



University of
Zurich^{UZH}

A Minimal-Solution Method toward Edge-to-Blockchain Transactions

Andreas Knecht, Tim Strasser
Zurich, Switzerland
Student IDs: 11-916-152, 12-742-573

Supervisors: Eder John Scheid, Bruno Bastos Rodrigues
Date of Submission: November 6, 2019

Abstract

In diesem Masterprojekt erweitern wir ein existierendes Produkt, welches auf Internet of Things (IoT) und Blockchain (BC) aufbaut, um die Sicherheit in der Lieferkette von pharmazeutischen Produkten zu erhöhen. Die Arbeit zeigt einen Proof of Concept (PoC), der auf zwei existierenden Projekten aufbaut: *modum.io*, das ein IoT Gerät und unterstützende Systeme umfasst, mit dem Ziel, die Sicherheit in Logistikprozessen der Pharmabranche zu erhöhen, indem Sensordaten unveränderbar in der BC gespeichert werden, und *BC4CC*, das darauf abzielt, die Anbindung von Edge Geräten an mehrere BCs flexibler und intelligenter zu machen. Der PoC umfasst ein funktionierendes System, das Transaktionen für eine private und eine öffentliche BC direkt auf dem IoT Gerät signiert und dessen Hardware Security Modul (HSM) nutzt. Eine Diskussion der Implikationen der benötigten Datenflüsse auf Sicherheit und Benutzerfreundlichkeit zeigt sowohl Vor-, wie auch Nachteile auf.

In this master project we extend an existing Internet of Things (IoT) and Blockchain (BC) based project for security improvements in the pharmaceutical supply chain. A proof of concept (PoC) is built upon two existing projects: *modum.io*, which provides an IoT device, as well as interfacing systems with the goal of improving security in pharmaceutical logistics processes by immutably storing sensor data from an IoT device in the BC and *BC4CC* which aims at making the interface from edge devices to multiple BCs more flexible and intelligent. The PoC provides a working system that signs transactions for a public and a private BC directly on the IoT device, leveraging its hardware security module (HSM). A discussion of the implications of the required data flows concerning security and usability identifies both advantages and drawbacks.

Acknowledgments

We like to thank both supervisors, Eder John Scheid and Bruno Bastos Rodrigues for their continuous support, guidance, feedback, valuable input and related work with BC4CC throughout the whole development of this master project. Further, we'd like to thank modum.io for their support and supportive contributions coming from a practical and industrial standpoint. We'd also like to express our gratitude towards the Communication Systems Group and Prof. Dr. Burkhard Stiller for the opportunity to conduct this master project.

Contents

Abstract	i
Acknowledgments	iii
1 Introduction	1
1.1 Motivation	1
1.2 Description of Work	2
1.3 Thesis Outline	2
2 Related Work	5
2.1 Data Storage in the Blockchain	5
2.2 Private versus Public Blockchains	6
2.3 Ethereum	6
2.4 Hyperledger Sawtooth	8
2.5 BC4CC	9
2.6 Modum	11
2.7 MODSense T Temperature Logger	12
2.8 Bluetooth Low Energy	13
2.9 ISO7816	14
2.10 ECDSA	15

3 Design	17
3.1 Decisions of Blockchains	18
3.2 Architecture	18
3.3 Transaction Preparation	19
4 Implementation	21
4.1 Mobile Application	21
4.2 Firmware	24
4.3 Applet	27
4.4 Smart Contract	28
5 Evaluation	31
5.1 Security and Usability	31
5.1.1 Checking of external values	34
5.2 Power Measurements	35
5.3 Challenges	36
6 Summary and Conclusions	39
6.1 Future Work	39
Abbreviations	45
List of Figures	45
List of Tables	47
A Installation Guidelines	51
A.1 Mobile Application	51
A.2 Smart Contract	51
A.3 Firmware	52
A.4 Applet	52
B Contents of the CD	53

Chapter 1

Introduction

In the last few years, Blockchain (BC)-enabled technologies have become more and more important as a part of the development towards more intelligent supply chains and logistics [1], ranging from storing ownership history of goods in a non-repudiable way in order to support provenance tracking to Smart Contracts (SC) enabling automatically triggered events such as notifications about violations of legal conditions or the automatic conduction of payments upon delivery. Necessary for those developments is that real-life events throughout the supply chain can be collected in a reliable way, which is made possible by technological advancements of Internet of Things (IoT) devices concerning connectivity, battery life, processing power and sensor capabilities. With the amount of collected events further increasing other aspects have become equally important, such that real-life events can not be altered (integrity) until they are immutably stored in the BC and seen as “truth” and that there exists no doubt regarding the origin of the event (authenticity). Focusing on these aspects, this master projects presents a proof of concept (PoC) for leveraging cryptographic functionalities of an IoT hardware security module to prepare signed raw transactions on-device for later publication on both a public and a private BC.

1.1 Motivation

The goal of this master project is to contribute an addition to two already existing projects that are currently work in progress. Both of these projects are in the field of using BC, IoT and Artificial Intelligence (AI) technology to develop a system that supports temperature monitoring in the medical drug distribution industry.

The first project is called BC-based Temperature Monitoring for Cold Chains in Medical Drug Distribution (BC4CC) [2]. One of the goals of the project is to offer an Application Programming Interface (API) that accepts temperature data measured during transport and stores them on one or multiple BCs, while being agnostic to both the device used to measure temperatures and to the BC implementations that are used to store the data. It is currently being developed at the Communication Systems Group (CSG) at the Department of Informatics (IfI) at the University of Zurich (UZH) as a joint project together

with the second relevant project for this work, called modum.io [3]. Modum.io is a startup spun off the CSG in 2016. A first prototype was developed as the practical work of two bachelor theses at the CSG [4, 5]. This prototype consisted of a temperature logger developed using a Texas Instrument SensorTag Development Kit, a mobile application to communicate with the temperature logger via a Bluetooth Low Energy (BLE) connection, and a backend system for data storage and communication to the Ethereum BC, where a SC was deployed. The current modum.io product catalog contains the MODsense T temperature data logger, which can be preconfigured wirelessly with shipment-specific profiles for alarm criteria. For regulatory compliance, the logger also offers immutable and tamper-proof recording, which is achieved by digitally signing data using a built-in hardware security module that holds a private and public key pair.

Thus, this master project combines the monitoring capabilities and the hardware-based cryptography capabilities of the MODsense T data logger with the BC4CC project. The goal is to digitally sign raw BC transactions using the hardware security module in order to directly publish them on the BC and not to rely on credentials that are stored on centralized backend services. Using this approach, the BC4CC project can be further decentralized and the number of intermediate systems that are involved can be reduced, resulting in a decreased amount of trust placed on individual components of the system architecture and a higher system stability.

1.2 Description of Work

The first stage of work related to this master project involved a research of related works on different BC implementations, such as Ethereum [6], HyperLedger Sawtooth [7], and Bitcoin [8], the creation of raw BC transactions, the use of hardware security modules in the context of BCs and signing those raw transactions and on iOS and Android mobile application development.

The second stage involved the design and implementation of components of the PoC, which includes JavaCard Applet code running on the Hardware Security Module (HSM), a basic firmware for the MODSense T temperature data logger, a mobile application that runs on both Android and iOS platforms and interacts with the temperature data logger via BLE technology, and backend systems to forward the transaction packets to the respective BCs.

The final stage of this master project covered an evaluation with respect to its impacts on resources of the edge device (*i.e.*, the temperature data logger) and a discussion of the implemented PoC.

1.3 Thesis Outline

First, in Chapter 2 of this report related work is covered. In detail, information about data storage in the BC is presented in Section 2.1, as well as the difference between private and

public BCs in Section 2.2. In Section 2.3 Ethereum is discussed, Hyperledger Sawtooth in Section 2.4, the BC4CC project in Section 2.5 and modum.io in Section 2.6. Further, this Chapter covers the temperature logger in Section 2.7, BLE technology in Section 2.8, the ISO786 standard in Section 2.9 and Elliptic Curve Cryptography in Section 2.10. In Chapter 3 the high-level design of the PoC is covered and in Chapter 4 the work is described in technical detail. In Chapter 5 the PoC developed in this master project is evaluated. The evaluation covers aspects such as security and usability of the PoC and power consumption. Further, the challenges we faced over the course of the project are discussed. Finally, the report is concluded with Chapter 6, presenting conclusions and future work.

Chapter 2

Related Work

This chapter contains information about work related to this master project. In Section 2.1, data storage in the BC is discussed as it is a central functionality for the developed PoC. As both modum.io and the BC4CC project use both public and private BCs, Section 2.2 discusses their differences. Section 2.5 and 2.6 discuss the BC4CC project and modum.io, as this master project is an addition to both of their systems. BLE, the ISO7816 standard, and elliptic curve cryptography are essential technologies for the development of the PoC and are therefore covered in Section 2.8, 2.9 and 2.10.

2.1 Data Storage in the Blockchain

BCs were not designed as databases, however some BCs, such as Ethereum, provide data storage as an intentional and central part of the design. For others, such as Bitcoin, storing data was not a design goal when the system was conceived. [9] discusses how data can be stored anyway in the Bitcoin BC, *e.g.*, by using the script part of a Bitcoin transaction, but none of the methods presented is without drawbacks [10]. A big advantage of those BC platforms that provide data storage as a core function is that the stored data can be retrieved in a standardized way using a public interface.

For example, in Ethereum all interaction with stored data is implemented in a SC and if the SC code is made public, everybody can understand the interface. In Bitcoin data stored is “hidden” in other data structures whose primary purpose is another. If a well-described method to store and retrieve data is used, such as the ones presented in [9], the data can be retrieved by anybody. However, data could also be stored in many undocumented ways that essentially hide the data in the BC. The two BC platforms supported by the PoC provide an easy way of storing and retrieving data. For Ethereum all interaction with stored data is via the SC whose source code is published together with this report. For Hyperledger Sawtooth we use the *IntegerKey* (*cf.* Section 2.4) method that is implemented in a default transaction processor. By implementing a unique transaction processor, the data storage format could be customized for specific requirements and by publishing the transaction processor source code, open access to the data interface would be guaranteed.

2.2 Private versus Public Blockchains

A public BC is a BC system that can be used by anybody and where the nodes are run by a variety of individuals. The design of the BC allows its use without trusting every single individual in the network. A private BC is designed for a use case where nodes are run only by a small number of companies or individuals. Thus, access to the data stored on the BC is possible only for the participants of the private BC. This is enticing for companies who fear that putting data on a public BC could be associated with privacy concerns [11].

In the modum use case sensitive shipment information is protected by the fact that it is hashed before the data is put on the BC. Nevertheless, the integration of a private BC platform is of interest for Modum because customer requirements may dictate that only a private BC can be used. For this reason, the PoC is able to communicate with a public BC (*i.e.*, Ethereum) and a private BC (*i.e.*, HyperLedger).

2.3 Ethereum

Ethereum, as described in [12], is a “global, open-source platform for a decentralized applications” and the “world’s leading programmable BC”. Ethereum consists of a native cryptocurrency called Ether (ETH) and the Ethereum Virtual Machine (EVM), a Turing-complete VM that accepts a predefined set of opcodes which are obtained by the compilation of SCs. SCs are pieces of code that can be used to enforce certain rules without a third party, *i.e.*, they can be used to enforce accountability and non-repudiation between parties without the need for any trust. Using those SCs, all kinds of different applications have been developed on top of the Ethereum BC and its EVM, including but not limited to wallets that facilitate payments with ETH, crowdfunding platforms, casino games, loans, financial applications or arbitrary ownership declarations.

In contrast to Bitcoin where balances are determined based on transactions and their inputs and outputs (*i.e.*, Unspent Transaction Output (UTXO) model), Ethereum is an account-based BC. This means that balances are stored directly on the BC.

Transactions that are broadcast and included in the Ethereum BC must contain different fields. The necessary fields are:

- **To:** The recipient, also a 20-Byte long address. This can be another account or a contract account, or left empty in the case of a new contract being created.
- **Value:** The amount of funds sent in weis (1 Ether = 10^{18} weis).
- **Data/Input:** This field is meant to hold contract related data, *i.e.*, encoded signature of a function and parameters to call a contract function. In case of a new contract creation, this field holds the bytecode and encoded arguments.

