind authorization grants and access through a TEE with blockchain payments. Using the same interledger mechanisms for cryptographically linking transactions on different blockchains, distributed ledgers, and TEEs, which are viewed as a local trusted ledger, allow for simplicity that can provide higher security and efficiency.

4.2.1 Evaluation

Our implementation of the IoT resource is based on the OP-TEE (Open Portable Trusted Execution Environment) open source port for the Raspberry Pi,[25] which uses ARM's TrustZone. OP-TEE follows the GlobalPlatform TEE system architecture [Glo18]. The secure world TEE runs the Linux OP-TEE operating system. The module providing the IoT resource access runs as a Trusted Application in the OP-TEE OS and performs the security operations that include generating the secret *s*, computing its hash *h = Hash(s)*, and verifying that the client provides the true secret *s* to obtain resource access. The module for providing the IoT resource's service is also executed in the TEE; this ensures[26] the IoT data's integrity and confidentiality and that the IoT resource will provide the intended service to the client if the latter provides the true secret *s*. A Client Application running outside the TEE uses GlobalPlatform's TEE client API to communicate with the Trusted Application.

For the evaluation we used a local Ethereum node running Go-Ethereum that was connected to the public Ethereum testnet Rinkeby. Smart contracts were written in Solidity with the Remix Web-based editor. The AS was based on a PHP implementation of the OAuth 2.0 framework. The AS used Web3.js to interact with the Rinkeby blockchain.

[25] https://www.op-tee.org/docs/rpi3/

[26] The degree to which this is guaranteed depends on the frequency/time that attestation is performed.

Table 7. Gas cost and delay for IoT resource access with TEE; gas price = 2.5 GWei, 1GWei = $1.1 x 10^{-7} on Feb 2, 2019

Transaction	**Gas** (and cost in $)	**Delay** (s)
Contract creation	473508 ($0.130)	-
Set payment information (Step 6)	32231 ($0.009)	13.6
Client deposit (Step 7)	28287 ($0.008)	14.9
Send secret s (Step 8)	41949 ($0.012)	15.0

Table 7 shows the gas, which quantifies the amount of EVM (Ethereum Virtual Machine) resources used, for the contract creation and the three transactions in Steps 6, 7, and 8 of Figure 9. The transactions are submitted with gas price 2.5 Gwei. The creation of the contract has the largest gas cost, which is one order of magnitude larger than the cost of the three transactions. Step 8 has higher gas than Steps 6 and 7 because it includes checking that the secret *s* submitted by the AS satisfies *h = Hash(s)*. Also, Step 6 has higher gas than Step 7 because the former submits two hashes to the blockchain, while Step 7 submits only the deposit, as shown in Figure 9.

The transaction delays shown in Table 7 are the average values from 20 executions. The 95% confidence interval was smaller than ±0.7 seconds of the averages shown. Both the average transaction delay and its standard deviation depend on the gas price: using a gas price smaller than 2.5 Gwei would result in both a higher average delay and a higher standard deviation. As expected, the blockchain transaction delay is significantly higher than the delay when the client interacts with the AS, Steps 1 and 2 in Figure 9, and when the client interacts with the IoT resource, Steps 3 and 4, which are both less than 0.08 seconds.

4.3 Constrained client and resource devices

In this section we present models for IoT resource access that consider different network connection capabilities of the client and the IoT resource:

- The IoT resource does not have continuous network connectivity, but only Device-to-Device (D2D) connectivity, whereas the client requesting resource access has continuous network connectivity.
- Both the client and the IoT resource do not have continuous network connectivity, i.e., they both having only D2D connectivity.
- The client has only D2D connectivity, whereas the IoT resource has continuous network connectivity.

In addition to whether a device (client or IoT resource) has continuous network connectivity or only D2D connectivity, a second dimension is whether the device is constrained in processing and memory. A device without processing and memory constraints can perform asymmetric key cryptographic functions, while a device that is constrained can only perform symmetric key cryptographic functions. Hence, processing and memory constraints influence the type of access tokens that can be used and in particular the type of integrity verification that will be incorporated in the JWT/CWT token; if the device is capable, then signatures using public/private keys can be used. On the other hand, if the device is constrained, then MAC integrity verification must be used.

In all three cases there is a Client Authorization Server (CAS) and an Authorization Server (AS) that handle requests and responses on behalf of the client and IoT resource, respectively [Ger+18]. The CAS and AS also handle interactions with the blockchain, in order to link authorization grants to blockchain payments. To take actions on behalf of the client and resource, the CAS and AS must have the consent of the client owner and the resource owner; one way to provide such consent is through verifiable credentials (VCs) [Spo+18], a more general approach than assuming that the client and resource owners control the CAS and AS, as assumed in [Ger+18].

During their initialization, both the client and the IoT resource establish with the CAS and AS, respectively, shared keys to be able to securely communicate over insecure D2D links and/or through intermediate nodes. If a device, either the client or the IoT resource, is constrained in terms of processing, then during its initialization it must establish a common secret key with the CAS or the AS. This shared secret key is used to add MAC integrity verification to the messages exchanged between the client and the CAS and between the AS and the IoT device. If the device has sufficient processing capability to perform asymmetric key cryptographic functions, then the CAS and/or the AS can use public key cryptography to sign messages they send to the client and/or the IoT resource, respectively. Note that if the IoT resource has continuous network connectivity, then instead of using signed or MAC integrity protected access tokens, simple access tokens can be used; in this case, the IoT resource can use introspection to verify the validity and scope of the access token.

A. Connected client and disconnected IoT resource

In the first model we discuss, the client has continuous network connectivity whereas the IoT device does not have continuous network connectivity, but only D2D connectivity. This is the case investigated in the previous sections. The difference with the model considered in this section is that the client, despite having network connectivity, does not interact directly with the blockchain, as shown in Figure 9.

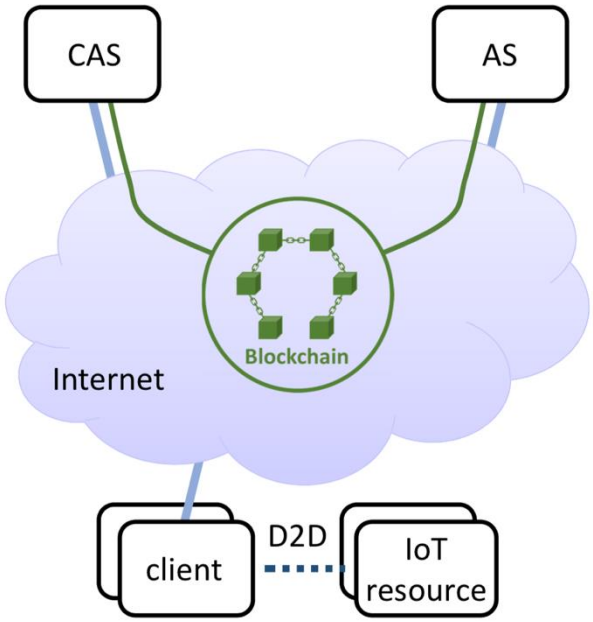


Figure 9. Client has Internet connectivity while the IoT resource has only D2D connectivity. The client acts as an intermediate node that forwards messages between the IoT resource and the AS, which handles authorization requests on behalf of the IoT resource. The client AS (CAS) interacts with the blockchain and the AS on behalf of the client.

Because the client does not interact directly with the blockchain, the CAS performs blockchain transactions on behalf of the client. The client can send authorization requests to the CAS, which handles the exchange of authorization messages with the AS and interacts with the blockchain. The goal of this interaction is to link authorization grants with blockchain payments. Specifically, the CAS obtains the necessary access token and PoP key from the AS only if it performs the blockchain payment, on behalf of the client. When the CAS receives the authorization credentials, it forwards them to the client. The client can then use the credentials to request service from the IoT resource. As we will see in more detail when we discuss the message exchange, when the client requests access to the IoT resource, the client acts as an intermediate node that forwards messages between the IoT resource and the AS, which handles authorization requests on behalf of the IoT resource. Specifically, the AS accepts authorization requests from the CAS and provides authorization credentials once it verifies that the appropriate blockchain payment has been performed. As shown in Figure 9, a single CAS and a single AS can handle multiple clients and IoT resources, respectively.

B. Disconnected client and disconnected IoT resource

Next we discuss the case where both the client and the IoT resource are constrained devices. As in the previous scenario, the authorization requests for the IoT resource are handled by the AS and the authorization requests on behalf of the client are handled by the CAS, see Figure 10. Moreover, both the CAS and the AS directly interact with the blockchain. The client, prior to communicating in D2D mode with the IoT resource, must obtain the necessary authorization credentials (access tokens and PoP keys) from the CAS. This may be achieved at any point prior to the time the client requests resource access, as the client has intermittent connectivity to the CAS using D2D communication. Once it has obtained the authorization credentials, the client can request access to the IoT resource through its D2D communication link, without requiring synchronous network connectivity or simultaneous D2D connectivity with the CAS. The communication between the CAS and the AS, to request resource access on behalf of the client and to obtain the authorization credential after the corresponding blockchain payment, is the same as the message exchange in the previous scenario.

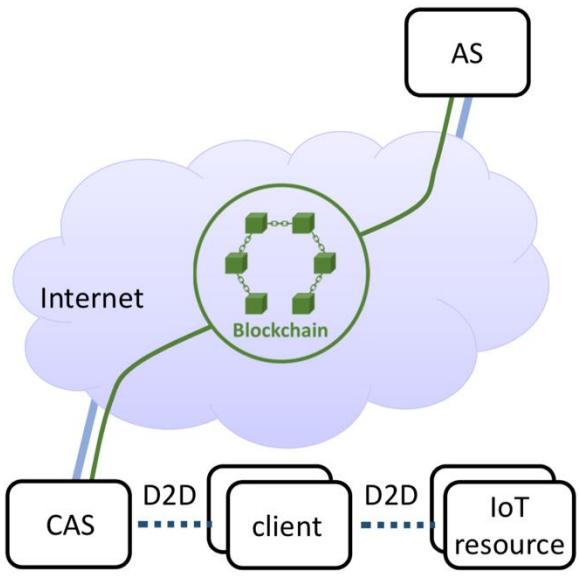


Figure 10. Both the client and IoT resource have only D2D connectivity. Prior to requesting access, the client must obtain the authorization credentials from the CAS. Once it has the credentials, the client can request access to the resource using D2D communication, without requiring synchronous network connectivity or simultaneous D2D connectivity with the CAS.

C. Disconnected client and connected IoT resource

In the third model, the client is disconnected while the IoT resource has continuous network connectivity. As in the model of the previous subsection, the CAS submits authorization requests to the AS and interacts with the blockchain on behalf of the client. The client communicates with the CAS using the connected IoT device as the intermediate node. The AS is responsible for handling authorization requests on behalf of the IoT resource; see Figure 11.

The CAS and AS interact in the same way as in the first two models. Once the CAS obtains the authorization credentials, which include the access token and the PoP key, it must transfer these to the client before the client requests service from the IoT resource; this transfer is performed through the connected IoT resource.

Because the IoT resource has continuous network connectivity, it can use introspection to verify the validity and scope of the access token [Har+12]. Hence, unlike the first two models, the access token does not need to contain a signature or a MAC for verifying its authenticity.

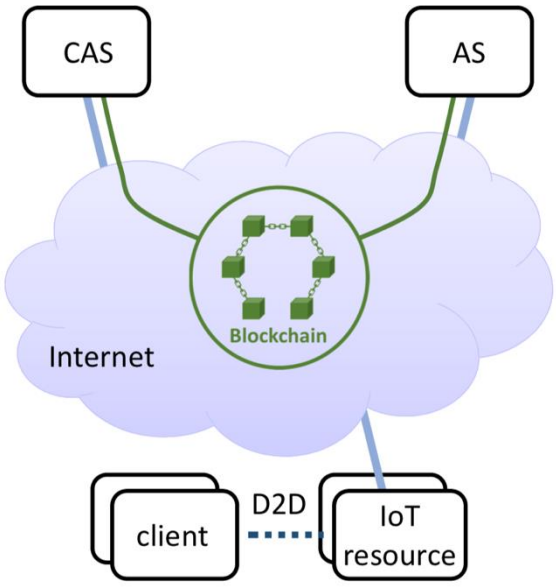


Figure 11. The client has only D2D connectivity while the IoT resource has continuous network connectivity. The IoT resource acts as an intermediate node that forwards messages between the client and the CAS, which handles authorization requests on its behalf.

4.3.1 Message exchange

In this section we present the message exchange between the various entities, namely the client, IoT resource, CAS, AS, and blockchain.

A. CAS-AS message exchange

We present two approaches for the message exchange between the CAS, which operates on behalf of the client, and the AS, which operates on behalf of the IoT resource. In the first approach, the authorization requests and responses are communicated directly between the CAS and AS. In this approach the blockchain is used only to record hashes of the authorization information exchanged between the CAS and the AS and to link blockchain payments to authorization grants. The motivation for recording hashes of the authorization information exchanged between the CAS and AS is that they serve as indisputable receipts in the case of disputes.

In the second approach, authorization requests and responses go through a smart contract, which is owned by the resource owner. Because smart contracts are executed by all blockchain nodes, a blockchain provides a secure execution environment with high availability. This offers higher protection against DoS attacks, compared to the first approach where access requests are sent directly to the AS. Moreover, in this approach a smart contract can be used to transparently record prices and other authorization policies defined by the resource owner. Examples of such policies include permitting resource access to specific CASes/clients, determined by their public keys on the blockchain, and dependence of access authorization on IoT events that are recorded on the blockchain. An additional advantage of allowing a smart contract to handle authorization requests is that the smart contract can securely bind the IoT resource with the AS responsible for providing authorization grants for that resource.

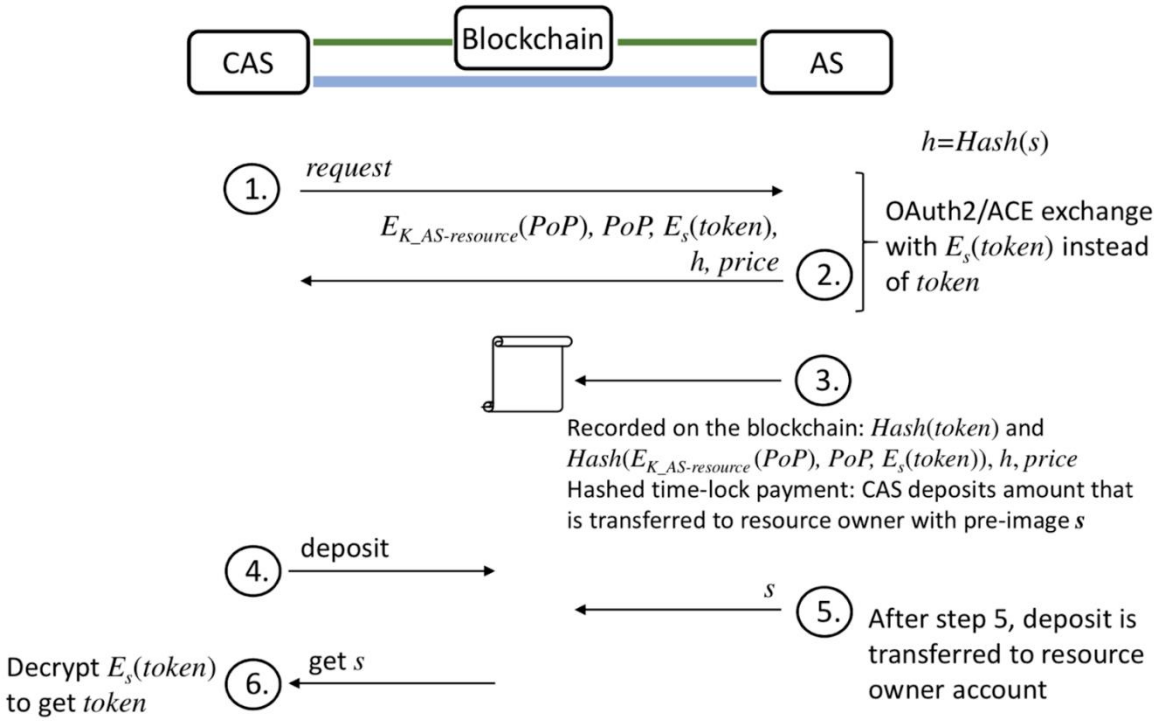


Figure 12. CAS-AS message exchange when authorization requests are sent directly to the AS. Hashes of the authorization information are recorded on the blockchain, which provide indisputable receipts in case of disagreement. Disclosure of authorization credentials is linked to blockchain payments.

