Institute of Architecture of Application Systems

University of Stuttgart
Universitätsstraße 38
D–70569 Stuttgart

Masterarbeit

Conception of a Blockchain Client for Secure Transmission of Production Data

David Michel

Course of Study: Informatik

Examiner: Prof. Dr. Dr. h. c. Frank Leymann

Supervisor: M. Sc. Ghareeb Falazi,
M. Sc. Tobias Korb

Commenced: January 8, 2018

Completed: July 5, 2018

Abstract

Blockchain technology has its origins in the financial industry, but also other areas of application are currently being investigated by many sides. One potential application field is the manufacturing industry. The immutability of once stored data makes blockchains interesting for the storage of various kinds of production data. For this purpose, a secure way from a production machine into a blockchain has to be ensured. It may not happen that data can be manipulated before becoming part of the immutable ledger. Currently used gateway solutions often run on general purpose computers, communicate with a production machine and simultaneously operate a blockchain client. Attacks on such gateways pose severe risks to the integrity of the measured data. This work focuses on the conception of a more secure way for saving data into a blockchain. The goal is the creation of a concept using hardware-near components and a prototypical implementation of it. The resulting prototype should enable both the collection of data and the transport of said data into a blockchain. At the same time, the use of lower level hardware in comparison to gateway solutions should reduce potential attack vectors.

Kurzfassung

Die Ursprünge der Blockchain Technologie finden sich in der Finanzindustrie, doch auch andere Anwendungsbereiche werden derzeit von vielen Seiten untersucht. Ein mögliches Anwendungsgebiet ist die Fertigungsindustrie. Die Unveränderlichkeit einmal gespeicherter Daten macht Blockchains für die Speicherung verschiedener Arten von Produktionsdaten interessant. Für diesen Zweck muss ein sicherer Weg von einer Produktionsmaschine hinein in eine Blockchain gewährleistet sein. Es darf nicht passieren, dass Daten manipuliert werden können, bevor sie Teil des unveränderlichen Speichers werden. Die gegenwärtig verwendeten Gateway Lösungen laufen häufig auf Universalcomputern, kommunizieren mit einer Produktionsmaschine und betreiben gleichzeitig einen Blockchain Client. Angriffe auf solche Gateways bergen schwerwiegende Risiken für die Integrität der gemessenen Daten. Diese Arbeit konzentriert sich auf die Konzeption eines sichereren Weges zum Speichern von Daten in eine Blockchain. Ziel ist die Erstellung eines Konzepts, das hardwarenahe Komponenten nutzt, sowie eine prototypische Implementierung desselben. Der entstehende Prototyp sollte sowohl das Sammeln von Daten als auch den Transport dieser Daten in eine Blockchain ermöglichen. Gleichzeitig sollte die Verwendung einer Hardware, die einfacher ist verglichen mit der von Gateway Lösungen, mögliche Angriffsvektoren reduzieren.

Contents

1 Introduction **15**
1.1 Motivation . . . 15
1.2 Scope of Work . . . 16
1.3 Document Structure . . . 16

2 Fundamentals **19**
2.1 Blockchain Technology . . . 19
2.1.1 History and Definition . . . 19
2.1.2 Blockchains in Cryptocurrencies . . . 20
2.1.3 Distinctions and Features of Blockchains . . . 22
2.2 Different Blockchain Implementations . . . 25
2.2.1 Bitcoin . . . 25
2.2.2 Ethereum . . . 27
2.2.3 Hyperledger . . . 29
2.2.4 IOTA . . . 32
2.3 Blockchains in the Industry . . . 34
2.3.1 Industrial Application Examples . . . 34
2.3.2 Remarks on Current Solutions . . . 36

3 Conception **39**
3.1 Requirements . . . 39
3.1.1 General Requirements . . . 39
3.1.2 Functional Requirements . . . 39
3.1.3 Non-functional Requirements . . . 40
3.2 Choice of Technology . . . 40
3.2.1 Hardware Technology . . . 41
3.2.2 Software Technology . . . 42
3.3 Transaction Detail . . . 44
3.3.1 Client . . . 44
3.3.2 Communication Interface . . . 45
3.3.3 Building a Transaction . . . 48
3.4 Different System Architecture Approaches . . . 58
3.4.1 Sign on Ethereum Client . . . 58
3.4.2 Sign on Microcontroller . . . 59

4 Prototypical Implementation **63**
4.1 General Microcontroller Program Information . . . 63

4.2 Concrete Implementation 63

5 General Blockchain Security Aspects **73**
5.1 Possible Attacks on Blockchains 73
5.2 Degree of Trust Measurement 75
5.2.1 Possible Parameters 76
5.2.2 Designing a Metric 78

6 Evaluation **81**
6.1 Evaluation of the Implementation 81
6.1.1 Smart Contract 81
6.1.2 Code Execution Time Analysis 86
6.2 Retrospect of Requirements 87
6.3 Conclusion 89

7 Summary & Future Work **91**
7.1 Summary 91
7.2 Future Work 92

Bibliography **95**

List of Figures

2.1 Simplified illustration of a blockchain structure [Nak08] 20
2.2 Signing transaction data with asymmetrical cryptography 21
2.3 Transaction data signature verification with asymmetrical cryptography 22
2.4 Mining and sending transactions in the Bitcoin P2P network 27
2.5 Steps of a transaction in Hyperledger Fabric 32
2.6 Comparison of a classic blockchain with the IOTA tangle 33
2.7 Attack vectors on blockchain gateway solutions 37

3.1 Layout of NodeMCU pins [Sah17] 42
3.2 Effect of nonces in the Ethereum blockchain 49
3.3 Different steps for generating a signed Ethereum transaction 54
3.4 System architecture with Ethereum node signing a transaction 60
3.5 System architecture with microcontroller signing a transaction 61

4.1 UML diagram of the different implementation components 64
4.2 Schematic of the hardware setup 67
4.3 UML flow chart for code execution on the microcontroller 70

5.1 Schematic of a successful double-spending attack 74

6.1 Serial output of the setup phase 84
6.2 Serial output of one loop iteration 85

List of Tables

2.1 Comparison of different consensus algorithms 24

3.1 Overview of different blockchain implementations 43

4.1 Configuration parameters for the microcontroller code that should not be changed 69
4.2 Configuration parameters for the microcontroller code that may be adapted by a user 71
4.3 Transaction specific configuration parameters for the microcontroller code 71

6.1 Duration of the single steps in the transaction creation process in milliseconds . . 86

List of Listings

3.1 Exemplary JSON object for the user of a web shop 46
3.2 Unsigned Ethereum transaction as a JSON structure 50
3.3 Step 1: Creating an unsigned Ethereum transaction 53
3.4 Step 2.1: Organizing the transaction data in an ordered list 55
3.5 Step 2.2: RLP encoding of an unsigned transaction 55
3.6 Step 3: Keccak-256 hash of the RLP encoded unsigned transaction 55
3.7 Step 4: Signing the Keccak-256 hash with ECDSA 56
3.8 Step 5.1: Signed transaction ready to be RLP encoded 56
3.9 Step 5.2: RLP encoded signed transaction 57
3.10 Optional Step 6: Calculate the transaction ID 57

6.1 Smart contract code for storage of sensor data 82

List of Abbreviations

API application programming interface. 48

B2B business-to-business. 30

DAG directed acyclic graph. 29

DAO decentralized autonomous organization. 37

dApp decentralized application. 27

DSA Digital Signature Algorithm. 20

ECDSA Elliptic Curve Digital Signature Algorithm. 20

EOA externally owned account. 48

ETH Ether. 27

EVM Ethereum Virtual Machine. 28

HTTP Hypertext Transfer Protocol. 47

HTTPS Hypertext Transfer Protocol Secure. 66

IDE integrated development environment. 44

IoT Internet of Things. 29

JSON JavaScript Object Notation. 45

LED light-emitting diode. 68

M2M machine-to-machine. 33

MCU microcontroller unit. 41

P2P peer-to-peer. 23

PoS Proof of Stake. 23

PoW Proof of Work. 23

RLP recursive length prefix. 49

RPC remote procedure call. 45

SPI Serial Peripheral Interface bus. 41

UART universal asynchronous receiver-transmitter. 41

UML Unified Modeling Language. 63

USB Universal Serial Bus. 41

1 Introduction

This chapter explains the motivation behind the work. Derived from this motivation, the scope of work is presented. Finally, an overview on the structure of the thesis is given.

1.1 Motivation

With the upcoming of what is generally referred to as the “Fourth Industrial Revolution” [Sch17], the image of production machines has drastically changed. They are no longer centrally controlled and run in isolation. Instead, production systems have become distributed and interconnected cyber-physical systems. This means that both software components and hardware like actuators or sensors share common data infrastructure and thereby communicate with each other.

Simultaneously, the blockchain technology is on the rise and it is observed with great interest from different perspectives. Though originally mainly applied in the financial world, the fundamental technology underlying blockchains is promising for various other fields of application, e.g. also for intertwined production systems [Swa15]. What makes blockchains so interesting for industrial applications is one of their main features: the immutability of recorded data. Once data was added to a blockchain, it cannot easily be deleted or altered. Through this property, production data can be immutably stored and also a proof for product quality can be made transparent.

Yet, the immutability of stored data requires for a secure path beginning from data acquisition and ending with the successful storage inside a blockchain. If there is the danger of data being altered on this path, the risk of manipulated data getting immutably stored arises.

At this time, there already exist a variety of interconnections between production machines and blockchain instances [GPM17]. Many of them are solutions using a gateway. Such a gateway is usually a software that is run on a general purpose computer. The gateway is connected to both the production machine and a blockchain instance. This blockchain connection is achieved by running a blockchain client which communicates with other clients via the Internet.

While such gateway solutions offer fast and easy data processing as well as blockchain communication, they can also be targets for attackers. If a general purpose computer is connected to the Internet, this opens a variety of attack vectors. The way from the production machine to the gateway creates further possibilities for attacks, namely manipulation of the measured data.

It is therefore necessary to explore new ways of enabling machines to communicate with a blockchain, while simultaneously reducing potential attack vectors.

1.2 Scope of Work

The goal of this work is the development of a concept for achieving storage of production data within a blockchain. This concept has to be a hardware-near system architecture. This means that it should not be a concept suitable for a general purpose computer, but a more low-level hardware.

Before that, existing approaches and technologies for communication between data sources and blockchains have to be analyzed and weak points identified.

Taking these results into account, an appropriate system architecture has to be developed. The design then has to be prototypically implemented in order to verify its practicability. The resulting prototype has to be able to both collect data directly and store it in a blockchain instance, while offering a higher security level than usual gateway solutions.

In addition to analyzing existing concepts, developing and implementing an own one, there also has to be a discussion on blockchain finality, i.e. when can a transaction be considered immutable. As a result, an approach to measure the degree of trust that data is durably stored within a blockchain has to be found.

1.3 Document Structure

The work is structured as follows:

Chapter 2 – Fundamentals: This chapter gives an overview on the prerequisites necessary to grasp the work. This comprises a definition and description of blockchains, their usage and underlying concepts such as hashing or asymmetrical encryption. Furthermore, different implementations will be presented and the application areas as well as shortcomings for industrial applications are discussed.

Chapter 3 – Conception: Here, the different requirements that are posed on this work's result are given. Choices in terms of software and hardware technology are discussed and important concepts for the generation of valid transactions in the chosen implementation are presented. Resulting from these observations, two system architecture approaches are developed and the better suited one is chosen for implementation.

Chapter 4 – Prototypical Implementation: In this chapter, the chosen system architecture is prototypically implemented as a proof of concept. The different components of the design as well as the program's structure are explained.

Chapter 5 – General Blockchain Security Aspects: Possible security issues inherent in blockchains are presented here. Afterwards, an approach for a metric measuring the chance of a transaction becoming immutable is designed.

Chapter 6 – Evaluation: This chapter evaluates both the design and the implementation in form of the prototype. Code execution timings and output will be analyzed and a comparison with the requirements shown earlier is made here.

Chapter 7 – Summary & Future Work: The final chapter wraps up the work and summarizes its results. Finally, an outlook on possible future work is given.

2 Fundamentals

This chapter gives an overview on blockchain technology in general and different implementations of it. Furthermore, actual projects using a blockchain structure are discussed.

2.1 Blockchain Technology

In this section, the history and also a definition of blockchains is introduced. The features of different blockchain concepts are discussed as well as their relation to cryptocurrencies.

2.1.1 History and Definition

In order to fully grasp the concept of the whole blockchain technology, one must first define the term itself. Up to now, there is no strict definition of what a blockchain is, what belongs to it and what does not [Mat16]. Though the term itself does not appear in his paper, Satoshi Nakamoto is often credited the invention of blockchain technology [Nak08]. In his paper called "Bitcoin: A Peer-to-Peer Electronic Cash System" from the year 2008, he describes how blocks of data could be organized in a chain where one block always references its predecessor. The author uses the concept of chained data blocks as the foundation for a decentralized and purely virtual currency. Later, the first cryptocurrency "Bitcoin" emerged from this idea.

In fact, the basic blockchain idea reaches even further in the past. In the year 1991, S. Haber and W. Scott Stornetta described how digital signatures and time stamps for various kinds of documents could be created by the use of hash functions. Furthermore, they propose to link the different data for verification purposes [HS91].

A few years later, R. J. Anderson described a data storage concept called "The Eternity Service". The author suggests to store data decentralized all over the globe. That way, information cannot be easily destroyed or kept under tight wraps by government regulations [And96].

This work follows the general and widely accepted definition of a blockchain: different data blocks are generated and linked with the use of hash functions and cryptography. The result is a chain of blocks where each block depends on its predecessor, thereby also creating a time consistency throughout the blockchain.

More precisely, a block consists of several transactions and a header. The transactions represent the data that is to be stored within a block. The header consists of two hash values: the hash value of the previous block header and the hash value calculated from the current block's transactions. Figure 2.1 shows a simplified illustration of such a blockchain structure.

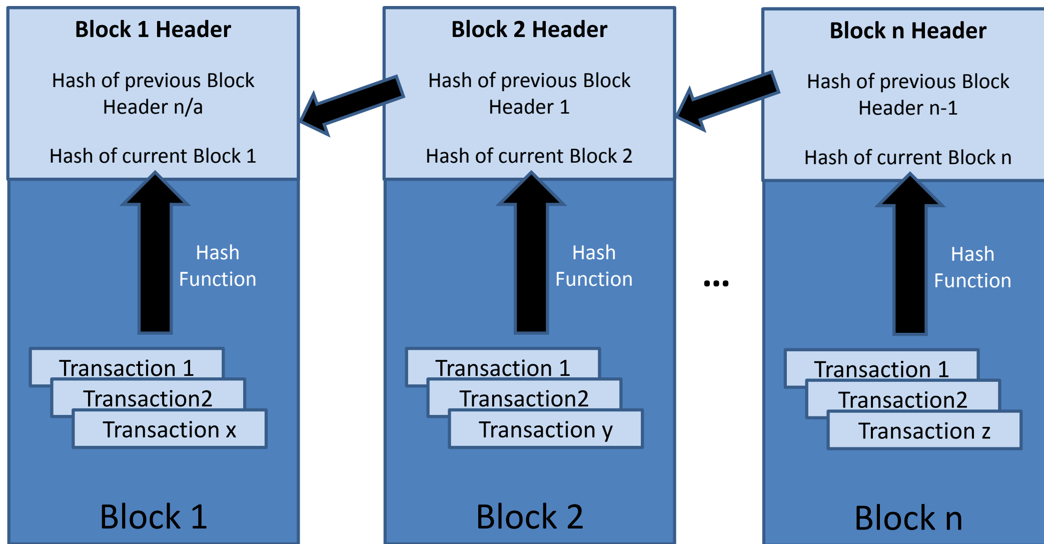

Figure 2.1: Simplified illustration of a blockchain structure [Nak08]

In the last few years, the success and growing prominence of Bitcoin drew more and more attention to its underlying and at this time innovative blockchain concept. This led to the development of additional cryptocurrencies, most of which use some variation of the original blockchain technology.

2.1.2 Blockchains in Cryptocurrencies

The core element of a cryptocurrency are tokens which can be sent from one user to another, like a real currency, e.g. the US-Dollar.

More specifically, the tokens are sent from one address to another. An address is either calculated from the public key of an asymmetrical cryptographic key pair or it is the public key itself. Such a key pair consists of two parts, a public and a private key.

As mentioned above, the address that a user can send coins to is related to the public key. The key is called "public" because itself or the derived address should be known by other parties who want to send funds to it. To each public key, there belongs exactly one private key. More precisely, a public key is calculated via a cryptographic function from the private key. The private key allows the address owner to access coins that were sent to the respective public key. Therefore, the private key is to be kept secret by the user who owns it.

The keys as well as signatures in many cryptocurrencies are created via a cryptographic process called Elliptic Curve Digital Signature Algorithm (ECDSA). The basic algorithm was proposed back in 1999 [JM99] and combines the classic Digital Signature Algorithm (DSA) [Nat13] with elliptic curve cryptography.

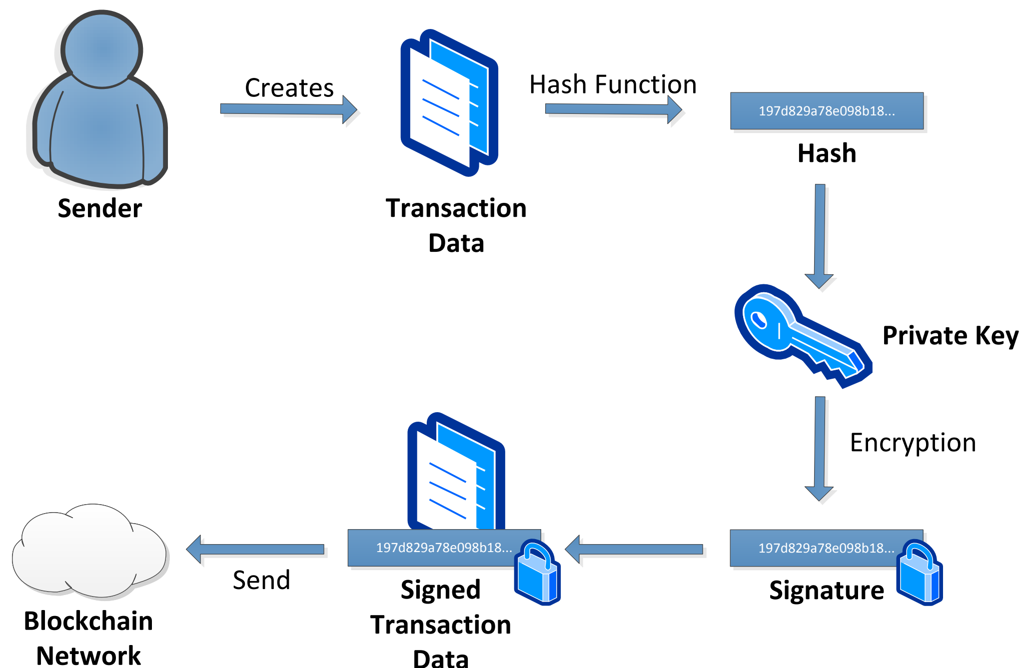

Figure 2.2: Signing transaction data with asymmetrical cryptography

The main feature of asymmetrical cryptography is the following: it is easy to verify that data was encrypted with a private key belonging to a certain public key. On the other hand, the public key itself does not give away any information about the corresponding private key. This means that it is an extremely hard mathematical problem to calculate a secret private key from a known public key.