- **Gas Price and Gas Limit:** Both fields are needed to deal with the transaction processing. Each processing step of a transaction (the execution of opcodes) costs a predefined amount of gas units. Gas price is the amount of currency the transactions initiator is ready to pay for a single gas unit. Gas limit is the maximum amount of units this transaction should consume during processing.
- **Signature Fields:** Three signature fields (r , s and v) encode the signature of the transaction. The public key can be recovered from the hashed transaction and the signature. The Ethereum sender address can be computed by hashing the public key.

In addition to the above fields, every transaction contains a number used once (nonce), which is an incrementing number equal to the number of transactions sent by this account to network.

There are two types of accounts on the Ethereum BC: Externally owned accounts (EOAs), which represents an individual user in the external world identified by a 20-Byte long address derived by a private key, and contract accounts. Contract accounts are created once the bytecode of a contract, which is derived by compiling the human-readable code into a machine-readable format, is deployed via a contract creation transaction.

Based on this, there are three different kinds of transactions that can occur on the Ethereum BC:

- i Transferring funds between two accounts
- ii Deploying a contract
- iii Executing a contract function

Executing a function of a SC deployed on the Ethereum BC (iii) leverages the same data field as the deployment of a SC.

Assuming a SC with two callable functions, `get()` and `set()` and a transaction that calls the function `get()`. The first step is to take the method signature as the input for the Keccak variant of the SHA3 (Keccak-SHA3) hash function. The method signature in this case is the simple string “get()”. The output of the hash function yields the following hash:

```
6d4ce63caa65600744ac797760560da39ebd16e8240936b51f53368ef9e0e01f
```

The function selector is then obtained by taking the first 4 Bytes of the hash `0x6d4ce63c`. Using this function selector, it is possible to call the `get()` function of the contract by using it as the value for the “data” field. For the successful realization of this project’s goals, we will use all of the three types of Ethereum transactions.

2.4 Hyperledger Sawtooth

At its core, Hyperledger [13] is “a multi-project open source collaborative effort hosted by The Linux Foundation, created to advance cross-industry BC technologies.” It was started in 2016 in order to further develop two different business BC frameworks: Fabric, which includes work by IBM and Sawtooth, which was developed at Intel’s incubation group.

Sawtooth is a modular framework for developing business BC solutions. Its core components are [7]:

1. A P2P network
2. A distributed ledger/log
3. A state machine/Smart Contract logic layer
4. A distributed state storage
5. A consensus algorithm

In order to communicate with each other, nodes in a network using Sawtooth send messages to each other using the Transmission Control Protocol (TCP). Those messages are serialized using the Protocol Buffer (Protobuf) technology by Google [14]. Sawtooth nodes are able to control various factors at the communication layer, such as [7]:

1. Who can connect and synchronize with state of the distributed ledger
2. Who can participate in the consensus mechanism
3. Who can submit transactions to the network and change the ledger’s state

Using these factors to different degrees, Sawtooth is suitable for deployment in business environments, where permissioned access, privacy and confidentiality is often of special importance.

Nodes that are permitted to change the network’s state are able to do so by sending creating and applying transactions. As depicted in Figure 2.1, transactions are committed to the state as part of a batch, *i.e.*, they are the atomic unit of changes to the state [15].

Transactions contain a “Header” field, which contains a serialized (using Protocol Buffers) version of the transaction header signed by a private key. The “Payload” field contains the data that is used during transaction execution to apply the state change. The signature of the transaction header is also included in the “Header Signature” field. Transactions are collected in a list, which is included in structure of a batch. Their IDs are included in the batch header. Both transaction headers and batch headers are signed using a ECDSA (See Section 2.10) key using the *secp256k1* curve. The network’s validator expects a 64 Byte compact signature [15] (*cf.* Section 2.10).

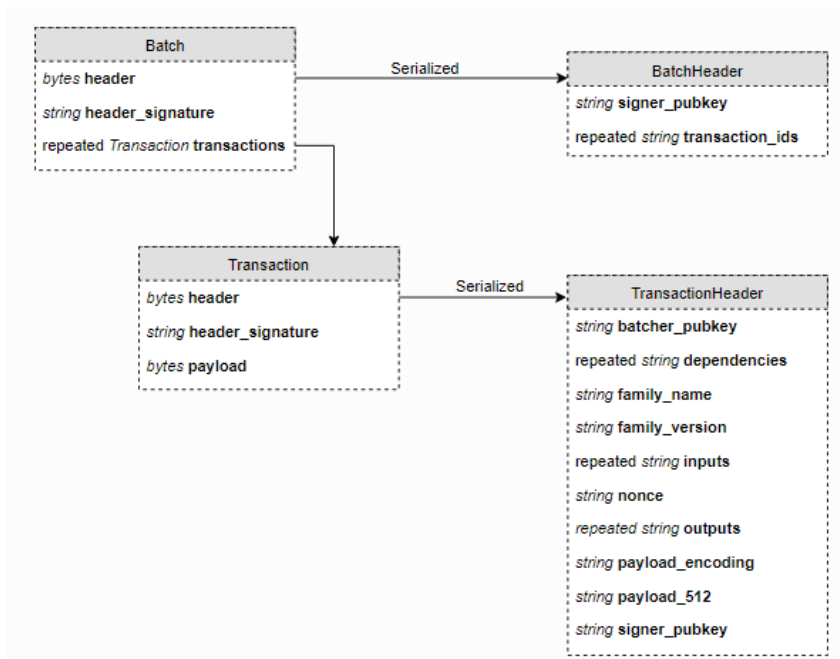


Figure 2.1: The structure of transactions and batches in Hyperledger Sawtooth [15]

Transactions are processed in Sawtooth by so-called “Transaction Processors”. This is an extension of the concept of SCs, as introduced by Ethereum [6] [12]. Transaction Processors support a set of possible transactions defined by a so-called “Transaction Family”. Transaction Families can be defined in multiple different languages, including Python, Javascript, Go or Java. For this project, the *IntegerKey* transaction family was used, which allows to simply store and change integers tied to a string as keys [7].

2.5 BC4CC

The BC Based Temperature Monitoring for Cold Chains in Medical Drug Distribution (BC4CC) project is a research and industrial project conducted by the CSG and modum.io with funding by the Commission for Technology and Innovation (CTI). The project has been running from September 2017 to December 2019. The main goal of the project is to “develop novel methods for supporting the cold-chain distribution process (supply-chain) of medical drugs using BC technology” [2]. The commercial goal is to “reduce the cost of delivery, while assuring a regulation-compliant temperature tracking using IoT-sensors” [2]. The project aims to offer an OpenAPI to customers, suited for the upload of sensor data, and to provide a BC-agnostic framework for the storage of data that can be configured based on customer-defined policies, which guide the selection of the most suitable BC on a case-to-case basis taking into consideration BC-related data, such as transactions costs, performance, and regulations.

The architecture of the BC-agnostic framework [16] and its components is depicted in Figure 2.2. It can be seen that the framework consists of an externally visible Application Programming Interface (*i.e.*, Connector/API). Moreover, the framework includes a

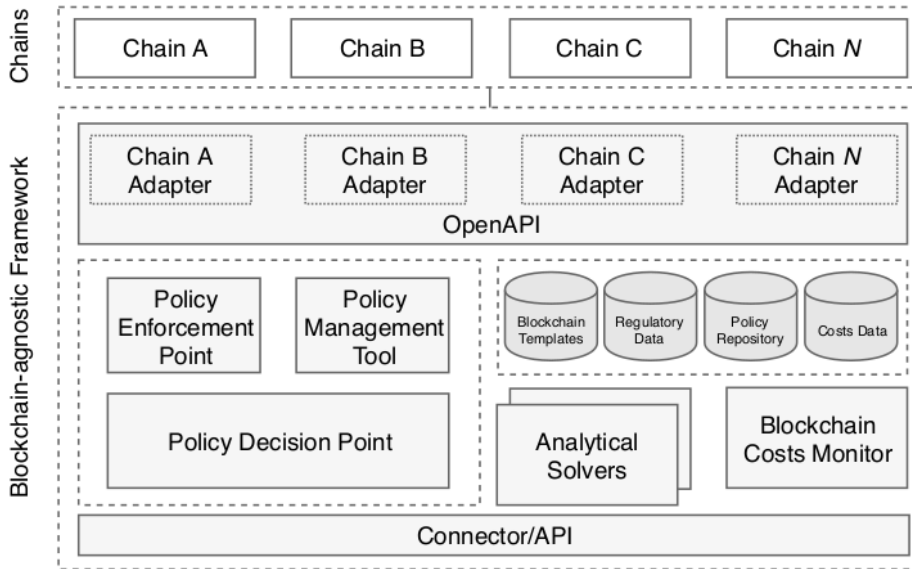


Figure 2.2: The architecture of the Blockchain-agnostic framework

Policy Decision Point (PDP) and an analytical solver that includes cost monitoring and the selection logic. Figure 2.3 presents a high-level system overview of the different stakeholders involved in the usage of the BC4CC system. Temperature data is recorded during the transport of medical drugs. Using the framework this data can be stored in one or multiple BCs and audited by the government.

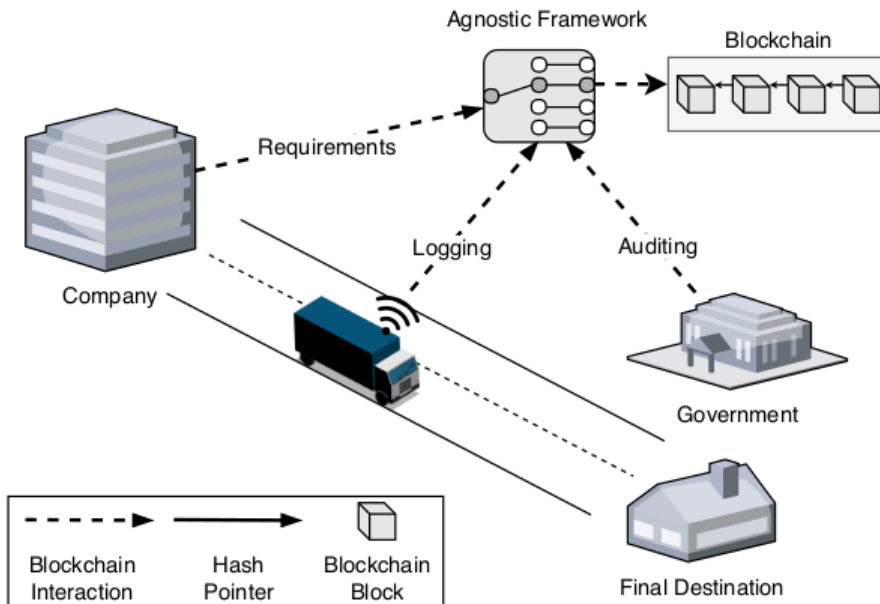


Figure 2.3: The stakeholders involved in the system

Further, Figure 2.4 shows the workflow of the OpenAPI [17]. The API offers a store method that accepts the identification (ID) of the BC in which the data should be stored. The BC correspondent adapter creates a new raw transaction and signs it using the cre-

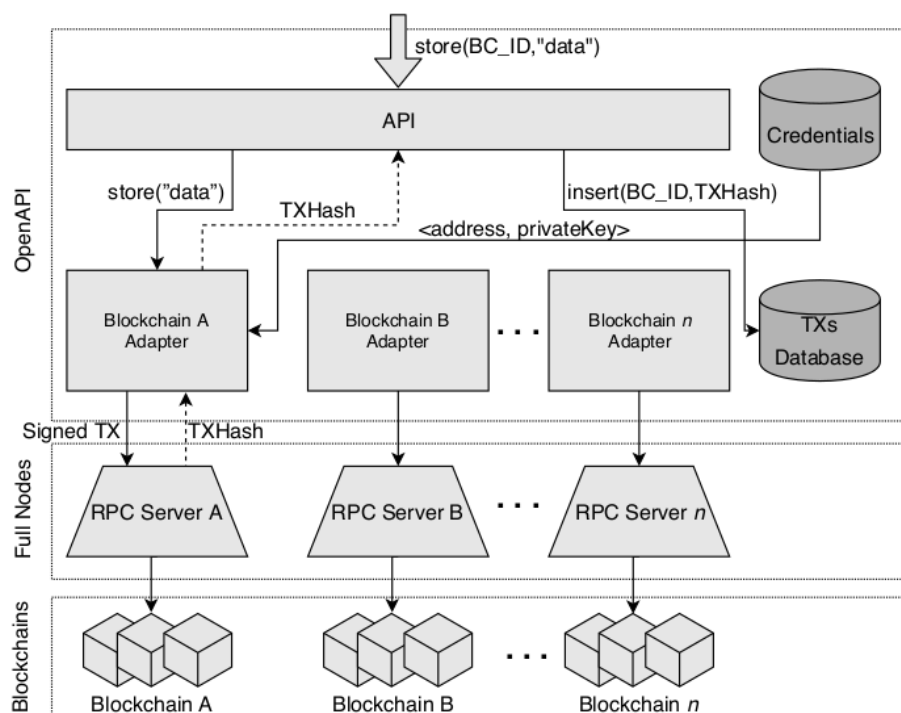


Figure 2.4: The workflow of the OpenAPI

dentials stored in the system. For later management the transactions hashes are stored in a database. The adapter forwards the transaction to the respective Remote Procedure Call (RPC) interface of the local node running the BC to which the transaction is broadcasted. The RPC server returns the transaction hash once the transactions has been included in the mining pool. This hash is used for retrieval of the data stored in the transaction.