Both approaches use a message exchange similar to that of the solutions presented in the previous sections, where the authorization message exchange occurred between the client and the AS, under the assumption that the client had continuous network connectivity and could interact directly with the blockchain.

1) Linking authorization grants to blockchain payments and recording hashes of authorization information: With this approach the initial communication between the CAS and the AS occurs as in the normal OAuth 2.0 framework, Figure 12. However, instead of the AS providing the CAS with authorization credentials after consent is given by the resource owner, the authorization credentials are disclosed only after the payment for resource access is recorded on the blockchain.

Specifically, in Step 1 the CAS sends to the AS on behalf of the client a request for accessing the IoT resource. The AS generates a random PoP key which it sends to the CAS[27] together with the PoP key encrypted with the secret key $K_{AS-resource}$ shared by the AS and the IoT resource, which is set during the IoT resource's initialization;[28] the client will later use the PoP key to establish a secure D2D link with the IoT resource. Also, the AS sends to the CAS the access token encrypted with a secret *s*, i.e., $E_s(token)$, the hash $h = Hash(s)$ of the secret *s*, and the price for the requested resource access scope. The secret *s* is a secret randomly generated by the AS and is required for the CAS to decrypt $E_s(token)$ and obtain the access token; the AS will reveal the secret *s* once it confirms that the payment for resource access has been committed on the blockchain. Communicating the price from the AS to the CAS allows different levels of resource access to be offered for different prices.

In Step 3, two hashes are submitted to the blockchain: the first is the hash of the token that the AS will reveal to the CAS once payment has been confirmed. The second is the hash of three items: $E_{KAS-resource}(PoP)$, the PoP key, and $E_s(token)$; the second hash serves as proof of the authorization information that is exchanged using OAuth between the AS and the CAS. Note that the above authorization exchange does not ensure that the access token the client obtains from the AS indeed allows access to the IoT resource.

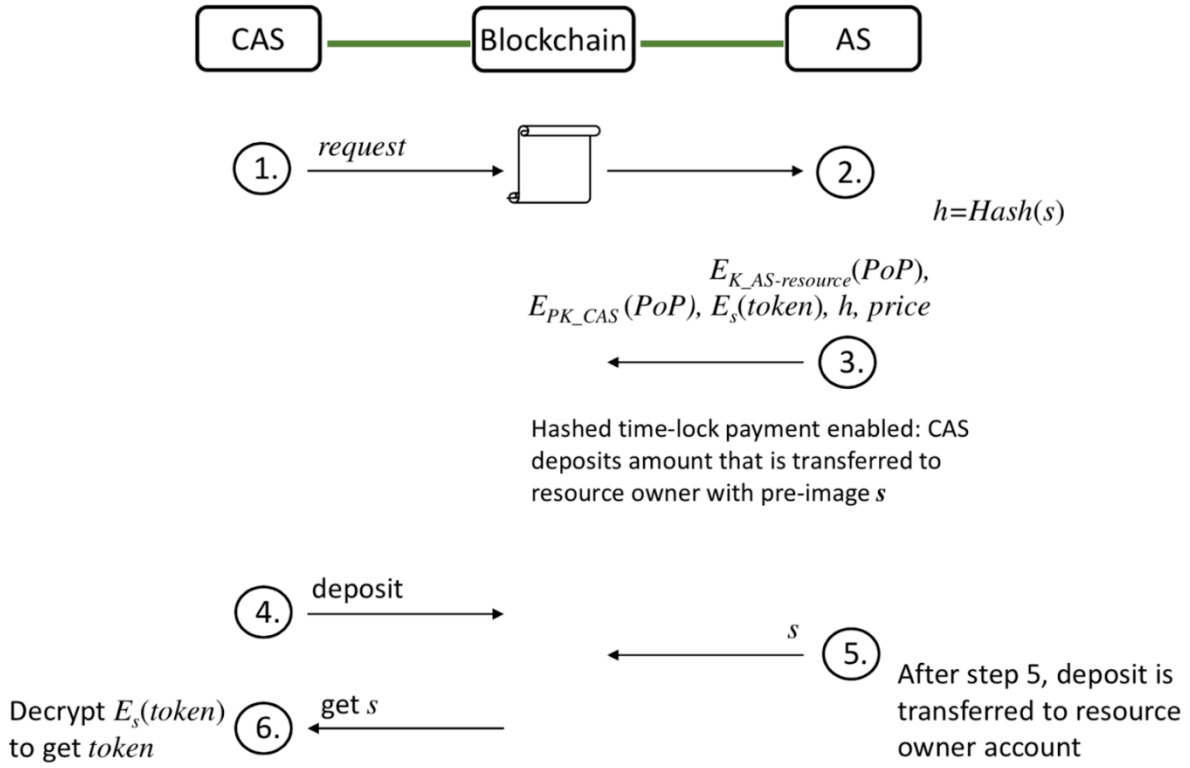


Figure 13. CAS-AS message exchange when a smart contract handles authorization requests. Authorization information is exchanged through the blockchain. As in the approach of Figure 12, disclosure of authorization credentials is linked to blockchain payments.

Also, in Step 3 a hashed time-lock payment is initiated on the blockchain, which allows the CAS to deposit the requested amount (Step 4). This amount will be transferred to the resource

[27] The communication link between the CAS and the AS is secured, hence the PoP key cannot be leaked through eavesdropping.

[28] If the resource has sufficient processing power, then the AS can use asymmetric cryptography and encrypt the PoP key with the resource's public key.

owner's account if the secret *s* (hash-lock) is submitted to the contract by the AS (Step 5) within some time interval. If the time interval is exceeded, then the CAS can request a refund of the amount it deposited. Once the secret *s* is revealed, the CAS can get *s* from the blockchain (Step 6) and decrypt $E_s(token)$, thus obtaining the access token. After Step 6, the CAS has all the credentials that are necessary for the client to request access from the IoT resource.

2) Smart contract for handling authorization requests: Unlike the previous approach where the CAS and the AS communicated directly, in the approach discussed next the interaction is through the smart contract, corresponding to Steps 1 and 2 in Figure 13.

In response to the authorization request it received from the CAS, in Step 3 of Figure 13, the AS sends to the smart contract the PoP key encrypted both with the secret key shared by the AS and the IoT resource, $E_{KAS-resource}(PoP)$, and with the public key of the CAS, $E_{PKCAS}(PoP)$. Note that in the previous approach the PoP key was sent from the AS to the CAS over a secure communication link, hence encrypting the PoP key was not necessary.

As in the first approach, a hash time-locked payment is enabled, allowing the CAS to deposit the amount corresponding to the resource access price (Step 4). The amount is transferred to the resource owner's account if the secret *s* that unlocks the hash-lock is revealed (Step 5). Once revealed, the CAS can obtain the secret *s* (Step 6), together with the other necessary authorization credentials that will allow the client to access the protected resource. If the blockchain is public, then *s* can be read by anyone, hence everyone can obtain the access token. However, the access token cannot be used alone, since the PoP key is also required for accessing the resource. Nevertheless, if privacy of the access token is important, then the secret *s* can be encrypted using CAS's public key PK_{CAS} and the hash-lock set to $h = Hash(E_{PKCAS}(s))$.

B. Client-CAS and client-IoT resource message exchange

The message exchange between the client, the CAS, and the IoT resource when the IoT resource does not have continuous network connectivity is shown in Figure 14. Note that this message exchange applies to both the case where the client has continuous network connectivity and the case where the client has only D2D connectivity. Initially the client communicates with the CAS by sending a message with its intent to access the IoT resource (Step 1). After receiving the request from the client, the CAS performs either of the two message exchanges presented in the previous section. Next, in Step 2 the client receives the authorization credentials from the CAS and in Step 3 it sends its access request to the IoT resource.

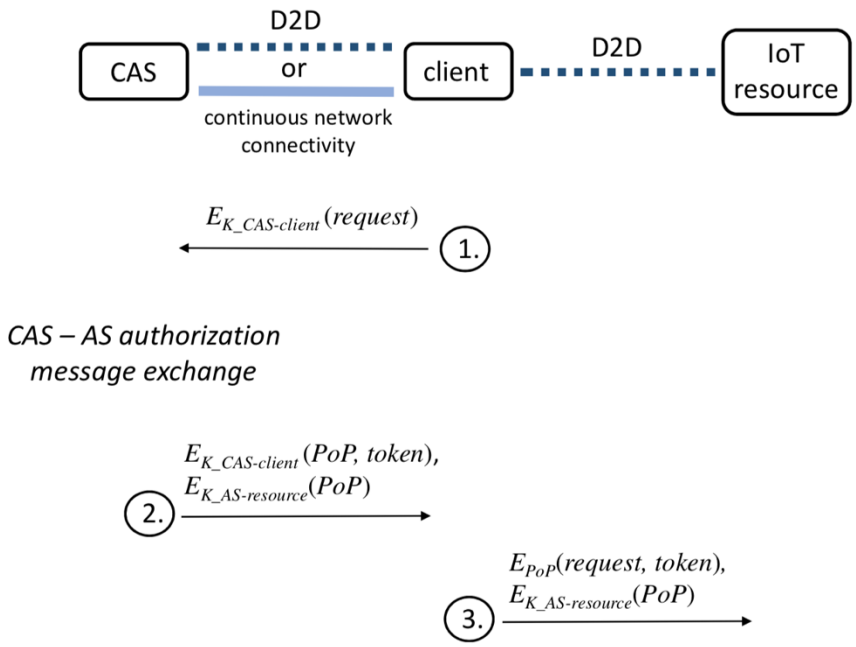


Figure 14. Message exchange between CAS, client, and resource when the IoT resource has only D2D connectivity (Figures 9 and 10). The CAS-AS message exchange can follow the sequence in Figure 12 or 13.

The message exchange when the client has only D2D connectivity and the IoT resource has continuous network connectivity is shown in Figure 15. Now, the client communicates with the CAS that handles authorization requests on its behalf using the connected IoT resource as an intermediate node. Note that the communication of the client and the CAS is secured, since they share a secret key $K_{CAS-client}$ that was configured during the client's initialization.

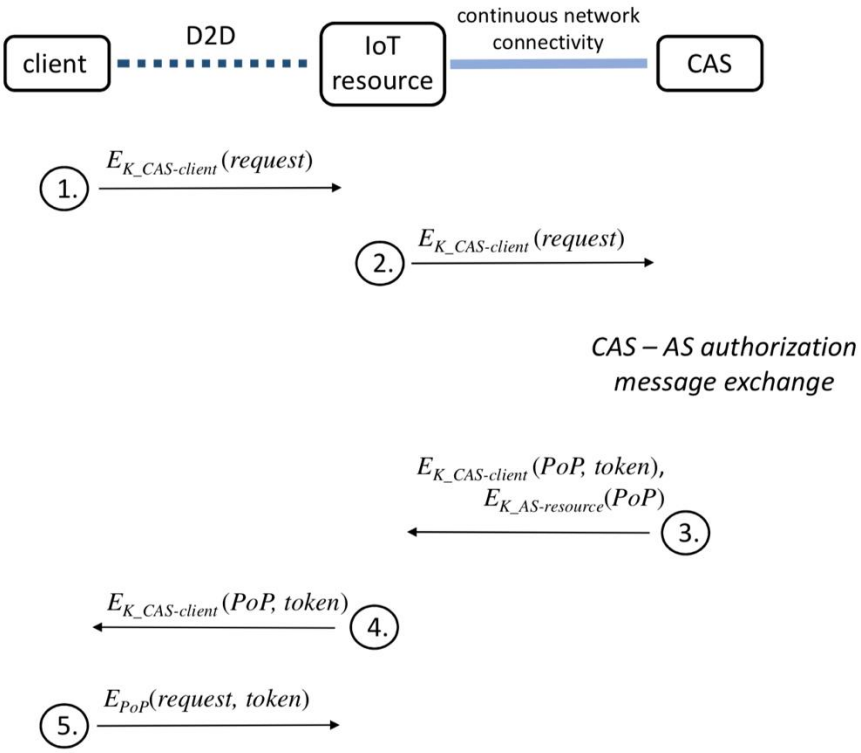
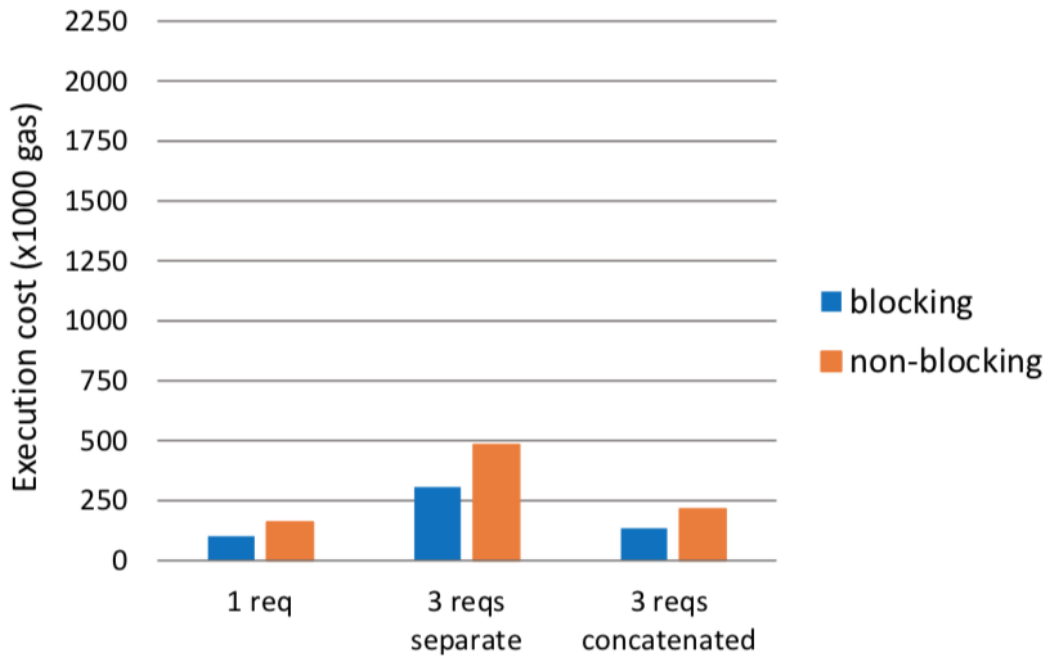


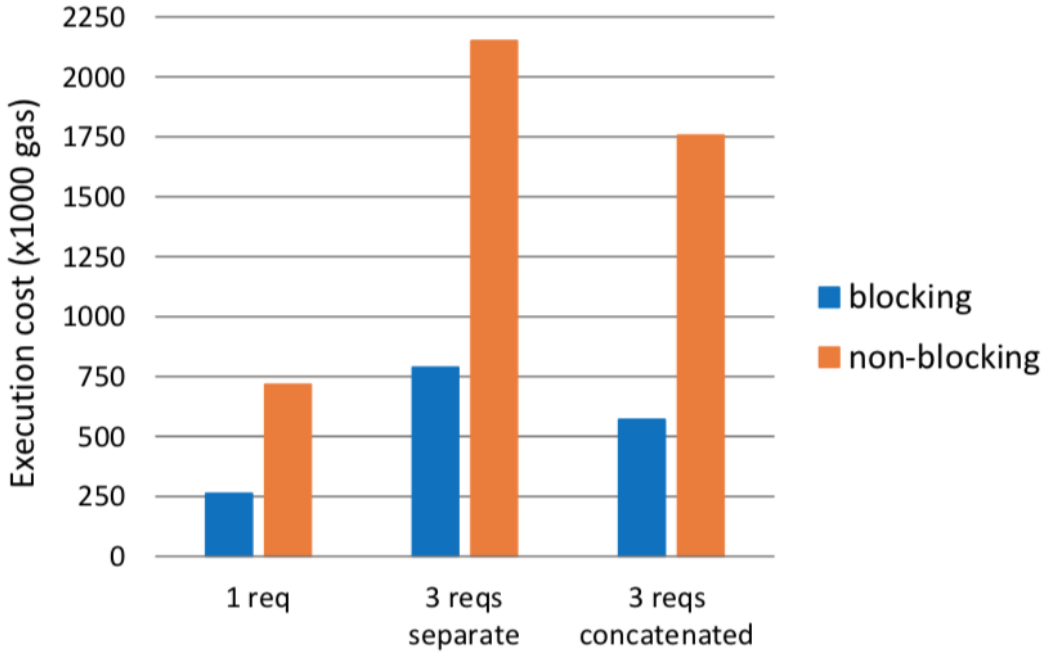
Figure 15. Message exchange between client, resource, and CAS message when the client has only D2D connectivity while the IoT resource has continuous network connectivity (Figure 11). The CAS-AS message exchange can follow the sequence in Figure 12 or 13.