In order to send coins from one address to another, a transaction is executed. The transaction states how many coins are sent from which sending address to which receiving address. For reasons of authentication, the issuer of a transaction signs the transaction with its private key. This is where ECDSA comes into play. Signing means creating a digital signature of some data with the use of a private key via a cryptographic function. The transaction data (or rather a hash of it) gets signed with the private key, resulting in a digital signature. This signature gets added to the original data and the signed transaction can then be issued. This process is pictured in Figure 2.2.

Receiving parties can now easily verify the authenticity of the sender. Therefore, they first decrypt the signature with the senders public key. The result of this operation is the hash value of the initial transaction data. Then the verifying parties just have to hash the transaction data they received and compare the obtained hash with the hash from the decrypted signature. If the two hashes match, the integrity of the data and the authenticity of the sender are proven.

The decryption process is illustrated in Figure 2.3.

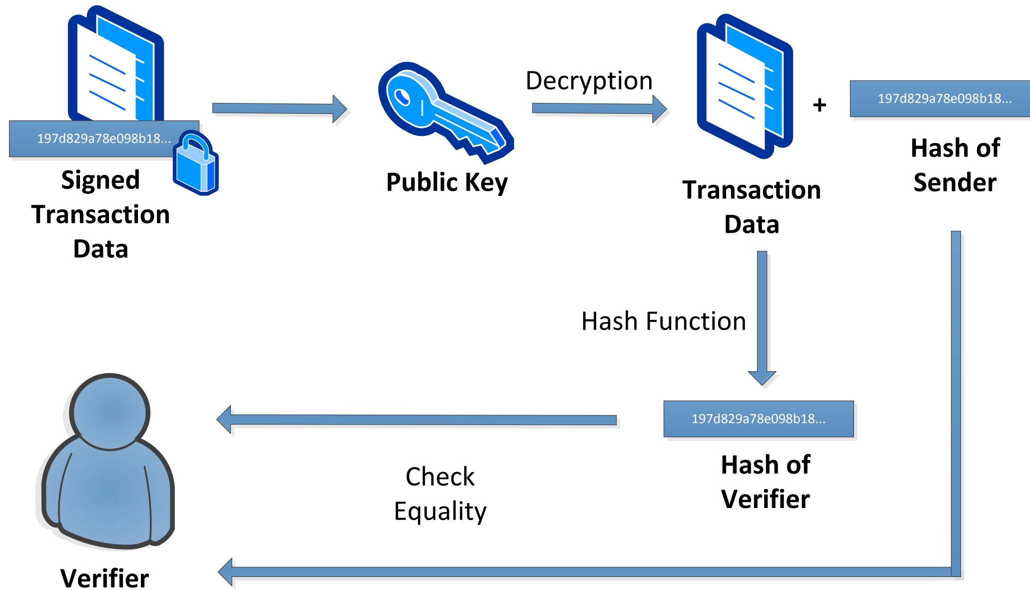

Figure 2.3: Transaction data signature verification with asymmetrical cryptography

With the use of an asymmetrical encryption method for signature generation, the following three properties are ensured [SSUF16]:

- The transaction was issued by the person that was in possession of the private key, i.e. the owner of the coins
- It is not possible to change the transaction once it was sent
- The sender cannot deny the intention of actually wanting this transaction to be executed

In a cryptocurrency context, the blockchain is a ledger where every transaction that was ever made is stored in. Yet, the blockchain itself does not directly store how many coins are held by which address. But by containing a complete list of each transaction, the current state of the token distribution can be calculated at any arbitrary point in the past and at present [Ant14].

2.1.3 Distinctions and Features of Blockchains

Abstracting from the financial aspect, the blockchain itself is nothing more than a data structure, comparable to a classic database.

A first distinction of different kinds of blockchains can be made by considering them permissioned or permissionless. A permissioned blockchain only allows access to a certain group of users, e.g. when it is a blockchain used within a company only by the company's employees, or if several companies work together on a shared blockchain.

Permissionless blockchains on the other hand allow access and participation to everyone [Bit15]. Every user who follows the given protocol can interact with the blockchain, and all users have the same rights.

Another distinction that resembles the one in permissioned and permissionless is public and private blockchains. Public blockchains can be used by anyone willing to do so, private blockchains restrict their userbase according to certain regulations.

A key feature of any blockchain is decentralization. A decentralized blockchain consists of many nodes where every node runs an own blockchain client [PP15]. This corresponds to a peer-to-peer (P2P) network [MKL+02]. Each node stores a version of the blockchain. The nodes communicate with each other and try to reach consensus over the current valid state. That way, no centralized authority is needed to verify the validity of the current blockchain.

A decentralized blockchain is resistant to failures of single nodes. As it is always the majority of active nodes that decides the current blockchain state, it is irrelevant whether single nodes fail. More importantly, it is also resistant to manipulation enforced by a corrupt node. The corresponding mathematical problem is known as the "Byzantine Generals Problem" described by Lamport et al. [LSP82].

In the original problem, a group of Byzantine generals communicate with each other to coordinate an attack on a city. They all have to agree on a certain maneuver, but some of the generals might be traitors trying to sabotage consensus or even create a bad consensus due to manipulated information.

This problem resembles the problem of independent nodes reaching consensus over the current blockchain state. One of the solutions that is used in many blockchain implementations is the following: The "correct" version of the blockchain is always the longest chain [Zoh15]. This makes sense if the creation of new blocks follows the Proof of Work (PoW) principle. That means that a mathematical problem has to be solved via guessing the correct solution in order to add a new block containing recent transactions to the blockchain. As it is a hard problem, a fair amount of time and especially computational power is needed to solve it. Thus, the longest chain is the one where the most computational effort was put it and thereby also the one that is least likely to have been manipulated by an attacker. The people trying to solve this PoW puzzle are called miners. They get rewarded with newly created tokens whenever they are the first to discover a new block.

Apart from the PoW consensus mechanism, there exist many other mechanisms with their respective advantages and disadvantages. Several of them are presented in the following:

- **Proof of Stake (PoS)**

  This consensus mechanism considers a user's stake, i.e. the amount of tokens held by an address, in order to determine the creator of a new block. The user adding a new block is usually chosen randomly, but the higher a user's stake, the higher is also the probability that the user's block is chosen [Nxt14]. There are different variants, e.g. also taking into account how long an address already possesses the tokens [KN12].

- **Proof of Activity**

  Proof of Activity combines both PoW and PoS mechanisms [BLMR14] [TS16]. First, an empty block, i.e. a block without transactions in it, is mined in a PoW manner. When it is found, a certain amount of stakeholders are chosen. Each of these stakeholders must sign the block with their respective private keys. The higher a user's stake is, the higher the probability to get chosen as one of the signatories. This is the PoS component. If one of the signatories is offline, i.e. the block does not get signed within a certain period of time, a new set of stakeholders is chosen. This means that inactive users are left out of the block generation process. Hence the name, Proof of Activity. The transaction fees are split between the one discovering the block and the signatories.

- **Proof of Authority**

  Another consensus algorithm is the Proof of Authority [Par18]. Here, a set of trustworthy authorities is chosen beforehand. Only they have the right to create new blocks, and the majority of these authorities must agree upon a created block. This concept is rather likely to be used in private blockchains.

The advantages and disadvantages of the different presented consensus mechanisms are discussed in Table 2.1.

Table 2.1: Comparison of different consensus algorithms

Mechanism	**Advantages**	**Disadvantages**
Proof of Work	+ Rewards distributed in fair manner relative to computational power + High block generation rewards encourage mining	- Low transaction throughput - High energy consumption - Rewards get smaller over time, might drive away miners
Proof of Stake	+ Low energy consumption + No new tokens have to be issued upon block creation + Transaction scalability	- Wealthy parties decide over network and get richer - Hoarding instead of spending tokens is encouraged
Proof of Activity	+ Active users are rewarded + Attacker needs both computational power and stake	- Combines disadvantages of PoW and PoS
Proof of Authority	+ High transaction throughput + Fast + Cheap in terms of computational power	- Blockchain gets centralized in the hands of few - Problem how to determine trustworthy authorities

After a certain number of blocks are appended to a specific block, the transactions in said block can be considered as safe [BMC+15]. The amount of necessary blocks heavily depends on the respective implementation.

In conclusion, the most important features of any blockchain are:

- Decentralization via multiple nodes that form some sort of P2P network
- Resistance to failure or manipulation of single nodes
- Immutability of data that was once successfully added into the blockchain

2.2 Different Blockchain Implementations

In the following, several implementations of blockchains and the differences between them are investigated.

2.2.1 Bitcoin

Bitcoin is the oldest and probably best known implementation of a blockchain [Nak08].

It is a decentralized blockchain which is used as a digital currency system. There are several thousand nodes participating in the P2P network all over the world [Bit18c]. As already described in Section 2.1.2, Bitcoin uses asymmetrical encryption for generating addresses, i.e. public keys, and private keys that allow a user to send funds from an address. A public key always starts with either a 1 or a 3 and has a length between 26-35 characters [Bit18a]. The private keys are 256 Bit numbers, which equals to 64 hexadecimal digits.

Transactions are often performed by the use of a wallet application. This application saves the private and public key and performs all the necessary steps to satisfy the Bitcoin transaction protocol. In order to perform a transaction, the sender only needs to know the address of the recipient [Fra15]. The transaction steps are as follows [Ant14]:

First, unspent incoming transactions of the sender have to be found. As the blockchain only stores transactions, the difference between incoming and outgoing Bitcoins determines the current balance of an address. If one or more yet unspent transactions that equal or surpass the new transaction's value are available, the wallet can begin to build up the transaction. Therefore, a stack-based, not Turing-complete scripting language is used [Fra15]. The wallet creates a script that contains one or more previous incoming and unspent transactions, as well as a signature generated with the sender's private key, to verify the possession of the Bitcoins. As an output, the sender specifies one or more addresses and also the amount of tokens that is to be sent to each address. For a valid transaction, the sum of Bitcoins of incoming transactions must be greater or equal to the sum of Bitcoins of the outgoing transactions.

Because incoming transactions are always fully used up, the sender can specify its own address as a receiving address in order to send resulting change back to itself. The difference between incoming and outgoing Bitcoins is used as a transaction fee, which will be explained later.

In order to access the funds within a transaction, the receiver also creates a script. This script contains a signature created with the receiver's private key.

The two scripts are being concatenated and run sequentially. If the script evaluates to “True”, the transaction can be considered completed.

Before the receiver can try to solve the sender’s script, the transaction must be added to the blockchain. In order to achieve this, the transaction is first sent to a network node. Each transaction has a unique ID. As the network nodes form a P2P network, the unexecuted transaction is propagated through the network to other nodes. It is now in the “mempool”, which is a term for the set of issued but not yet executed transactions. Transactions from the mempool can be appended to the blockchain via a process called “mining”.

A mining node takes transactions from the pool of not yet executed transactions and puts them together in a block. Such a block contains a number representing its size, a block header, a transaction counter for the amount of transactions included in the block, and the actual transactions’ data (which is by far the biggest part of each block’s size). The block header itself consists of several elements, namely [Ant14]:

- A version number of the Bitcoin protocol that was used
- The hash of the previous block in the blockchain
- A hash of the Merkle-Tree root for the block’s transactions, which is a special encoding of all the transaction hashes
- A timestamp of the blocks creation (created during the mining process)
- The difficulty target, i.e. a parameter for the PoW algorithm (created during the mining process)
- A nonce used for the PoW algorithm (created during the mining process)

After putting together the required information, the miner tries to solve the PoW puzzle. The task is to create a hash which has a value smaller than a certain target. The hashing algorithm is performed multiple times with different nonces until a miner finds an appropriate hash. The miner is now allowed to add the block to the blockchain and the result is again propagated to the other network nodes.

The used hashing algorithm is SHA-256. Its output is always fixed-length and can be interpreted as one big number. Though the amount of digits is always the same, it is not possible to determine the size of the output number beforehand. So finding an appropriate block is based on pure luck. With the difficulty target, adjustments can be made so a block is found approximately every 10 minutes, independent of how many miners perform how many hashes.

The miner who is allowed to append a new block is rewarded with the fees of the block’s transactions as well as a fixed amount of newly generated Bitcoins for each block. The amount of new Bitcoins halves multiple times, the longer the blockchain gets. Because this is the only way how new Bitcoins are generated, the total amount can never surpass 21 million Bitcoins.

Because miners may keep all the fees within their found block, it is a common approach for miners to include transactions with higher fees first. This leads to the general assumption: the higher the fees, the faster a transaction will be executed.

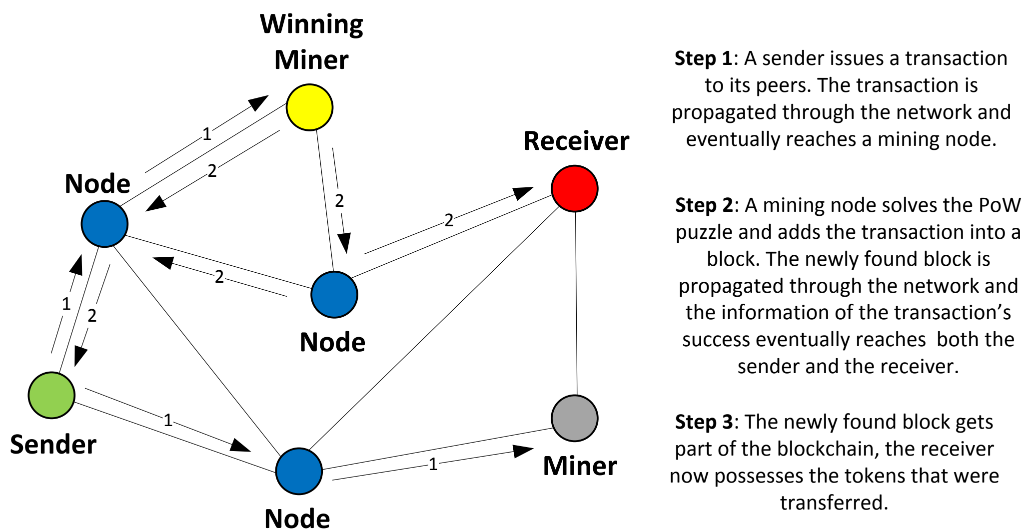

Figure 2.4: Mining and sending transactions in the Bitcoin P2P network

Figure 2.4 shows the different roles of participants in the Bitcoin network. Sender, Receiver and Miner in this figure can be active P2P network nodes, too, but they do not necessarily have to be. This means that Bitcoin accounts and miners do not need to operate their own nodes. It is sufficient for them to communicate with an active node, e.g. over a wallet or a website.

The Bitcoin blockchain is optimized to save space, as it grows larger with every transaction. That is why it does not allow a user to store big amounts of data within it. In fact, there is no mechanism intended to store arbitrary data at all. A transaction can have a description that both the sender and the receiver see, though this data does not become part of the blockchain. Other ways of storing data result from abuse of fields that accept user input. One way is sending a transaction to a receiving address which consists of the data that is to be stored. This method allows the storage of 35 hexadecimal digits, i.e. 17 and a half bytes. Another way is the "OP_RETURN" script opcode of the Bitcoin scripting language. It marks transaction outputs invalid and allows for the storage of up to 80 bytes, though this is not a recommended procedure [Bit17]. All in all, storing arbitrary data is not naturally supported and very expensive considering the transaction fees.

2.2.2 Ethereum

The Ethereum Project was developed after the potential and success of Bitcoin were proven. Many of the concepts from Bitcoin were absorbed directly or slightly changed. In addition to transferring tokens in a blockchain, the founder Vitalik Buterin also added a Turing-complete programming language for smart contracts and decentralized applications (dApps) [But14]. Like in Bitcoin, tokens named Ether (ETH) can be sent from one address to another. Addresses are either part of an asymmetrical key pair, or, in contrast to Bitcoin, smart contract addresses.

The concept of smart contracts was first introduced by Nick Szabo back in 1996 [Sza96]. He describes how real world contracts about various things like property rights or liens could be implemented into computer soft- or hardware. The goal why such contracts should be virtualized is making them harder to be breached or manipulated.

In Ethereum, smart contracts are pieces of code which can be written in different programming languages. The most prominent one is a programming language called Solidity, which was exclusively developed for Ethereum [Sol16]. Each deployed smart contract has an own address and is stored in the Ethereum blockchain. A runtime environment called the Ethereum Virtual Machine (EVM) executes parts of the smart contract code whenever a transaction is sent to the smart contract's address. This can also result in the contract receiving or sending Ether.

The term dApp is used in the Ethereum context to describe an application which is written in a smart contract and interacts with users [Git18a]. As the application's code is stored in the blockchain, it is per se decentralized. The Ethereum project distinguishes between three different types of dApps:

- Financial Applications

  A dApp can create and issue its own tokens which are different from ETH, basically creating its own currency. DApps could also be used to create saving accounts or manage wills. All these application fields focus exclusively on the monetary aspect a dApp can have.

- Semi-financial Applications

  This comprises dApps which have a financial aspect but also a real world reference. An example would be dApps that send money to the first person solving a specific real world problem.

- Non-financial Applications

  The third and last category can be used for dApps that handle e.g. online voting or public administration.

The transactions in the Ethereum network resemble the ones of Bitcoin, though there are some slight differences. A transaction in Ethereum has a receiving address, the sender's signature, an amount of ETH to be sent (can be 0, e.g. when interacting with smart contracts), a data field where arbitrary data can be put into, a gas limit and a gas price.

Gas is used to determine the transaction fee. Each computational step consumes 1 gas, and each byte of transaction data costs another 5 gas. The user can specify a limit, how much gas should not be exceeded by the transaction. If the gas limit is reached, the transaction fails.

The gas price is usually given in "wei", which is a fraction of ETH. One wei is equivalent to 10^{-18} ETH. The gas limit multiplied with the price per gas used determines the maximum fee that a user is willing to pay for a transaction.

The idea behind this concept is preventing users from spamming and clogging the network with unnecessary transactions or huge data blocks. The fees are a reward for the miner who finds the block where the transactions were put into. Like in Bitcoin, the only way of generating new ETH, in

addition to the amount that was issued initially, is the mining reward. Currently, a miner is rewarded 3 newly generated ETH for each new block. This reward was and will be reduced further to cap the amount of maximum ETH circulating.

The PoW algorithm in Ethereum is called Ethash [Eth16][Git17a]. The goal is once again to produce a hash that is smaller than a certain target. The hash is generated on a subset of a big directed acyclic graph (DAG) which changes every 30,000 blocks. As the average time between the finding of two blocks is about 12 seconds, the DAG gets recreated every 100 hours [Git18e]. The used hashing algorithm is a variant of the SHA-3 standard which is called Keccak-256.