2.6 Modum

Modum.io [3, 1] is a start-up based in Zurich, founded in 2016 by entrepreneurs coming from a background in technology and pharmaceutical manufacturing. Moreover, modum.io offers solutions for supply chain intelligence and automation using BC, IoT (see Section 2.7), and AI technology for a wide range of industries.

The company has so far been collaborating with many different global technology partners, including SAP and Swiss Post and has been well recognized by the industry as the various awards such as the 2018 Swiss Digital Economy Award and the 2018 Swiss Logistics Award have shown.

2.7 MODSense T Temperature Logger

MODSense is Modum’s solution for temperature monitoring, designed for sensitive shipments. Customers can pre-define requirements per shipment and automatically evaluate conformity upon readout of data.

Besides the web application for shipment management and a mobile application for shipment initialization and readout of data, the MODSense T Temperature logger is an important component of the solution. It evaluates the collected temperature data against the alarm criteria that was defined for each shipment and can be configured over the air using BLE (see Section 2.8).



Figure 2.5: The MODSense T Temperature Logger

The device features a tamper-evident casing, a maximum storage capacity of forty-five thousand temperature measurements, Near Field Communication (NFC) support, and traceability via the QR code. Critical for this project is the built-in on-board hardware security module that holds a private key initialized in the factory at manufacturing time yet completely inaccessible to anyone. The private key enables the logger to digitally sign any kind of data. In the remainder of this document we also refer to the MODSense device as the *modum logger*.

The modum logger firmware runs on the microcontroller unit (MCU) which contains a central processing unit (CPU), volatile and non-volatile memory, hardware peripherals for bus communication and other tasks, as well as all necessary hardware for a BLE connection except the antenna. The HSM contains its own processor and can run code independent of the MCU.

2.8 Bluetooth Low Energy

BLE is a wireless technology standard that enables the connection of devices in a range of about ten meters, but with significantly less power consumption and costs compared to the classical Bluetooth standard [18]. It has been integrated into the Bluetooth standard in 2009 as an optional part of the 4.0 specification. Because BLE is an optional part of the Bluetooth 4.0 specification, devices supporting 4.0 do not have to necessarily support communication with devices that support Bluetooth Smart connections [19].

An specialized logo was created to market BLE technology, calling the standard "Bluetooth Smart". The logo can be found on multiple sorts of devices, *e.g.*, smartphones that have been extended with the technology. Using two separate hardware modules, those devices can connect both to devices using classic Bluetooth and devices using BLE technology.

Using BLE, data can be transmitted with a rate of 1 MBit/s, which is identical with the Basic Rate (BR) of the Bluetooth standard. The typical range of BLE devices is 40 meters, although devices have been developed supporting ranges up to 200 meters. A clear difference to the classic Bluetooth is the shorter connection setup time (3 milliseconds compared to 100 milliseconds in the classic Bluetooth standard) and the minimum data transmission time of six milliseconds.

	Preamble	Access Address	LL Header	Data	CRC
Length in Bytes	4	0 - 251	2	0 - 255	3

Figure 2.6: The structure of a BLE packet [19].

	L2CAP Header	ATT Data
Length in Bytes	4	0 - 251

Figure 2.7: The structure of the "Data" field of a BLE packet [19].

Figure 2.6 shows the basic structure of a BLE packet, while Figure 2.7 shows the underlying structure of the "Data" field. The data field has a variable size dependent on the Bluetooth specification. In Bluetooth 4.0 and 4.1 the maximum size of the data field was 27 Bytes. Bluetooth 4.2 extended the size to a maximum of 255 Bytes [19]. Contained in this data field is a Logical Link Control and Adaptation Protocol (L2CAP) header. L2CAP is a core protocol in the Bluetooth standard responsible for multiplexing and segmentation of packets [19]. Thus, the maximum size of Bytes that can be transmitted in the data field is 23 Bytes for Bluetooth 4.0 and 4.1 devices or 251 Bytes for Bluetooth 4.2 devices.

Contained in the "ATT Data" field of the link layer "Data" field is an Attribute Protocol (ATT) packet. Figure 2.8 shows the structure of such a packet. The "Op-code" field

	Op-Code	Attribute Handle	Data
Length in Bytes	1	2	0 - (ATT MTU - 1)

Figure 2.8: The structure of an ATT packet

is necessary for the protocol to determine the desired operation in this packet, such as writing data, reading data or registering notifications. For certain commands, The “Data” field in the ATT packet is dependent on the ATT Maximum Transmission Unit (MTU). Duan additional “Attribute Handle” field is required. During MTU negotiation, both devices exchange the maximum MTU size they are able to support and the lower one of those values is then assumed as the new MTU for the rest of the connection.

2.9 ISO7816

The Hardware Security Module (HSM) on the modum logger is connected using a bus conforming to the ISO 7816 standard. ISO7816 [20, 21, 22, 23] specifies a full communication stack from the physical to the application layer and, thus, governs the physical encoding of bits on the wire as well as the binary encoding of commands. It originates in the credit card industry and is not very common in IoT devices. Commands sent to the HSM are encoded in Application Protocol Data Units (APDUs). Each APDU contains a class Byte (CLA), an instruction Byte (INS) and two parameter Bytes (P1 and P2). Each command (as identified by the CLA and INS Byte) falls into one of four categories (named cases):

- **Case 1 commands:** do not have a command payload and don’t expect a response payload (*e.g.*, cycle the HSM into a different internal state).
- **Case 2 commands:** have a command payload and don’t expect a response payload (*e.g.*, store a given certificate on the HSM).
- **Case 3 commands:** don’t have a command payload and do expect a response payload (*e.g.*, a read out a public key from the HSM).
- **Case 4 commands:** have a command payload and do expect a response payload (*e.g.*, calculate the hash of some given data).

Figure 2.9 depicts the format of the APDU commands in all four cases. Depending on the command case, the Lc field encodes the length of the command payload and the Le field encodes the length of the expected response payload. Figure 2.10 depicts the format of APDU responses. The response message contains the optional response payload and finally two mandatory status code Bytes (SW1 and SW2). The response code for “success” is 0x90 0x00.

Case 1:	CLA	INS	P0	P1			
Case 2:	CLA	INS	P0	P1	Lc	Payload	
Case 3:	CLA	INS	P0	P1	Le		
Case 4:	CLA	INS	P0	P1	Lc	Payload	Le

Figure 2.9: APDU command format

Response without payload:	SW1	SW2	
Response with payload:	Response Payload	SW1	SW2

Figure 2.10: APDU command response format

The ISO7816 standard governs which CLA and INS codes can be used by an application, which ranges of values are reserved for future use and which bits in the CLA and INS codes have a special meaning.

Traditionally, HSMs were programmed in assembly language. JavaCard [24, 25] is a standard that defines an operating environment, a subset of the Java programming language and some Java libraries that allow development, deployment and execution of applications written in this Java subset language on a JavaCard enabled HSM. JavaCard runs a simple Java virtual machine on the HSM and provides abstraction and sandboxing mechanisms so that multiple applications (named applets) can be loaded onto the HSM and be run independently of each other.

When the HSM is booted, APDU commands are processed by the JavaCard platform. A SELECT command allows selection of an applet, after which most APDU commands are forwarded to the applet and can be processed by it. Some commands, such as the SELECT command are never forwarded to the applet and, thus, allow switching to a different applet by issuing another SELECT command.

GlobalPlatform [26] is a standard that specifies the management of a HSM platform and its applets. Thus, it specifies commands to load, configure and delete applets, as well as a definition of life cycle states of the HSM, and multiple protocols for secure communication with the HSM or its applets.

2.10 ECDSA

The Elliptic Curve Digital Signature Algorithm (ECDSA) is a signature algorithm that relies on elliptic curves to create a signature. For an ECDSA signature a value k is required that must never be reused to sign some other data. Thus, a random value is usually used. This leads to signatures becoming non-deterministic. RFC6979 [27] defines a scheme to create deterministic signatures. It is achieved by deriving the value k from a function of the input data to be signed. Most implementations of BC platforms use the deterministic variant of ECDSA.

The signature consists of two numbers r and s . Each number can be represented as a 32 Byte integer. There are two encodings commonly used for ECDSA signatures. One concatenates r and s as 32 Byte integers each, yielding a 64 Byte signature. The other uses Distinguished Encoding Rules (DER) of an Abstract Syntax Notation One (ASN.1) structure [28] that contains both r and s as signed positive integers. Thus, if either the r or s component of the signature have a positive highest-order yields encoded signatures with common sizes between 70 and 72 Bytes.

The *secp256k1* curve [29] is one of many elliptic curves that can be used with ECDA. As opposed to many other curves, its parameters were not randomly chosen, which reduces the risk that the publishing organization has some undisclosed knowledge that enables them to attack encrypted material with a higher success rate than the average attacker. It is for this reason that this curve is popular with cryptocurrencies and BC platforms, including Bitcoin and Ethereum.

For a given signatures (r and s) there exists an easy transformation that is also a valid signature (r and $-s \bmod n$ where n is the curve order of the used elliptic curve). This is a problem in Ethereum, for example, because the hash of a transaction is used as its identifier. With the above transformation another transaction can be generated that is also valid, but has a different hash and, thus, a different identifier. The Homestead Hard-fork Changes proposed in Ethereum Improvement Proposal 2 (EIP-2) [30] call for Ethereum signatures to apply a restriction on the value s so that it cannot be larger than half the curve order of the *secp256k1* curve. Hyperledger applies the same restriction to its signatures.

The public key of a key pair used to sign some data can be recovered from the data and the signature [31]. Given a signature and a message there are four public keys that could have generated the given signature [32]. In most BC applications accounts are associated with hashes of public keys [8, 12]. Since a cryptographic hash is an irreversible operation, no public key can be recovered from a given hash. But it is feasible to test all four hashed public key recovered from a given signature and message against the transaction sender address. In Ethereum a value *recovery ID* is calculated that defines which of the four options corresponds to the public key used so that no testing of all four options is required. The *recovery ID* is included in the transaction together with the *chain ID* in the value v .

Chapter 3

Design

This Chapter covers core design decisions concerning the system architecture of the PoC implemented in the context of this master project. The current system stores a hash of shipment data, including temperatures, tracking number, shipment nonce, start timestamp and stop timestamp in the BC, thus preventing modification of shipment information after immutable storage in the BC. The requirements for an edge-device-signing-enabled extension to the modum.io system respectively the BC4CC system are:

- Sign transactions directly on the modum logger. The private key never leaves the HSM on the modum logger
- The signed transactions result in a hash of shipment information to be stored in the BC (*e.g.*, for Ethereum the transaction can call a SC method)
- The hashed shipment information must include a hash of the temperatures of that shipment – that is, after all, the data we want to protect from modification

In order to focus on the core aspects (*i.e.*, transaction signing within the modum logger), the requirements were refined to the following:

- Sign transactions directly on the modum logger so that the private key never leaves the HSM on the modum logger
- The signed transactions result in a hash of shipment information to be stored in the BC (*e.g.*, for Ethereum the transaction can call a SC method)
- The hashed shipment information must include some data that originates on the modum logger. It can be random data generated on the modum logger. Using actual temperature measurements is not a priority as the type of data does not affect the functionality of the PoC.
- Furthermore, the data structure used to store the data in the BC does not have to be optimized to store all shipment parameters or allow efficient lookup. Thus, a simple SC that stores an array of hashes proves the feasibility of the PoC, as the main PoC functionalities are not related to the SC.