4.3.2 Evaluation

For the evaluation, we used a local Ethereum node running Go-Ethereum that was connected to the public Ethereum testnet Rinkeby. Smart contracts were written in Solidity with the Remix Web-based editor. The AS was based on a PHP implementation of the OAuth 2.0 framework. The CAS and AS used Web3.js to interact with the Rinkeby blockchain. We compare the two approaches presented in the previous section: the first records hashes of the authorization information on the blockchain (Figure 12) and the second involves a smart contract handling authorization requests (Figure 13). For each of the two approaches we compare four implementations: The first is the baseline implementation where the smart contract operates in blocking mode where only one authorization request can be handled at a time: “1 req” in Figure 16(a). The second implementation also operates in blocking mode, but each message includes three authorization requests—“3 reqs concatenated” in Figure 16(a)—which are sent by the same CAS; similarly to the requests, we assume that the responses are also concatenated, which requires that the authorizations are handled by the same AS. The third implementation operates in non-blocking mode, allowing more than one authorization request, each in a separate message, to be pending at the same time—“1 req” in Figure 16(b). Finally, the fourth implementation operates in non-blocking mode, as the previous (third) implementation, but each message includes three authorization requests—“3 reqs concatenated” in Figure 16(b). The “3 reqs separate” columns in Figures 16(a) and 16(b) correspond to the case where three authorization requests and their responses are sent and received separately with blocking and non-blocking operations, respectively.



(a) Recording only hashes on the blockchain



(b) Smart contract handling authorization requests

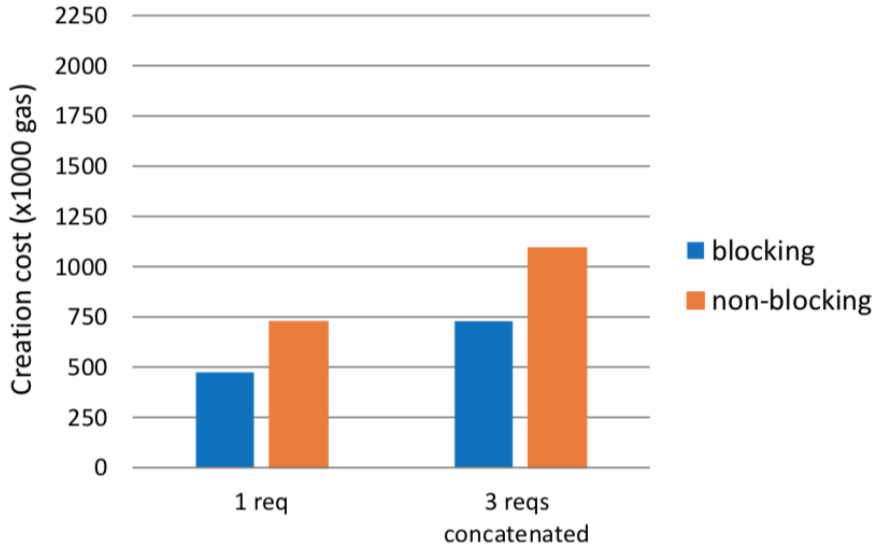
Figure 16. Smart contract execution cost. The top graph corresponds to the approach in Figure 12 where only hashes are recorded on the blockchain. The bottom graph corresponds to the approach in Figure 13 where a smart contract handles authorization requests.

Figure 16 shows the execution cost (gas), which quantifies the amount of EVM (Ethereum Virtual Machine) resources (computation and storage), for each of the above implementations.

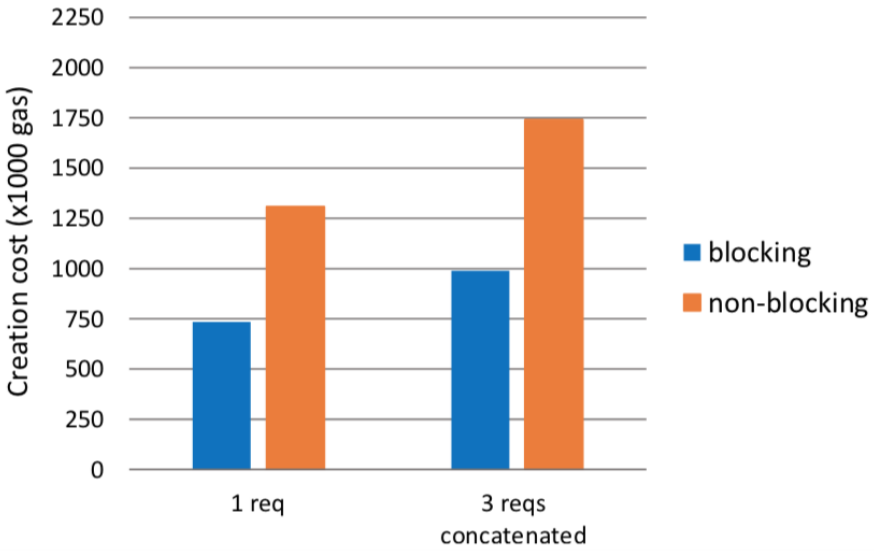
A comparison of the corresponding columns in Figures 16(a) and 16(b) shows that, for a blocking implementation, a smart contract that handles authorization requests requires approximately 2.5 times more gas than the approach that records only hashes of the authorization information on the blockchain. For the non-blocking implementation, the ratio is larger and close to 4 times. Figure 16(a) shows that the gas is 88% higher for the non-blocking implementation compared to the blocking when only hashes are recorded on the blockchain, while Figure 16(b) shows that it is approximately 190% higher in the case of a smart contract handling authorization requests. The above results quantify the higher execution cost for smart contracts, with more functionality.

Comparison of columns “3 reqs separate” and “3 reqs concatenated” in Figure 16(a) shows that, for the blocking implementations, the gas, when three requests and their responses are concatenated, is smaller than the gas when the requests are sent separately by 56% when only hashes are recorded, and smaller by 28% when a smart contract handles requests. The gains for non-blocking, Figure 16(a), are 55% and 18%, respectively, when only hashes are recorded and when a smart contract handles authorization requests. These results show that concatenation of requests can provide gains in terms of reduced execution cost; indeed, the gains are significantly higher for simple contracts that record only hashes. Additional experiments (not shown) indicate that, as expected, the gains are higher when more requests are concatenated. Specifically, for non-blocking, when 9 requests are concatenated the gains are 67% (higher than the 55% gain when 3 requests are concatenated) when only hashes are recorded and 25% (higher than the 18% gain when 3 requests are concatenated) when a smart contract handles requests.

Concatenation of authorization requests can be performed in the space domain, when CASes and ASes handle multiple clients and IoT resources. Alternatively, concatenation can be performed in the time domain by aggregating requests received by a CAS in a time interval, before sending them to the AS. Such time domain aggregation of requests adds a delay to the authorization process, which needs to be considered along with the blockchain transaction time.



(a) Recording only hashes on the blockchain



(b) Smart contract handling authorization requests

Figure 17. Smart contract creation cost. The top graph corresponds to Figure 12, where only hashes are recorded on the blockchain. The bottom graph corresponds to Figure 13, where a smart contract handles requests. The contract creation cost for "3 reqs separate" in Figure 16 is the same as the cost for "1 req."

The contract creation cost is shown in Figure 17. Note that this figure does not contain the contract creation cost for "3 reqs separate," since it uses the same contract as "1 req." The figure shows that the increase of the contract creation cost for the second approach, where authorization requests are handled by the smart contract, compared with the simpler scheme, where only hashes of authorization information are recorded on the blockchain, is smaller for the non-blocking compared to the blocking implementation. A comparison of the corresponding columns in Figures 17(a) and 17(b) shows that the contract creation cost for smart contracts handling authorization requests is 36 to 80% higher than the creation cost for contracts that record only hashes. An additional conclusion from the comparison of Figure 17 and Figure 16

is that for simple contracts that record only hashes and are blocking, the contract creation cost dominates the execution cost, while for more complex smart contracts such as the ones handling authorization requests and are non-blocking, the execution cost becomes comparable to the creation cost.

Finally, 20 executions of each of the non-blocking implementations have shown that the average transaction delay when only hashes of authorization information are recorded on the blockchain is 44 seconds, with a 95% confidence interval ± 5 seconds; the delay for the blocking implementation with three separate requests is higher by approximately 29 seconds, due to the serialization that blocking imposes. For a smart contract handling authorization requests the delay is 58 seconds, with a 95% confidence interval ± 6 seconds. The above results show that the delay is approximately 32% higher for the smart contract approach compared to the approach that records only hashes. This result is expected, since recording only hashes involves three transactions on the blockchain (Figure 12) whereas a smart contract handling authorization requests involves four transactions (Figure 13).

5. Evaluation scenarios

This section follows the SOFIE pilots, generalizes them into pilot inspired use cases by including alternatives not selected in the SOFIE pilots, abstracts them out to appropriate degree and uses emulation and simulation to consider the various tradeoffs of many potential alternative design decisions and their impact.

5.1 Food supply chain

5.1.1 Overview

The food supply chain pilot aims at leveraging distributed ledgers to provide traditional supply chains with dependable provenance data. More specifically, it targets the supply chain transferring agricultural products from producers to supermarkets and it aims at providing the following features:

- Traceability of agricultural products from the producer to the consumer
- Traceability of transport and storage conditions
- Resolution of disputes in case of customer complaints

This pilot assumes smart boxes (or, simply, boxes) as the end-to-end unit of transfer. That is, products are packed into boxes by the producer and they remain in these boxes throughout the entire transfer until they reach the consumer. Each smart box is equipped with an RFID tag, which is scanned and registered when the box is handed over by one stage of the supply chain to another.

The chain consists of five stages, shown in the following table.

Table 8. The stages of the food supply chain pilot use case

Stage no.	Stage name	Abbre-viation	Role
1	Table Grapes Field	TGF	Grows table grapes and packs them into boxes
2	Transportation A	TRA	Transfers boxes from TGF to SDC
3	Storage & Distribution Center	SDC	Collects, stores, and dispatches boxes
4	Transportation B	TRB	Transfers boxes from SDC to SM
5	Supermarket	SM	Displays boxes and sells them to consumers

5.1.2 Events

The SOFIE platform will record all **handovers** between consecutive stages, as well as **periodically reported conditions** in each stage. This leads to the following list of events governing the Food Chain pilot at a high level. These events can be split up into the three categories shown in the following three tables.

Table 9. Handover events of the food supply chain pilot

Event Type 1: Handovers	
Event	**Parameters**
Box handover: producer → TRA employee	`time, box, weight, producer, employeeTRA`
Box handover: TRA employee → SDC employee	`time, box, weight, employeeTRA, employeeSDC`
Box handover: SDC employee → TRB employee	`time, box, weight, employeeSDC, employeeTRB`
Box handover: TRB employee → SM employee	`time, box, weight, employeeTRB, employeeSM`

Table 10. Action events of the food supply chain pilot

Event Type 2: Actions	
Event	**Parameters**
Driver picks up specific truck (assuming multiple drivers & trucks in company)	`time, employee{TRA|TRB}, truck`
Driver parks and leaves truck	`time, employee{TRA|TRB}, truck`
Store box in specific shelf of the Storage & Distribution Centre	`time, box, employeeDC, room, rack, shelf`
Store box in specific shelf of the Supermarket	`time, box, employeeSM, store, aisle, shelf`

Table 11. Periodic logging events of the food supply chain pilot

Event Type 3: Periodic logging	
Event	**Parameters**
Field: log temperature, humidity, precipitation	`time, field, temp, humid, rain`
Transportation: log temperature, humidity, GPS location	`time, truck, temp, humid, location`
Distribution Centre: log temperature, humidity	`time, room, temp, humid`
Supermarket: log temperature, humidity	`time, store, aisle, temp, humid`

5.1.3 Emulation overview

In order to evaluate alternative architectures for the Food Chain use case, we implemented two different scenarios and we considered two types of deployment, resulting in a total of four distinct emulations.

Scenario 1: We have a single smart contract, providing an API to handle all aforementioned events. All five stages of the chain have access to this smart contract and use it to record all events taking place. The advantage of this scenario is its simplicity, as all data reported is stored in a single smart contract's storage. As a consequence of this simplicity, this scenario uses fewer blockchain resources, thus, less gas.

Scenario 2: We have a total of five distinct smart contracts, one per stage. Each smart contract provides the API needed to handle that stage's actions, logs, and handovers, as well as to interact with smart contracts of adjacent stages. This scenario's advantage is that it gives higher flexibility to organizations to manage their own smart contracts, as long as they respect the interface to neighbouring stages' smart contracts. Its downside is the increased use of blockchain resources, hence the extra gas it spends.

Deployment 1: Use of public Ethereum only. All smart contracts are deployed on a public Ethereum instance, such as the original Ethereum, or a testnet, such as Rinkeby or Ropsten. This deployment offers the highest transparency and immutability guarantees, however, with a significant monetary cost for executing the respective smart contracts.

Deployment 2: Use of both private and public Ethereum. In this deployment, all smart contracts (of either Scenario 1 or 2) are deployed and executed on a private Ethereum instance to avoid the high execution costs associated with public Ethereum instances. Only a single smart contract is deployed on a public Ethereum instance for the sole purpose of *anchoring*, that is, for periodic public recording of the private instance's block hash in order to increase immutability guarantees.

5.1.4 Evaluation results

We start by presenting a table of the cost of certain types of smart contract calls in gas, which are the same independently of the scenario used.

Table 12. Execution cost for each periodic logging event of the food supply chain pilot

Event	Gas	Cost (ETH*)	Cost (EUR*)
Periodic log (TGF)	(mean) 85435	0.0000854	€0.021
Periodic log (TRA/TRB)	(mean) 85435	0.0000854	€0.021
Periodic log (SDC)	(mean) 72535	0.0000725	€0.018
Periodic log (SM)	(mean) 85435	0.0000854	€0.021
Anchoring	47744	0.0000477	€0.012
(*) Based on June 2019 prices			

Note that, although the gas used for each individual anchor is precisely 47744, for periodic logging events we present *average* values. The exact gas used for 100 consecutive TRA or TRB (transportation) logs is shown in the following figure. The periodic pattern repeating every eight logs, is due to the fact that EVM reserves storage in 256-bit increments. As our log data are encoded in 32-bit values, at every 8th log each array (e.g., the temperatures array, etc.) has to be extended by another 256 bits, thus spending more gas.