In the future, the consensus algorithm will change from PoW to PoS. In Ethereum's proposal of a PoS algorithm, consensus is reached by the participants voting for the valid blockchain and their votes get weighted with the amount of ETH that a participant owns. This process would completely substitute the regular PoW-based mining [Git18f]. The idea behind this procedure is to reduce electricity costs, hinder centralization and mitigate the token inflation.

2.2.3 Hyperledger

Hyperledger was started at the end of 2015 by the Linux foundation [Lin15]. In the meantime, multiple companies joined the project to collaboratively work on the development of open source blockchains. Hyperledger aims for a broad variety of application fields, e.g. manufacturing, banking or Internet of Things (IoT). Therefore, it provides several permissioned blockchain implementations.

What it offers is a modular architectural framework that all Hyperledger projects are built on. The modular components comprise [Hyp17]:

- A Consensus Layer

  Defines how agreement on blocks and transactions is reached.

- A Smart Contract Layer

  Executes business logic embedded in smart contracts.

- Communication Layer

  Handles the message transport between the participating nodes.

- Data Store Abstraction

  Allows different data stores to be used by other modules.

- Crypto Abstraction

  Defines and abstracts the crypto algorithms that are used.

- Identity Services

  Provides authentication, authorization and handles network identities.

- Policy Services

  Handle policies, e.g. for endorsement, consensus or group management.

- APIs

  Defines interfaces for applications and clients to interact with the blockchain.

- Interoperation

  Supports the interoperation between different blockchain instances.

Several implementations of blockchain frameworks have been published since the start of Hyperledger. Each project puts an emphasis on different blockchain aspects. Hyperledger Iroha focuses on mobile application development [Sor18], while Hyperledger Indy has its main advantage in identity management [Git18c].

Another prominent project in this environment is Hyperledger Fabric [Hyp18a]. It is a platform for distributed ledger solutions that looks at blockchains more as data structures and not as a foundation for token-based cryptocurrency applications. Like other Hyperledger projects, it also supports smart contracts (here called "chaincode") and the key mechanism are transactions sent from one participant to another. The architecture is modular so that different components can be replaced or interchanged.

One of the main differences to the other presented blockchain implementations is the fact that it is private and permissioned [ABB+18]. Therefore, only trusted members that were previously approved may participate in a Hyperledger Fabric instance. This shows that small groups like companies that share a production process are the targeted audience. Business-to-business (B2B) blockchain designs demand special requirements that differ from those of public distributed ledgers.

Some of those key features that Hyperledger Fabric focuses on are identity management, privacy and a modular design.

The blockchain itself consists of two elements: The first is a so called "world state" which is a representation of the blockchain's state at a certain point in time. The second element is the "transaction log" which stores all performed transactions. The world state is therefore the result of all executions of transactions in the transaction log up to a fixed point in time.

Hyperledger Fabric supports transactions in multiple channels, and each channel has its own blockchain and privacy management. A dataset in the blockchain is represented by a key-value pair that can either be created, updated or deleted.

The nodes participating in a Hyperledger Fabric instance can be distinguished into three different categories:

- (submitting-)Clients

  This is the entity that an end-user runs in order to issue transactions. For communicating with the blockchain, a client must be connected to a peer.

- Peers

  These nodes maintain copies of the ledger and execute transactions on them. They can also have an endorsing functionality. Every chaincode has an endorsement policy which specifies which peers have to endorse transactions in which way. Only if the endorsement policy gets fulfilled, the corresponding transaction is executed.

- Ordering Service Nodes

  These nodes provide a shared communication channel between peers and clients within a channel. They are also responsible for reaching consensus in a blockchain instance as they order transactions and put them together into blocks.

As there is no native currency that gets sent between users, the core element of interaction is defined in the chaincode. Each channel that multiple peers share with each other has to define functions for interaction. Such a function could e.g. define trading two goods for each other.

Let's assume that both trading parties have a client connected to a network peer, a channel set up between these peers and they have both verified their respective identity and authenticated to the network. The two peers are the only ones connected via the used channel and they both have to endorse each transaction. The steps of a transaction then look as follows [Hyp18b], with a visualization given in Figure 2.5.

1. **Transaction Initiation and Proposal**

   One client initiates a transaction via the peer it is connected to. As a result of the transaction, some part of the chaincode has to be invoked. The initiator's request gets built, signed and sent to both endorsing peers as a proposal.

2. **Signature Verification and Transaction Execution Proposal**

   Both peers have to verify that the transaction proposal is well-formed and that it has not been submitted at some point in the past. Additionally, they have to verify the validity of the signature and the authorization of the initiator. If these steps are successful, the addressed chaincode gets executed with the passed arguments and a proposal for a result is created. At this point, no change to the ledger's state has occurred yet.

3. **Proposal Inspection**

   The two parties' proposed results are compared by the initiator. If they are the same and the transaction would result in a change of the ledger state, the transaction has to be forwarded to an Ordering Service. This is also done by the initiating client.

4. **Building of Transaction**

   The Ordering Service receives a transaction message containing the datasets that have to be read or written as well as the signature of the endorsing peers. The Ordering Service then orders all the transactions it received in the corresponding channel and creates blocks containing the transactions of the channel.

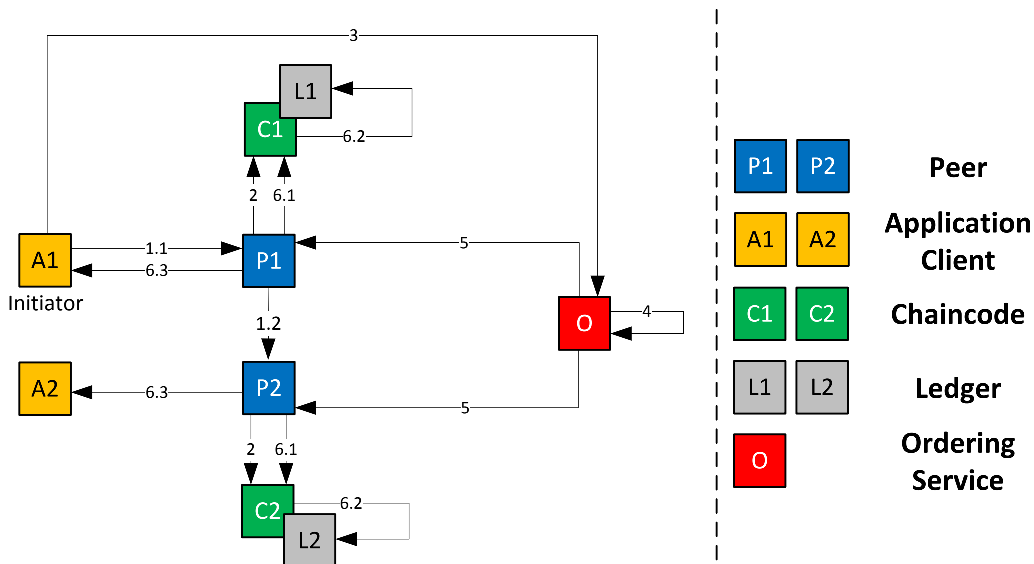

Figure 2.5: Steps of a transaction in Hyperledger Fabric

5. **Transaction Validation and Execution**

   All peers in the channel receive the blocks constructed by the Ordering Service. The requested ledger updates are again checked before the final execution. If this check is successful, the transaction is tagged as being valid.

6. **Ledger Update**

   As a last step, each peer updates its copy of the ledger by executing the transaction. Afterwards, the client which issued the transaction gets informed that the transaction was successfully executed.

In contrast to other blockchain implementations, there exists not one certain consensus algorithm in Hyperledger Fabric. Consensus is rather reached over multiple steps performed by different parties. Each authentication and signature generation along this way can be considered a part of the consensus algorithm. The pure act of constructing blocks though is performed by the Ordering Service nodes.

2.2.4 IOTA

The fourth and last presented distributed ledger technology is IOTA. It was released in 2016 and is currently under the control of the IOTA Foundation [The18b]. Alike Ethereum and Bitcoin, the distributed ledger is permissionless and open source. The main field of application differs though.

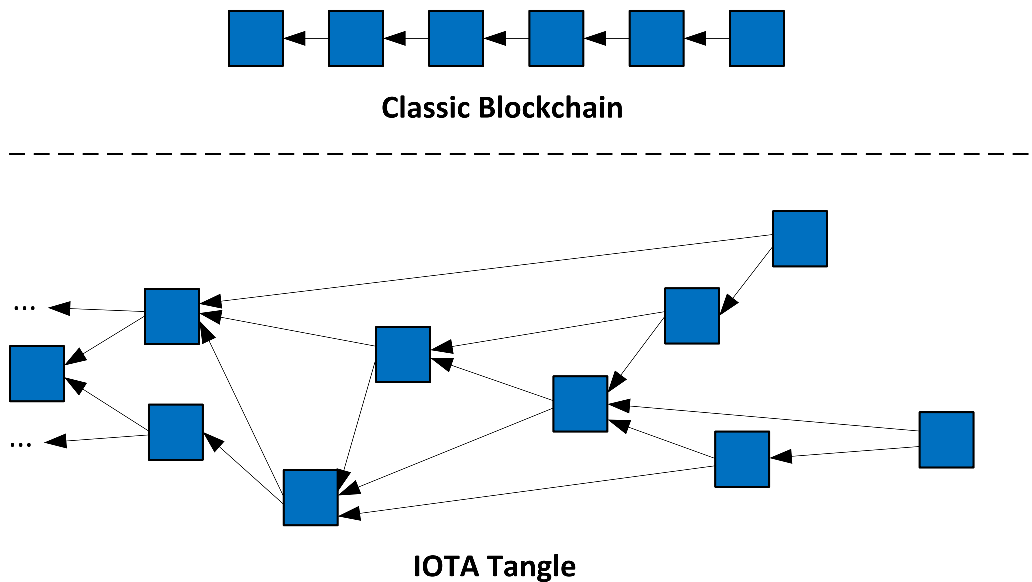

Figure 2.6: Comparison of a classic blockchain with the IOTA tangle

IOTA targets machine-to-machine (M2M) transactions in an IoT environment [Pop17]. Therefore exists a currency within the ledger which is also called IOTA. As the total amount of IOTA tokens is over 2.7 quadrillion, the token is often referred to as MIOTA, which means 10^6 IOTA.

The feature where IOTA most differs from the already presented distributed ledgers is the data structure itself. Instead of a linear blockchain where each block contains the hash of its predecessor, IOTA uses a DAG for connecting single transactions. This DAG is called "the tangle". Like a classic blockchain, the tangle holds all transactions that were ever issued in a chronological order.

Let us now further investigate how the tangle is being constructed: each transaction is represented by a node in the graph. When a user issues a new transaction, this transaction must approve two previous ones. For each approval, a directed edge gets added to the graph. The direction of the edge is always from the approving transaction to the approved one. Every block further backwards in such an approval path gets also approved by the new transaction. This is then called indirect approval.

The structural difference between a classic blockchain and the tangle is illustrated in Figure 2.6. While each block in a classic blockchain contains a multitude of transactions, the IOTA tangle holds single transactions, each of which approves two previous transactions.

In the beginning there was one initial transaction called the "genesis transaction". It held all the tokens that would ever get distributed. This means that no new tokens will ever be created, and consequently there is no process like mining new tokens. Instead of the classic PoW done by miners, every transaction issuer must solve a PoW puzzle themselves in order for the transaction to get sent. This once again prevents spamming the network with a multitude of transactions.

In addition to solving the PoW puzzle, a transaction issuer also has to verify two previous transactions. When a conflict is discovered while approving a transaction, the approval should fail. If not, there is the risk that the transaction approving an invalid one gets considered invalid by following transactions, too.

It still happens that two conflicting transactions both get added to the tangle. The network participants then have to decide which transaction is more likely to be valid. The other one then gets orphaned, meaning it will not be indirectly approved by new transactions anymore.

A transaction is more likely to be valid, the higher its cumulative weight is. Each transaction has a certain weight representing its importance. It is of course not possible for a single user to create many transactions with high weight in a short period of time. The cumulative weight is the weight of a transaction itself plus the weight of all transactions directly or indirectly approving a transaction.

The concept of cumulative weight resembles the block confirmations in other blockchain implementations. The more transactions follow a certain transaction, the safer one can be that it is durably stored.

As there is little PoW necessary to issue a transaction or to sign one, it is easy for an attacker to manipulate the tangle state, as long as there is a low frequency of transactions and participating nodes [IOT18]. Out of that reason, the IOTA Foundation decided to use a coordinator until the network throughput gets high enough to resolve security issues. This coordinator signs every transaction and stores its location in the tangle. Transactions cannot be considered successful until referenced by the coordinator.

This basically makes the current tangle a centralized distributed ledger, as one party has the power to decide which transactions get signed and can be deemed confirmed.

The developers claim that IOTA enables almost instant and feeless transactions. Also, in contrast to classic blockchains, the more transactions are issued, the faster other transactions get verified.

2.3 Blockchains in the Industry

The potential application fields of blockchain technology in the industrial context are numerous. In the following, a few exemplary concepts are presented. Afterwards, reasons and challenges for an industrial blockchain adoption are discussed.

2.3.1 Industrial Application Examples

Three industrial applications where the use of blockchains is discussed or even already implemented are presented in this section.

Machine-to-Machine Electricity Market

Sikorski et al. propose the use of blockchain technology in the electricity market for M2M interaction [SHK17]. The idea is that electricity producers publish their offers, i.e. how much a consumer currently has to pay for a certain amount of electric energy. The consumer can observe and compare different offers and choose the cheapest or generally best fitting electricity provider. Both the offer and the consumer's purchase process take place in a blockchain. The participants have to lock their trading goods, i.e. the electricity provider guarantees the delivery of the offered amount of energy, and the consumer ensures the possession of the required amount of money to pay for it.

The benefit of this model is the fact that no human interaction is needed in the whole process. Consuming machines can specify their energy demand and take care of their power supply themselves. The machine of the electricity provider on the other hand can produce the energy on demand.

The blockchain is used here both for automation of the trading process and as a trustful intermediary, as it guarantees both sides that the trading goods actually exist.

Construction Engineering Management

Another application field proposed by Wang et al. is the use of blockchains in construction engineering management [WWWS17]. They identify three different categories.

The first are notarization-related applications. The need to have a notary verify the validity and integrity of documents is expensive and time-consuming. By using a distributed ledger to track every document creation, update and deletion, the blockchain could replace a notary in some specific points.

A further category are transaction-related applications. If the ownership of goods is controlled by the blockchain, a lot of trades can be automated. This can comprise tangible goods like building materials or construction machines or even intangible goods, e.g. the transfer of certain technology or knowledge.

The third and last identified category is provenance-related applications. As all the supply chains of a construction project would become transparent, it is easy to check if quality standards were fulfilled. If material-related problems arise during the construction, responsible parties can be identified and made accountable.

As an exemplary use case, a process for leasing construction cranes based on a Hyperledger blockchain instance is introduced. In this prototypical blockchain, a crane manufacturer can register a new crane in the blockchain which is then offered for lease to a customer. Therefore, the manufacturer invokes a certain piece of chaincode that creates a record for the new crane. Afterwards, a customer can lease the crane via another piece of chaincode for a certain period of time. If the customer's payment is verified, the crane can be used by the customer for construction works. During the lease period, the customer updates the operational status of the crane once again on the blockchain. This data, e.g. breakdown events, electricity consumption or daily lifting load, updates the crane record.

Like in the previous concept, the blockchain is used here mainly to automate steps that would normally require human interaction. Additionally, the blockchain's immutability is used for property rights and document validity.

3D Printing Supply Chain

Blechschmidt et al. describe how to improve the manufacturing industry by the use of blockchain technology [BS16]. They state that the cost overhead in 3D printing supply chains could be reduced if producers and customers both collaboratively used a blockchain. Therefore, they brought into being a project called "Genesis of Things" [The18a]. Their platform connects owners of 3D printers, customers who need parts printed as a service and designers of 3D parts.

The creator of a 3D design file which can be printed uploads the encrypted design to the blockchain. A customer can now decide to buy a certain amount of the product. Via smart contracts, the pricing as well as other terms and conditions are negotiated between the three parties automatically. Additionally, the best fitting 3D printer in terms of price and local proximity gets chosen. Each product is assigned a unique ID and its whole production history is made transparent via blockchain records.

The design file creator gets a royalty each time a design is printed. The owner of the printer gets paid for every print job. The customer can find the nearest and cheapest 3D printer for the production of otherwise costly spare parts or small series products.

As a proof of concept, cufflinks made of titanium were designed and the 3D printing file was uploaded to the blockchain. Along with the design itself, meta-information like used material are stored as well. A customer can then order any amount of cufflinks produced on demand with the already uploaded design. Each item gets a serial number and, through the transparency of the production process, also a digital product memory.

2.3.2 Remarks on Current Solutions

The exemplary solutions proposing or even actually using blockchains in an industrial context show certain advantages, but also challenges.

Benefits of Blockchain Adoption

Although many of them are just prototypical suggestions, a lot of use cases exist for the application of blockchains in various industrial fields. The main reasons why these applications try to use blockchain technology are automation and a need for trust. If several parties interact with each other and do not know each other, or even worse, do not trust each other, a blockchain can facilitate this interaction. Together with smart contracts, the technology offers transparency, automated execution of previously defined actions and resistance to manipulation by single participants.

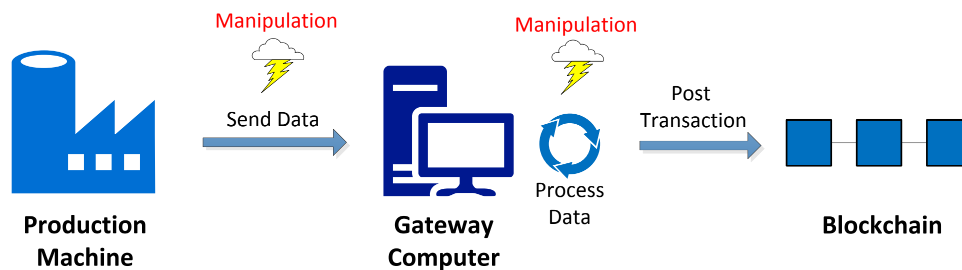

Figure 2.7: Attack vectors on blockchain gateway solutions

When multiple parties work together on a blockchain-based platform, they can form a so called decentralized autonomous organization (DAO). The rules of interaction between the members of a DAO are written in computer code, i.e. smart contracts define the business structure [Cho17]. That way, the need for a neutral but also costly intermediary is no longer present, as the blockchain can fulfill this task.

A further important factor is the immutability of stored data. The decentralized blockchain structure is resistant to both failure and manipulation of single nodes. Once data was successfully stored, it cannot be deleted or altered as long as the majority of nodes does not agree.

Challenges

When considering blockchain technology within a manufacturing process, the integrity of data must be ensured on the whole way from production machine to blockchain. Machine data obtained e.g. by sensors must not be altered before reaching the blockchain [Sob17]. Otherwise, wrong or manipulated data gets immutably stored, destroying the whole concept of trust.