3.1 Decisions of Blockchains

The first design decision that had to be discussed was the choice of which BCs to support in the PoC. Supporting Ethereum was a decision made on the basis of the large expertise by the writers of this master project based on their bachelor theses [5] [4] and their affiliation with modum.io, which also uses Ethereum SCs in their solution. Because Ethereum is a public BC, Hyperledger Sawtooth was chosen as the additionally supported BC in order to study both the feasibility for public and private respectively permissioned BCs in the PoC.

3.2 Architecture

Figure 3.1 shows a broad overview of the architecture and information flow of the PoC system for Ethereum transactions. As described in Section 2.3, an Ethereum transaction includes several fields, including a nonce (specific to an address), the gas price and the gas limit. The BC4CC API described in Section 2.5 offers these fields via a Representational State Transfer (REST) API. Note that the figure does not reflect the information flow in a chronological order.

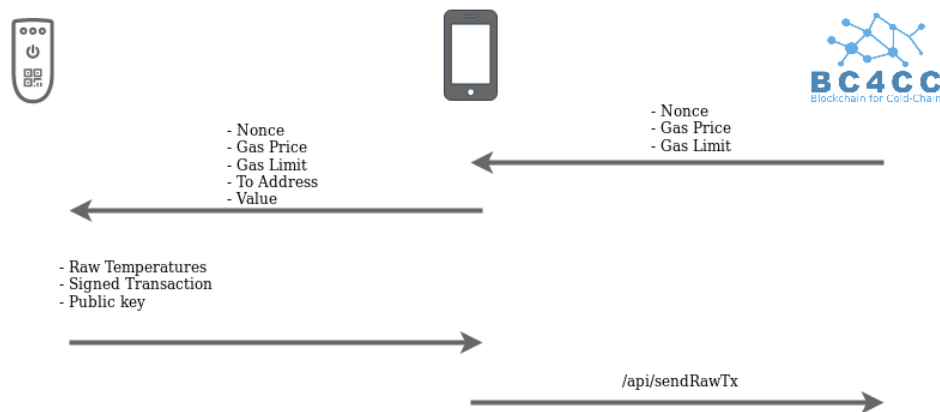


Figure 3.1: The broad architecture and information flow of the PoC system for Ethereum transactions.

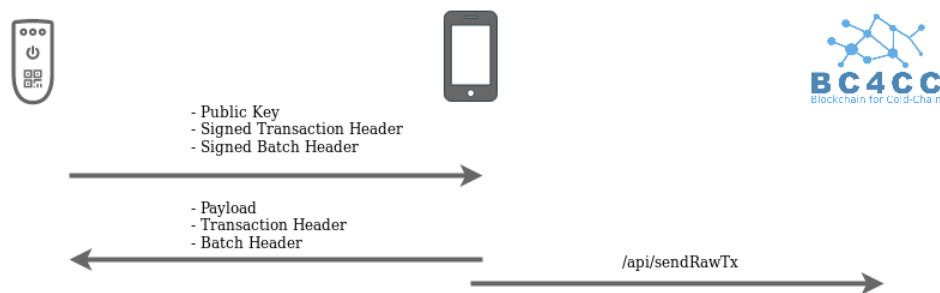


Figure 3.2: The broad architecture and information flow of the PoC system for Hyperledger Sawtooth transactions.

Figure 3.2 shows the same overview of the architecture for Hyperledger Sawtooth transactions. No information is required from the BC4CC API in order to construct a transaction template.

The mobile phone respectively the mobile application is the central part of the overall architecture. It retrieves the necessary information to construct a raw transaction. In the case of Ethereum, this transaction includes the nonce, the gas price, the gas limit, the to address, which is the address of the SC, value, and data. The data field contains a data field encoding a method call to the SC as described in Section 2.3. The transaction is then transmitted to the modum logger using BLE. After transmitting the transaction, the transaction is signed on the modum logger side.

The modum logger in turn offers various values over Bluetooth, such as the measured temperatures, the signed transaction, and its public key.

Using the API described in Section 2.5, the signed transaction is transmitted to the BC4CC system.

3.3 Transaction Preparation

As the core goal of the PoC is to be able to sign and transmit raw transactions to the BC4CC system, decisions had to be made about how to prepare the transactions in order for the modum logger to be able to sign these transactions and include its measured data in these transactions. Concerning the signing of these raw transactions, the modum logger can adopt different levels of involvement which had to be discussed in the context of overall architectural design decisions.

As described in Section 2.3, Ethereum transactions are serialized using Ethereum's own serialization format *Recursive Length Prefix* (RLP) [33], serializing a list of nonce, gas price, gas limit, receiver address, value, data, *recovery ID* and signature fields.

The information about the nonce, gas price and gas limit all originate from outside the modum logger. These are all integers, although not limited to any fixed number of Bytes. Thus, sending these values to the modum logger to include in a transactions requires the definition of some kind of serialization format of these values. It was decided to prepare an RLP encoded transaction on the mobile application and have the modum logger inject only the shipment information hash into the already prepared transaction, followed by signing it. Thus, the PoC avoids de- and reserializing the nonce, gas price, gas limit, receiver address and value fields of the transaction structure. In other words, the serialization format of the transaction data sent to the modum logger is the same as is used to send the transaction to the Ethereum network.

Due to size restrictions of the data that can be exchanged in one message between the modum logger and mobile application (*cf.* the implications of the MTU limitation on the BLE interface in Sections 4.1 and 4.2 and a general description of the MTU in BLE in Section 2.8), only the raw signature and shipment information hash are returned from the modum logger. The signature and shipment information hash are then incorporated into

the prepared transaction on the mobile application. The outcome of this approach does not differ from returning the full encoded final transaction and has no security implications because any modification of the transaction in a way that does not mirror those performed on the modum logger for signing would result in an invalid signature.

For Hyperledger the same approach was followed. As described in Section 2.4 Hyperledger Sawtooth transactions are a Protobuffer [14] encoded structure of *Concise Binary Object Representation* (CBOR) [34] encoded payload. Thus, all the parts that do not rely on the private key, are prepared in the mobile application, while the modum logger only injects the shipment information hash into the transaction and signs the transaction with its private key.

For processing the transactions, the modum logger can have different levels of responsibility:

- **minimal**: The modum logger injects the shipment information hash into the transaction at a hardcoded offset. The full encoded transaction is signed and only the signature is returned. The mobile application integrates the signature into the final transaction.
- **intermediate 1**: The modum logger injects the shipment information hash into the transaction at an offset that is also passed on to the modum logger from the mobile application. This allows more flexibility in changing, for example, the API of the SC that receives the transaction without having to modify the modum logger firmware. This is also required if the serialized transaction contains variable-length values before the shipment information hash so that a hardcoded offset would not work.
- **intermediate 2**: The modum logger can parse the serialization format of the transaction and, thus, inject the shipment information hash at the correct location without receiving an offset from the mobile application. No further processing of the other fields of the transaction is performed.
- **maximal**: The modum logger understands the full semantics of the transaction and is able to parse the serialization format of the transaction. This allows the modum logger to perform checks on the values of the transaction and reject signing it in certain cases.

The current PoC was implemented as an **intermediate 2** level for Ethereum and a **minimal** level for Hyperledger. A discussion of the value of having the modum logger firmware understand more of the structure of the prepared transactions and applying checks on the values before signing the transaction is included in Section 5.1.1.

Chapter 4

Implementation

In this Chapter the implementation of the PoC system is presented in full detail. The components are the HSM applet, logger firmware, Ethereum mobile app, Hyperledger mobile app and the Ethereum SC. (connection to the BC4CC backend). These components combined implement the PoC system that targets the Ethereum and Hyperledger platforms.

4.1 Mobile Application

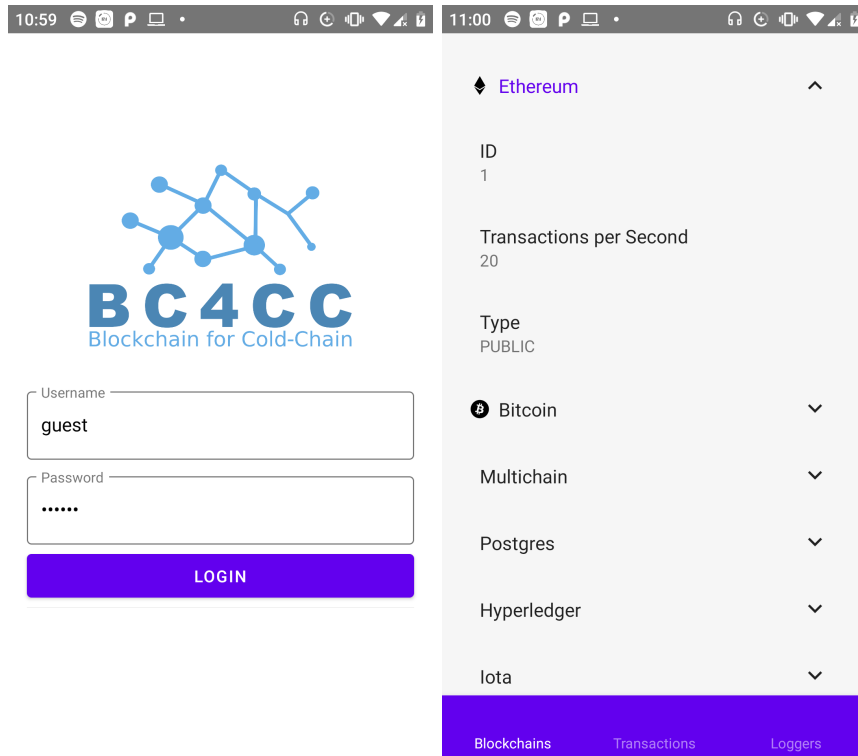
This Section describes the cross platform mobile application that communicates with the modum logger using BLE. The mobile application was developed using the React Native framework [35].

The application consists of several screens. Before being able to use the application, one has to enter the credentials of an account registered in the BC4CC system, consisting of a username and a password. Upon successful login, the user can choose between three different screens. On the “Blockchains” screen depicted in Figure 4.1b, the user is able to get information about the different BCs that are offered by the BC4CC system, such as transactions per second or whether the BC is public or private.

On the “Transactions” screen depicted in Figure 4.2, the user can browse through the transactions once a transactions is processed on the BC. The user can see when and on which BC the transaction and whether it was considered to be valid.

On the “Loggers” screen depicted in Figure 4.3a, the user is prompted to enable the Bluetooth functionality of the smartphone the application is running on in order to scan the surroundings for modum loggers. The modum loggers are listed and upon selecting a modum logger from the list, the application will connect to the device and display relevant information to the user, as shown in Figure 4.4a

On the “Single Logger Screen” shown in Figure 4.4a, the user is able to start a recording on the modum logger, read out data from the device if it is already recording by pressing



(a) The “Login” screen of the mobile application.

(b) The “Blockchains” screen offers an overview over the different blockchains.