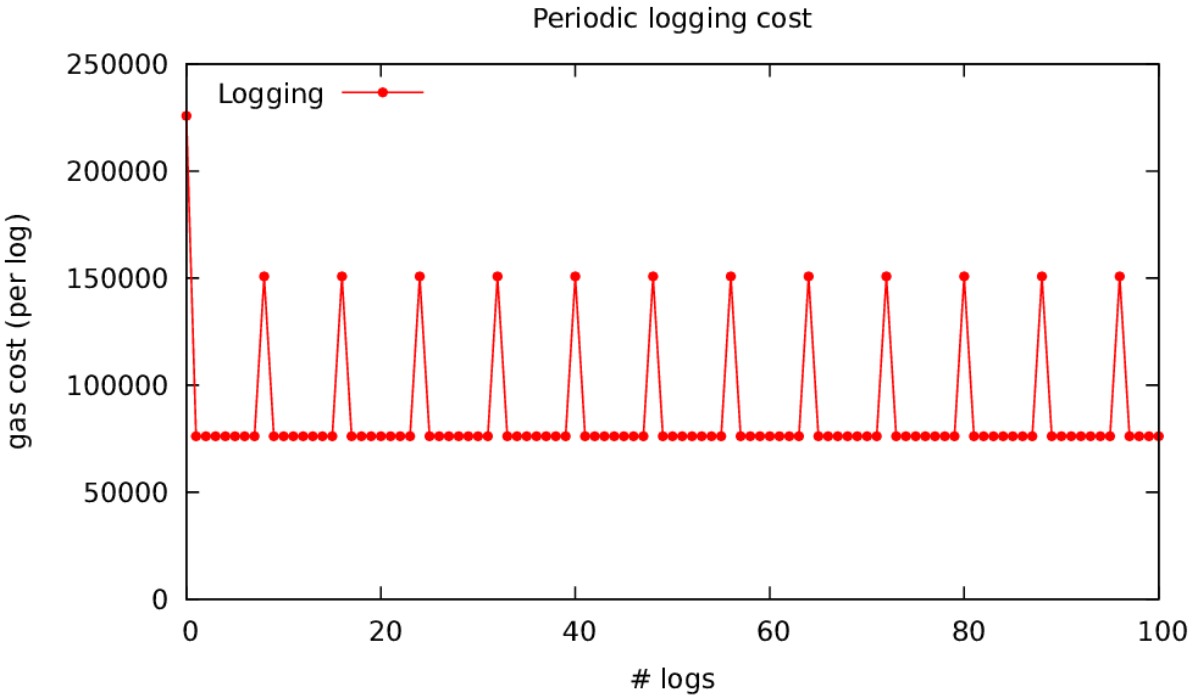


Figure 18. Periodic logging cost for 100 consecutive transportation logs

In contrast, anchors (i.e., block hashes) are 256-bit long, so every single anchor registration reserves an extra 256-bits of storage, resulting in uniform gas cost.

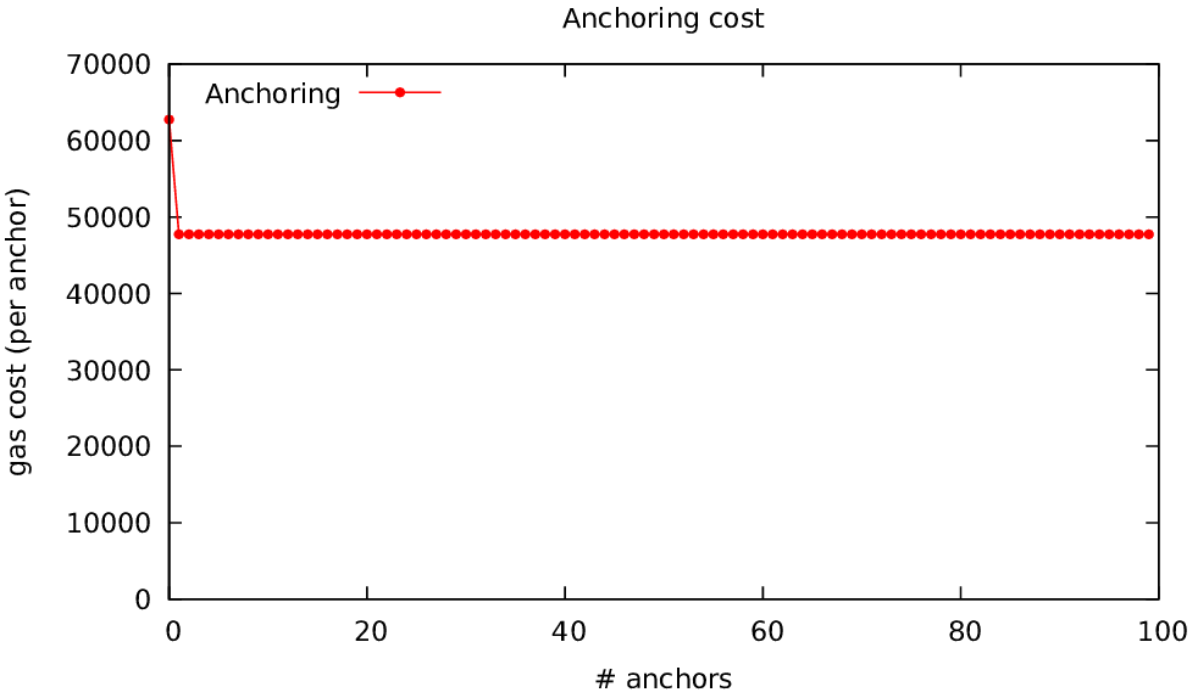


Figure 19. Anchoring cost

Let us now focus on the functions that are not symmetric across the two proposed scenarios. The first one is the function that registers a new box with a producer (stage 1: TGF). Counterintuitively, this function is almost twice as costly in Scenario 1 compared to Scenario 2. The reason is that Scenario 1, storing all data in a single smart contract, allocates more storage when a new box is registered at the beginning of the supply chain, to accommodate data relevant to all five stages. In contrast, in Scenario 2, where each chain stage uses their own smart contract, registering a new box with a producer allocates just enough storage to record data relevant to the producer.

The following figure shows the exact gas cost for 100 box registrations for both scenarios. The periodic artifact discussed above (due to 256-bit storage allocations vs. 32-bit box indices) is clearly visible as spikes repeating every 8 new boxes.

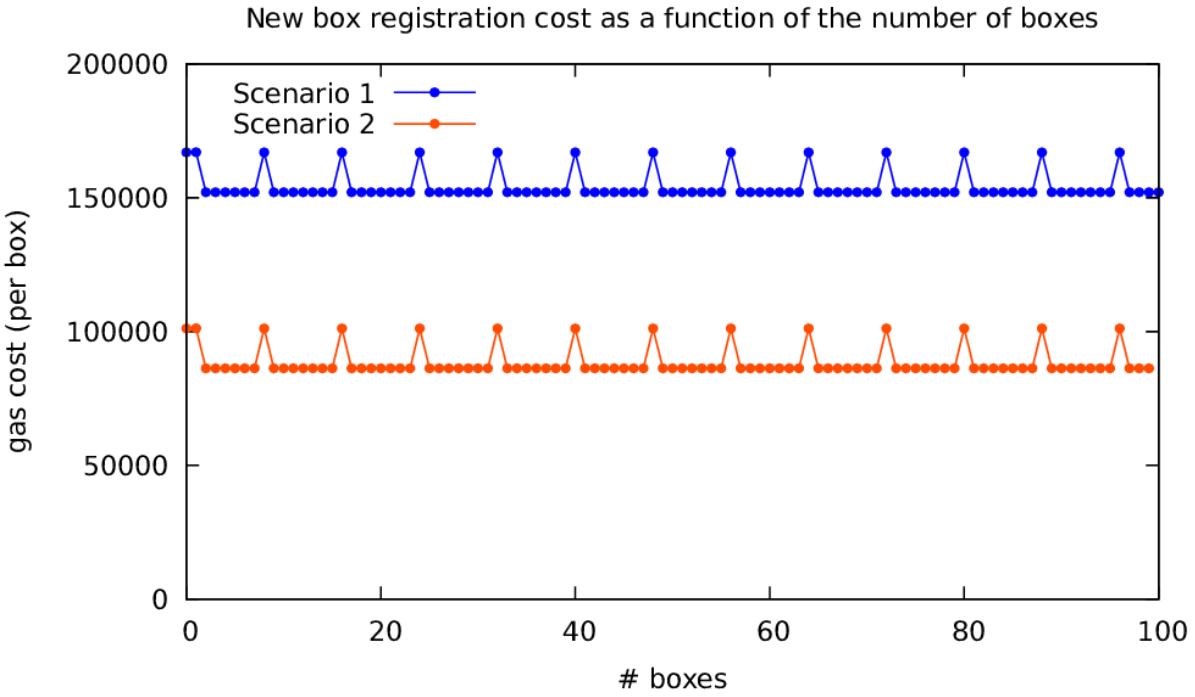


Figure 20. Execution cost for 100 box registrations

The second type of function that differs across the two scenarios is the handover function. In our emulation, handovers are symmetric across all stages. The following figure shows the gas spent for handing over a box from one stage to the next one, as a function of the number of boxes that each of these two stages is currently hosting. Besides the aforementioned artifact appearing with a period of eight boxes, we notice a clear cost increase as stages become full. This is due to the data structures used to maintain the list of boxes currently at a given stage. More specifically, a box is removed from an array in stage *i* and then appended to an array in stage *i+1*.

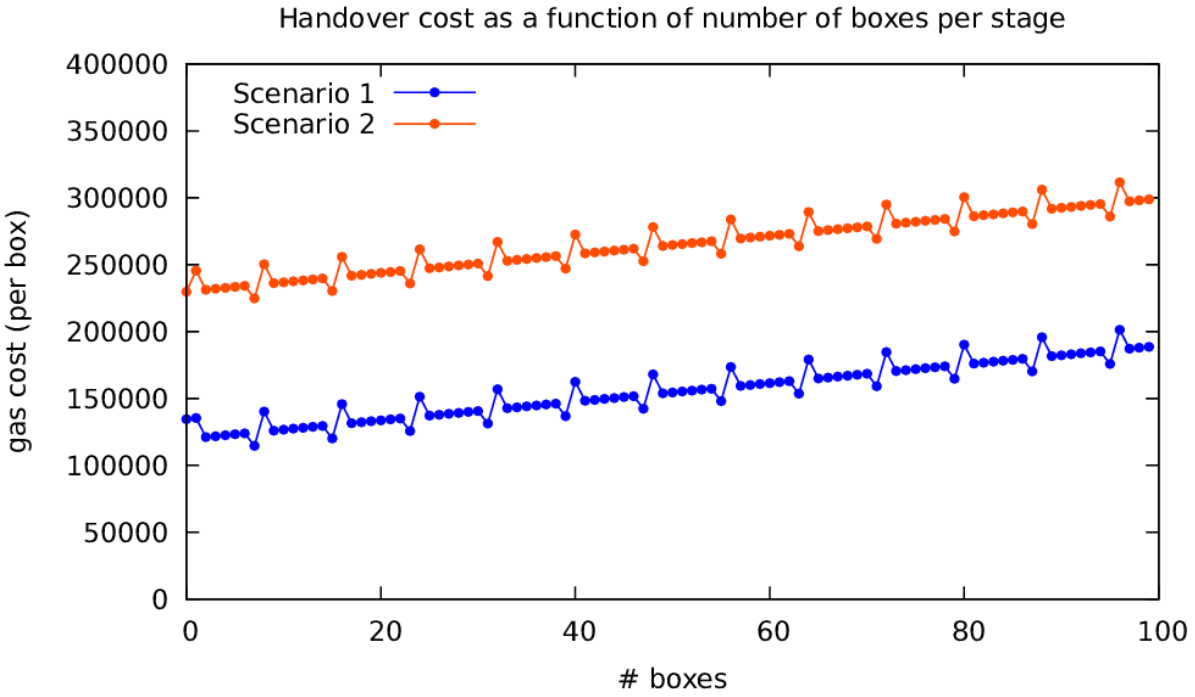


Figure 21. Execution cost for handover function from one stage to the next one

We also notice that Scenario 2 is far costlier for performing handovers. This is due to two facts. First, in Scenario 2 each handover call at stage *i* results in an external call to the smart contract of stage *i+1*, as the handover needs to also be registered there. Second, data is essentially stored twice, as both stage *i* and *i+1* need to keep a record of each handover between them.

The data presented above is summarized in the following table.

Table 13. Execution cost for both food supply chain pilot handover scenarios

Event	Scenario 1			Scenario 2		
	Gas	ETH	EUR	Gas	ETH	EUR
Register Box	153974	0.000154	€0.037	88133	0.0000881	€0.021
Handover TGF→TRA	~150000	0.00015	€0.036	~250000	0.00025	€0.060
Handover TRA→SDC	~150000	0.00015	€0.036	~250000	0.00025	€0.060
Handover SDC→TRB	~150000	0.00015	€0.036	~250000	0.00025	€0.060
Handover TRB→SM	~150000	0.00015	€0.036	~250000	0.00025	€0.060
Sum	**~753974**	**0.000754**	**€0.181**	**~1088133**	**~0.001**	**€0.261**

We can see that the initial registration and four subsequent handovers for a given box will result in an aggregate cost of €0.18 for scenario 1, or €0.26 for scenario 2, when running on the

original public Ethereum network. This is a non-trivial cost, as it concerns each individual box. As expected, using the public Ethereum blockchain is an expensive and non-scalable option.

Instead, a far more cost-effective solution is to run the aforementioned smart contracts in a private Ethereum, where gas does not reflect real money. In this case, anchoring will be important for providing immutability guarantees. The cost of anchoring (€0.012) is negligible compared to a 15 to 20 times higher cost *per box*. In addition to that, anchoring does not need to be configured for every single block generated. It can be applied periodically every hour, or every day.

5.1.5 Evaluation conclusions

In order to develop an intuition about the overall costs of our four emulation scenarios and deployment combinations for a full day operation, we make the following assumptions, based on discussions with SOFIE's advisory board. We assume a total of **6000 smart boxes** entering the food-chain per day, and a total of **10 trucks** needed to transfer these 6000 smart boxes between two stages (i.e., a total of 20 trucks, given that we have two transportation stages).

Table 14. Overall cost of the food supply chain pilot stages

Stage	Description	Count	Logging period	Single log cost	24-hour cost
TGF	Synelixis sensors collecting precipitation, temperature, and humidity logs	1000	1 hour	€0.021	€504.00
TRA	Sensors for reporting temperature and humidity (1 per truck, for a total of 10 trucks)	10	5 min	€0.021	€60.48
SDC	Sensors for reporting temperature and humidity	100	5 min	€0.018	€518.40
TRB	Sensors for reporting temperature and humidity (1 per truck, for a total of 10 trucks)	10	5 min	€0.021	€60.48
SM	Sensors for reporting temperature and humidity	20	5 min	€0.021	€120.96
Sum	-	-	-	-	€1264.32

For estimating the cost of handover registration, we have to compute it separately for each scenario.

Table 15. Handover registration cost for each of the scenarios

Description	Box count per day	Scenario 1		Scenario 2	
		Single box	All boxes	Single box	All boxes
Cost for handovers across all stages.	60000	€0.181	€10860	€0.261	€15660

Finally, we should consider the cost of anchoring, in case of the second deployment type, i.e., in case we use a private Ethereum instance for storing all logs and handovers, and we periodically store the block hash on public Ethereum. For simplicity we will consider as a baseline approach storing the hashes of every single block.