Many of the existing prototypes use some kind of gateway between blockchain and production machine [Ste17]. Such a gateway consists of a software that communicates with a production machine, collects data and stores it in the blockchain. The software is run on a general purpose computer, together with a full blockchain node. That way, both the production machine and the blockchain interaction can be achieved by a single computer.

While this is the easiest way to connect a production machine to a blockchain, gateway solutions can pose severe security risks. Possible attack vectors can be seen in Figure 2.7. On the way from the production machine to the computer, the sent data could potentially be manipulated. That way, measurement values could be involuntarily changed before reaching the blockchain. Further on, the computer running the blockchain node and communicating with the production machine could be attacked, too. The computer needs an Internet connection and therefore offers a variety of attack

vectors. Every virus or malware running on the computer poses severe risks to the production measurement process. If the collected data gets altered before a blockchain transaction is built up, the integrity of the blockchain can then no longer be guaranteed.

That is why the concept of blockchains for production data only makes sense if a secure way from the sensor measuring data to a finished blockchain transaction is ensured.

One way to reduce the described security risks is changing the gateway design. If instead of a general purpose computer, a more primitive, single-purpose computer is used, this could potentially increase the security of the whole system.

The goal of this work is to answer the question: is it possible to develop a concept that enables communication with a blockchain without the necessity of a general purpose computer running a blockchain client? In the following chapters, this topic will be further investigated.

3 Conception

This chapter comprises the development of a concept for a secure transmission of production data. Beforehand, requirements are presented and used technologies as well as functionalities are introduced.

3.1 Requirements

In the following section a concept for data transmission between a production machine and a blockchain will be developed. Therefore, one first needs to identify requirements that the design has to meet.

3.1.1 General Requirements

- **Use Appropriate Blockchain Implementation**

  As already stated in Section 2.2, a lot of different implementations based on blockchain technology exist. As the underlying concepts differ from each other, a technology has to be chosen that is best fitting for both the functional and non-functional requirements.

- **Hardware-near Implementation**

  The developed prototypical component has to run on a platform that is close to hardware, so it can both retrieve data from a production machine and send it to the blockchain.

- **Direct Sensor Connection**

  Retrieving data has to be able without the need for an intermediary between a sensor or a production machine and the prototype.

3.1.2 Functional Requirements

- **Sending Data to the Blockchain**

  The component must be able to communicate with a blockchain. As transactions are the core element of blockchain communication, the component must be able to issue transactions.

- **Permanent Storage**

  The designed prototype must be able to send production data to the blockchain which is then put into a block and stored immutably. The successful sending of a transaction must be checked.

- **Retrieving Blockchain Data**

  The prototype has to be able to access data that is already stored in the blockchain.

- **Minimize Time Between Transactions**

  As the production machine may have to update values frequently, building and sending a transaction may not take longer than 10 seconds.

- **Real World Interaction**

  The designed component must offer means to communicate with the real world and obtain data that is to be stored.

3.1.3 Non-functional Requirements

- **Minimize Security Risks**

  Gateway solutions are vulnerable to attacks like data manipulation on the way from machine to gateway or while a transaction is built in the gateway, see Section 2.3.2. The developed system architecture must have as little attack vectors from the outside as possible to minimize security risks.

- **Measure Degree of Trust**

  Along with the prototype concept development, it is also required to develop an approach to measure the degree of trust that some data is durably stored in the blockchain.

3.2 Choice of Technology

Before developing a design, different decisions concerning the used technologies have to be made. In this case, it comprises both hardware and software technologies.

3.2.1 Hardware Technology

The key point of the design is that it differs from the usual gateway solutions. This means, it is not making sense to run the design on a general purpose computer. The platform should have as little underlying operation system functionalities as possible. This is useful for various reasons:

First, it reduces the possibility for attacks. For an attacker it is easier to manipulate a machine the more interfaces it has to the outside. By having a hardware as primitive as possible but still powerful enough to complete its task, the amount of attack vectors is minimized.

Secondly, the less overhead caused by a full operation system, the faster simple calculations and operations can be made.

Besides, the component should be used to retrieve and process machine data. A lot of machines, especially older ones, do not offer high-level interfaces for communicating with the world outside. They often use a serial data bus interface to exchange data, and a general purpose computer might not be able to interpret the data without the use of additional software.

That is why a microcontroller is the ideal platform to program the required prototype on. A microcontroller unit (MCU) can be defined as "...a computer present in a single integrated circuit which is dedicated to perform one task and execute one specific application" [Tec18].

One central feature that the MCU is required to have is a WiFi capability. This is essential for the MCU to exchange data with a blockchain that is only accessible via the Internet. This requirement massively narrows down the choice of an appropriate MCU.

Computational power and memory storage are negligible criteria, as the MCU will not be needed for complex timing-critical calculations or as a storage of big amounts of data.

A further criterion is the price of the MCU. It should be kept low in order to be able to equip multiple machines with the MCU in the future.

All these criteria led to the NodeMCU with an EPS8266 WiFi module as the technology of choice [Nod18b]. NodeMCU is the name of a firmware as well as the corresponding MCU board. The firmware is based on eLua and it runs on the ESP8266 WiFi module which has a Tensilica Xtensa L106 processor [Nod18a] [Esp18].

Programming the MCU can be done via a Universal Serial Bus (USB) interface and with different tools. Apart from Lua, there are several other programming languages supported by the ESP8266. The programming language used in the scope of this work is Arduino C/C++ [Ard18][Gro18].

The NodeMCU Development Kit board is powered with 5V via micro USB either from a regular USB port or with a power adapter. It has a total of 30 pins which can be plugged e.g. in a development breadboard. Some of the pins have general purpose digital input/output functionality, others are ground pins or constant 3.3V output. Certain pins do also support interfaces like universal asynchronous receiver-transmitter (UART) or Serial Peripheral Interface bus (SPI).

Figure 3.1 shows the layout of the board and the functions of the individual pins.

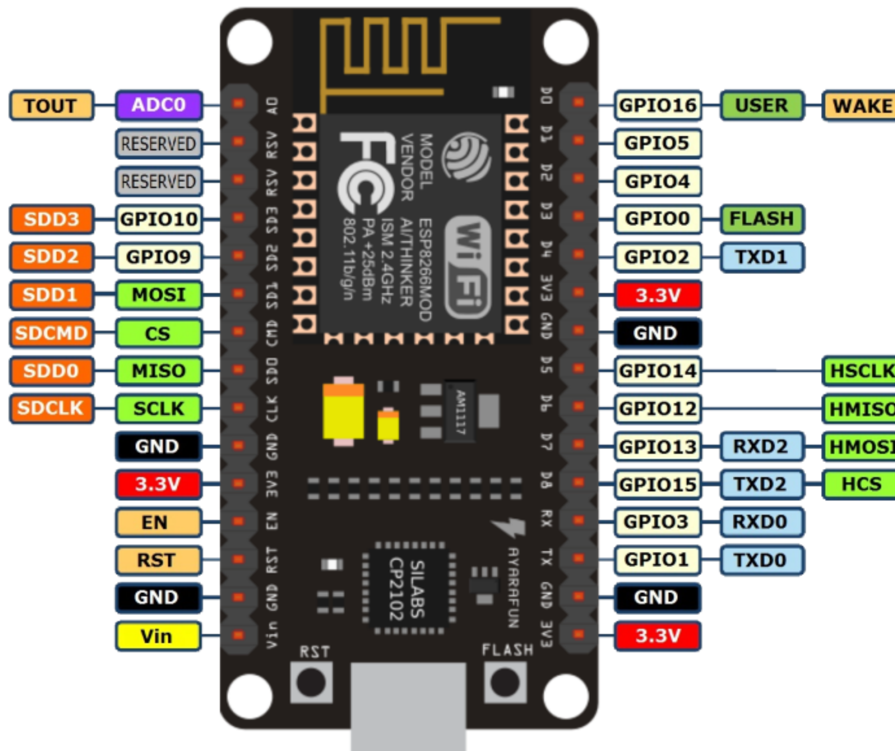

Figure 3.1: Layout of NodeMCU pins [Sah17]

The board can be purchased on the Internet for less than 10 Euro a piece. The low price together with the broad functionality and WiFi capability makes the NodeMCU the ideal choice for programming the desired prototype.

3.2.2 Software Technology

In the following, some decisions about the used software components are discussed. They comprise the targeted blockchain implementation as well as the prototype's programming language and development environment.

Blockchain Implementation

The four blockchain implementations that were presented in Section 2.2 are some of the best known and most widely used ones. While Bitcoin and Ethereum are similar to each other, a comparison to Hyperledger Fabric or IOTA is more sophisticated and will be done as a second step.

Table 3.1 gives a brief overview of the four presented blockchain implementations. Bitcoin and Ethereum both offer an open source blockchain implementation with their respective tokens, BTC and ETH. In both technologies, transactions can be issued sending tokens from one address to

Table 3.1: Overview of different blockchain implementations

Implementation	**Main Features**
Bitcoin	- Public, permissionless, decentralized - Transactions with a block time of approximately 10 minutes - Sending and receiving of tokens, little to no user defined data can be stored
Ethereum	- Public, permissionless, decentralized - Smart contracts enabling dApps - Arbitrary data can be added to transactions, though increasing fee
Hyperledger Fabric	- Private, permissioned, decentralized - Groups of users set up own blockchain instances - No tokens, no mining, focus on exchange of data
IOTA	- Public, permissionless, currently centralized (to be decentralized in the future) - DAG with single transactions, no classic blockchain - Fast and cheap M2M transactions

another. For an industrial application, sending tokens plays a subordinate role. More important is the ability to send arbitrary data, e.g. information about a fabrication process or a machine's status. But the data should not be just sent to the blockchain, but also processed there and potentially trigger further events or actions. That is where smart contracts come into play. Bitcoin does not offer any smart contracts or similar possibilities in contrast to Ethereum. The latter was specifically designed to expand Bitcoin's concept e.g. for the development of smart contracts [But14]. As smart contract functionality is essential for an industrial application, Ethereum is superior to Bitcoin in this aspect. In addition to smart contracts, Ethereum also has a shorter block time (10-15 seconds in contrast to 10 minutes) which makes new transactions visible faster. That is why Ethereum is superior to Bitcoin for the required use case.

As Bitcoin has been ruled out as a potential technology, Ethereum must now be compared to Hyperledger Fabric. This comparison cannot be done as easily as the one before, because Hyperledger Fabric has a fundamentally different design and therefore different properties. Hyperledger Fabric targets a group of users who want to work together on one Hyperledger Fabric instance, which consists of private and permissioned blockchains. An instance has to be initially set up and rules for endorsement and interaction have to be agreed upon. While this makes sense for a private, B2B blockchain, it also poses a huge overhead of work that has to be invested before there is an up and running instance. For a prototypical client implementation, a blockchain with one single peer is rather pointless. As the main task of this work is designing a prototype, it makes more sense to use an already existing blockchain instance. Additionally, a blockchain client for Hyperledger Fabric would result in a very special solution tailored for the many specific properties that this implementation has. When designing a prototype for e.g. Ethereum, the concepts can be transferred to many similar blockchain implementations with little effort. Therefore, this also facilitates the portability of the design. That is why Ethereum is favored over Hyperledger Fabric in this work.

Last but not least, IOTA is to be compared with Ethereum. IOTA focuses on M2M transactions which are fast and cheap. Yet it does not support any kind of native smart contracts. This means that a transaction can be issued and also measured data from a production environment can be put into the transaction, but the data would only get stored. IOTA is still undergoing a lot of development and the amount of transactions in the network is rather low, with a few transactions per second [The18c]. Another point of criticism is the de facto centralization via the coordinator which is still present at the time of this work's creation in June 2018. If one day the IOTA Foundation decided to lock out certain accounts, there is little that could be done about it. Out of all these reasons, Ethereum should be preferred over IOTA, at least until IOTA overcomes the majority of the above stated problems.

Consequently, Ethereum is the technology of choice for this work's blockchain prototype.

Integrated Development Environment

As mentioned in Section 3.2.1, the NodeMCU will be programmed with Arduino C/C++. Therefore, the use of the Arduino integrated development environment (IDE) is recommended. The version used for developing the prototype is 1.8.2. The IDE can be configured to program a variety of different MCU boards with their respective properties. Downloading and managing libraries for the supported boards can also be done directly in the IDE.

A design, here called sketch, is written and compiled in the IDE. If there are no errors, the IDE uploads it to the connected MCU board via the computer's COM port. With a serial interface, the connected MCU generates output which is readable on the computer screen, e.g. for debugging purposes. The monitor necessary for this is included in the Arduino IDE.

3.3 Transaction Detail

The core element of each blockchain are transactions which are executed and stored there. The details of the Ethereum network as well as transactions within it are investigated in the following.

3.3.1 Client

The Ethereum network is formed by a multitude of P2P nodes all around the world. The specific instance of a node is called client. There exist several implementations in different programming languages. The two most used implementations are "go-ethereum" written in Go and "Parity" written in Rust. Apart from these, there are also implementations in C++, Python, Java and a few other programming languages [Eth18a]. The clients communicate with each other, build transactions and retrieve information about the blockchain's status. Therefore, the clients are constantly connected to the Internet in order to regularly update the ledger. Additionally, a client requires a huge amount of hard drive space, as it has to download and maintain the whole blockchain. The exact amount of necessary space also depends on the chosen client implementation. The following formula describes the daily growth of the Ethereum blockchain size.

Let λ be the daily growth rate of the blockchain in megabytes, s the size of a block in kilobytes and f the time between the generation of two blocks in seconds. The daily growth rate then is:

$$\lambda = \frac{s \cdot 60 \cdot 60 \cdot 24}{1000 \cdot f}$$

When assuming a block size of 15kb, which is slightly below the average block size in the time window from May 2017 to May 2018 [Eth18c] and an average time between two blocks of 12 seconds, the equation results in the following:

$$\lambda = \frac{15kb \cdot 60 \cdot 60 \cdot 24}{1000 \cdot 12s} = 108mb/d$$

This means that the size of the Ethereum blockchain is currently growing with approximately 100 mb a day, or 3 gb per month.

The problem of growing size led to the development of so called “light clients” which download only the headers of all the blocks. If necessary, additional information gets downloaded on demand, which drastically reduces the required hard drive space [Git17b].

For the most prominent client implementation, go-ethereum, there exists a command line interface called “geth”. When running an Ethereum node, geth allows the execution of remote procedure calls (RPCs) to interact with the blockchain.

3.3.2 Communication Interface

This section explains the mechanics of RPCs and the data interchange format JavaScript Object Notation (JSON). They are both components of the main interface for interaction with the Ethereum blockchain, which is JSON-RPC.

Remote Procedure Calls

The term RPC comes from the distributed computing domain and describes a concept for calling functions on a remote machine [BN84]. The goal of an RPC is the execution of a function on a remote machine as easy as if it was executed on the local machine. The concept is often used in a client-server environment. The server offers functions for clients to call. When a client calls one of these functions, it sends the necessary parameters. The server then processes the request with the given parameters and generates an appropriate response which is sent back to the client. This form of communication is therefore called a request-response protocol.

In a classic RPC, the client is blocked while waiting for the server’s response. That is why RPC is considered a synchronous protocol. In contrast to this design, asynchronous protocols prevent the client from actively waiting for a response. It can execute other functions or process data while passively waiting.

A client also has to handle the situation that the server is not sending any response at all. This can happen due to network failures or the server currently being offline.

Listing 3.1 Exemplary JSON object for the user of a web shop

```
{
  "customer": {
    "username": "Randomuser",
    "email": "Randomuser@mailprovider.com",
    "shippinginformation": {
      "country": "Somecountry",
      "city": "Famouscity",
      "zipcode": "12345",
      "street": "Firststreet",
      "housenumber": 7
    },
    "recentorders" : [
      "thisitem",
      "thatitem",
      "anotheritem"
    ],
    "useractive": true
  }
}
```

JavaScript Object Notation

JSON is a text-based data interchange format [Bra14] [JSO18]. It is language-independent and easily readable for humans. In addition to that, it is also easy for machines to parse or generate the JSON format. It is often used when structured data has to be serialized. JSON offers two different structures: collections of name-value pairs (called objects) and ordered lists (called arrays). Some forms of these two data structures exist in any modern programming language, making JSON widely usable.

Apart from objects and arrays, JSON has four primitive data types. They are strings consisting of arbitrary unicode characters, boolean values true or false, numbers and the null value.

An object is surrounded by curly brackets ({...}) and contains an unordered set of name-value pairs, separated with commas. An array starts and ends with square brackets ([...]) and contains ordered values, also separated with commas.

JSON structures can also be arbitrarily nested, enabling the representation of more complex objects.

Listing 3.1 shows an exemplary JSON object representing the user data of a web shop customer. Note that "useractive" is a boolean value and "housenumber" is a number, hence they both do not require quotation marks. Several objects are nested in this example and "recentorders" consists of an ordered list with three elements.

JSON-RPC

The two previously described concepts are combined together in a protocol called JSON-RPC [JSO13]. The version currently used in Ethereum is 2.0. JSON-RPC is a stateless RPC protocol built on JSON data structures and it can be used in various message passing environments, e.g. Hypertext Transfer Protocol (HTTP) or sockets.

Alike other RPC implementations, it is request-response-based. A request object has these members:

- **jsonrpc**

  A string specifying which JSON-RPC protocol version is used. In Ethereum, this string should always be “2.0”.

- **method**

  This string is the name of the remote method that is to be called.

- **params**

  If present, this is a JSON structure containing the parameters for the function call. It can either be an object containing name-value pairs for each parameter of the called function, or an array where the parameters have to be in a fixed and correct order. If no parameters are needed for the called function, this member may also be omitted.

- **id**

  This identifier can be a string, a number or null. It is used by the client to signalize that it expects a response from the server. The server’s response then contains the identifier of the client’s request. If this field is omitted, the client indicates that it does not want to receive a response. Such a request without corresponding response is called a notification.

Upon receiving a request, the server processes it with the delivered parameters. After the request is processed, the server sends a response object of the following form back to its client:

- **jsonrpc**

  A string specifying which JSON-RPC protocol version is used. In Ethereum, this string should always be “2.0”.

- **result**

  This member only exists if the function call was successful. The absence of the member indicates that an error occurred in the remote procedure. The value of this member depends on the function that was called on the server. It contains the result of the function call that the client made.

- **error**

  If this member is present, an error occurred during the RPC. It then contains an error object giving further information about the error as well as an error code. This member may not be sent if the function was executed successfully.

- **id**

  This member is required in a server's response. It must be the same id as in the client's corresponding request.

Ethereum nodes, i.e. clients, offer an application programming interface (API) based on this JSON-RPC protocol [Git18d]. A multitude of methods allow the interaction with the Ethereum blockchain. The RPC endpoint is localhost. An application that runs on the same machine as the blockchain client can call these methods. Note that the blockchain client hereby becomes the RPC server, with the application being the RPC client.