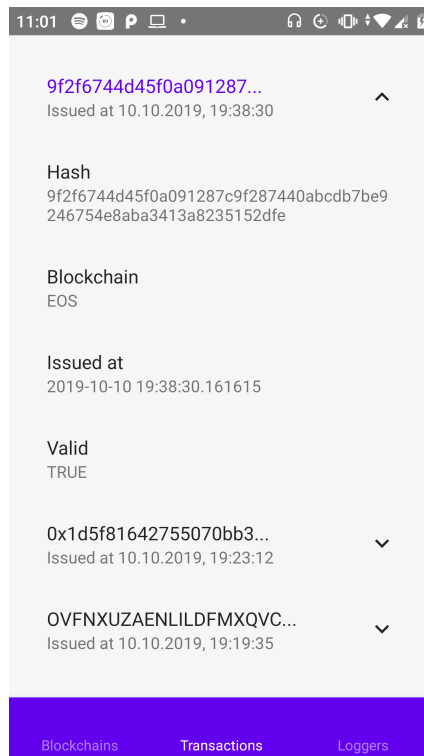
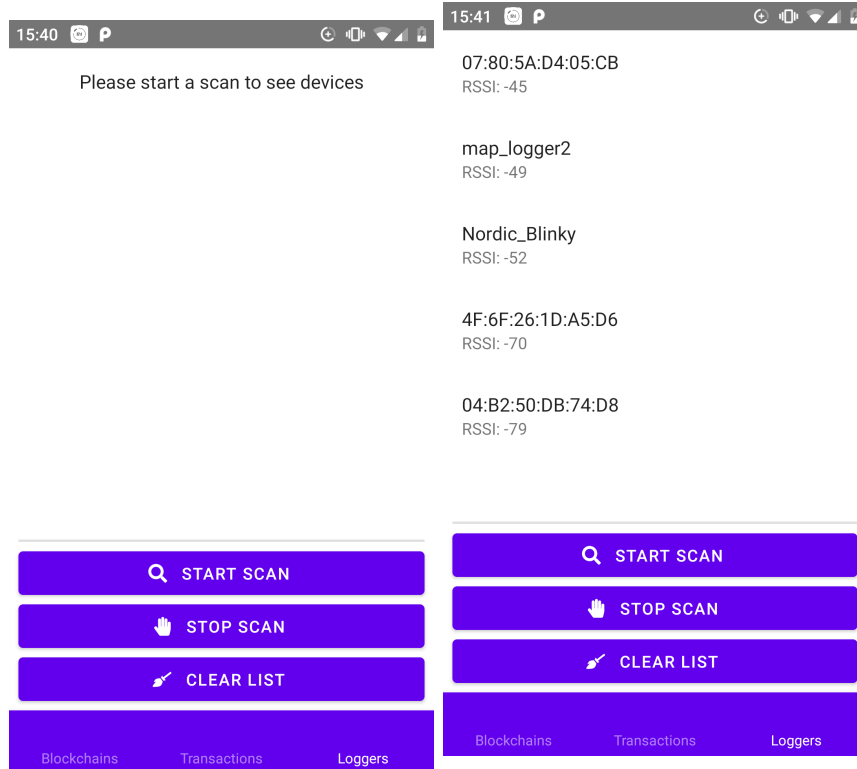


Figure 4.2: The “Transactions” screen offers an overview over the processed transactions.



(a) The “Loggers” screen allows scanning for modum loggers.

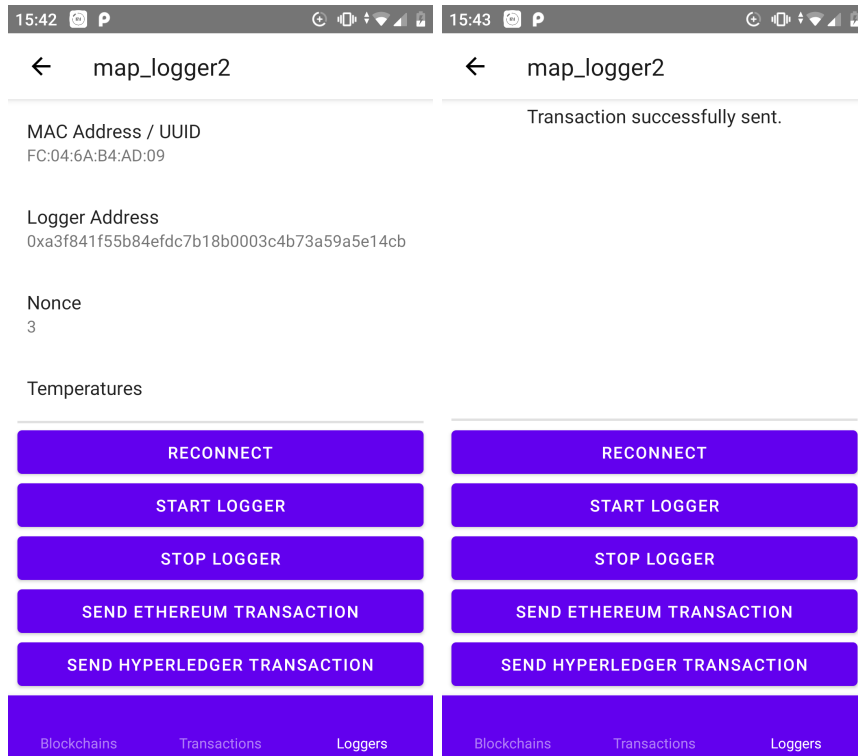
(b) After a scan, the “Loggers” screen shows the modum loggers found nearby.

the “Stop Logger” button, and to generate a BC transaction and send it to the BC4CC system for processing by pressing the “Send Transaction” button. The “Send Transaction” button triggers several operations that are conducted using BLE and the interface offered by the modum logger firmware as described in Section 4.2.

Once the “Send Ethereum Transaction” or the “Send Hyperledger Transaction” button is pressed, the mobile application will start to construct a transaction according to Section 3.3. The transaction prepared by the mobile application is a template for an encoded method call to the `storeShipment()` method of the SC described in Section 4.4. The transaction prepared for Hyperledger Sawtooth is a template for a `set` operation creating a new key-value pair with the first 20 characters of the shipment id being the key and the first four Bytes of the temperature data hash being the value. Storing the entire temperature data hash is not possible since the *IntegerKey* transaction family limits valid values to integers in the range of 0 to 2^{32} , *i.e.*, four Byte integers, while keys are limited to 20 characters [36].

Both Ethereum and Hyperledger apply the restriction of EIP-2 (see Section 2.10) on their signatures, limiting the range that the *s* component is allowed to assume. Since the HSM is unaware of these restrictions it generates a signature that is not in all cases immediately a valid Ethereum or Hyperledger signature. We apply the EIP-2 fix to the signature’s *s* component in the mobile app and, in the case of Ethereum, compute the *recovery ID* to derive the value *v*.

Usually, the minimum of the largest MTU (see Section 2.8) that both devices can support



(a) The “Single Logger Screen” shows information about the modum logger to which the mobile phone is connected to. (b) After constructing and sending a transaction, the “Single Logger Screen” displays a status message.

is negotiated as the connection’s MTU. The largest MTU on the modum logger side is limited to 510 Bytes, where 2 Bytes are deducted for the “Op-code” field and one Byte for the “Attribute Handle” field. Maximum MTU sizes for Android phones vary wildly. With higher-end phones, maximum MTUs of 256 Bytes and more are common, but some phones only support the default BLE MTU of 23 Bytes. The noble Javascript library supports a maximum MTU of 256 Bytes. For the PoC we deployed the mobile application on a phone supporting a large enough MTU. The way we defined the Bluetooth interface for the PoC, a client must support at least an MTU of 244 Bytes. This restriction is of no concern since a fragmentation layer on top of the Bluetooth serial channel is easily implemented and is present in the next generation of the modum logger firmware, thus, allowing transmission of large data structures independent of the underlying channel’s MTU.

4.2 Firmware

The modum logger firmware provides an interface of a very basic temperature logger. Endpoints are provided to start and stop a recording, as well as reading out the recorded data. However, no real temperature data is acquired during a recording. One endpoint implements preparing and signing of an Ethereum transaction and four endpoints implement the preparing and signing of a Hyperledger transaction.

The common application layer protocol for BLE is Generic Attribute (GATT). Endpoints are realized in “characteristics”, which are grouped into “services”. Characteristics can be read-only or read-write. Reads and writes can be rejected by the server based on the circumstances and can have side-effects, such as starting a recording on the modum logger upon write of a defined characteristic. Due to memory limitations the number of characteristics on a Bluetooth device is limited. Further, a single read or write of a characteristic has a maximum size, preventing exchange of larger data structures. For these reasons a different approach was chosen for the next generation of modum logger firmware. Only a single read-write characteristic provides a serial communication channel for arbitrary data. Endpoints are realized through prefixes of written data. A fragmentation layer sitting on top of the BLE layer allows transparent exchange of larger data structures.

A similar approach was chosen for the PoC system. A single read-write characteristic provides a serial data exchange channel. No fragmentation is implemented in a dedicated protocol layer. Fragmentation is implemented at the application layer if needed (*e.g.*, by splitting the Hyperledger transaction header signature into endpoints 0x10 and 0x11). Table 4.1 describes the endpoints of the modum logger interface.

The Universal Unique Identifier (UUID) of the single characteristic is:

```
93490001-9d71-4d26-bc1f-0d768fcb3d85
```

The service UUID is:

```
93490000-9d71-4d26-bc1f-0d768fcb3d85.
```

The ISO7816 bus driver for the modum logger has an important limitation. The bus communication for commonly used bus specifications on IoT devices, such as Inter-Integrated Circuit (I2C), the Serial Peripheral Interface (SPI) and Universal Asynchronous Receiver-Transmitter (UART) is usually partially implemented in hardware. Thus, the most timing-critical aspects of communication can be handled without software involvement. Since ISO7816 is not a common bus in the IoT world, the driver must perform even timing-critical actions in software. This does not work if a Bluetooth client device is connected (because Bluetooth communication also requires handling of important timing-critical events in software). Thus, no concurrent connection with a Bluetooth client and the HSM is possible. We overcome this limitation by disconnecting the Bluetooth client in the modum logger firmware as soon as a signature is requested (*i.e.*, endpoints 0x04, 0x11 and 0x13). After the signature has been computed the modum logger restarts its Bluetooth advertising. The client scans for Bluetooth devices until it rediscovers the modum logger. It can then read the result from the BLE protocol endpoint command by reading the BLE characteristic.

Ethereum transactions are hashed using Keccak-SHA3. After finding that implementing Keccak-SHA3 on the HSM is unsatisfactory (see Section 5.3), we decided to implement hashing of the transaction in the modum logger firmware. Thus, only a pre-computed hash is sent to the HSM, which signs it with its private key and returns the signature. There is no security difference between sending an arbitrary hash to the HSM to be signed and sending arbitrary data to the HSM to be hashed and signed. The only security improvement to be gained from sending data instead of a hash could be if the HSM

Table 4.1: Logger interface endpoints

ID	Name	Acc.	Description
0x01	Get Pubkey	ro	Returns the public key of the keypair that was generated on the HSM during provisioning
0x02	Start Recording	rw	Starts a temperature recording
0x03	Stop Recording & Readout	rw	Returns the hardcoded temperature values of the "last shipment"
0x04	Get Ethereum Signature	rw	Receives as payload the raw prepared Ethereum transaction prepares the transaction with the hash of the shipment information and returns the signature concatenated with hash of the transaction
0x09	Put Hyperledger Payload	rw	Receives as payload the raw Hyperledger "intkey" payload and stores it internally
0x10	Get Hyperledger Transaction Header Signature 1	rw	Receives as payload the first 240 Bytes of the raw Hyperledger transaction header and stores them internally
0x11	Get Hyperledger Transaction Header Signature 2	rw	Receives as payload the remainder of the raw Hyperledger transaction header, inserts (part of) the shipment information hash into the previously stored payload, prepares the transaction using the updated payload, signs it and returns the signature concatenated with the updated payload
0x13	Get Hyperledger Batch Header Signature	rw	Receives as payload the raw Hyperledger batch header, signs it and returns the signature

The ID corresponds to the prefix of the endpoint, Access (abbreviated Acc.) is either "ro" for read-only or "rw" for read-write

understands some semantics of the data. For example, if the HSM understands the nonce part of the transaction, it can enforce that the nonce is strictly incrementing and refuse to sign transactions for apparently invalid nonces. A discussion of the security implications of this decision can be found in Section 5.1.1. We pursue the same approach for Hyperledger transactions and pre-compute the hash to be signed in the modum logger firmware.

4.3 Applet

The applet’s function is to receive a pre-computed hash of a transaction and sign it using ECDSA using the *secp256k1* curve [29]. The key pair is generated during installation of the applet and stored in the HSM ROM.

Table 4.2 describes the HSM interface endpoints. Retrieving the public key is a Case 2 command and signing a hash is a Case 4 command (see Section 2.9).

For source code licensing reasons the HSM applet was developed without any prior implementation. For the PoC system only the most important functions were implemented. Encryption or authentication for the connection between the MCU and the HSM was not developed. The modum.io system already implements an encrypted and authenticated connection between the MCU and HSM adhering to the GlobalPlatform [26] Secure Connections SCP 03 standard. In Section 5.1.1 we discuss the potential for involvement of the HSM in validation of external data that is included in the transactions.