Table 16. Anchoring cost for the second deployment type of the food supply chain pilot

Description	Logging period	Single anchor cost	24-hour cost
Cost for storing a private Ethereum's block hash on a public Ethereum smart contract ("anchoring")	15 s (the default block generation period in Ethereum)	€0.012	€69.12

In conclusion, an entire day's cost based on the aforementioned assumptions is estimated to be as follows:

Table 17. Entire day's cost for each evaluation scenario of the food supply chain pilot

Scenario	Deployment	24-hour cost
Scenario 1 (single smart contract for all stages)	Public Ethereum	€12124.32
	Private + Public Ethereum	€69.12
Scenario 2 (separate smart contract per stage)	Public Ethereum	€16924.32
	Private + Public Ethereum	€69.12

This aggregate cost comparison further emphasizes the need to deploy and maintain private Ethereum instances instead of replying on the public one alone, as the cost implications are severe.

5.2 Decentralised energy flexibility marketplace

The decentralised energy flexibility marketplace pilot aims to balance the load on a real energy network, namely the distribution grid of the city of Terni, located in central Italy, by charging electrical vehicles.

In Terni, a significant amount of energy is produced locally by distributed photovoltaic plants, which on occasion can cause Reverse Power Flow, when unbalances between produced and consumed energy occur. To avoid this abnormal operation, electrical vehicles (EVs) will be offered incentives to match their EV charging needs with the distribution network's

requirements, through the decentralized marketplace, which allows electricity producers and consumers to place offers and bids for selling and buying electricity.

The actors of the pilot use case are:

DSO: Distribution System (Grid) Operator, responsible for grid management

CSO: Charging Station Operator, operates multiple charging stations

CS: Charging Station, that can charge electric vehicles

EV: Electric Vehicle

EVU: EV User

FM: Fleet Manager, represents a group of EVs in the energy price negotiations

EP: Electricity Provider (that may be included in a later stage of the pilot, but is included in the use case and) that would act between the DSO and electricity users, such as FMs

The flow of the use case is:

1. The DSO puts flexibility requests to the decentralized marketplace (that utilizes blockchain), asking for a specific amount of energy (kWh) to be drawn at specific time intervals, at a specific location (expressed as GPS coordinates), while providing specific incentives (expressed as tokens) in order to shave peaks of locally produced energy.
2. The FM places offers to the marketplace in order to maximize the incentives. The offers include: user type (e.g. electricity load imposed), current location, residual autonomy (i.e., how long the vehicle can withstand before it has to be charged), and EV's current status (e.g., parked).
3. The marketplace identifies potential candidates taking both offers (by DSO and FM) into account and notifies selected EV users that they will receive a token incentive if they fulfil the conditions of the DSO's offer (i.e., charge the vehicle with a specific amount of electricity, using the assigned charging station, or group of charging stations, within a specific time interval).
4. Some EV users accept the offer and the acceptance is recorded in the blockchain used by the marketplace.
5. The EV user (who has accepted the offer) charges the EV to fulfil the conditions of the offer. The charging event will be recorded in the blockchain.
6. The smart contract (running on the Ethereum blockchain) notices that the conditions of the accepted offer have been satisfied and sends the agreed amount of incentive tokens to the EVU on (the Ethereum) blockchain.
7. For accepted bids that failed to fulfil their requirements, the EVU should be “fined” by sending a corresponding transaction on the (Ethereum) blockchain.

Note that while in D4.2 it was mentioned that DSO bids could be generated automatically based on energy use or supply forecast, this is considered out of scope of the pilot, but is included in the use case to be considered for evaluation.

No evaluation results are available yet for this use case, however some preliminary planning has been undertaken in order to perform it. Some Interesting metrics to be considered for the evaluation include:

- Latency of various transactions
- Cost of smart contracts (e.g. gas, in case a public ledger is used)
- Whether the events can be reported to the system in a secure and authenticated manner
- User experience of the FM and the EVU, based on various offer options and mechanisms (e.g., auction, immediate pricing, etc.)

5.3 Decentralised Energy Data Exchange

The initial evaluation of the decentralised energy data exchange use case consists of two directions:

- The first investigates the case where a smart contract uses smart meter measurements to calculate a discount.
- The second investigation considers a model that captures the cost tradeoffs related to the frequency with which hashes of the smart meter measurements are recorded on a public blockchain.

For the first direction, we investigate the gains of utilizing two blockchains. We have two scenarios. The first scenario uses a Public Ethereum (Rinkeby), while the second one extends the structure with a permissioned blockchain (Private Ethereum). In the first case, we have one smart contract to keep records of measurements and calculate discounts (Public Ethereum). The second case uses the same contract in the permissioned blockchain and an extra contract in the Public ledger, for keeping hashes of smart-meter measurements. The comparison of the two scenarios, is in terms of the gas cost and delay.

The second direction considers the two blockchain cases discussed above and presents numerical investigations that illustrate how the cost tradeoffs depend on various system parameters.

5.3.1 Smart contract for flexible energy services

We first investigate, with experiments on a private Ethereum network and the Rinkeby public Ethereum testnet, the performance when smart meter measurements are recorded by a smart contract, which periodically computes a discount based on the recorded measurements. The components involved are the following:

- Smart Meter Component: This component submits smart meter measurements to the Smart Meter Contract.
- Smart Meter Contract: Records smart meter measurements and periodically computes a discount based on the recorded measurements. This component is executed either on a private Ethereum network (scenario 2) or on the Rinkeby public Ethereum testnet (scenario 1).
- Hash Recording Component: This component performs an interledger function for the second scenario that involves periodically recording hashes of the average consumption and the calculated discount on the public ledger.
- Hash Recording Contract: This is the smart contract responsible for recording hashes in the second scenario.

The records that are submitted by the Smart Meter Component to the corresponding contract have the general form [consumption, time unit], indicating the amount of energy consumption per time unit.

Smart contracts execute code that implements business logic and rules in a transparent manner. The smart contract in the scenarios we investigate performs a simple operation that periodically calculates a discount based on the smart meter measurements. We investigate two scenarios that differ in where the smart contract is executed:

Scenario 1: Smart contract running on a public ledger. In this scenario the smart contract is executed on a public Ethereum blockchain, namely the Rinkeby public Ethereum testnet.

The UML diagram for this scenario is shown in Figure 22, below.

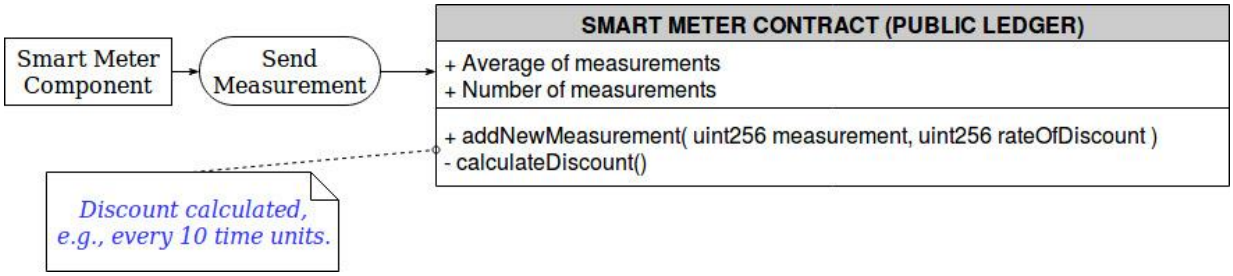


Figure 22. UML component diagram for the first scenario where a smart contract handling smart meter measurements runs on a public ledger

The Smart Meter Component sends measurement records to the Smart Meter Contract, running on a Public Ethereum network (Rinkeby). This smart contract has two main functions. The first (addNewMeasurement) is a public function that is called by the Smart Meter Component whenever it wants to record a measurement. The second (calculateDiscount) is an internal function that calculates a discount, after a standard number of measurements.

Scenario 2: Recording only hashes on a public ledger. In this scenario the smart contract that records the smart meter measurements and periodically calculates a discount is executed on a private Ethereum network. When the discount is calculated, a hash of the discount and the corresponding average consumption is recorded on the Rinkeby public Ethereum testnet.

The UML diagram for this scenario is shown below.

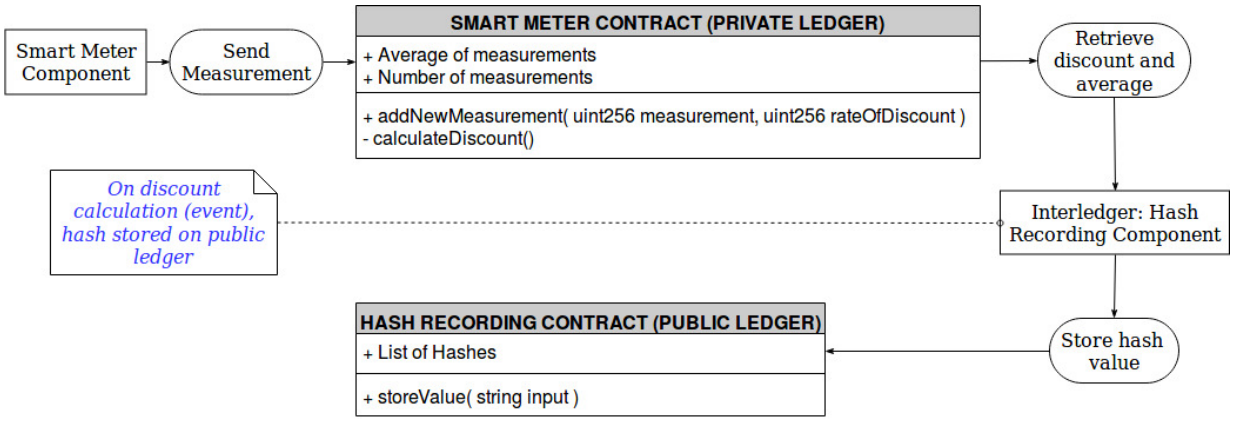


Figure 23. UML component diagram for the second scenario, where only hashes are stored on a public ledger

Recording of hashes is performed by an interledger gateway. The frequency at which hashes are recorded on the public Ethereum network is assumed to be the same as the frequency that the smart contract running on the private Ethereum network calculates discounts. This is achieved by having the interledger gateway listen to events that are generated each time that the Smart Meter Contract running on the private Ethereum network calculates a new discount.

Table 18 below summarises the smart contract functions (written in Solidity the main Ethereum smart contract programming language).

Table 18. Smart contract functions for decentralised energy data exchange scenarios

Function	Arguments	Scenarios used
addNewMeasurement	-uint256 measurement -uint256 rateOfDiscount	Scenario 1 (Smart contract running on a public ledger) Scenario 2 (Recording only hashes on a public ledger)
calculateDiscount	(none)	Scenario 1 (Smart contract running on a public ledger) Scenario 2 (Recording only hashes on a public ledger)
storeValue	-string value	Scenario 2 (Recording only hashes on a public ledger)

5.3.1.1 Evaluation results

The figure below shows the EVM execution cost (gas) for the two scenarios described above. The results are from experiments executed for 30 time units. Smart meter measurements are recorded at a frequency of one record every time unit. For the specific execution one time unit was equal to one minute. However, we note that the results are independent of the actual value of the time unit.

We consider three values for the rate of discount, which correspond to the three columns in the figure: one discount calculation every 5, 10, and 30 time units.

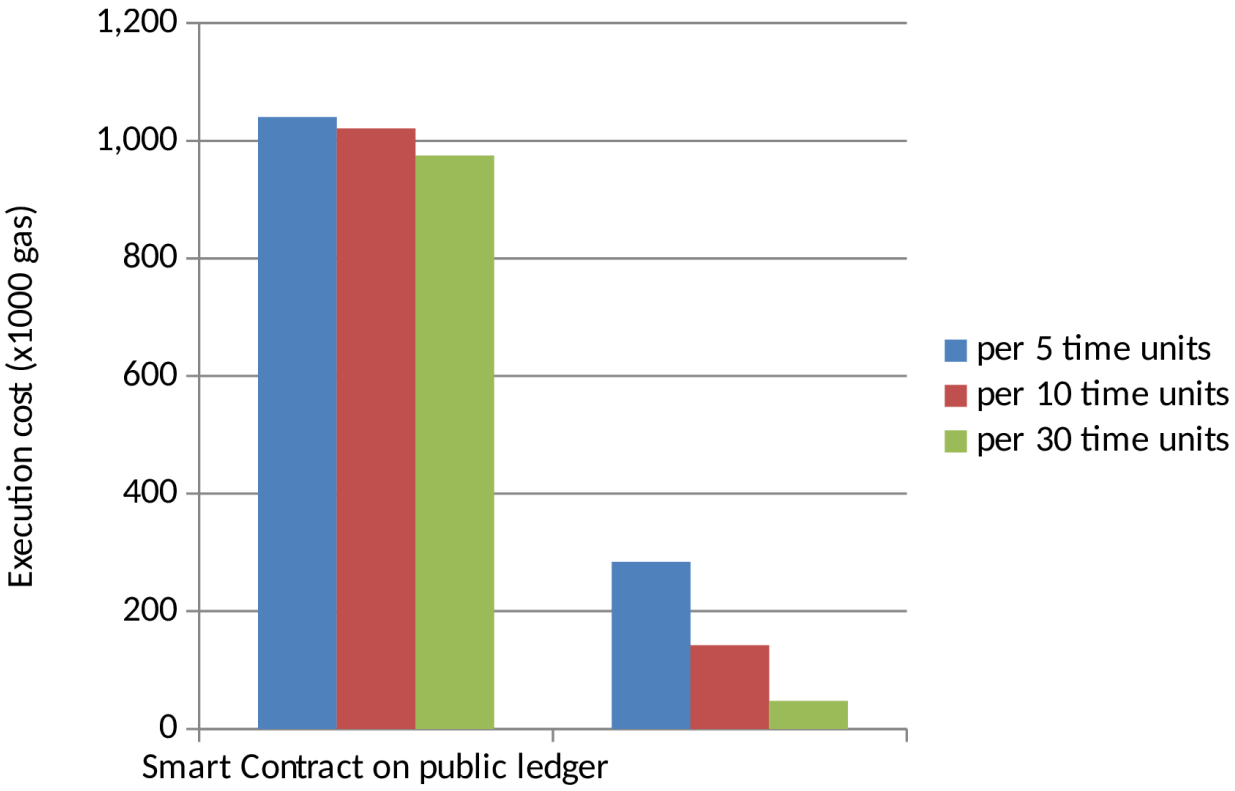


Figure 24. EVM execution cost for decentralised energy data exchange scenarios

The results show that, as expected, recording data measurements directly on a public blockchain has a very high execution cost. Note that such an approach also has low privacy.

Moreover, when a smart contract on a public ledger handles smart meter measurements, the time interval that discounts are calculated has a small influence on the total execution cost.

The cost when only hashes are recorded on the public ledger is significantly lower. Moreover, the results for this case verify that the execution cost when only hashes are recorded on the public chain is approximately inversely proportional to the time between consecutive hash recordings.

5.3.2 Hash recording frequency

In this subsection we present a simple model that captures the cost tradeoffs and the impact of the frequency with which hashes of the smart meter measurements are recorded on a public blockchain. The model applies to the second scenario presented in the previous subsection that corresponds to Figure 23. Specifically, the proposed model captures the following costs:

- Cost (monetary) for recording data on a public blockchain. Alternatively, this can refer to the cost for using a timestamping service.
- Cost for verifying that the data (smart meter measurements stored on the platform) is consistent with the hashes recorded on the public blockchain. This cost corresponds to the processing cost for performing the verification computations.
- Cost that quantifies the opportunity to modify or the impact from actually modifying the data from the time the last hash was recorded on the public chain until the time the next hash will be recorded.

The cost per unit of time for recording hashes on the public chain (or for using a timestamping service) can be expressed as a function P(f), which we assume is a linear function of the hash recording rate f. Alternatively, the function P(f) can be a concave function, if the incremental cost for recording hashes decreases as the hash recording rate f increases.