3.3.3 Building a Transaction

Transactions in Ethereum can be sent between any two accounts. The accounts can either be externally owned accounts (EOAs), i.e. an account controlled by a private key, or contract accounts [Eth18b]. A transaction always has to contain the following information: the address of the receiver (EOA or smart contract), a signature identifying the sender and proving the intention to send the transaction, an ETH amount to be sent, a data field for arbitrary data, a maximum limit of gas to be used by the transaction and a price per unit of gas. In addition to these values, a nonce is used to count the outgoing transactions of an address. With each transaction that the sending address issues, the nonce gets increased by one. That way, an order can be obtained when many transactions are issued within a short time frame. If the nonce of a transaction is higher than the account's expected nonce, the transaction does not get processed immediately by the Ethereum network. The transaction is pending until enough other transactions from the same account have been made. Finally, when the expected nonce equals the nonce of the pending transaction, it gets executed.

A schematic for the effect of nonces is given in Figure 3.2. If the two transactions with the same nonce are issued shortly after each other and the second transaction pays a higher gas price, then the first transaction is canceled. This only works if the earlier transaction was not yet added into a block. The transaction with nonce 3 does not get put into a block before a further transaction with the nonce 2 is issued.

Optionally, a further field can be added to transactions, the chain ID. This is an identifier for the different Ethereum blockchains, i.e. the main blockchain and several test blockchains.

Before a transaction's signature is calculated, the transaction is called unsigned. As a JSON object, an unsigned transaction looks as shown in Listing 3.2. All the values except for the chain ID are given in strings starting with the prefix "0x" and are interpreted as hexadecimal values. v, r and s together form the signature. Because this is an unsigned transaction, the values for the signature elements are not yet filled in.

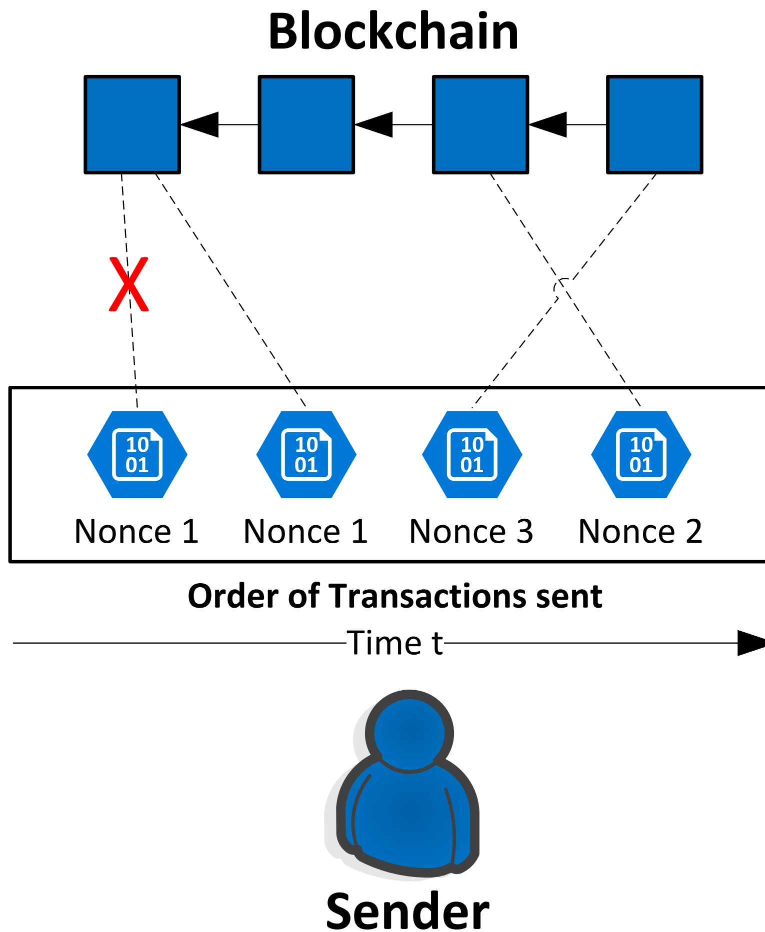

Figure 3.2: Effect of nonces in the Ethereum blockchain

The necessary mechanisms to get from an unsigned to a signed Ethereum transaction are investigated in the following.

Recursive Length Prefix

The purpose of the recursive length prefix (RLP) is the serialization of arbitrarily nested arrays containing data in the Ethereum network [Git18g]. The output is a special encoding consisting of one single array containing all the information of the nested arrays. The now serialized data can be sent as a single string and afterwards the receiver can decode it. RLP does not encode the data contained in the arrays itself, but only the structure of the nested arrays.

Listing 3.2 Unsigned Ethereum transaction as a JSON structure

```
{
  "nonce" : "0x00",
  "gasPrice" : "0x1234567890",
  "gasLimit" : "0x5500",
  "to" : "0x1234567891234567891234567891234567890000",
  "value" : "0x1234",
  "data" : "0xabcd",
  "chainId" : 1,
  "v" : "",
  "r" : "",
  "s" : ""
}
```

In Ethereum, the data that is to be serialized consists of strings or integer values. Characters of a string get their ASCII value as a hexadecimal number assigned and therefore have a size of 1 byte. Hexadecimal values (as seen e.g. in Listing 3.2) take one byte per two hexadecimal digits.

The rules for the RLP encoding then are the following [Tu18]:

- If the input to the RLP encoding is just a single byte in the range [0x00, 0x7f], then the output is the byte itself.
- The output of an empty string or an empty byte is 0x80. This may not be confused with the byte 0x00, because the encoding of this byte would be itself.
- If the input is a single byte in the range [0x80, 0xff], it is a special character as it is out of the alphanumeric ASCII table range. Such a byte is encoded with a 0x81 followed by the byte itself.
- If the input is not just a single byte, but a string with a length of 2 to 55 bytes, the encoding consists of two parts. The first part is the byte 0x80 plus the length of the input string in bytes. The second part is the bytes of the string.
- Should the input string be longer than 55 bytes, then the encoding consists of three parts. The first part is a byte with value 0xb7 plus the length of the second part in bytes. The second part is a hex value for the length of the input string represented in bytes. The third and last part is again the bytes of the input string as hex values.
- Not just strings, but also arrays can be encoded. The encoding of an empty array returns 0xc0 as an output.
- If the input is a list with a total of at most 55 bytes, the RLP encoding is a byte with value 0xc0 plus the length of the list followed by the encoded values for each list item.
- Should the input be a list which is more than 55 bytes long, the encoding once again consists of 3 parts. The first part is a byte with the value 0xf7 plus the length of the second part. The second part is a hex value for the length of the input in bytes. The third and last part is a concatenation of the RLP encodings for each item in the list.

This set of rules allows for the encoding of strings and arbitrarily nested lists. The RLP decoding is done by the receiver and is therefore not needed when building a transaction. Thus, it is not further explained here.

An RLP encoding of the string “Ethereum” would result in the output [0x88, 0x45, 0x74, 0x68, 0x65, 0x72, 0x65, 0x75, 0x6D]

Keccak-256

Keccak-256 is a hashing algorithm used in several steps throughout the whole Ethereum network [BDPA11]. It is similar to the well known hashing standard SHA-3, but the two algorithms indeed produce different outputs and may not be confused with each other.

The hashing algorithm takes an arbitrarily long input string and generates an output of exactly 32 bytes. Hashing the same input string again results in the same output. As with every hashing function, the initial input cannot easily be calculated from a known output.

One important application area for Keccak-256 in Ethereum is the generation of account addresses. An EOA’s address is not the public key of the asymmetrical key pair itself, but it is derived from it. More precisely, the public key gets hashed with Keccak-256 and the rightmost 20 bytes of the output yield the EOA’s address [Woo18]. The addresses are normally represented as hexadecimal values, and so every Ethereum address has a length of 40 hexadecimal digits.

Another crucial step were Keccak-256 is used happens right before a transaction is signed. The signature creation algorithm does not take the whole transaction data and generates a signature for it. Instead, it signs only the hash of a transaction. The hash is generated on the RLP of the unsigned transaction.

Keccak-256 is also used to derive an identifier for issued transactions, i.e. a transaction ID. As an input to the hash function, the unsigned transaction data as well as a valid signature are given. The output of the function is a 32 byte long string which unambiguously identifies a transaction. This identifier is also stored in the blockchain along with the transaction detail.

Elliptic Curve Digital Signature Algorithm

ECDSA is a cryptography method which became a standard in the late 1990s [JMV01]. The classic DSA relies on the difficulty of solving the discrete logarithm problem. The problem statement is the following [BG04]:

$$\text{Given } a,\ y, \text{ find } x \in \mathbb{Z} \text{ solving the equation } y = a^x \bmod G$$

If an adequate group is chosen, there is no method known yet to solve the discrete logarithm problem efficiently. This property makes it an interesting candidate for the use in cryptography. ECDSA differs from the classic DSA by the choice of its underlying group. In ECDSA, the group consists of points on an elliptic curve over a finite field. As the discrete logarithm problem of elliptic

curves is believed to be harder solvable than the classic discrete logarithm problem, ECDSA offers several advantages in comparison to DSA. The strength-per-key-bit in ECDSA is greater than in DSA, resulting in smaller parameters offering the same level of security. This also enables a faster calculation of keys or signatures and reduces the storage space for them.

Though the basic steps of the algorithm are clearly described, there exist different variants of it. This is due to the use of different elliptic curves. An elliptic curve E_k defined over a field K of characteristics $\neq 2$ or 3 is the set of solutions $(x,y) \in K_2$ to the equation [Kob87]:

$$y^2 = x^3 + ax + b, \text{ with } a,\ b \in K$$

Ethereum uses the secp-256k1 curve for both key pair generation and transaction signatures [Woo18]. The algorithm creates key pairs with a length of 32 bytes for the private key and 64 bytes for the public key. A once generated key pair can be used for an arbitrary amount of transactions.

The act of signing a transaction though is needed more frequently than the generation of new key pairs. We recall that each Ethereum transaction has three parameters for its signature, v, r and s. The input to the ECDSA sign function is the hash of the unsigned message's RLP encoding and the private key (both 32 bytes long). The generated signature has a length of 2 x 32 bytes, which then get assigned to r and s. The ECDSA algorithm includes randomness, which leads to the following behavior: even if the same data gets signed with the exact same private key, the resulting signature is different each time.

ECDSA has the property that the public key of a signature can be obtained by knowing the signature itself as well as the data that was signed. Because both of these parameters are known once a transaction is signed and issued, the receiving nodes can calculate the sender's public key, hash it and thereby receive the sending address. This procedure of re-calculating the address from the signature has two advantages: first, the transaction data gets smaller, because the sending address does not have to be explicitly added to the transaction. Apart from that, the signature verifies that the sender is indeed the owner of an account's private key.

Unfortunately, the so calculated public key is ambiguous, as there are always two possible values solving the equation. This means that out of the two possible solutions, only one public key is the correct one. That is what the v component of the signature is used for. It is called the "recovery parameter", as it is used to recover the correct public key of a transaction. Necessary for the correct recovery is the information whether the calculated point on the elliptic curve has an odd or even y coordinate. This information can be stored in 1 bit. However, the v component also stores the chain ID, i.e. an identifier distinguishing the different Ethereum blockchain instances. Thus, the value for v is calculated with the following formula [Git18b]:

$$v = \text{ Chain ID } \cdot 2 + 35 + \text{ Parity Bit}$$

After a transaction was successfully signed, it is not possible to change any of its data without the signature becoming invalid. A receiving node checks the validity of the signature against the transaction data. If any data is changed on the way from signature generation to a receiving node, the signature check will fail. So if a node receives a transaction that can be considered invalid, it gets rejected with an error code.

Steps of Building a Transaction

Now that all necessary prerequisites and functions used for building valid Ethereum transactions are presented, we will take a closer look at the single steps. They are shown in Figure 3.3.

Step 1: Building an Unsigned Transaction As a first step, the different parameters of an unsigned transaction have to be determined. If this is done, the result is a transaction as shown in Listing 3.3. It does not have to be organized as a JSON structure, but for easier readability a transaction will be represented as such in the scope of this work.

Listing 3.3 Step 1: Creating an unsigned Ethereum transaction

```
{
  "nonce" : "0x0c",
  "gasPrice" : "0x3b9aca00",
  "gasLimit" : "0xc350",
  "to" : "0x9876543210987654321098765432109876543210",
  "value" : "0x1388",
  "data" : "0xc0de",
  "v" : "0x03",
  "r" : "",
  "s" : ""
}
```

The nonce indicates that it was the 12th transaction outgoing from this account. The gas price is exactly one billion wei (1 gwei). The gas limit is 50000 wei, meaning the transaction fee will be not higher than 0.00005 ETH. The amount of sent ETH to the receiver address is 5000 wei. Note that the Chain ID has the value 3 (for the Ropsten test net) and is used as a value for the parameter v of the unsigned transaction. r and s do not have a value yet.

Now that all relevant data for a transaction is collected, one can proceed with the next step.

Step 2: Creating an RLP Encoding of the Unsigned Transaction The transaction data is interpreted as strings (omitting the “0x” prefix) and the RLP encoding is performed. Because the names of the different parameters are not part of the RLP, the data has to be arranged in a specific order. The order is the same as in the JSON object shown in Listing 3.3, though a JSON object is a set of values rather than an ordered list. The plain data organized in a list is shown in Listing 3.4.

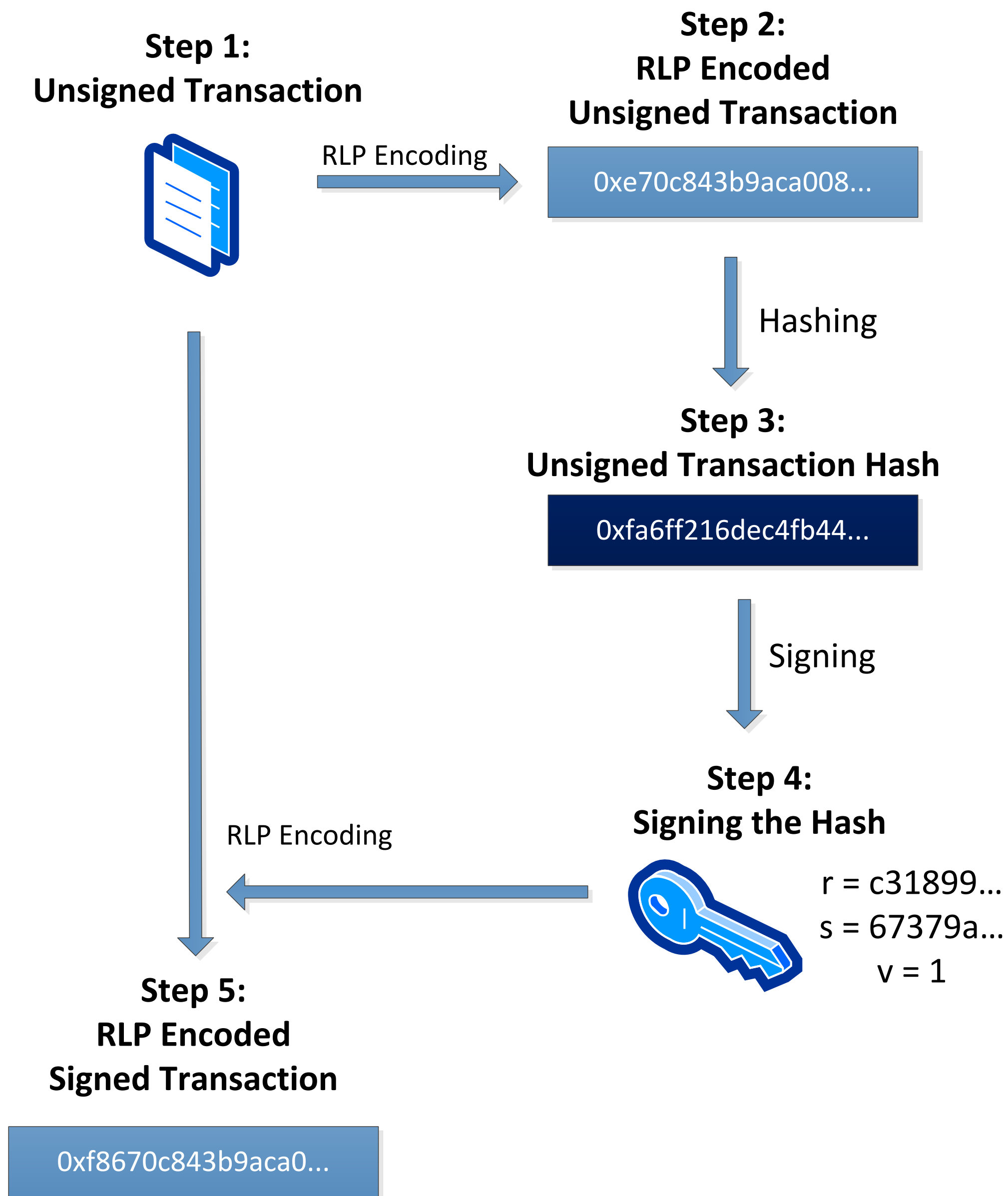

Figure 3.3: Different steps for generating a signed Ethereum transaction

Listing 3.4 Step 2.1: Organizing the transaction data in an ordered list

```
["0c", "3b9aca00", "c350", "98765432109876543210987654321098765432l0", "1388",
 "c0de", "03", "", "" ]
```

This list now gets RLP encoded, resulting in the string shown in Listing 3.5:

Listing 3.5 Step 2.2: RLP encoding of an unsigned transaction

```
RLP(["0c", "3b9aca00", "c350", "9876543210987654321098765432109876543210", "1388",
     "c0de", "03", "", "" ])

=
"0xe70c843b9aca0082c35094987654321098765432109876543210987654321082138882c0de038080"
```

Now the whole transaction structure is serialized without the loss of any information.

Step 3: Hashing the RLP Encoded Unsigned Transaction The string shown in Listing 3.5 now has to be hashed with the Keccak-256 hash function. The input to this hash function can be arbitrarily long, the output is always 32 bytes long, representable as a 64 digit hexadecimal string. The hash function output for the previous RLP encoding is shown in Listing 3.6.

Listing 3.6 Step 3: Keccak-256 hash of the RLP encoded unsigned transaction

```
Keccak-256(
"e70c843b9aca0082c35094987654321098765432109876543210987654321082138882c0de038080")

= "fa6ff216dec4fb44bc1a6381600a910bea00f3f52a68e4e0ced0b8123be2679c"
```

We have now obtained the hash for this specific unsigned transaction. When performing the hashing algorithm multiple times on the same data, the result will always be the same (in contrast to the ECDSA algorithm).

Step 4: Signing the Obtained Hash with ECDSA At this point, all the information for creating a signed transaction is available. The 32 byte long Keccak-256 hash is given to the ECDSA function along with the private key which also consists of 32 bytes. The result of the signature calculation are values for the parameters v, r and s of the unsigned transaction. This is shown in Listing 3.7.

Listing 3.7 Step 4: Signing the Keccak-256 hash with ECDSA

```
ECDSA_sign("fa6ff216dec4fb44bc1a6381600a910bea00f3f52a68e4e0ced0b8123be2679c",
PRIVATE_KEY)

=
{
  "r" : "c3189977506593aacc96149481d6cdd33737649bc4aa73171054f36253d58298",
  "s" : "67379abbd178970890fe17b36ac71a53873fe6dfb278fe4e5f4f518761e8f7aa",
  "v" : "1"
}
```

The signature as well as the recovery bit are now successfully calculated from the unsigned transaction. It can now be considered signed. Before it is issued, all the data has to be serialized once again.