To develop the applet and debug the code, a special smart card fitted with the same HSM that is programmed with a debug interface was utilized. The smart card can be inserted into any USB smart card reader and connected to a PC. The IDE then allows applet installation on the smart card and stepping through the code as with any other debugger.

The HSM’s implementation of the ECDSA algorithm encodes signatures in the DER format. Thus, the applet also contains code to extract the raw *r* and *s* values from the DER encoded binary data to return a 64 Byte raw signature (*cf.* Section 2.10).

Table 4.2: HSM applet endpoints

CLA	INS	P1	P2	Lc	Payload	Le	Response Payload
0x80	0x02	0x00	0x00	n/a	n/a	0x40	Public key of the HSM’s keypair as raw x and y value, each 32B
0x80	0x04	0x00	0x00	0x20	32B hash to sign	0x40	Signature as raw r and s value, each 32B

4.4 Smart Contract

To ensure end-to-end security of the PoC, a basic SC was developed using the Solidity [Tim => ref] programming language. In addition to the temperature hash that the modum logger injects into the transaction, the SC also stores the twenty Bytes long Ethereum address of the modum logger, the six Bytes long MAC address of the modum logger and a sixteen Bytes long unique number and combines this information in a “Shipment” structure. The SC provides two main functionalities:

- Storing new shipments
- Comparing the temperature data hash of an already stored shipment with temperatures in their human-readable form

As the code in Listing 4.1 shows, the SC internally stores two mappings, one of MAC addresses to a list of shipments, representing modum loggers and their list of recorded shipments, and one mapping nonces to a shipment, facilitating the later retrieval of single shipments. Both of these mappings are modified in the `storeShipment()` method taking shipment information as parameters and deriving the modum logger’s address from the transaction sent to the contract (`tx.origin`). A modification to these mappings are a change of the contract’s state, which is stored on the public Ethereum blockchain. The mapping between a modum logger and a temperature data hash stored in the “Shipment” structure is thus immutably and non-repudiably stored on the blockchain.

The correctness of a temperature hash can be checked via the `checkTemperatures()` method, taking a nonce and temperatures in the form of raw Bytes as parameters. The method will then retrieve the shipment uniquely identified by the provided nonce respectively the temperature data hash linked with this shipment from the mapping, hash the provided temperatures, compare this new hash to the already existing hash and finally return as a boolean value whether or not they match. This method returning `true` not only cryptographically proves that the shipment stored in the contract has indeed the provided temperatures associated in a hash form but also that the modum logger associated with this shipment has recorded the provided temperatures.

```

1  pragma solidity >=0.5.10;
2
3  pragma experimental ABIEncoderV2;
4
5  /// @author Tim Strasser
6  /// @title BC4CC Contract
7  contract BC4CC {
8
9      struct Shipment {
10         address loggerAddress;
11         bytes6 macAddress;
12         bytes16 nonce;
13         bytes32 temperaturesHash;
14     }
15
16     mapping(bytes6 => Shipment []) macAddressToShipments;

```

```
17
18     mapping(bytes16 => Shipment) nonceToShipment;
19
20     /// @notice Stores a new shipment
21     function storeShipment(bytes6 macAddress, bytes16 nonce, bytes32
22         temperaturesHash) public {
23         Shipment memory existingShipment = nonceToShipment[nonce];
24         require(existingShipment.loggerAddress == address(0), "Shipment
25             should be new");
26         Shipment[] storage existingShipments = macAddressToShipments[
27             macAddress];
28         address loggerAddress = tx.origin;
29         Shipment memory newShipment = Shipment(loggerAddress, macAddress
30             , nonce, temperaturesHash);
31         existingShipments.push(newShipment);
32         macAddressToShipments[macAddress] = existingShipments;
33         nonceToShipment[nonce] = newShipment;
34     }
35
36     /// @notice Returns all shipments for one macAddress
37     function getShipments(bytes6 macAddress) public view returns (
38         Shipment[] memory) {
39         return macAddressToShipments[macAddress];
40     }
41
42     /// @notice Returns the shipment with this nonce
43     function getShipment(bytes16 nonce) public view returns (Shipment
44         memory) {
45         Shipment memory existingShipment = nonceToShipment[nonce];
46         require(existingShipment.loggerAddress != address(0), "Shipment
47             should exist");
48         return nonceToShipment[nonce];
49     }
50
51     /// @notice Allows to check the correctness of temperatures for the
52     shipment with this nonce
53     function checkTemperatures(bytes16 nonce, bytes memory temperatures)
54         public view returns (bool) {
55         Shipment memory existingShipment = nonceToShipment[nonce];
56         require(existingShipment.loggerAddress != address(0), "Shipment
57             should exist");
58         Shipment memory shipment = nonceToShipment[nonce];
59         bytes32 existingTemperaturesHash = shipment.temperaturesHash;
60         bytes32 hashToCompare = sha256(temperatures);
61         return existingTemperaturesHash == hashToCompare;
62     }
63 }
```

Listing 4.1: The BC4CC Smart Contract code, written in the Solidity programming language.

Chapter 5

Evaluation

In this Chapter the PoC system and its advantages and disadvantages over the existing modum.io system are discussed. Further, the power consumption of the PoC system using the HSM is compared to a variant of the PoC system that performs all cryptographic algorithms in software on the modum logger's MCU. Additionally, the challenges that we faced during development of the PoC system are discussed.

5.1 Security and Usability

We base the discussion of the differences between our PoC system and the modum.io system in terms of security, advantages, and disadvantages on the following concerns:

- **Flexibility with BC selection:** Systems that are integrated with BC4CC offer a greater flexibility in selecting the appropriate BC platform to store the shipment information on.
- **Checking integrity of past shipments:** How complex is it to verify a shipment's data against an immutably stored record of the shipment in the BC? Systems that do not sign the BC transaction directly on the modum logger require an additional step in verification of the shipment data. In addition to the shipment data hash, a signature is stored on the BC and must be verified against the shipment data hash and the modum logger's public key. Systems that sign transactions directly on the modum logger only require comparison of the stored hash against the given shipment data. The fact that the transaction led to the inclusion of the record in the BC proves that the signature was valid.
- **Complexity of the modum logger:** Updates to the modum logger firmware are not a trivial task. Although the modum logger has an over-the-air firmware update feature, firmware updates in the field are error-prone and affect user experience. Furthermore, it is impossible to guarantee that all modum loggers in circulation are updated. Therefore, it is an advantage to select a system that keeps the modum logger firmware simple and static.

- **Centralization:** Less centralization leads to higher resilience against system failure because not all transactions have to be routed via a central server.
- **Distribution of funds:** If a central server is responsible for transaction generation, there is a single BC account that must hold funds in order to pay for transactions. If modum loggers sign transactions, all modum loggers must hold funds and may have to receive regular top-up transactions to keep the balance positive.
- **Smart contract complexity:** A SC that allows checking of an external structure of signed shipment information is more complex than one that only stores a hash of shipment information. Thus, a system where the modum loggers prepare transactions themselves tends to require a less complex SC (but this is contrasted by the **Rejection of transactions from third parties** concern below).
- **Rejection of transactions from third parties:** In the case of Ethereum, if only one account on a backend system is responsible to send transactions to a SC, a check can be built in that returns immediately if the sender address differs. Thus, modum can easily ensure that only they can send transactions to the SC. If the modum loggers sign transactions such a check becomes more difficult. If no such check is present, anyone can send a transaction to the SC for non-existent shipments or even random data. This could be prevented by issuing digital certificates to the modum loggers and verifying certificates in the SC. For Hyperledger the problem is of much less concern since it is a private BC.
- **Importance of a single key:** If only one account on a backend system is responsible to send transactions to a SC, it becomes a sole target of an attack. The account probably holds more funds than if the keys were distributed over thousands of modum loggers. The account may also be hardcoded in some validation checks (*e.g.*, with the aforementioned checks against a transaction sender in the SC), which makes replacement of a compromised account harder. If the modum loggers sign transactions the loss of a single key has less impact.

In addition to the modum.io system as it is currently implemented and our PoC system, two additional theoretical systems are introduced and discussed regarding how these affect the previously defined concerns.

- **modum.io system status quo:** Defines the modum.io system in its current implementation, *i.e.*, the system in production.
- **modum.io + BC4CC TX in BE:** Defines an integration of the modum.io system and BC4CC that supports multiple BC platforms, but prepares all the BC transactions in the (BC4CC) backend.
- **PoC transactions on mobile app:** Defines a system similar to our PoC where the modum logger still signs temperature measurements and shipment information in the format of the original modum.io system. The transactions, however, are constructed in the mobile application, can target a variety of BC platforms and are sent directly to the respective BC network from the mobile application.

Table 5.1: Evaluation of Aspects of Study #1

System	Flexibility with Blockchain selection	Checking integrity of past shipments	Complexity of modum logger	Centralization
modum.io system status quo	none	-	-	-
modum.io + BC4CC TX in BE	increased	same difficulty	same	same
PoC transactions on mobile app	increased	same difficulty	same	less
PoC transactions on modum logger	sl. increased	easier	higher	less

Table 5.2: Evaluation of Aspects of Study #2

System	Distribution of funds	Smart Contract complexity	Rejection of transactions from third parties	Importance of a single key
modum.io system status quo	-	-	easy	high
modum.io + BC4CC TX in BE	same difficulty	same	easy	high
PoC transactions on mobile app	risky – user must own	same/higher	harder/verify Cert.	low
PoC transactions on modum logger	harder	lower/higher	harder/verify Cert.	low

- **PoC transaction on modum logger:** Defines the PoC system that prepares the transactions for multiple BC platforms directly on the modum logger (we count both signing in software on the MCU and signing on the HSM into this category).

Tables 5.1 and 5.2 illustrate the performed assessment of the given systems. The following paragraphs explain and elaborate the differences between the systems for each concern.

The current modum.io system offers no flexibility with BC selection. The only option is Ethereum. Integrating BC4CC and/or preparing transactions on the mobile app greatly improves this flexibility. Preparing transactions on the modum logger as in the PoC system also improves the flexibility, but to a lesser degree since it contrasts with the goal of keeping the modum logger firmware as static as possible. Every new BC platform that needed to be supported would require a firmware update.

Checking the integrity of a past shipment record stored in the BC is slightly easier if the BC transaction is signed directly on the modum logger because the verification process requires one less signature verification. For all the other system variants, integrity checking of past shipments does not differ from the current modum.io system.

Signing transactions directly on the modum logger increases complexity of the modum logger firmware for each BC platform that must be supported. This mainly stems from the different transaction serialization formats that the modum logger must be able to parse. For all the other system variants the modum logger firmware can stay exactly the same as current modum.io system's firmware.

Integrating modum.io with the BC4CC backend does not on its own reduce centralization. However, when transactions are broadcast to the BC network from the mobile application,

centralization is lowered. Thus, the third and fourth system variant offer the advantage of lower centralization.

As summarized in Table 5.2, the distribution of funds becomes more complex when the modum loggers sign transactions with their own private keys who need to have funds associated — on all supported BC platforms. If transactions are signed in the mobile application, theft of funds becomes a concern. Mobile applications are easily disassembled and stored keys retrieved. Thus, it is advisable that the user own the funds on his mobile application — as opposed to modum.io. This lowers user experience, however, since the user must now manage his own balance on the application.

A simple integration of BC4CC and the modum.io system (second system variant), as well as a system where mobile applications sign transactions (third system variant) could reuse the same SC that is currently used in the modum system. If modum loggers sign transactions (fourth system variant), the SC complexity can be slightly reduced because verification of the modum logger’s signature is an inherent part of the transaction validation. On the other hand certificate verification (see the concern **Rejection of transactions from third parties**) (required for the third and fourth system variant) leads to an increase of SC complexity.

Rejection of transactions from third parties is easy for those systems where a single key can create valid transactions to interact with a SC (first and second system variant). For the other systems, rejection of third-party transactions unless certificate verification is implemented in the SC. For Hyperledger the problem is of much less concern because it is a private BC.