The verification cost can be expressed as $r_{ver}V(D)$ where r_{ver} is the rate of verification requests and V(D) is a function of the amount of data D that is required in order to perform verification. The verification cost is expressed as a cost per unit of time, similar to the hash recording cost. The shape of the function V() depends on how the hashes are computed. If the hash that is recorded on the public chain is computed by applying a hash function on all the data that has been produced since the last hash was recorded, then the verification cost V(D) is a linear function of the amount of data D produced between two consecutive hash recordings. D is equal to r_{data}/f, where r_{data} is the rate at which data is produced and f is the hash recording rate. On the other hand, if a Merkle tree is used to compute the hashes that are recorded on the public chain, then the verification cost V(D) is a logarithmic function of $D=r_{ver}/f$.

The cost that quantifies the opportunity to modify or the impact from modifying the data from the time the last hash was recorded on the public chain until the time the next hash will be recorded can be expressed as a function Q(D) of the amount of data D produced between consecutive hash recordings. The actual shape of Q() is application and scenario specific. Possible shapes are the following:

- Concave: Such a shape corresponds to the case where modifying data initially has large impact, which becomes smaller as more data is modified.
- Convex: This shape corresponds to the case where modifying data has a small impact up to some point, after which the impact increases when more data is modified.
- Sigmoid or stepwise function: In this case, initially modifying data has small impact (or zero impact in the case of a stepwise function), which increases sharply at some point. Then the impact from modifying additional data is smaller (or zero in the case of a stepwise function).

5.3.2.1 Evaluation results

In this subsection we present preliminary numerical results that involve the hash publication cost and the verification cost. The cost that quantifies the opportunity to modify data will be considered in future investigations.

Based on the model presented in the previous subsection, if c_{pub} is the cost for recording a hash on the public ledger, then the cost per unit of time for recording hashes on the public chain is $P(f)=f\, c_{pub}$. The verification cost is $r_{ver}\, V(r_{data}/f)\, c_{ver}$, where r_{ver} is the rate of verification requests, r_{data} is the rate at which data is produced, and c_{ver} is the cost for each verification. The cost c_{ver} represents the cost for a unit of computation that is necessary for performing verification. In the numerical results presented below, we assume that $c_{pub}/c_{ver} = 2$.

If the hash recorded on the public blockchain is computed by applying a hash function on all the data that has been produced since the last hash was recorded, then $V(r_{data}/f)$ is a linear function. On the other hand, if a Merkle tree is used, then $V(r_{data}/f)$ is a logarithm of r_{data}/f.

The verification cost $r_{ver}\, V(r_{data}/f)$ and the total cost $P(f) + r_{ver}V(r_{data}/f)$ as a function of the hash recording rate f in the case of a linear function V() are depicted by the lines containing the label "linear" in the figure below. We have assumed that $r_{data}/r_{ver}=1000$. Observe that the optimal hash recording frequency, which is the frequency with the lowest total cost, is approximately 16. Note that the time units for the hash recording frequency are the same time units that the rate r_{data} data is produced and the verification request rate r_{ver}. The verification cost and total cost for the case of a logarithmic function V() correspond to the lines with the label "log." Observe that in this case, the optimal frequency is smaller than 1, i.e., it is much smaller than the optimal frequency in the case of a linear function V(). The reason for the smaller optimal frequency in the case of a logarithmic function V() is because of the concave dependence of the corresponding verification cost on the hash recording frequency.

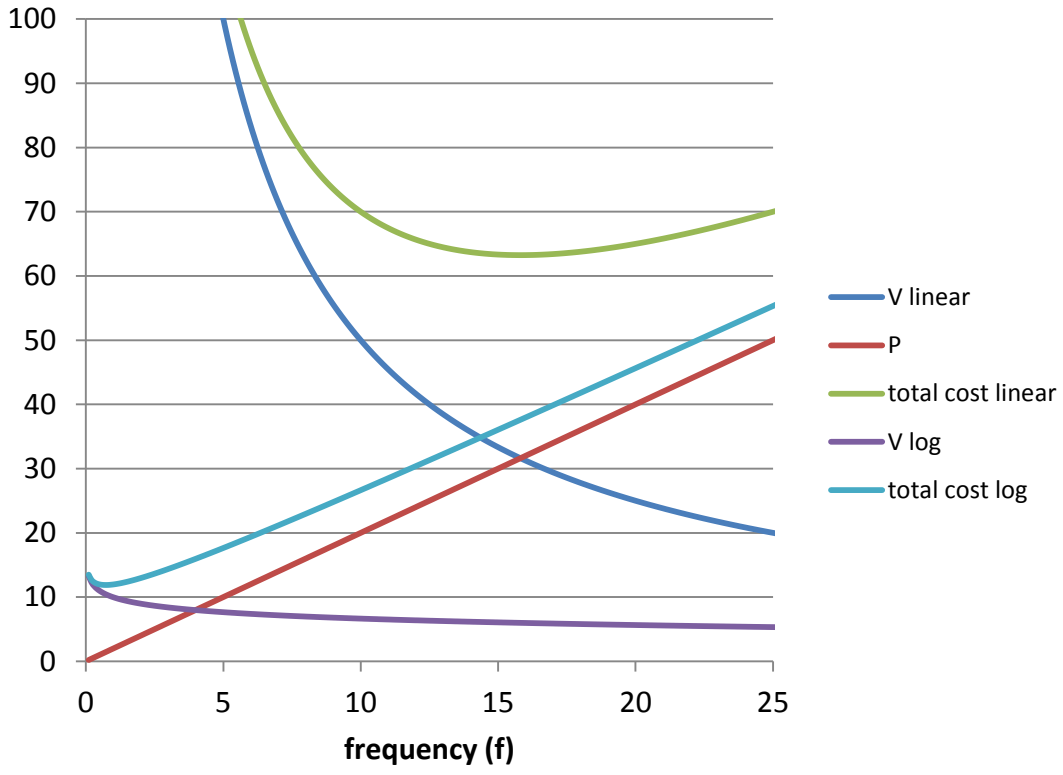


Figure 25. Cost as a function of frequency for a linear and logarithmic verification function V(f)

5.4 Mixed Reality Mobile Gaming

5.4.1 Overview

The mixed reality mobile gaming pilot will prototype a scavenger hunt location-based game. The goal of the game is to find the IoT device locations. The players will solve a riddle provided by the company or other players through a QR code. Solving the riddle will reveal the location of the next IoT device, in order for the user to go there, collect his points and download another riddle. This procedure continues until the last IoT device is reached. Players can bypass any challenge using the In-App tokens, which can be bought, awarded by viewing advertisements, awarded by completing a challenge or traded on the marketplace.

The entities identified within the pilot use case, including organizations and people are shown below:

- Game administrator/developer – Game company
- Game player
- Challenge designer
- Ads administrator – Advertisement company
- Point of Interest (POI) employee – POI company

IoT beacons will be used to provide the location of the IoT devices. The beacons will communicate with user smartphones using Bluetooth Low Energy (BLE). Furthermore, blockchain technology will be used to manage and record various actions and properties, e.g. player arriving at a beacon, points collection, trading tokens, etc. One of the main challenges, of this use case is to exploit scalable DLTs to support millions of active users per day with thousands of transactions per second in a cost-effective manner.

5.4.2 Events

The main events of this use case involving the game player are:

1. Game player downloads and installs the scavenger hunt application, granting all the appropriate permissions. Then, a private key is generated for the blockchain wallet and stored on the mobile.
2. Game player joins any challenge by scanning a QR code.
3. When the player visits the POI (IoT beacon), a task is shown to the player.
4. Player completes the challenge:
    a. By solving the task.
    b. By viewing ads.
    c. By paying with in-App tokens.
    d. By paying.
5. Player, having an approved user account, creates a new challenge.

The main event involving the game administrator/developer is:

- Game developer creates a new challenge.

The main event involving the POI employee is:

- The POI employee adds a new IoT beacon.

The main event involving the ads administrator:

- The ads administrator publishes an advertisement.

Adding a reward for completing a challenge or watching an advertisement video can be performed by the respective employees.

5.4.3 Emulation

To initially evaluate the performance of the mixed reality gaming pilot we will use an environment which emulates various aspects of this use case. Based on this emulation, we will be able to have both quantitative and qualitative evaluation. On the other hand, with the emulation environment we cannot investigate various other issues, such as game discovery, IoT devices scale, etc.

To start with, we emulate the scavenger hunt mobile application, as a Web application with a simple UI, implemented in React, which is a JavaScript library for building user interfaces. For emulation purposes, we assume that the user/player has already created a metamask account, thus he owns a private key for his blockchain wallet. Challenges are emulated as unique integer IDs. Challenge designers can create a new challenge by adding the *id* of the challenge in the Web App and clicking the appropriate button. When a designer adds a new challenge, the appropriate function of the smart contract is called and a transaction is sent to the blockchain network to be mined. Then, the player can select any challenge he wants by entering the *id* of the challenge. This process emulates the interaction with the IoT beacon. Afterwards, the player is able to click the “complete” button, which emulates that the player has solved the riddle. When the player completes the challenge, the smart contract automatically adds the calculated points to the player’s account. The player is able to skip any challenge by paying in ether, by paying in In-App tokens or by viewing advertisements. To emulate the In-App tokens, we create a smart contract that implements the ERC20 standard. Moreover, the player can obtain In-App tokens by paying in ether, by viewing advertisements, or by redeeming his points. Advertisements are emulated as smart contracts. When a user “watches” an advertisement the viewAds function of the ads contract is called. Only if this function returns true, i.e. the player watched the advertisement, then he gets the appropriate reward.

The initial evaluation of this use case is performed through various emulated scenarios with different configurations. Two scenarios are presented below.

5.4.3.1 Scenario 1

The first scenario investigates the case where only public Ethereum is used as the blockchain technology. The experiments in this scenario, took place on the Rinkeby (public Ethereum) testnet. The components of this scenario are:

- Web application: This component performs the interaction between the players and the blockchain technology.
- Ads smart contract: This component checks whether the user watched the advertisement or not.
- Tokens smart contract: This component creates and manages the In-App tokens.
- Game smart contract: This component implements all the required functionality of the game. First of all, it records all the challenges by their unique IDs to the blockchain. It also records a mapping between the players and the challenges and whether the player has completed the challenge or not. It automatically calculates the points that the player should be rewarded when he completes a challenge. This contract implements three functions that give the opportunity to players to skip any challenge by the aforementioned ways. Finally, it calls the other two contracts to offer rewards to players.

The UML diagram for this scenario is shown below.

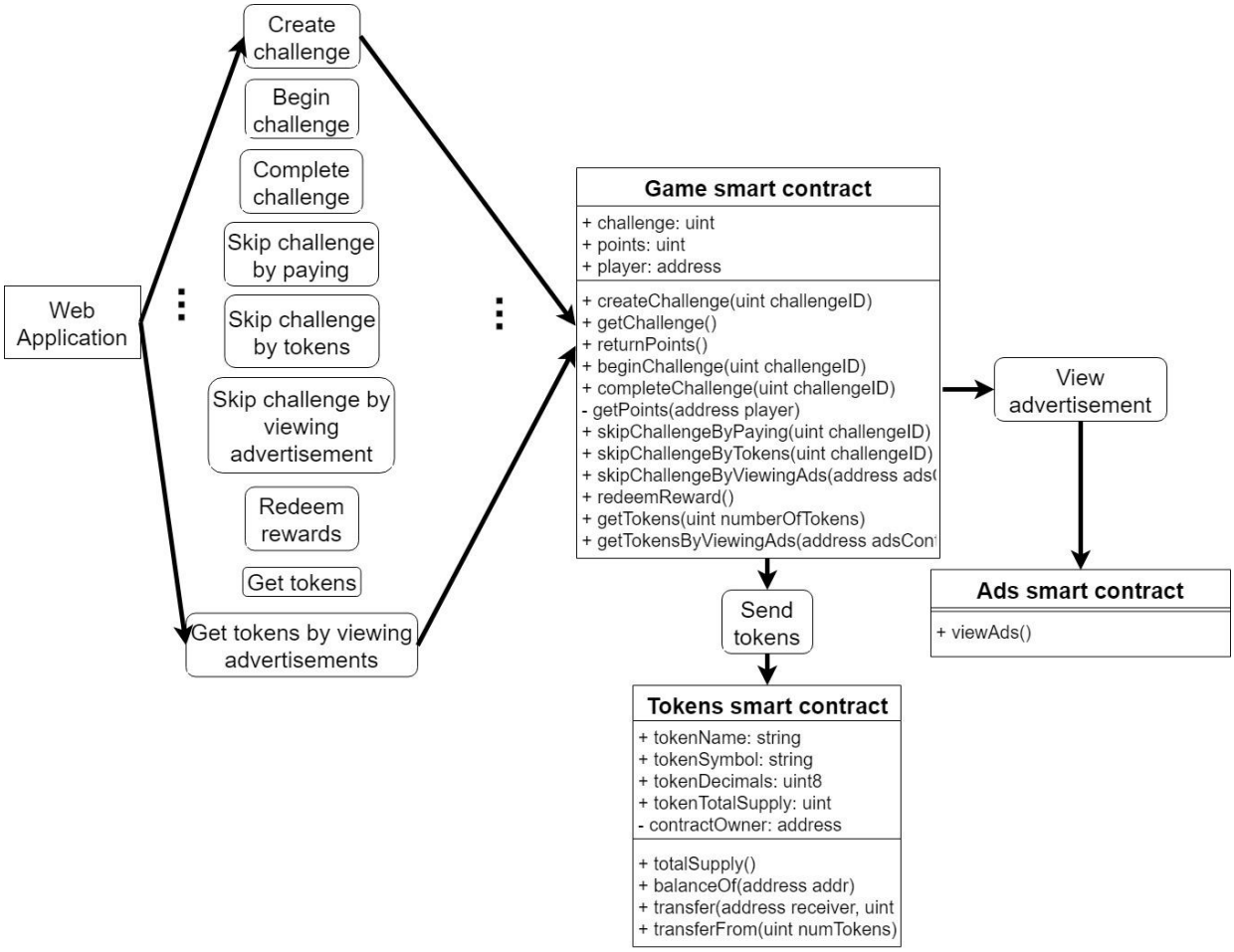


Figure 26. UML component diagram for the first mixed reality mobile gaming scenario

5.4.3.2 Scenario 2

The second scenario investigates the gains from utilizing two (different) blockchains instead of one. In this scenario, the first blockchain is a public blockchain and the second one is a permissioned blockchain. We assume that the public blockchain is a public Ethereum and the permissioned blockchain is a private Ethereum chain. The components of this scenario are the same ones shown above for Scenario 1 plus the one shown below.

- Interledger gateway: This component is listening to events on both Ethereum blockchains (public and private), which are generated each time the player invokes a function of the game smart contract that needs to perform an action involving the In-App tokens or the advertisements.

The game smart contract is executed on the private Ethereum network. On the other hand, the tokens smart contract and the ads smart contract are executed on the Rinkeby testnet.

The UML diagram for this scenario is shown below.