Step 5: RLP Encoding the Signed Transaction One last step before serializing the signed transaction is calculating the correct v parameter. According to the previously presented formula, the v value for recovery bit = 1 and Chain ID = 3 (Ropsten test net) is 42, or as a hexadecimal number 0x2a. A JSON structure for the signed transaction ready to be RLP encoded one last time can be seen in Listing 3.8:

Listing 3.8 Step 5.1: Signed transaction ready to be RLP encoded

```
{
  "nonce" : "0x0c",
  "gasPrice" : "0x3b9aca00",
  "gasLimit" : "0xc350",
  "to" : "0x9876543210987654321098765432109876543210",
  "value" : "0x1388",
  "data" : "0xc0de",
  "v" : "0x2a",
  "r" : "0xc3189977506593aacc96149481d6cdd33737649bc4aa73171054f36253d58298",
  "s" : "0x67379abbd178970890fe17b36ac71a53873fe6dfb278fe4e5f4f518761e8f7aa"
}
```

After the final RLP encoding was performed on the signed transaction data, the serialized string can be seen in Listing 3.9.

Listing 3.9 Step 5.2: RLP encoded signed transaction

```
RLP(["0c", "3b9aca00", "c350", "9876543210987654321098765432109876543210", "1388",
     "c0de", "2a",
     "c3189977506593aacc96149481d6cdd33737649bc4aa73171054f36253d58298",
     "67379abbd178970890fe17b36ac71a53873fe6dfb278fe4e5f4f518761e8f7aa"])

= "0xf8670c843b9aca0082c350949876543210987654321098765432109876543210821388820de2a
   a0c3189977506593aacc96149481d6cdd33737649bc4aa73171054f36253d58298a067379abbd178
   970890fe17b36ac71a53873fe6dfb278fe4e5f4f518761e8f7aa"
```

This string can now be issued as a valid and signed Ethereum transaction. Therefore it has to be sent to an Ethereum node's JSON-RPC. Once a node picks up the transaction and acknowledges its validity, it is broadcasted to other nodes. At this point in time, the transaction gets assigned an ID. Eventually, a mining node picks up the transaction and puts it into a block.

Optional Step 6: Calculating the transaction ID Once a transaction is received by the first node, an ID for this transaction is calculated and usually also sent back inside the JSON response object. In order to check the correctness of the received ID or if out of some reason no ID is received at all, the transaction issuer can calculate the ID itself. Therefore, the RLP encoded string of the signed transaction has to be hashed once again with the Keccak-256 function. The result for the exemplary transaction is given in Listing 3.10.

Listing 3.10 Optional Step 6: Calculate the transaction ID

```
Keccak-256(
"f8670c843b9aca0082c350949876543210987654321098765432109876543210821388820de2a
a0c3189977506593aacc96149481d6cdd33737649bc4aa73171054f36253d58298a067379abbd17
8970890fe17b36ac71a53873fe6dfb278fe4e5f4f518761e8f7aa")

= "09d30c7553af79cab6fddd0b107aa16d41ffefd3f9ae9907b32d9cc466dac67f"
```

The transaction creation is now completed and can be checked e.g. with a blockchain explorer. The necessary reference for the transaction is only its ID[1].

[1] The exemplary transaction shown here was indeed built and sent to the Ropsten test net Ethereum blockchain. It can be checked e.g. with the block explorer etherscan.io under the link https://ropsten.etherscan.io/tx/0x09d30c7553af79cab6fddd0b107aa16d41ffefd3f9ae9907b32d9cc466dac67f

3.4 Different System Architecture Approaches

Now that it is clear which steps are required to build a valid transaction, the best fitting system architecture approach has to be found. The MCU does not have the required storage (see Section 3.3.1) and bandwidth capacity to act as a full Ethereum node. In fact, this is also not needed for the use case of issuing transactions. Ethereum nodes offer a lot functionality, they have to communicate with each other, maintain a blockchain copy and perform further tasks. This causes a huge computational power overhead and demands for a more complex hardware. That is why a full Ethereum node has to run somewhere, and the MCU has to be able to call the node's JSON-RPC methods. For distributing the transaction building process between the MCU and the Ethereum node, two different approaches with different levels of security will be presented.

3.4.1 Sign on Ethereum Client

The first approach requires an Ethereum node with a JSON-RPC running in the same network that the MCU's WiFi connection is set up to. The Ethereum client holds the private key in this scenario and RPC calls requiring the use of the private key need to first unlock it with a password. The MCU could then issue RPC calls for various functions of blockchain interaction. Building a valid and signed transaction would then work like this:

The MCU collects and processes the data that it wants to store on the blockchain. This means that the MCU builds an unsigned transaction analog to the first step in the previous section. If this is done, the MCU makes an HTTP request to the Ethereum client's JSON-RPC interface. The adequate RPC method for such unsigned message transaction is "eth_sendTransaction". The parameters necessary for this RPC call have to be sent in a JSON object. They comprise [Git18d]:

- "from" - The 20 bytes long address of the sender.
- "to" - The 20 bytes long address of the receiver, or, if left empty, the transaction is interpreted as a contract creation.
- "gas" - This is the maximum amount of gas that the transaction may cost. This parameter can be left empty, resulting in a default gas value of 90000.
- "gasPrice" - Price for each unit of used gas in wei.
- "value" - The amount of ETH sent to the receiving address. If left empty, no ETH is sent (e.g. for calling smart contract functions).
- "data" - Arbitrary data of the transaction that also gets stored in the blockchain. Can alternatively be used to call smart contract functions.
- "nonce" - This gives an order to transactions. If a nonce is used in two transactions and the first one was not yet put in a block, the older transaction can be overwritten.

After the “eth_sendTransaction” RPC call with appropriate parameters, the Ethereum client executes the remaining steps of the transaction building protocol. It finally signs the transaction and broadcasts it to its peer nodes. After a short while (also depending on the gas price), the transaction gets executed and becomes a part of the blockchain history. A schematic for this system architecture is shown in Figure 3.4.

Though this system architecture is fairly easy to understand and implement, it has some major downsides, especially in terms of security. The MCU is used here mainly to collect and process data coming from a production machine. After this is done, the data is sent via HTTP to the Ethereum client running on a general purpose computer or a very powerful single-board computer (e.g. a Raspberry Pi). The computer receives the data, performs the cryptographic operations and then issues the transaction to the network. With this setup, the same problems already described in Section 2.3.2 arise. The measured data could potentially be manipulated on its way from the MCU to the Ethereum node. Additionally, the computer that runs the Ethereum client could be attacked or already be infected with malware. Therefore, a secure transmission from a production machine into the blockchain cannot be guaranteed. Moreover, the MCU would have to somehow unlock the Ethereum private key on the computer. If the password for this is sent via HTTP, it might also get exposed.

A setup like this would work, but it can definitely not be considered secure.

3.4.2 Sign on Microcontroller

A more sophisticated but also more promising approach is this one: the whole data processing and signature generation is performed on the MCU. In order to send the transaction, there is still the need for an Ethereum client. But in contrast to the first approach, this client does not have to run in the same network. How exactly this works is explained in the following:

After obtaining the desired data from a production machine, the MCU creates an unsigned transaction, the same way as in the previous approach. Instead of calling the RPC of an Ethereum client in the same network, the MCU performs the signature generation itself. It creates one serialized transaction containing a valid signature. Therefore, the private key only needs to be stored on the MCU. The signed transaction string can then once again be sent via HTTP. The Ethereum client receiving the transaction does now not have to run in the same network. In fact, any node offering the JSON-RPC interface to the public can be used. A schematic for this approach can be seen in Figure 3.5.

Once the transaction left the MCU, it is not possible to alter any data without the signature becoming invalid, which would result in the rejection of the transaction by the Ethereum network. This approach prevents production data getting manipulated on the way from a production machine into the blockchain. It could still happen that the transaction gets lost on its way from the MCU to the client, or that the client is corrupt. The data would then not be stored, but still there would never be manipulated data stored.

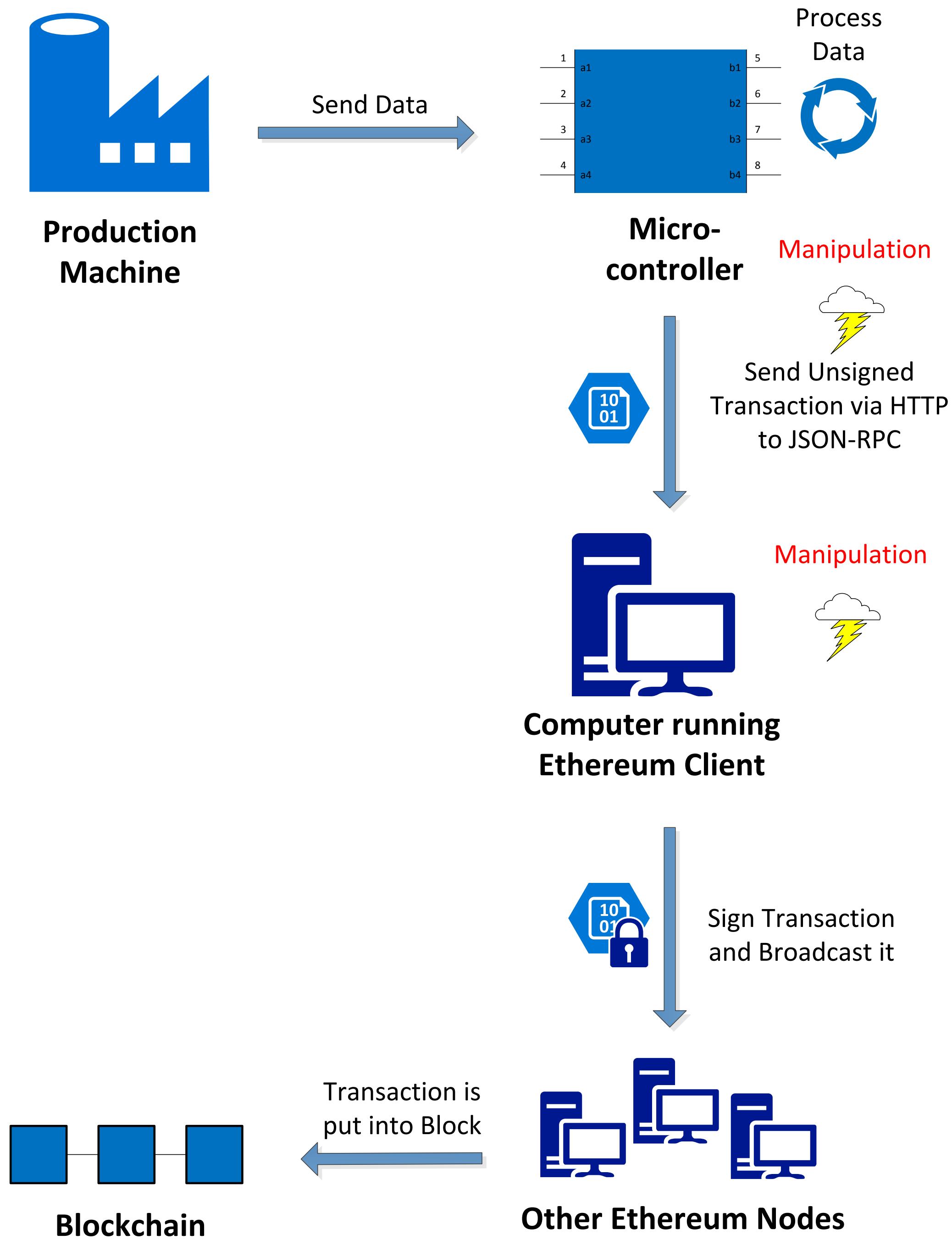

Figure 3.4: System architecture with Ethereum node signing a transaction

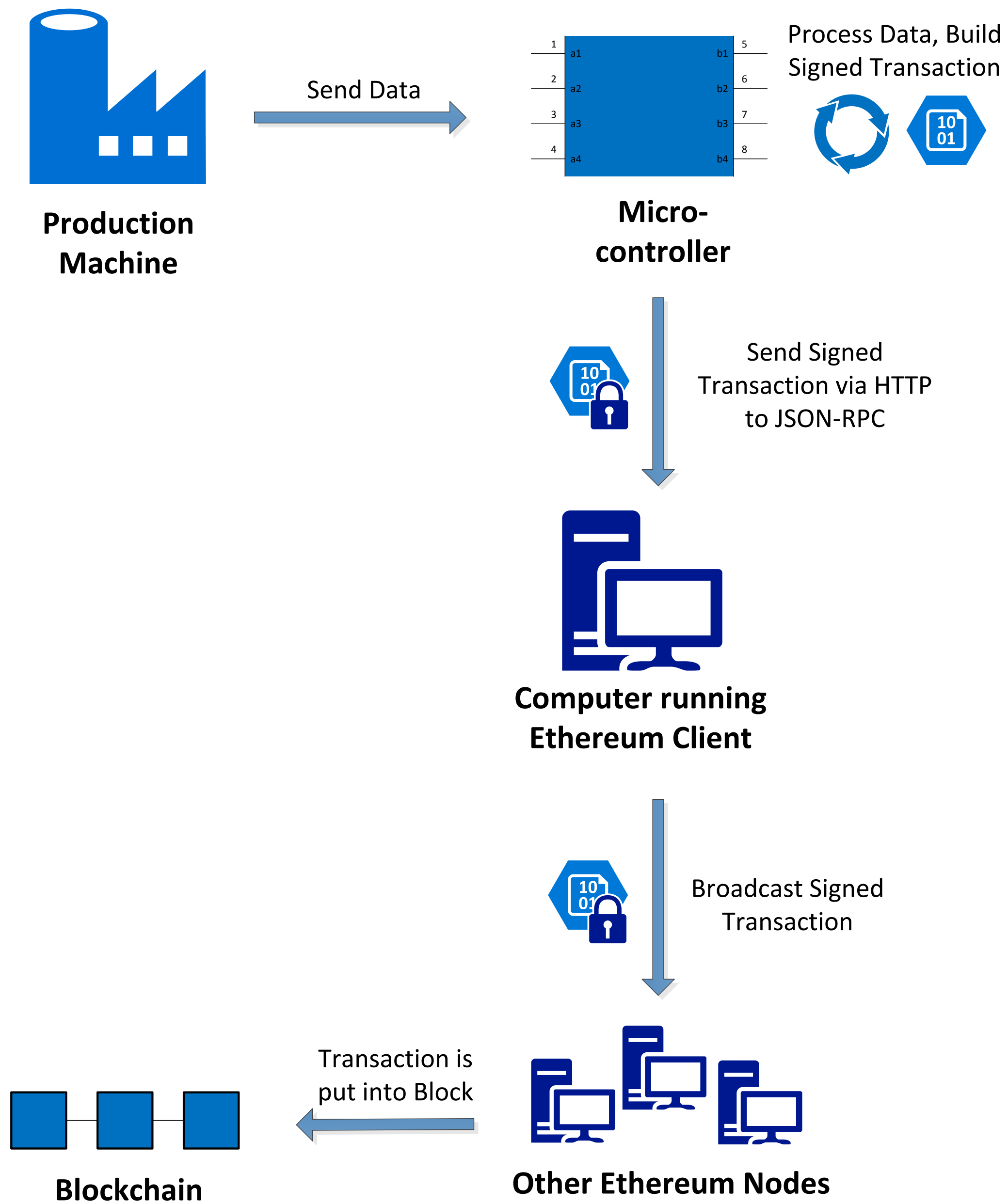

Figure 3.5: System architecture with microcontroller signing a transaction

Of course, another JSON-RPC call has to be made for the described approach. Apart from the "eth_sendTransaction" function, there is a second one for sending transactions. It is called "eth_sendRawTransaction" and takes only one parameter, which is the signed transaction RLP encoding. As a result, the RPC call returns the 32 byte long Keccak-256 hash of the transaction string inside a JSON response object. This ID can also be calculated by the MCU itself, if necessary.

4 Prototypical Implementation

This section shows how the prototypical implementation of the concept described in Section 3.4.2 can be realized. It demonstrates that the previously presented concept indeed works and that it can be adapted for different upcoming use cases.

4.1 General Microcontroller Program Information

Each Arduino C/C++ program running on the NodeMCU has to have at least two basic functions. The first one is the function “void setup()”. This function is called whenever the MCU is supplied with power. When the setup routine is finished, a second function is called automatically, which is the “void loop()” function. As the name already gives away, this is the MCU’s main program execution loop. As long as the power supply is provided, this loop runs and executes code. Once it has reached its end, the loop function is started again automatically.

The prototype’s code consists exclusively of .ino files (which is the Arduino C/C++ format), and regular C/C++ files (.c/.cpp) along with their respective header files (.h). After successful code compilation, a binary file is created and uploaded on the device. This binary file contains all used libraries, as the MCU later runs autonomously without a connection to a programming device.

When the MCU is operating while connected to a computer via a USB interface, a serial monitor can be used. Outputting data to the serial monitor resembles the “printf” function of C. With the serial output, human-readable data can be generated on the connected computer. This can be used to output measured or calculated values and facilitate debugging.

With the “delay()” function, the code execution can be halted for a specified period of time.

4.2 Concrete Implementation

The implementation of the prototype consists of several components, each of which complete different tasks. This encapsulation with well-defined interfaces in the form of functions makes it easy to change or update specific parts of the code. The different components along with their functionalities are explained in the following. An illustration of the single components and their interactions in form of a Unified Modeling Language (UML) class diagram can be seen in Figure 4.1. The attributes of the components as well as parameters and return values are omitted in the graphic to ensure a better readability. As the cryptographic component is a library included and used in the main component, it is also not part of the diagram.