The sensitivity of a single key is lower in those systems where many keys are used (*i.e.*, signing in the mobile application or on the modum logger) because there is not a single key that is the sole account allowed to interact with the system’s SC. This is contrasted with the increased complexity of funds distribution.

Overall, we conclude that all discussed system variants have advantages and drawbacks. Signing transactions on the modum logger – as is implemented in the PoC of this work – provides the advantage of a slightly increased flexibility with BC selection, easier checking of the integrity of past shipments, decreased centralization and a lower importance of a single key in the system. But this is contrasted with a higher complexity of modum logger firmware, the requirement of funds distribution to all modum loggers and increased SC complexity.

5.1.1 Checking of external values

In our PoC the modum logger accepts external values that go into the transaction without checking them. Concretely, those are the nonce, gas price, gas limit, receiver address, value and chain ID for Ethereum. The data field is not fully checked either – the modum logger only inserts the shipment data hash into the correct position. For Hyperledger Sawtooth the external values that the modum logger doesn’t check are the transaction family name, family version, inputs, outputs and dependencies. Malicious or accidental

modification of those fields allows construction of a transaction that is invalid (and, hence, will be rejected by the network), that unnecessarily wastes funds or that doesn't point to the correct method of the correct SC on the correct network (thus, not leading to an inclusion of the shipment information at the expected location of the BC network). But none of those modifications threaten the integrity of shipment data because it is directly included into the transaction by the modum logger and signed, preventing any external modification thereof.

Denial of service is always possible and cannot be mitigated on a purely technical level. No system can prevent the actor who reads out a modum logger's temperatures from refusing to upload the data or signature to any backend system. But on an organizational and process level, such an actor would quickly be sanctioned or avoided by the employer or other companies along the supply chain.

Since the modum logger is not directly connected to the Internet, it cannot interface the BC platform directly and obtain a trusted copy of the aforementioned external values. But an intermediate system (*e.g.*, BC4CC) could provide these values digitally signed and, thus, vouch for their correctness. This would prevent modification of this data along the chain via the mobile application.

5.2 Power Measurements

Figure 5.1 depicts a power measurement of the modum logger during the process of obtaining a signed Hyperledger Sawtooth transaction structure where the signatures are computed on the HSM (top) and in software (bottom). The measurements were obtained using the Nordic Semiconductor Power Profiler Kit [37]. The Hyperledger process running on the device under test (DUT) includes signing the transaction header, as well as the batch header. The sequence is evident from the characteristic peaks during each of the phases. First the modum logger advertises (red) until the client connects (green). After reading other data such as the public key, the transaction header signature is requested,

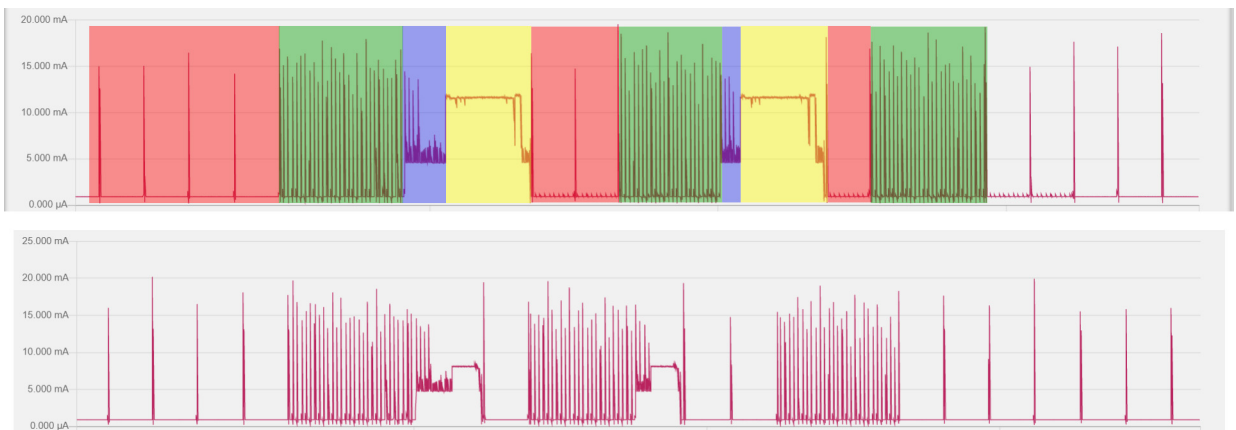


Figure 5.1: Power measurements for the transaction preparation process of the modum logger using an HSM (top) and signing in software (bottom)

Table 5.3: Power consumption of HSM-enabled PoC

Phase	Avg. current	Max. current	Color
BLE advertising	1.00mA	19.92mA	red
BLE client connected	1.47mA	17.94mA	green
Hashing on MCU	4.82mA	13.76mA	blue
HSM operation	11.60mA	11.69mA	yellow

Table 5.4: Power consumption of software-crypto PoC

Phase	Avg. current	Max. current
BLE advertising	1.00mA	20.19mA
BLE client connected	1.37mA	18.09mA
Hashing on MCU	5.16mA	13.79mA
Signature computation	8.10mA	8.16mA

after which the modum logger disconnects the client and starts computing the transaction header hash (blue). The signature is then computed on the HSM (yellow) or in software (not colored). As soon as the signature is computed the modum logger starts advertising again (red). During the next connection (green) the previously calculated signature is read from the modum logger. A second cycle of similar operations is performed for the batch header. Table 5.3 visualizes the average and maximum power consumption during each of the distinct phases of the modum logger shipment cycle and also references the overlay colors used to mark these phases in the figure.

Analyzing this data, it is evident that using the HSM incurs a higher power consumption than running all cryptography in software. Additionally, the HSM signature takes longer than its counterpart in MCU software. However, the security advantages of the HSM still justify its use. Generally, the power consumption is fairly high in these measurements because the UART peripheral used for transmitting log statements to the development machine has a high power consumption. Logging via UART would normally be deactivated for a distribution build of the firmware.

5.3 Challenges

Table 5.5 lists some of the challenges that we faced during the project and their resolution. The *Ref.* column references the section where the implementation details of the resolution is documented. Before deciding to implement hashing of transactions in the modum logger firmware, we evaluated the use of a software implementation of Keccak-SHA3 on the HSM. Computing a hash took around two minutes on the given HSM. Thus, we do not see any realistic scenario where computing the Keccak hash on the HSM could be used. The

Table 5.5: Challenges encountered during development of the project

Challenge	Solution	Ref.
BLE MTU size is too small to exchange full transaction	Implement simple fragmentation on the application layer by splitting the affected endpoints	4.1,4.2
Keccak-SHA3 is too slow on the HSM	Implement hashing in the modum logger firmware	4.2
The HSM's signatures consist of r , s values without EIP-2 restrictions	Apply EIP-2 restrictions in the mobile app by transforming s if necessary, recover the v value in the mobile app	4.1
ISO7816 driver does not allow concurrent HSM and BLE connection	Disconnect BLE before using the HSM and reconnect after the signature has been computed	4.2
Nonce in Ethereum TX must exactly match the number of TXs successfully introduced to the Blockchain by the given node	Retrieve the nonce from the BC4CC backend	4.1

consequence of this is that semantic checking of raw transaction fields is not possible on the HSM. But as Section 5.1.1 discusses, checking of external values is not implemented in the PoC anyway and is not required to ensure integrity of shipment data.

Generally, we conclude from building the PoC that a system can satisfy the requirements of an integration of edge-device signing into the modum use case and that the challenges we faced do not pose insurmountable obstacles.

Chapter 6

Summary and Conclusions

A PoC system was presented that signs transactions for Ethereum and Hyperledger Sawtooth directly on the modum logger's HSM. The evaluation discussion notes that the PoC system provides security and usability advantages, such as slightly increased flexibility with BC selection, easier checking of the integrity of past shipments, decreased centralization and a lower importance of a single key in the system. However, drawbacks include higher complexity of the modum logger firmware, the requirement of funds distribution to all modum loggers and increased SC complexity. In the PoC system, the logger accepts multiple external values without any means to check their validity. This does not, however, present a threat to the integrity of shipment data because that data is directly included into the transaction by the logger itself and signed before it leaves the logger. Furthermore, the power consumption of a system using the HSM is higher than if all cryptography were done in software. Nevertheless, the PoC system fulfils the requirements defined in Chapter 3 and is a step towards reducing centralization of the modum system by eliminating transaction preparation in a backend server. The source code of the PoC system is provided under a permissive open-source license (mostly Apache) so that the functionality could be integrated into a future version of the modum.io system.

6.1 Future Work

We see various opportunities in further developing the system presented in this master project by pursuing work and research in the following areas:

- **Logger Authentication in the Smart Contract:** In the current modum.io system, it is verified that transactions are signed by the modum.io backend system. This prevents unauthorized and unauthenticated actors from including transactions. By directly using the HSM on the modum logger to sign transactions, this kind of verification is not sufficient anymore. Instead, certificates have to be issued for loggers in circulation and have to be sent in addition to the signed transactions. A SC would then be able to verify the signature of the certificate respectively the whole certificate chain.

- **Signing of External Transaction Values:** As mentioned in section 5.1.1, the transactions templates sent to the modum logger contain various external values that are not verified to be consistent or valid, leading to vulnerabilities such as a malicious actor being able to inject invalid or expensive transactions. Using asymmetric cryptography, those values could also be signed and verified. This could either be achieved by the logger having knowledge of one specific public key that is associated with the BC4CC system or another single trusted actor or system or by using public key certificates.
- **Improved Distribution of Funds:** In order for an Ethereum transaction to be successfully included in the Ethereum BC, the signer of the transaction has to have sufficient funds to pay for the needed amount of *gas* (see section 2.3). For the PoC developed in the context of this master project, funds were allocated manually by sending them to the addresses of the modum loggers used. A potential for future work could be to manage addresses of modum loggers and their funds in a more intelligent and scalable way.
- **Support for Additional Blockchains:** The PoC was developed to support the Ethereum and Hyperledger Sawtooth BC platforms. Support for additional BCs could be of interest to align the range of supported BCs of the PoC with those supported by BC4CC. Integration of some BCs, such as Bitcoin might entail managing a complex multitude of input values that are required as input into a transaction.
- **Usability Improvements of the Overall System:** There are multiple opportunities to improve the usability of the overall system because the PoC was developed with the main goal of technical demonstrability in mind. For example, the screen of the mobile application presenting a list of modum loggers nearby to select from could be replaced by a feature enabling the user to directly scan the barcode of the targeted modum logger.

Bibliography

- [1] T. Bocek, B. B. Rodrigues, T. Strasser, and B. Stiller, “Blockchains Everywhere - a Use-Case of Blockchains in the Pharma Supply-Chain,” in *IFIP/IEEE Symposium on Integrated Network and Service Management (IM 2017)*, pp. 772–777., May 2017.
- [2] Communication Systems Group (CSG), “Blockchain Based Temperature Monitoring for Cold Chains in Medical Drug Distribution (BC4CC).” <https://www.csg.uzh.ch/csg/en/research/BC4CC.html>. last visit October 21, 2019.
- [3] “modum.io website.” <https://www.csg.uzh.ch/csg/en/research/BC4CC.html>. Accessed: 2019-10-24.
- [4] T. Strasser, “Managing Goods Distribution with Smart Contracts.” Bachelor Thesis, Communication Systems Group, Department of Informatics, University of Zurich, October 2016.
- [5] A. Knecht, “Securing Goods Distribution with Smart Contracts and Sensors.” Bachelor Thesis, Communication Systems Group, Department of Informatics, University of Zurich, October 2016.
- [6] V. Buterin, “Ethereum: A next-generation smart contract and decentralized application platform.” <https://github.com/ethereum/wiki/wiki/White-Paper>, 2014. Accessed: 2019-10-24.
- [7] K. Olson, M. Bowman, J. Mitchell, S. Amundson, D. Middleton, and C. Montgomery, “Sawtooth: An introduction.” https://www.hyperledger.org/wp-content/uploads/2018/01/Hyperledger_Sawtooth_WhitePaper.pdf, 2018. Accessed: 2019-10-24.
- [8] S. Nakamoto *et al.*, “Bitcoin: A peer-to-peer electronic cash system,” 2008.
- [9] A. Sward, I. Vecna, and F. Stonedahl, “Data insertion in Bitcoin’s Blockchain,” *Ledger*, vol. 3, 2018.
- [10] R. Matzutt, J. Hiller, M. Henze, J. Ziegeldorf, D. Müllmann, O. Hohlfeld, and K. Wehrle, “A Quantitative Analysis of the Impact of Arbitrary Blockchain Content on Bitcoin,” 02 2018.
- [11] J. F. Hoops, “An introduction to Public and Private Distributed Ledgers.” Seminar Innovative Internet Technologies and Mobile Communications SS2017, Chair of Network Architectures and Services, Departments of Informatics, Technical University of Munich.