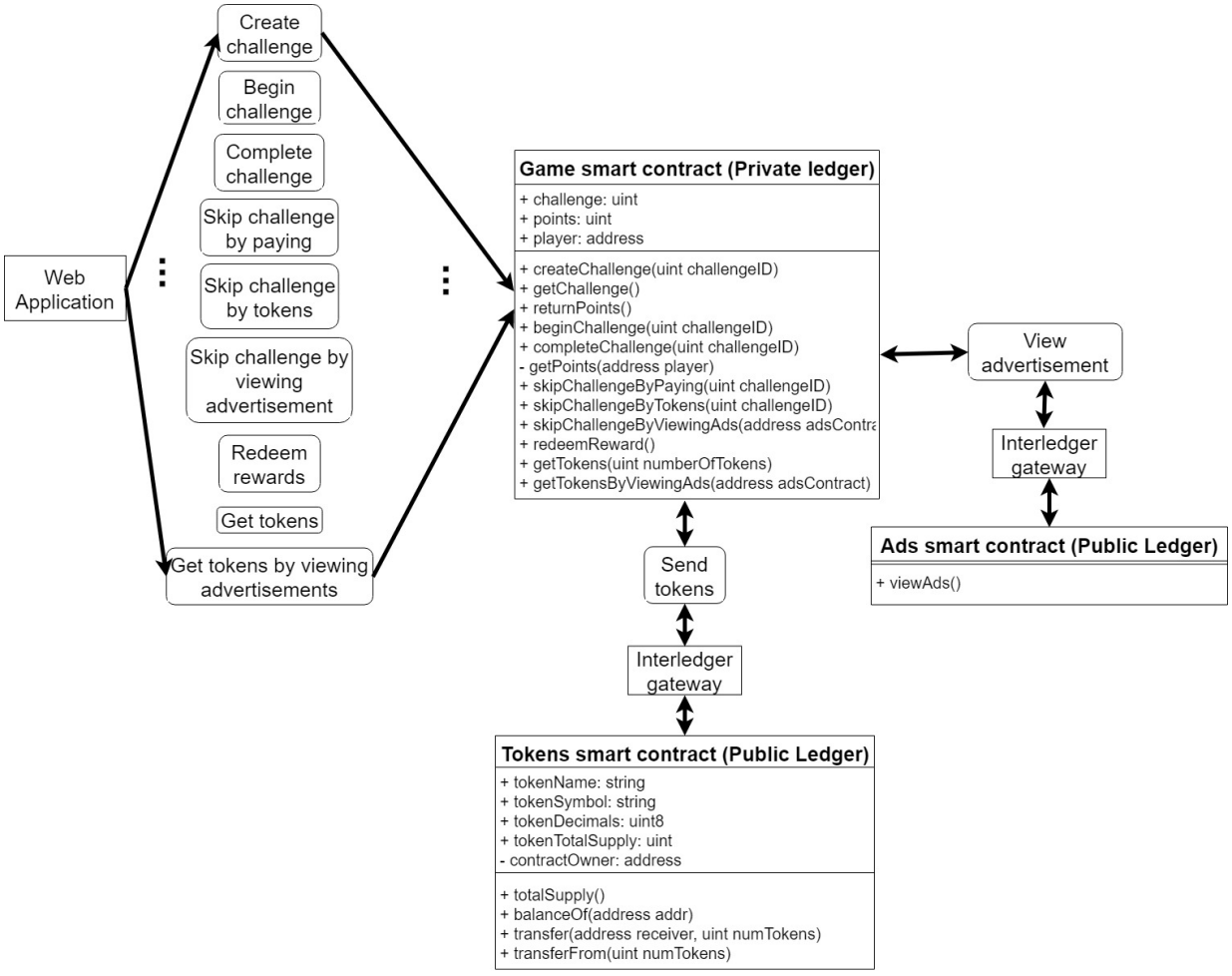


Figure 27. UML component diagram for the second mixed reality mobile gaming scenario

5.4.4 Evaluation Results

The initial evaluation of the mobile gaming pilot use cases begins with the comparison of the two aforementioned scenarios, in terms of the gas cost. The following table shows the EVM execution cost in terms of gas for the scenarios described above.

Table 19. EVM execution cost for mixed reality mobile gaming scenarios

Function	Scenario 1 cost (gas)	Scenario 2 cost (gas)
Add challenge	47050	N/A
Begin challenge	52432	N/A
Complete challenge	53529	N/A
Skip challenge by paying	61867	N/A
Skip challenge by paying in In-App tokens	63877	33438
Skip challenge by viewing advertisements	53926	21462
Get tokens by paying	44199	35274
Get tokens by viewing advertisements	37981	56736
Redeem rewards	36618	35274

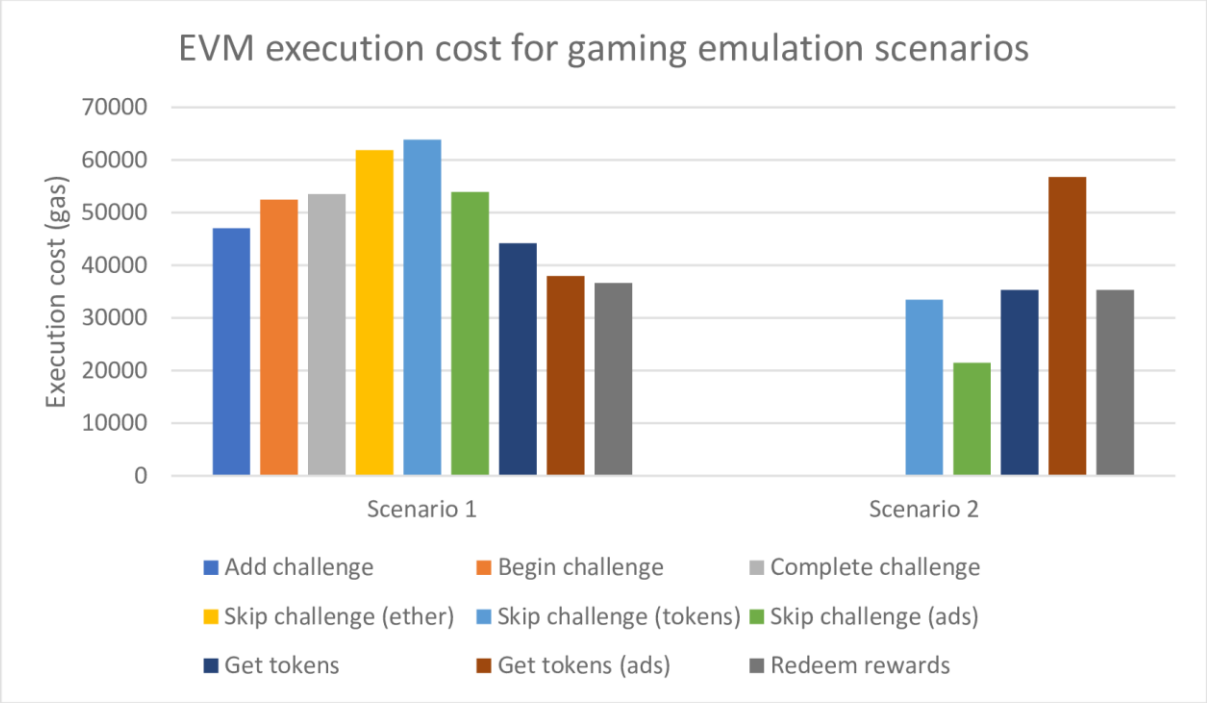


Figure 28. EVM execution cost for mixed reality mobile gaming scenarios

As we can see from the results, using only a public blockchain is very costly. On the other hand, the execution cost of using the public instance of Ethereum only for managing the In-App tokens, is far lower.

6. Business oriented evaluation

SOFIE is about *Secure Open Federation* of IoT Systems (silos) so that they can exchange data under their own rules and "voluntarily" cooperate to achieve various goals. Most of the rules, impediments, but also incentives and opportunities are about business aspects, rather than technical issues. Therefore, an evaluation of the SOFIE architecture and the mechanisms being developed from a business perspective is necessary and critical. In addition to the aspects discussed earlier, such as of a distributed system and an open (business) platform, other business aspects are discussed and evaluated here for a start and will be further refined and completed in the next WP4 deliverable, D4.4.

Of special interest is an investigation of whether and under what conditions a Federated Open Platform (System), without a pre-set leader governing the system can successfully arise, develop and prosper and whether the constituting (sub-)systems also grow and prosper and how potential gain are distributed across the constituents. We use a System dynamics approach to investigate these questions. An initial investigation is provided below, but the work continues, and further results are expected to be provided in D4.4.

6.1 On federation

One can consider a group of component systems that are governed by either a single central authority or distributed governance, in which each component system serves its own interests (but they can also demonstrate altruism, or desire for cooperation, sometimes against their narrow defined interests). The analogy in real life is with a group of states/countries, like the European Union or the United States of America. An example of a federated system in information technology is the interconnection of Social Networks.

The design of a federated system has to cover the following aspects:

- **Business foundations:** Covers the context of defining fully customizable applications. If an application has access to the system through one specific component, it must be able to use (or not) services from the other components.
- **Service foundations:** Includes creation and resource management of a service. The creation of a service in a component might be simple; the difficulty lies in defining the right authorization and privacy schemes. Storing and replicating the service' s data is also a hard task.
- **Heterogeneous infrastructure:** Employ Servers (hardware infrastructure) in order to support a variety of devices and reach the appropriate resource allocation level.
- **Communications foundation:** Last, but not least, is the ability for the components to communicate. Components must be able to identify the services offered among each other. As we mentioned in the introduction, Social Networks have already solved this issue. An example is post publishing. An Instagram photo is already published in Facebook and Twitter, if the user allows it.

Our Proposed Interledger Protocol is based upon distributed and decentralized nodes. Each node is a permissioned/permissionless ledger (Hyperledger, Ethereum, Bitcoin). The solution covers Communications/Service foundation and the Heterogeneous infrastructure. The pilots show examples of Business Foundations.

6.2 Modelling SOFIE business platform growth and sustainability

In order to understand the growth and sustainability of federated digital business platforms, we first need to understand the key component causal feedback loops. We develop models for (decentralized) Ethereum platforms and DApps (Decentralized Apps) that have their core business logic parts run on top of such platforms, in the form of smart contracts.

Modelling the Ethereum Platform with System Dynamics

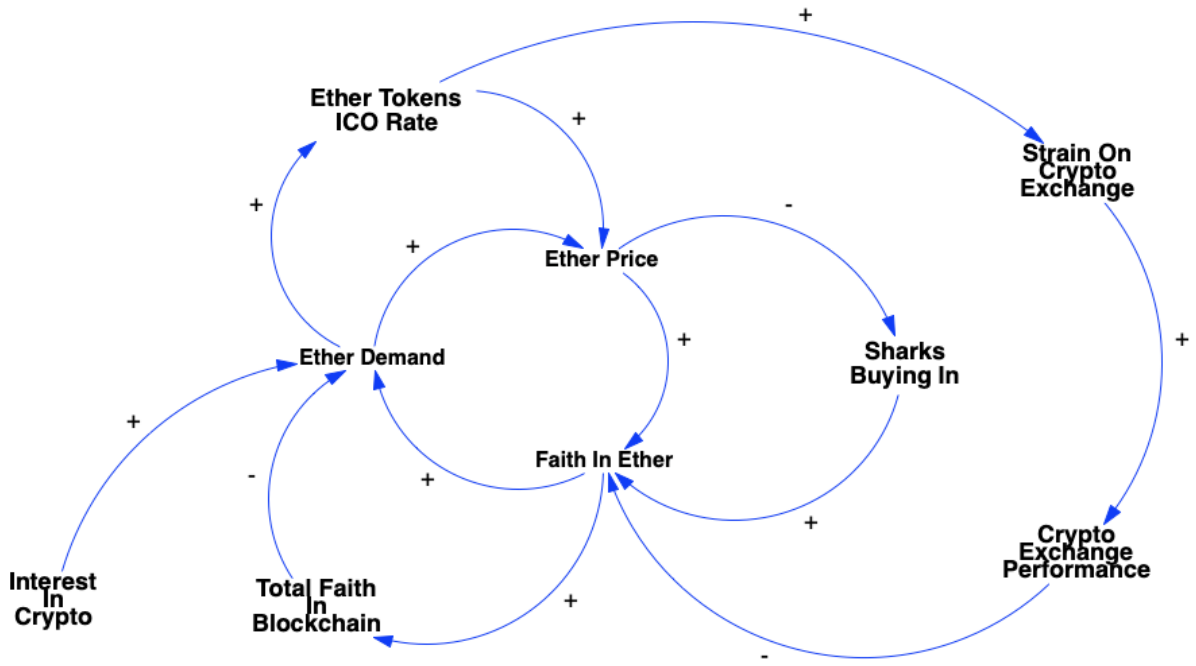


Figure 29. Ethereum platform sustainability causal feedback loops

There is a general interest in cryptocurrency, which in the current state of this model is assumed to be exogenous. This interest causes, among other cryptocurrencies, an increase in the demand for *Ether* (ETH), which is the token of the Ethereum platform that we use as our case study. An increase in the demand of the token causes an increase in its price, which in turn increases its attractiveness as this is perceived to be indicative of the token's steady successful growth. The more attractive the token, the more its demand, and this closes the reinforcing loop that depicts the network effects of our model.

As long as the demand of the token remains low, the more "sharks" are looking for an opportunity to buy as much of it as possible. This will give them the opportunity to inject money back to the platform and enhance the attractiveness of the token.

The total blockchain attractiveness is accumulated from all currently available cryptocurrencies in the market. It then has a negative causality to their demand, and this illustrates the balancing loop of the market's saturation. In other words, it models the natural limit for the demand of cryptocurrencies.

What makes the Ethereum platform differ from other cryptocurrency platforms is its unique feature of supporting other tokens. The higher the demand for Ether, the more people will seek opportunities to take advantage of its features. Hence, efforts to piggy back (or complement?) Ether's token creation protocol and launch Initial Coin Offering (ICO) companies rise. The more such companies enter the market, the higher the price of the Ether token.

However, the more ICO companies in Ether platform, the more transactions take place. Consequently, this strain in crypto exchange makes it less feasible to validate these transactions in a timely manner. Thus, we have lags in the platform's performance, which in turn has a negative effect on the platform's attractiveness.

Modelling a distributed application with System Dynamics

Here we present a model of an example application. We present a model of CryptoKitties, which is a decentralized game running on top of the Ethereum blockchain.

CryptoKitties[29] are collectable and breed-able property that reside and function as part of the Ethereum [Nak08], [Woo18]. They are the first widely deployed commercial Ethereum application, which broke into the public consciousness towards the end of 2017 and the beginning of 2018. We model the CryptoKitties on the Ethereum blockchain business platform. CryptoKitties tests the idea of digital scarcity. Before the invention of blockchains, digital scarcity could only be created via means of centralised control. However, centralised control is unable to create real scarcity in the sense that everyone needs to trust the central operator to actually produce said scarcity. If the central operator stops producing this scarcity for any reason, the users might not even know about it, much less prevent it.

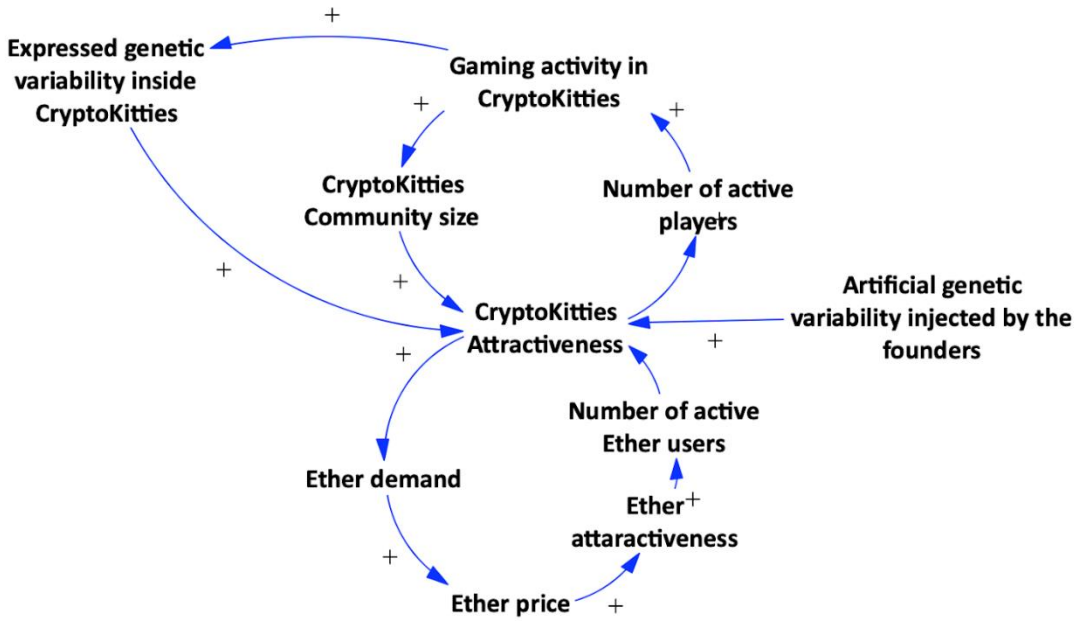


Figure 30. CryptoKitties application sustainability causal feedback loops

From the model we can see that there is a mutually reinforcing network effect between the two networks of CryptoKitties and the Ethereum platform. The more CryptoKitties users, the more the platform benefits. And the more there are platform users, the more there are potential CryptoKitties players.