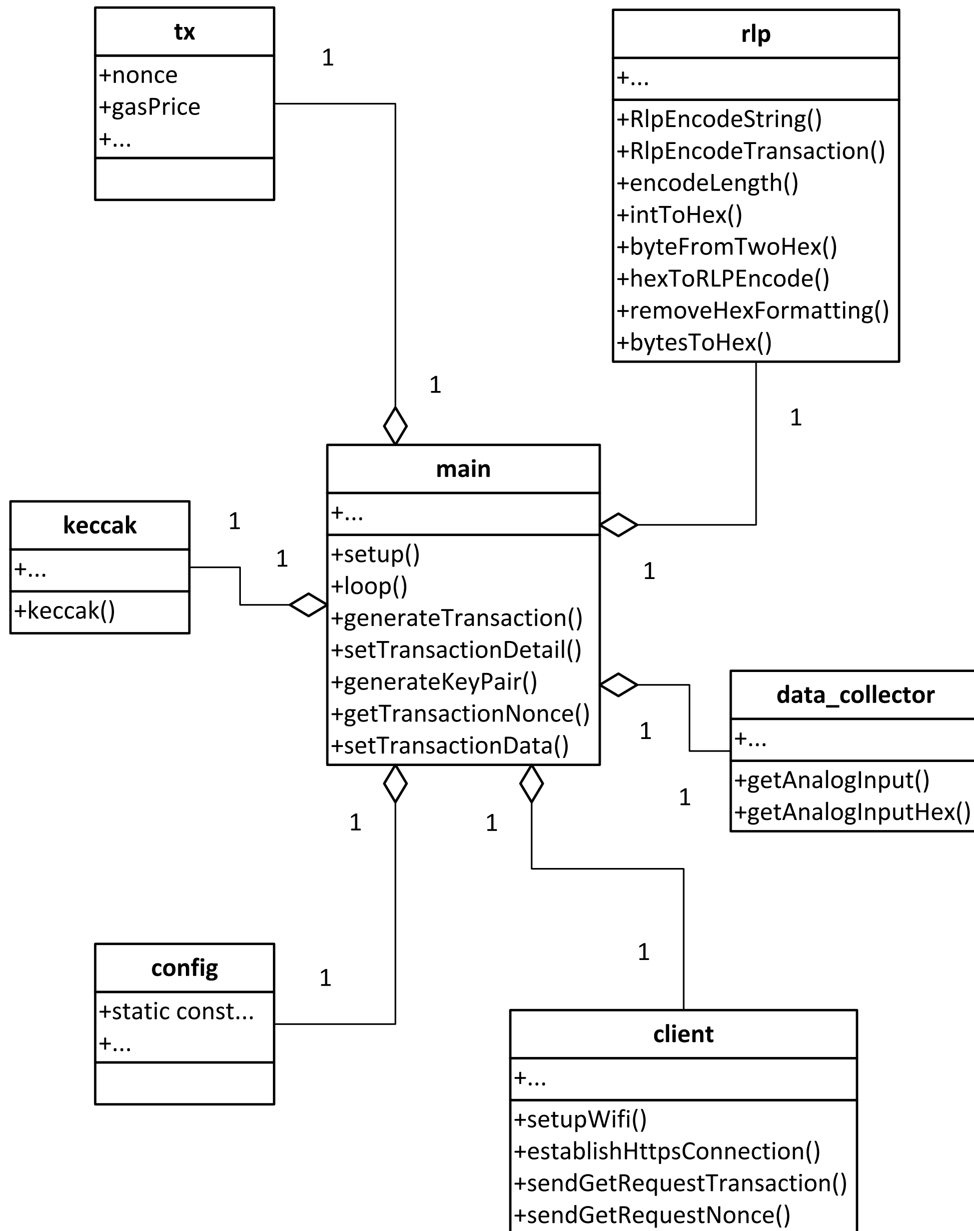

Figure 4.1: UML diagram of the different implementation components

Hashing Component

This portable C++ library implements the Keccak-256 hashing algorithm compliant with the Ethereum protocol [2].

It is used to obtain the Keccak-256 hash of the unsigned transaction, which is then signed afterwards. The library allows both hashing of strings and hashing of memory blocks in the form of bytes. In order to generate the same hash as an Ethereum node, the data must be input formatted as bytes (i.e. each two hexadecimal values have to be interpreted as one byte). Only then a valid hash is generated.

Cryptographic Component

In order to perform the necessary cryptographic operations for building a valid Ethereum transaction, the C library "uECC" is used [3]. It implements various cryptographic functions like asymmetrical key generation or signature verification with different elliptic curves.

The most important function for the prototype is the signature generation with the secp-256k1 curve. The function gets as an input the byte formatted private key of the sending Ethereum account and the byte formatted Keccak-256 hash of an unsigned transaction. As an output, the function returns the 64 byte long signature. This signature can then be split into Ethereum's r and s value, after they are converted back to hexadecimal numbers again.

One important Ethereum protocol change from March 2016 has to be taken into account when generating a signature. The solution space for valid s values of the signature was halved back then [Git16]. From this point in time on, an Ethereum transaction has to have an s value equal to or smaller than $\frac{secp256k1n_{max}}{2}$, with $\mathrm{secp256k1n_{max}}$ being the maximum possible value for s. This means that on average, every second generated signature will be considered invalid by the Ethereum network. When verifying the generated signatures with the uECC library, the result is of course always correct. To overcome this issue, the size of a resulting s signature component is checked directly after it was calculated. If the most significant byte is bigger than 128, i.e. $s > \frac{secp256k1n_{max}}{2}$, the signature creation function is called again. The probability to create a signature that will not be accepted by the Ethereum network with this procedure is the following:

$$\mathrm{P_n(X = no\ success)} = \left(\frac{1}{2}\right)^n$$

In this formula, n is the amount of times that the signature is generated. When generating the signature ten times, the probability that still not a single one was valid until then, is:

$$\mathrm{P_{10}(X = no\ success)} = \left(\frac{1}{2}\right)^{10} = \frac{1}{1024} < 0.1\%$$

[2]The C++ library for the Keccak-256 hashing function can be obtained under http://create.stephan-brumme.com/hash-library/

[3]The C library for ECDSA functions can be found under https://github.com/kmackay/micro-ecc

As the signature generation can be done quickly, it can be repeated multiple times, see Section 6.1.2.

An adaption that had to be made to the original library concerned the recovery parameter. Usually, an ECDSA algorithm only calculates the 64 bytes of the signature. Yet, Ethereum needs an additional bit in order to correctly calculate the sender's public key. The algorithm was modified so it also returns the required bit in an extra parameter. This way, the sending address can always be calculated correctly by the Ethereum network, without ever being transmitted.

Client Component

This component is used to get the signed transaction into the Ethereum blockchain. It offers a function to connect the MCU to a WiFi. Furthermore, it establishes an Hypertext Transfer Protocol Secure (HTTPS) to connection to the domain "api-ropsten.etherscan.io". Etherscan is an Ethereum block explorer which also offers tools for developers [Eth18d]. One of these tools is a geth/Parity proxy API. It enables a user to send HTTP(S) requests which then get transformed into JSON-RPC calls. The HTTP(S) requests can either be Get or Post requests and the parameters needed for the JSON-RPC call have to be provided as HTTP parameters. For the prototypical implementation, no Ethereum client running in the same network as the MCU and enabling a JSON-RPC is required. Instead, the signed messages are brought into the blockchain by the publicly available Etherscan API. That way, the MCU can always build and send transactions, as long as it is has a WiFi connection.

A further HTTPS call that was implemented retrieves the nonce of the Ethereum account whenever the MCU program is started. The nonce is useful for ordering transactions, e.g. when multiple transactions are issued before the first one is actually put into block. The MCU counts all its successfully sent transactions since the power on and adds them to the initial value that was retrieved via Etherscan's API. That way, a time consistency throughout all issued transactions is ensured.

RLP Component

The RLP component implements the rules for an Ethereum conform transaction object serialization. Its main function is "RlpEncodeTransaction", which gets a transaction object as an input and performs the RLP encoding steps. The component also offers helper functions for various data type conversions, e.g. generating one byte from two hexadecimal values or transforming hexadecimal numbers into integers.

Transaction Component

This component is a header file for a transaction. It is only used to capture a transaction's structure and the different parameters that a transaction has. The class itself does not offer any functions, as this is not needed. Instead, other components directly set certain values of the transaction object.

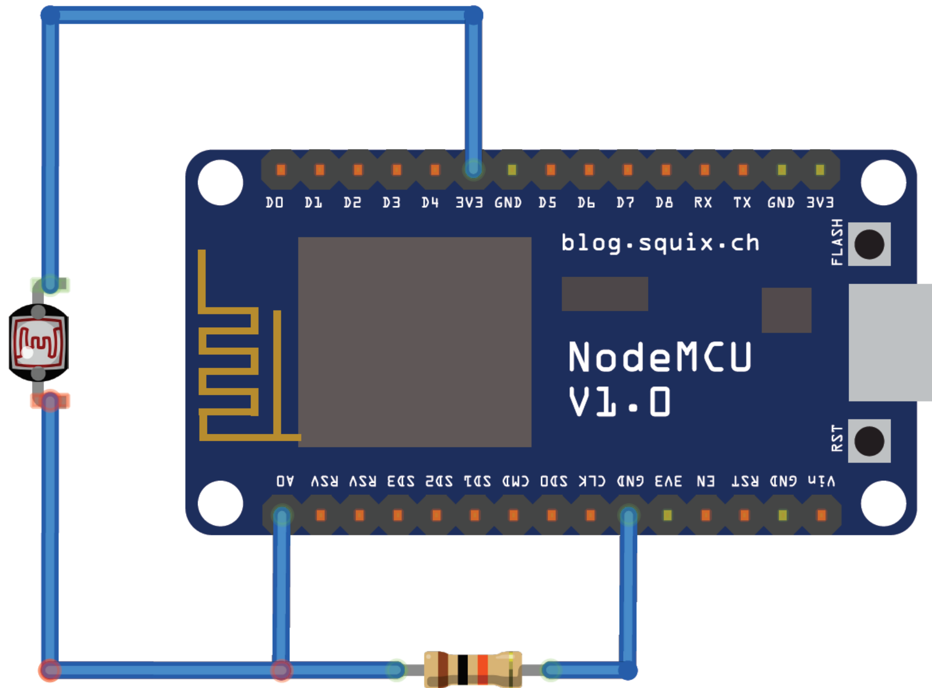

Figure 4.2: Schematic of the hardware setup

Data Collection Component

The goal of this prototype is the immutable storage of measured data in the blockchain. This component is responsible for obtaining data which has to be stored. In the prototypical implementation, this data is a brightness value of the MCU's surrounding. In order to measure a brightness value, external hardware components have to be connected to the MCU. The hardware setup is shown in Figure 4.2. The graphic was created with the tool Fritzing [4].

The part on the left side of the board is a photosensitive resistor. This means that the resistance value changes depending on how much light hits the resistor's surface. When strong light directly shines on it, the resistance is a few Ω. In complete darkness, the value goes up to several MΩ. The photosensitive resistor together with another resistor having a fixed value are used to build a voltage divider at the board's analog input pin. The analog input pin can measure a voltage in the range of 0 to 3.3V with a resolution of 10 bit, i.e. values from 0 to 1023.

We can now connect one of the constant 3.3V output pins over the photosensitive resistor to the analog input pin, and at the same time connect the analog input pin over a further resistor to the board's ground pin. With this setup, much light leads to a small resistor value and therefore little voltage drops at the photoresistor itself. Most of the voltage drops at the fixed value resistor. As the analog pin measures the voltage against ground, it consequently receives a high value when the brightness is high. The opposite happens for little light: the photoresistor has a high value, hence little voltage drops over the second resistor, leading to a low voltage at the analog input pin.

[4]The tool Fritzing can be obtained under `http://fritzing.org/home/`

The actual voltage that is measured at the analog input pin when using 10kΩ for the fixed resistor can be calculated with the following formula:

$$U_{analog} = \frac{U_{out} \cdot R_{fixed}}{R_{photo} + R_{fixed}} = \frac{3.3V \cdot 10000\Omega}{R_{photo} + 10000\Omega}$$

This measured data could e.g. result from a production machine or its environment. The data collection component can easily be modified or interchanged, measuring different data or directly communicating with a production machine.

The component offers two functions to retrieve the current brightness value, either as an integer value or as a hexadecimal representation prefixed with “0x”, which can then be further used for a transaction’s data.

Main Component

This file is the core element of the MCU program. It contains the required setup() and loop() function and coordinates all the other components. During the setup phase, it starts the serial output communication in order to send debugging information to a connected computer, if available.

It also provides the uECC library with a function for generating random numbers, which is utilized during signature generation. Values from multiple measurements of the analog input pin are taken to generate an appropriate random number. That way, the random values cannot be predetermined, as the analog input basically just measures noise. If necessary, the main component can also generate and output new key pairs with the uECC library.

In the last setup step, the WiFi connection is established over the client component and an initial HTTP request for obtaining the account’s current transaction nonce is performed.

Now that all prerequisites for the transaction generation and its transport are given, the loop function can start. It has a fixed delay time for each loop iteration which can be configured before flashing the MCU. The idea is the following: in each loop cycle, the MCU obtains a measurement value from the data collection component. This value is to be stored on the blockchain. A signed and valid transaction containing the measured value is built and sent to the Etherscan API via an HTTPS request. If the request was successful, i.e. the API returns a transaction ID and has broadcasted the transaction, the nonce for the next transaction is increased. Now that one loop cycle has finished, the MCU code execution is halted for a specified period of time before generating the next transaction.

A successful transaction is indicated by turning on a light-emitting diode (LED) which is built into the NodeMCU itself. The LED is turned off at the beginning of the transaction building process. When the transaction is sent and a positive response was received, the LED is turned on again until the generation of the next transaction. This gives a visual feedback, even if no computer monitoring the serial output is connected.

For measuring the duration of single algorithm steps, a flag can be commented in. If present, the “TIMING” flag performs timing measurements and puts the results out on the serial monitor.

The code execution starts as soon as the MCU is provided with power, either over a USB port or directly over a power socket. An UML flow chart for the order of MCU operations in given in Figure 4.3.

Instead of building transactions with a fixed cycle time, the creation could easily be triggered on specific events. For example, an external switch could trigger the transaction creation each time it is pressed. Or obtained sensor data could trigger creating a transaction, e.g. when a certain threshold brightness value is surpassed. Such external conditions would only require slight changes in the MCU's programming code.

config.h

The configuration header file contains constants and parameters for the configuration of the MCU. Some of them may be changed by the user, others should stay as they are. An overview on the different parameters is given in the following. Table 4.1 shows the configuration parameters that should not be changed in order for the program to work properly.

Table 4.1: Configuration parameters for the microcontroller code that should not be changed

Parameter Name	Type	Description
GET_NONCE	String	Has the value "eth_getTransactionCount", which is the Etherscan API GET function to retrieve an account's nonce.
SEND_TRANSACTION	String	Has the value "eth_sendRawTransaction", which is the Etherscan API GET function for sending a signed transaction.
HTTP_HOST	String	Has the value "api-ropsten.etherscan.io", the Etherscan domain where the HTTPS requests are sent to.
API_KEY	String	The API key which has to be sent with every HTTP request to the Etherscan server.
HTTPS_PORT	Integer	Port where the HTTP requests are sent to. Has the value 443 for the server's HTTPS port.
HTTPS_FINGERPRINT	String	SHA-1 hash of the server's TLS certificate. Can be checked with every request if necessary.

Table 4.2 lists parameters that may be changed by the user, e.g. for using another account or specifying the WiFi to which the MCU connects to.

In Table 4.3, the parameters defining static transaction data are given.

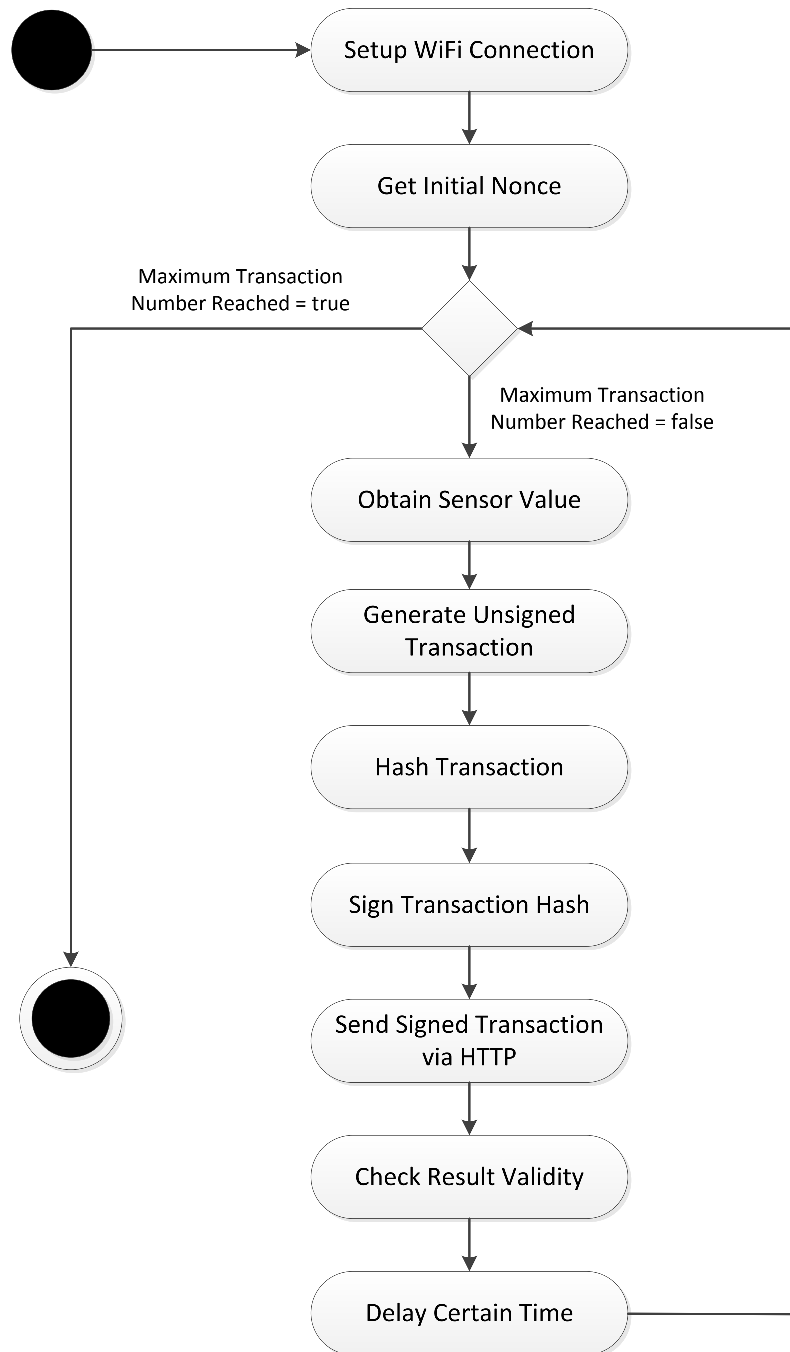

Figure 4.3: UML flow chart for code execution on the microcontroller

Table 4.2: Configuration parameters for the microcontroller code that may be adapted by a user

Parameter Name	Type	Description
PRIVATE_KEY	String	The Ethereum account's private key used to sign transactions.
PRIVATE_KEY _LENGTH	Integer	Length of the account's private key in hexadecimal digits, which is 64.
PUBLIC_KEY	String	The Ethereum account's public key. Can be used for signature verification.
PUBLIC_KEY _LENGTH	Integer	Length of the account's public key in hexadecimal digits, which is 128.
ACCOUNT_ADDRESS	String	40 hexadecimal digits long address of the account, prefixed with "0x".
WIFI	String	ID of the WiFi that the MCU connects to upon start up.
PASSWORD	String	Password of the WiFi network that the MCU connects to.
WAIT_BETWEEN _TRANSACTIONS_MS	Integer	Duration between end of one transaction and the start of the next one in milliseconds.
MAXIMUM_AMOUNT _TRANSACTIONS	Integer	Maximum number of transactions that the MCU should issue in one session.