- [12] G. Wood, “Ethereum: A secure decentralised generalised transaction ledger,” *Ethereum project yellow paper*, vol. 151, pp. 1–32., 2014.
- [13] “Hyperledger Website.” <https://www.hyperledger.org/about>. Accessed: 2019-10-25.
- [14] Google, “Protocol Buffers.” <https://developers.google.com/protocol-buffers>. Accessed: 2019-10-25.
- [15] “Sawtooth v1.0.5 Documentation, Transactions and Batches.” https://sawtooth.hyperledger.org/docs/core/releases/1.0/architecture/transactions_and_batches.html. Accessed: 2019-10-25.
- [16] E. J. Scheid, B. Rodrigues, and B. Stiller, “Toward a Policy-based Blockchain Agnostic Framework,” in *IFIP/IEEE Symposium on Integrated Network and Service Management (IM 2019)*, (Washington, D.C., USA), pp. 609–613., April 2019.
- [17] E. J. Scheid, T. Hegnauer, B. Rodrigues, and B. Stiller, “Bifröst: a Modular Blockchain Interoperability API,” in *IEEE Conference on Local Computer Networks (LCN 2019)*, (Osnabrück, Germany), pp. 1–9., October 2019. Accepted. To be published.
- [18] M. Honkanen, A. Lappetelainen, and K. Kivekas, “Low end extension for Bluetooth,” in *Proceedings. 2004 IEEE Radio and Wireless Conference (IEEE Cat. No.04TH8746)*, pp. 199–202, Sep. 2004.
- [19] Bluetooth SIG, “Bluetooth Core Specification.” https://www.bluetooth.org/docman/handlers/downloaddoc.ashx?doc_id=441541. v4.2.
- [20] ISO/IEC, “ISO7816-1,” 2011. v2.
- [21] ISO/IEC, “ISO7816-2,” 2007. v2.
- [22] ISO/IEC, “ISO7816-3,” 2006. v3.
- [23] ISO/IEC, “ISO7816-4,” 2013. v3.
- [24] Oracle, “JavaCard Specification.” v2.2.2.
- [25] M. Baentsch, P. Buhler, T. Eirich, F. Horing, and M. Oestreicher, “JavaCard-from hype to reality,” *IEEE Concurrency*, vol. 7, pp. 36–43, Oct 1999.
- [26] GlobalPlatform Technology, “Card Specification.” v2.3.1.
- [27] Internet Engineering Task Force, “RFC 6979: Deterministic Usage of the Digital Signature Algorithm (DSA) and Elliptic Curve Digital Signature Algorithm (ECDSA).” <https://tools.ietf.org/html/rfc6979>. Accessed: 2019-11-01.
- [28] B. S. Kaliski, Jr, “A Layman’s Guide to a Subset of ASN.1, BER, and DER.” <http://luca.ntop.org/Teaching/Appunti/asn1.html>, 1993. An RSA Laboratories Technical Note, Accessed: 2019-11-01.

- [29] D. R. L. Brown, “Sec 2: Recommended elliptic curve domain parameters.” Certicom Research, 2010. v2.
- [30] “Ethereum EIP-2.” <https://github.com/ethereum/EIPs/blob/master/EIPS/eip-2.md>. Accessed: 2019-11-01.
- [31] D. R. L. Brown, “Sec 1: Elliptic curve cryptography.” Certicom Research, 2009. v2.
- [32] “Ethereum Stackexchange: During ECDSA signing, how do I generate the Recovery ID?.” <https://ethereum.stackexchange.com/a/53182>. Accessed: 2019-11-01.
- [33] “Recursive Length Prefix.” <https://github.com/ethereum/wiki/wiki/RLP>. Accessed: 2019-10-24.
- [34] “Concise Binary Object Representation.” <https://cbor.io/>. Accessed: 2019-10-28.
- [35] Facebook Inc., “React Native - A Framework for Building Native Apps Using React.” <https://facebook.github.io/react-native/>. last visit November 3, 2019.
- [36] “IntegerKey Transaction Family.” https://sawtooth.hyperledger.org/docs/core/releases/1.0/transaction_family_specifications/integerkey_transaction_family.html. Accessed: 2019-10-31.
- [37] “Nordic Semiconductor Power Profiler Kit.” <https://www.nordicsemi.com/Software-and-Tools/Development-Kits/Power-Profiler-Kit>. Accessed: 2019-11-01.

Abbreviations

AI	Artificial Intelligence
APDU	Application Protocol Data Unit
API	Application Programming Interface
ASN	Abstract Syntax Notation
BC	Blockchain
BLE	Bluetooth Low Energy
CPU	Central Processing Unit
CBOR	Concise Binary Object Representation
DER	Distinguished Encoding Rules
ECDSA	Elliptic Curve Digital Signature Algorithm
EIP	Ethereum Improvement Proposal
EOA	Externally Owned Account
EVM	Ethereum Virtual Machine
I2C	Inter-Integrated Circuit
IoT	Internet of Things
ISO	International Organization for Standardization
HSM	Hardware Security Module
L2CAP	Logical Link Control and Adaptation Protocol
MAC	Media Access Control
MCU	Microcontroller Unit
MTU	Maximum Transmission Unit
PoC	Proof of Concept
PDP	Policy Decision Point
SC	Smart Contract
SPI	Serial Peripheral Interface
REST	Representational State Transfer
RLP	Recursive Length Prefix
ROM	Read Only Memory
RPC	Remote Procedure Call
TCP	Transmission Control Protocol
UART	Universal Asynchronous Receiver-Transmitter
UTXO	Unspent Transaction Output

List of Figures

2.1	The structure of transactions and batches in Hyperledger Sawtooth [15] . . .	9
2.2	The architecture of the Blockchain-agnostic framework	10
2.3	The stakeholders involved in the system	10
2.4	The workflow of the OpenAPI	11
2.5	The MODSense T Temperature Logger	12
2.6	The structure of a BLE packet [19].	13
2.7	The structure of the “Data” field of a BLE packet [19].	13
2.8	The structure of an ATT packet	13
2.9	APDU command format	14
2.10	APDU command response format	15
3.1	The broad architecture and information flow of the PoC system for Ethereum transactions.	18
3.2	The broad architecture and information flow of the PoC system for Hyperledger Sawtooth transactions.	18
4.2	The “Transactions” screen offers an overview over the processed transactions.	22

List of Tables

4.1	Logger interface endpoints	26
4.2	HSM applet endpoints	27
5.1	Evaluation of Aspects of Study #1	33
5.2	Evaluation of Aspects of Study #2	33
5.3	Power consumption of HSM-enabled PoC	36
5.4	Power consumption of software-crypto PoC	36
5.5	Challenges encountered during development of the project	37

Appendix A

Installation Guidelines

A.1 Mobile Application

The mobile application is set up as a React Native project. It uses the Node Package Manager (npm) version 6.7.0 and the Node.js runtime environment version 11.14.0. For instructions on how to set up the environment for React Native, see the steps under the “React Native CLI Quickstart” option at <https://facebook.github.io/react-native/docs/getting-started>. To run the application on an Android device in development mode, the development server has to be run using the `npm run start` command in the root folder of the Mobile Application project. With the development server running, the application can be run on an Android device by running the `react-native run-android` command and on an iOS device by running `react-native run-ios`.

A.2 Smart Contract

The Ethereum smart contract is developed using the Solidity programming language version 0.5.10. It can be compiled using a compiler compatible with version 0.5.10 or higher. Multiple possibilities to deploy the contract can be used, such as deploying it to the different existing variations of the Ethereum BC by using standalone clients for Ethereum, such as Geth (<https://geth.ethereum.org/>) or Parity (<https://www.parity.io/>) to setup a standalone node in the Ethereum network, or by using the online Remix Integrated Development Environment (<https://remix.ethereum.org>). Remix can connect to either a virtual Ethereum blockchain running in the browser memory, to a Web3 instance that is injected into the browser environment, using the Metamask browser extension (<https://metamask.io/>), or an external Web3 endpoint, using an Unique Resource Locator (URL). Web3 is a “collection of libraries which allow you to interact with a local or remote ethereum node, using a HTTP or IPC connection” (<https://web3js.readthedocs.io/en/v1.2.2/>).

A.3 Firmware

The firmware is written for the MODSense T logger, which has a Nordic Semiconductor nRF52832_xxAA MCU on it. The SoftDevice used is S132, version 6.1.1. The Nordic SDK version used is 15.3. After installing the gcc-arm-none-eabi cross-compiler for ARM processors and the Nordic Command Line Tools, the firmware can be built by navigating into the *armgcc* folder and calling `make`. The `SDK_ROOT` should be modified to point to the SDK location. The Makefile also references the ISO7816 driver, where a suitable driver should be located in the directory `../fw/app/drivers/iso7816/` relative to the project directory. Running `make flash` installs the firmware on the logger and `make flash_softdevice` installs the SoftDevice. The MAC address is set on the MCU during manufacturing, but can also be set manually in the firmware code. The advertised device name is `map_logger2` and can be changed in the firmware code.

A.4 Applet

The applet source code is standard JavaCard [24] and can be compiled for and installed on any JavaCard compliant HSM. The process of provisioning the HSM and installing the applet are out of scope of this work. We still include a high level description. The HSM is delivered by the manufacturer in an unconfigured factory state. In this state, all interactions with the HSM require a key that is not distributed by the manufacturer. Instead the manufacturer distributes APDU sequences to the customers that configure the HSM and switch it to a ready, open state with a known key. Since the sequence consists of commands encrypted with the key only known to the manufacturer, the customer can only apply the sequence, but not make any modifications to it.

If the customer wishes for another HSM configuration they can request a corresponding sequence from the manufacturer. After running the initialization sequence, the HSM is in an open state with a known key. Thus, we can authenticate ourselves toward the HSM and install any applet on it. A command allows locking the HSM before shipping it in a final product. In this state no more applets can be installed. It is not necessary, however, to lock the HSM before applets can be run, tested and used.

To install the applet on the logger's HSM we extracted the raw APDU commands from the IDE that install the applet on the debug target, applied a transformation to the APDU sequence that modifies the authentication commands to target the given HSM's key and sequence counter, and temporarily hardcoded the APDU sequence in the logger firmware for applet installation.

Appendix B

Contents of the CD

The CD contains the following file tree:

```
/
├── thesis.pdf ..... The thesis in PDF format
├── midterm.pdf ..... The midterm presentation slides in PDF format
├── applet ..... The applet source code
│   ├── src/io/modum/uzhMasterProject
│   │   ├── Crypto.java ..... Cryptographic operations
│   │   ├── EthereumSignature.java ..... Extract r and s from ASN.1
│   │   ├── Secp256k1.java ..... secp256k1 curve parameters
│   │   └── SecureTransactionApplet.java ..... Applet main code
├── logger_fw ..... Logger firmware source code
├── armgcc ..... Makefile for arm-gcc
├── blockchain ..... Signing transactions and RLP decoding
├── comm_stack ..... BLE communications code
├── common ..... Common helper functions
├── config ..... Configuration headers
├── crypto ..... Cryptographic functions
│   └── hsm ..... HSM communication code
├── drivers ..... HSM driver
├── License
├── main.c ..... Application entry point
├── mobile_app
│   ├── mobile ..... The React Native project for the mobile application
│   └── noble ..... A node client, preparing Hyperledger TXs on the logger
├── smart_contract
│   └── contract.sol ..... Smart Contract for the Ethereum Blockchain
```

The `logger_fw/crypto/sha3.c` file is in the public domain. The `applet/src/io/modum/uzhMasterProject/Secp256k1.java` file is licensed under the GNU Affero General Public License. Most other files are licensed under the Apache License v2. The latter two licenses are not compatible. Thus, for publication or deployment of the code, an alternative to the `Secp256k1.java` file must be found.