The role of the CryptoKitties application founders is also depicted in the model. Founders allowed themselves a time limited role in creating more genetic variability so that not all the variability inherent in generation 0 Kitties was present at the launch of the application. When additional genetic diversity can be injected later, it can be used as a marketing tool by timing the injection so that it produces the best economic results.

[29] CryptoKitties: Collect and breed digital cats! Available online: https://www.cryptokitties.co

Combining Platform and Application models in the SOFIE architecture

Here we apply our System Dynamics understanding to the SOFIE approach and, more specifically, its pilot cases.

In order for the model to be meaningful, we first need to understand the relevant DLT platforms and have a good understanding of the business level functionality of the pilot case. By modelling the interactions between DLTs and application business functionality, we arrive at the business model of the pilot.

Next, we develop a preliminary model of the food-chain pilot case to evaluate the sustainability of the created business platform through its causal feedback loops.

Food Chain Pilot Business Platform Model

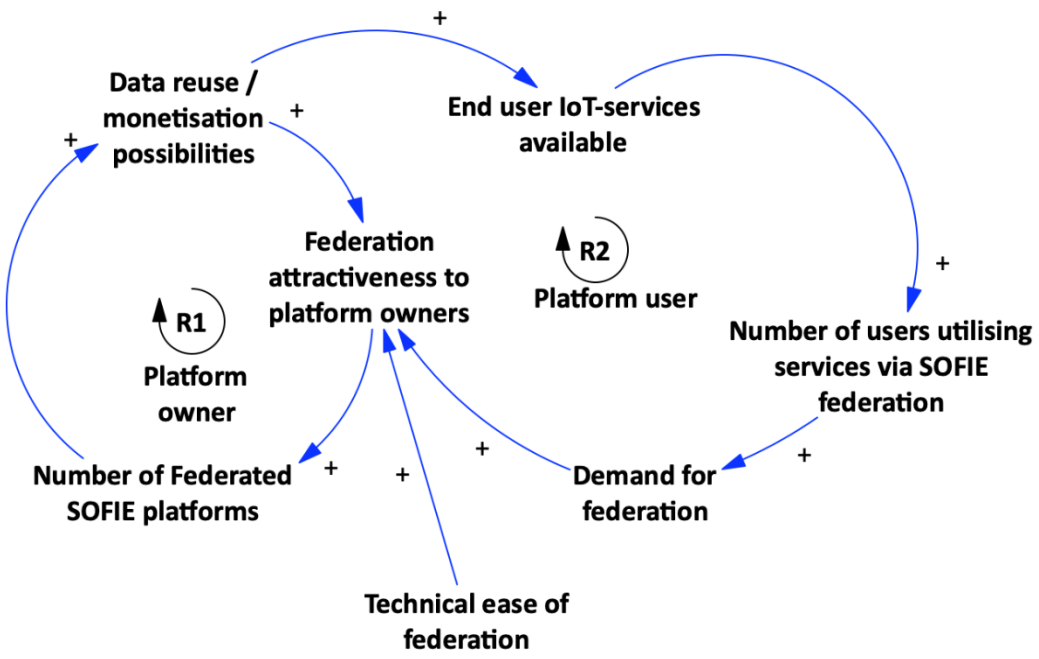


Figure 31. Federation growth and sustainability with platform owner and platform user reinforcing causal feedback loops

System Dynamics in Business Platforms Work Outlook

Up to this point the System Dynamics models produced and presented have been qualitative. The next step would be to produce quantitative models by determining some parameters, measuring some and making assumptions for others, and then fitting these to stochastic models and also using simulation, as in [AEN19] (e.g., see Fig. 2 of [AEN19]). The outcome of this next step can show under what conditions (parameter values) the business platform can thrive, to what extent, and even the impact and gains of the various constituent players.

7. Conclusion

This First Architecture and System Evaluation Report mainly provided initial component and technique evaluation results and also illustrations of the type of evaluation that has been and will be performed within the project. We expect to provide more concrete and integrated results in the next WP4 deliverable, D4.4: Second Architecture and System Evaluation Report (M28), after the Federation Architecture (D2.4), Initial Platform (D5.2), Federation Framework (D2.5) and the Business Platforms (D3.3, pilot release) have matured.

The evaluation methodologies were many and diverse, from simple presentation of arguments and qualitative evaluation, through modelling, analytical evaluation and simulation, to implementation and measurements in real components and systems. Since pilots have a central position in the SOFIE project, an important evaluation direction is inspired from each pilot, considering the more general use case represented by the pilot, generalizing from the specific choices made in the pilots, and evaluating the many potential alternatives around them.

In addition to the methodologies and tools for evaluation, the questions to be answered (i.e., the targets of the evaluation) are diverse. They range from traditional performance metrics, which typically have limited generality as they have to refer to fully specified systems, to more general questions such as security analysis, robustness, usability and even business analysis. It is therefore even more obvious that the tools used and to be used for evaluation must be diverse and applied at very different abstraction levels and under different assumptions.

Because of the diverse goals and evaluation methodologies, in this deliverable we have five main sections with different style and tools and the following structure. In Section 2, we performed a high-level qualitative evaluation of the architecture, focusing on the desired properties and general approaches and techniques to achieve them.

Section 3 presented an initial SOFIE Federation Framework component evaluation, starting here with three main components: (a) interledger, (b) privacy and data sovereignty, and (c) identification, authentication and authorization.

Section 4 covered IoT resource access detailed evaluation and it goes to much more depth and considers many more alternatives and their tradeoffs. The other sections do not intimately depend on it, but demonstrates a more detailed design and evaluation and many useful alternatives for the IoT world.

Section 5, on Evaluation Scenarios, followed the SOFIE pilots, generalized them into pilot inspired use cases by including alternatives not selected in the SOFIE pilots, abstracted them out to appropriate degree and used emulation and simulation to consider the various tradeoffs of many potential alternative design decisions and their impact. It thus provided guidance for specific designs related to these use cases under various assumptions and constraints.

Section 6 addressed decentralized Business Platforms evaluation through an illustrative application of a System Dynamics approach. It reflects preliminary work, qualitative only for now, but sets the stage for our future efforts in this area.

Note that this third WP4 (Evaluation) deliverable is being produced in parallel with D2.4 (Federation Architecture, 2nd version) and D5.2 (Initial Platform Validation) and before D2.5 (Federation Framework, 2nd version — August 2019) and D3.3 (Business platforms, pilot release — September 2019). The design of the architecture, the business platforms, the pilots, and the implementation of the federation framework components and the systems are in flux. Therefore, this deliverable focuses more on evaluating key aspects of the architecture, framework components, and pilot inspired use cases, rather than providing an evaluation of a specific data point in the development of one or many systems.

References

[AEN19] E. Arzoglou, T. Elo, and P. Nikander, "The Case of iOS and Android: Applying System Dynamics to Digital Business Platforms," Proc. ICCS 2019. In: J. Rodrigues et al. (eds) *Computational Science – ICCS 2019*, Lecture Notes in Computer Science, Vol. 11540, Springer, Cham.

[AJM14] R. Azarderakhsh, K.U. Järvinen, and M. Mozaffari-Kermani, "Efficient Algorithm and Architecture for Elliptic Curve Cryptography for Extremely Constrained Secure Applications," *IEEE Transactions on Circuits and Systems*, Vol. 61, No. 4, pp. 1144–1155, 2014.

[Ber06] D.J. Bernstein, "Curve25519: New Diffie-Hellman Speed Records," Proc. 9th International Conference on Theory and Practice of Public-Key Cryptography (PKC 2006), pp. 207–228, New York, NY, USA, April 2006.

[Ber+12] D.J. Bernstein et al., "High-speed high-security signatures," *Journal of Cryptographic Engineering*, Vol. 2, No. 2, pp. 77–89, Springer, 2012.

[But16] V. Buterin, "Chain Interoperability," R3 Report, September 2016. Available online: https://www.r3.com/wp-content/uploads/2017/06/chain_interoperability_r3.pdf

[Cle+14] R. de Clercq, L. Uhsadel, A. Van Herrewege, and I. Verbauwhede, "Ultra low-power implementation of ECC on the ARM Cortex-M0+," Proc. 51st ACM/EDAC/IEEE Design Automation Conference (DAC), San Francisco, CA, USA, June 2014.

[Del+16] A. Delignat-Lavaud, C. Fournet, M. Kohlweiss, and B. Parno, "Cinderella: Turning Shabby X.509 Certificates into Elegant Anonymous Credentials with the Magic of Verifiable Computation," Proc. IEEE Symposium on Security and Privacy (SP), May 2016.

[Dül+15] M. Düll et al., "High-speed Curve25519 on 8-bit, 16-bit, and 32-bit microcontrollers," in *Designs, Codes and Cryptography*, Vol. 77, No. 2-3, pp. 493-514, Springer, 2015.

[Fot+16] N. Fotiou, T. Kotsonis, G.F. Marias, G.C. Polyzos, "Access control for the Internet of Things," Proc. ESORICS International Workshop on Secure Internet of Things, pp. 29–38, 2016. Available online: https://mm.aueb.gr/publications/SIoT.pdf

[Fot+19] N. Fotiou, I. Pittaras, V.A. Siris, S. Voulgaris, G.C. Polyzos, "Secure IoT access at scale using blockchains and smart contracts," Proc. 8th IEEE WoWMoM Workshop on the Internet of Things: Smart Objects and Services (IoT-SoS),Washington DC, USA, June 2019.

[FSP18] N. Fotiou, V.A. Siris, G.C. Polyzos, "Interacting with the Internet of Things using Smart Contracts and Blockchain Technologies", Proc. Security, Privacy, and Anonymity in Computation, Communication, and Storage (SpaCCS 2018), Melbourne, Australia, December 2018.

[Ger+18] S. Gerdes et al., "An architecture for authorization in constrained environments," IETF Draft, October 2018.

[Glo18] Global Platform, "TEE System Architecture v1.2," December 2018. Available online: https://globalplatform.org/specs-library/tee-system-architecture-v1-2/.

[Har+12] D. Hardt et al., "The OAuth 2.0 Authorization Framework," RFC 6749, Standards Track, IETF, October 2012.

[HS13] M. Hutter and P. Schwabe, “NaCl on 8-Bit AVR Microcontrollers,” Proc. International Conference on Cryptology in Africa, published in *Progress in Cryptology – AFRICACRYPT 2013*, A. Youssef, A. Nitaj, and A.E. Hassanien (eds), Lecture Notes in Computer Science, Vol. 7918, Springer, 2013.

[JBS15a] M. Jones, J. Bradley, and N. Sakimura, “JSON Web Token (JWT),” RFC 7519, Standards Track, IETF, May 2015.

[JBS15b] M. Jones, J. Bradley, and N. Sakimura, “JSON Web Signature (JWS),” RFC 7515, Standards Track, IETF, May 2015.

[JBT16] M. Jones, J. Bradley, and H. Tschofenig, “Proof-of-Possession Key Semantics for JSON Web Tokens (JWTs),” RFC 7800, Standards Track, IETF, April 2016.

[Kor+19] Y. Kortesniemi, D. Lagutin, T. Elo, N. Fotiou, “Improving the Privacy of IoT with Decentralised Identifiers (DIDs),” *Journal of Computer Networks and Communications*, Hindawi, 2019.

[KL08] J. Katz and A. Y. Lindell, “Aggregate message authentication codes,” Proc. The Cryptographers’ Track at the RSA conference on Topics in Cryptology (CT-RSA), 2008.

[Lag+19] D. Lagutin, Y. Kortesniemi, N. Fotiou, V.A. Siris, “Enabling Decentralized Identifiers and Verifiable Credentials for Constrained Internet-of-Things Devices using OAuth-based Delegation,” NDSS Workshop on Decentralized IoT Systems and Security (DISS), San Diego, CA, USA, February 2019.

[Nak08] S. Nakamoto, “Bitcoin: A Peer-to-Peer Electronic Cash System,” White Paper, October 2008. Available online: https://bitcoin.org/bitcoin.pdf

[PD16] J. Poon and T. Dryja, “The Bitcoin Lightning Network: Scalable off-chain instant payments,” White Paper, January 2016.
Available online: https://lightning.network/lightning-network-paper.pdf

[Ree19] D. Reed et al., “Decentralized Identifiers (DIDs) v0.13: Data Model and Syntaxes for Decentralized Identifiers,” Draft Community Group Report, W3C, June 2019. Available online: https://w3c-ccg.github.io/did-spec/

[Sei+16] L. Seitz et al., “Use Cases for Authentication and Authorization in Constrained Environments,” RFC 7744, IETF, January 2016.

[Sei+19] L. Seitz et al., “Authentication and Authorization for Constrained Environments (ACE) using the OAuth 2.0 Framework (ACE-OAuth),” IETF Draft, March 2019.

[Sho01] V. Shoup, “A proposal for an ISO standard for public key encryption,” Cryptology ePrint archive, Report 112, 2001. Available online:
https://eprint.iacr.org/2001/112

[Sir+19a] V.A. Siris, D. Dimopoulos, N. Fotiou, S. Voulgaris, G.C. Polyzos, “Interledger Smart Contracts for Decentralized Authorization to Constrained Things,” Proc. 2nd Workshop on Cryptocurrencies and Blockchains for Distributed Systems (CryBlock 2019), in conjunction with IEEE INFOCOM 2019, Paris, France, April–May 2019. Available online: http://mm.aueb.gr/publications/2019-CryBlock.pdf

[Sir+19b] V.A. Siris, D. Dimopoulos, N. Fotiou, S. Voulgaris, G.C. Polyzos, “OAuth 2.0 meets Blockchain for Authorization in Constrained IoT Environments,” Proc. 5th IEEE World Forum on Internet of Things, Limerick, Ireland, 2019.
Available online: http://mm.aueb.gr/publications/2019-WF-IoT.pdf

[Sir+19c] V.A. Siris, D. Dimopoulos, N. Fotiou, S. Voulgaris, G.C. Polyzos, "IoT Resource Access utilizing Blockchains and Trusted Execution Environments," Proc. Global IoT Summit, Aarhus, Denmark, 2019.
Available online: http://mm.aueb.gr/publications/GIoTS2019_IEEE.pdf

[Sir+19d] V.A. Siris, P. Nikander, S. Voulgaris, N. Fotiou, D. Lagutin, G.C. Polyzos, "Interledger Approaches," *IEEE Access*, 2019 (to appear).

[SMS07] B. Sunar, W.J. Martin, and D.R. Stinson, "A Provably Secure True Random Number Generator with Built-In Tolerance to Active Attacks," *IEEE Transactions on Computers*, Vol. 56, Issue 1, pp. 109-119, January 2007.

[Spo+18] M. Sporny et al., "Verifiable Credentials Data Model 1.0: Expressing verifiable information on the Web," Draft Community Group Report, W3C, January 2018.

[VV15] V. Vogelsteller and B. Vitalik, "ERC-20 token standard," Tech. Rep., 2015.
Available online: https://github.com/ethereum/EIPs/blob/master/EIPS/eip-20.md

[Woo18] D.G. Wood, "Ethereum: A Secure Decentralised Generalised Transaction Ledger," Ethereum Yellow Paper, December 2018. Available online: https://ethereum.github.io/yellowpaper/paper.pdf