Table 4.3: Transaction specific configuration parameters for the microcontroller code

Parameter Name	Type	Description
CHAIN_ID	String	Hex prefixed string for the used chain. Set to "0x03" for the Ropsten test net.
GAS_PRICE	String	Hex prefixed string for the price per used gas of the transaction in wei. Set to "0x3b9aca00" for 1 gwei.
GAS_LIMIT	String	Hex prefixed string for the maximum amount of gas that the transaction may consume. Set to "0x186a0" for 100000.
RECEIVING _ADDRESS	String	Hex prefixed address of the transaction receiver. Can either be an EOA or a smart contract.
TRANSACTION_DATA	String	Hex prefixed string of the transaction data. Can be filled with placeholders that get their values at execution time.
REC_ID_FALSE	String	Hex prefixed string for the recovery parameter of the current blockchain instance if the bit is 0. For Ropsten test net: "0x29"
REC_ID_TRUE	String	Hex prefixed string for the recovery parameter of the current blockchain instance if the bit is 1. For Ropsten test net: "0x2a"

5 General Blockchain Security Aspects

The blockchain technology as a whole is a relatively new concept and its complete potential as well as risks are not yet fully investigated. In this section, we take a closer look at possible attacks on blockchains and also design a metric to estimate how secure it is that a transaction will actually be stored immutably in a distributed ledger.

5.1 Possible Attacks on Blockchains

One of the main features that most blockchains share is being distributed. This also changes the way that attacks on them are performed. Attacking a blockchain immensely differs from attacking a single centralized server. One corrupt node in the P2P network is highly unlikely to do any harm.

There are two different main goals that an attacker wants to achieve [CKLR17]: an attacker wants to either shut down the blockchain P2P network, i.e. prevent successfully sending transactions and mining new blocks, or perform transactions that favor the attacker in some way.

Many of the attacks are described to theoretically and also practically work on the Bitcoin blockchain, as it is the oldest and best known blockchain implementation. Nonetheless, the same attack mechanisms work on various other blockchain implementations with similar underlying concepts.

Double-Spending Attack

The term double-spending attack refers to all kinds of different attacks that allow a malicious user to spend the same token multiple times [KAC12]. The basic setup is always the following: an attacker sends two transactions that only differ in the receiver's address shortly after another. The first one transfers tokens to the address of the victim. This could e.g. be an online shop which accepts payments done in a cryptocurrency. The second, slightly later issued transaction goes to an address in control of the attacker. Only one of the two transactions will then be put into a block. The attacker's goal is to trick the victim into thinking that it was the recipient of the transaction, while the transaction that actually gets included into a block is the one favoring the attacker.

Such an attack works if the receiver is not waiting for the transaction to be put into a block, or if the transaction is put into a block which is later abandoned due to a previously happened fork. A fork describes the event that multiple instances of a valid blockchain exist, i.e. two or more different blocks have the same height [Ant14]. Such an event can either occur on purpose or accidentally.

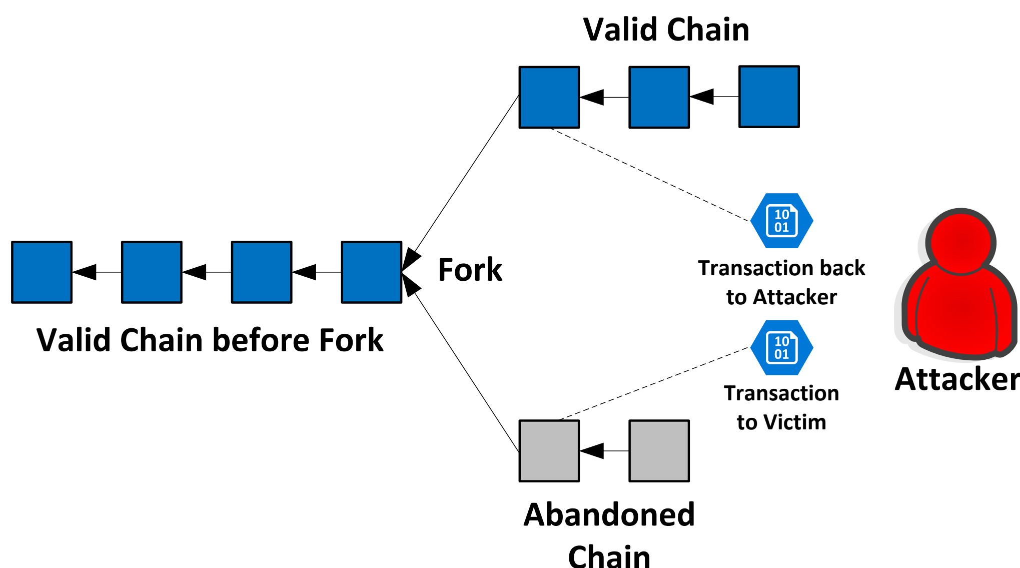

Figure 5.1: Schematic of a successful double-spending attack

Purposeful forks happen e.g. when the blockchain protocol gets updated. This can result in transactions following the old protocol no longer being accepted by nodes which implement the new protocol. On the other hand, nodes running the old protocol will still accept such transactions. That way, two different blockchain instances exist in parallel which are both valid according to their respective protocols.

At an accidental fork, two or more miners find a valid block at almost the same time. Multiple blocks contain a different set of transactions, but they are all valid blocks. Multiple chains are valid at this point and subsequent blocks get added to them. Over time, one chain will eventually grow faster than the other ones, resulting in the shorter chains to be abandoned.

If the transaction favoring the attacker is on the chain that will henceforth be considered the valid chain, and the transaction to the victim is in a block of the abandoned chain, the attack can be successful. A schematic for a successful double-spending attack with the help of a blockchain fork is given in Figure 5.1.

51% Attack

This potential scenario can occur when a group of miners holds more than 50% of the global mining power [KDF13]. Mining cryptocurrencies is often organized in pools. In such a pool, users solve the PoW puzzle together and split the reward proportional to each user's provided calculation power. That way, it is not unlikely for a single pool to contribute a large amount to the global mining power.

If a single group of miners would possess more than half of the global mining power, this group could decide what the blockchain's status is at any time. They could perform double-spending attacks, reject blocks or revert already confirmed transactions. The root cause for the possibility of such an attack lies in the consensus algorithm used in many blockchain implementations. As there is no central authority supervising the status of the blockchain, it is always the majority of hashing power deciding over it.

Routing Attack

Such an attack targets the route of IP packets in the Internet [AZV17]. An attacker tries to attract all the traffic that was destined to a certain IP prefix. This is possible due to the routing protocol BGP which is used to transport IP packets. Routers are offered a route along which they can send their packets, and the validity of the route does not get checked. If an attacker can trick a router into sending packets to a different receiver than intended, the attacker can either drop or reroute the packets.

In the Bitcoin blockchain for example, the P2P network operates as follows: after a network node is started, it first needs to find peers to connect to. This is achieved by issuing DNS requests to static host names which then return a list of active peers in the network. If this is not successful, there is also the possibility to communicate with nodes via hard coded IP addresses. If a connection is established, a node can query the addresses of peers that the other node is connected to [Bit18b].

The goal of an attacker is to create a partition of the P2P network. One set of nodes should be running on IP addresses hosted by IP prefixes under the attacker's control. The other set of nodes runs normally. If nodes have a communication crossing the boundary of the two sets, the attacker can just drop all the packets and thereby isolate one set from the other. Node traffic within the set that the attacker forged can be monitored.

The result of such an attack is that some nodes get different information about the blockchain's status than others. Depending on the size and constellation of the isolated nodes, such a routing attack can facilitate other attacks like the previously described double-spending attack. It can also be seen as a denial of service attack, because nodes are being disconnected from the majority of the network and computational power is wasted.

5.2 Degree of Trust Measurement

In the following, an approach to measure the degree of trust that data is durably stored in the blockchain will be developed. The considerations apply to public, decentralized blockchains using a PoW consensus algorithm.

The term "finality" of a transaction describes a state when it is mathematically highly unlikely that a transaction will be changed or abandoned. If a transaction was already successfully put into a block, measuring the degree of trust is rather simple. It suffices to count confirmations, i.e. the number of blocks that were appended to the blockchain after the block containing the observed

transaction. The more confirmations a block has, the more likely it is that it is not on a blockchain branch that will be abandoned. For each blockchain implementation, there can be defined a number of confirmations that make a transaction final under certain prerequisites. In the Bitcoin blockchain, which is the most prominent PoW blockchain implementation, the chances of an attacker reverting a transaction are as follows [Nak08]:

$$\mathrm{P(X = success)} = 1 - \sum_{k=0}^{z} \frac{\lambda^k \cdot e^{-\lambda}}{k!} \left(1 - (q/p)^{z-k}\right)$$

z is the amount of blocks to be reverted,

p the probability of an honest node finding the next block,

q the probability of an attacker finding the next block,

$\lambda = \frac{z \cdot q}{p}$.

This formula shows that if an attacker tried to create a longer, valid Bitcoin chain, the chances for success would be rather low, especially when the attacker has to catch up several blocks. When assuming that 11 blocks have to be reverted and the attacker possessed 20% of the global hash rate, a success occurs with a chance of less than 0.1% [Nak08].

In order to avoid getting wrong information from a corrupt or isolated node, it is recommended to check the confirmations for a specific block by querying different nodes.

As it is now clear how the chance for finality of transactions already contained in a block is measured, another aspect can be focused: can the degree of trust be estimated before a transaction is even part of a block?

5.2.1 Possible Parameters

Before an approach can be designed to measure the degree of trust for unconfirmed transactions, one has to identify parameters that enable an estimation.

Each of the following parameters is best accessible when running multiple blockchain nodes which can then be monitored. It is not required to have access to all existing nodes, but rather to a certain percentage of total nodes. The values for the whole network can then be extrapolated from the data that is known.

Number of Total Nodes

As we have seen that nodes can be attacked or isolated, a big P2P network creates more security. The more nodes participate, the less likely it is to do severe damage to the network. Therefore, the overall amount of active nodes plays a role in estimating the security.

Number of Nodes Aware of the Transaction

Of course, it does not help a transaction to be issued if the network only has a lot of participants. The nodes must also be aware that a transaction is pending. The more nodes know of the pending transaction, the more likely it is that a miner picks the transaction and puts it into a block. Therefore, if the pending transaction is spread over many nodes, it is less likely to get lost or be canceled.

Transaction Fee

Each miner can freely choose which transactions to put into the next block and which not. Though, a miner is usually willing to maximize the profit, i.e. the transactions offering high fees will be put into a block first. That is why a high fee increases the chance of a transaction being in the next block that is mined.

Relative Transaction Fee

Nevertheless, the amount of fee paid alone is not a good estimator. If all pending transactions offer a fee that is considered unusually high, it does not favor any of them. The fee has to be high relative to the fee of other competing transactions. One could think of a table containing all the unconfirmed transactions ranked by the fee that is paid in descending order. The higher the considered transaction's rank in such a table is, the more likely it is to succeed within a short time.

Total Amount of Incoming Transactions

The amount of overall transactions that are issued on a blockchain within a certain period of time is also relevant for the success of the observed transaction. Yet, the pure rate of transaction arrival does not give anything away about the overall performance.

Transaction Throughput of the Blockchain

Many blockchains have a certain average time between the finding of two blocks. Also, the amount of transactions that can be put into a block is limited. It can either be a fixed value (which makes sense if the size of transactions is always the same and cannot be enlarged by the sender) or dynamic, i.e. depending on the different sizes of all contained transactions. One way or the other, statistics can be generated estimating how many transactions are executed in a certain period of time. The higher the general transaction throughput is in relation to the overall incoming transactions, the higher the probability for a fast execution.

5.2.2 Designing a Metric

As already discussed, the number of nodes alone does not give away anything about how secure one can be that a transaction succeeds. The same goes for the number of nodes that are aware of the transaction. What is an important factor though, is the ratio of these two values. This can be expressed with:

$$c_1 = \frac{N_{aware}}{N_{total}} ,$$

with N_{total} being the amount of active P2P nodes and N_{aware} being the amount of nodes that have the targeted transaction in their unconfirmed transaction pool. The resulting value is in the interval [0,1]. The higher this value, i.e. the higher the ratio between nodes aware of the transaction and total nodes, the more likely the transaction will be executed.

A further component of a metric can consider the transaction fee. Let R be the rank of a transaction in a table of pending transactions as described in Section 5.2.1. The following estimator can then be calculated:

$$c_2 = \frac{1}{R}$$

This gives a value in the range [0,1], with a high rank, i.e. a low numerical value, resulting in a high c_2 value.

The last component results form the blockchain's throughput. Let t be the average time between the finding of two blocks in seconds and a the average amount of transactions in a block. The transaction throughput can then be calculated by $\frac{a}{t}$ [transactions/second]. Furthermore, let λ be the rate of newly arriving transactions to the network in transactions per second. A metric value can now be calculated by $\frac{a}{t \cdot \lambda}$. The result is in the interval [0, ∞[. A value over 1 means that more transactions are being executed than issued, which guarantees that each transaction will be part of the next block. As a value bigger than 1 should not have an effect, the result is:

$$c_3 = \begin{cases} \frac{a}{t \cdot \lambda} & \text{for } \frac{a}{t \cdot \lambda} < 1 \\ 1 & \text{for } \frac{a}{t \cdot \lambda} \geq 1 \end{cases}$$

Once again, a high value means that many of the pending transactions are executed, thus increasing the chance of a certain transaction to be executed.

Simply adding the three values c_1, c_2 and c_3 would not result in a good metric. There could be an extremely high throughput and an immense fee paid, but only one isolated node is aware of the transaction. In such a case, adding the single values would result in a relatively high overall value (2 out of a maximum of 3), but the transaction would probably never get executed. The same goes for extremely bad c_2 and c_3 values.

To overcome this issue, we additionally introduce positive weights w_i for each of the parameters. The sum of the parameters multiplied with their respective weights divided by the sum of weights gives us an overall estimator.

$$C = \frac{\sum_{i=1}^{3} c_i \cdot w_i}{\sum_{i=1}^{3} w_i} = \frac{c_1 \cdot w_1 + c_2 \cdot w_2 + c_3 \cdot w_3}{w_1 + w_2 + w_3} = \frac{\frac{N_{aware} \cdot w_1}{N_{total}} + \frac{w_2}{R} + c_3 \cdot w_3}{w_1 + w_2 + w_3}$$

with

$w_1, w_2, w_3 \in \mathbb{R} \geq 0 \mathrel{\widehat{=}}$ Positive weights for each parameter free to choose,

$N_{aware} \mathrel{\widehat{=}}$ Number of active nodes that have the transaction in their pending transactions pool,

$N_{total} \mathrel{\widehat{=}}$ Number of active nodes in the P2P network,

$R \mathrel{\widehat{=}}$ Rank of the transaction in the table of pending transactions, ordered descending by fee paid,

$c_3 = \frac{a}{t \cdot \lambda}$ or maximum 1,

$t \mathrel{\widehat{=}}$ Average time between the creation of two consecutive blocks,

$a \mathrel{\widehat{=}}$ Average amount of transactions in a block,

$\lambda \mathrel{\widehat{=}}$ Rate of newly arriving transactions to the network.

Note that C does not have a unit and will always be in the range [0,1]. A high value indicates a high probability for the transaction to be executed, a low value stands for a low probability.

The weights can be chosen freely, and it is recommended to perform multiple calculations with many different weights in order to get a good estimate.

With the formula that was built up here, the chances for transaction finality can be estimated. Different aspects influencing the chance of a transaction being executed in PoW blockchains are taken into account. By weighting and combining the single estimators, one overall estimator was found.

6 Evaluation

This chapter evaluates the prototypical implementation and thereby also the designed concept. Additionally, the requirements that were imposed on the design are checked against the prototype.

6.1 Evaluation of the Implementation

In order to evaluate the practicality of the prototype and the idea of immutably stored data, it is not enough to just successfully send transactions to the blockchain. While it is correct that the data inside a transaction becomes a part of the immutable ledger, it does not make much sense for production data to be distributed in an unstructured manner over multiple single transactions. What we want to achieve is storing the data at one easily accessible place in the blockchain. This can be done with the help of an Ethereum smart contract.

6.1.1 Smart Contract

For the evaluation of the prototypical implementation, a smart contract was deployed into Ethereum's Ropsten test net. The smart contract is written in the programming language Solidity and the code can be seen in Listing 6.1.

The name of the shown smart contract is "NodeMCU_Endpoint" (line 5). It defines *dataBlocks* (line 8) consisting of an Ethereum address and an unsigned 16 bit integer. A dynamic array of dataBlocks named *valueArray* is declared (line 17) which allows for appending new dataBlocks after the contract's initial deployment. Additionally, the contract stores the address of its *creator* (line 14). This is done in the constructor which has the same name as the contract (line 29). When the contract is deployed, the constructor is called automatically once and can never be called again for this contract. In the constructor, the deploying address gets stored within a variable of the contract. Together with the modifier *onlyOwner* (line 20), functions can be restricted to be called only by the contract creator.

The contract offers two functions and one event. The first public function is called *Send_Data* (line 35) and can only be called by the contract creator's address. As a parameter, an unsigned 16 bit integer has to be given. If the size of the parameter is smaller than 1024, the value is stored in a newly generated dataBlock together with the sender's address. Then, the dataBlock gets appended to the array of publicly visible dataBlocks. The idea behind this function is that the MCU can send its measured values to the blockchain and they all get stored in an array chronologically. The restriction of sensor values to the range $0 \leq \text{value} \leq 1024$ is done to filter unexpected sensor data out. The restriction to only one valid sender address creates additional security. This is done by

Listing 6.1 Smart contract code for storage of sensor data

```
pragma solidity ^0.4.19;

// This contract receives values from 0-1024 from the contract creator and stores them
// in a dynamic array together with the senders address.
contract NodeMCU_Endpoint {

    // Struct containing sender address and sensor value.
    struct dataBlock {
        address sender;
        uint16 value;
    }

    // Address of the contract creator. Only the creator is allowed to send values.
    address private creator;

    // Dynamic array of dataBlocks
    dataBlock[] public valueArray;

    // Modifier allowing only the contract creator to call a function.
    modifier onlyOwner() {
    require(msg.sender == creator);
    _;
    }

    // Create event log for each sent value.
    event OnSendData(address sender, uint16 sentValue);

    // Constructor defining the contract creator.
    function NodeMCU_Endpoint() public {
        creator = msg.sender;
    }

    // Allows the contract creator to send a sensor value in the range 0-1024.
    // The value gets stored in a data block together with contract creator's address.
    function Send_Data(uint16 amount) public onlyOwner {
        if (amount > 1024) return;
        valueArray.push(dataBlock({
            sender: msg.sender,
            value: amount
            }));
        OnSendData(msg.sender, amount);
    }

    // Returns the latest sensor value that was stored.
    function Get_Last_Value() public view returns (uint16) {
        if (valueArray.length == 0) return;
        return valueArray[valueArray.length - 1].value;
    }
}
```

giving the Send_Data function the previously defined onlyOwner modifier. In order to manipulate data in the smart contract, an attacker has to know the private key of the account stored on the MCU and issue transactions in its name.

When a dataBlock was successfully appended to the array, a previously defined event (line 26) is triggered. This event is appended to a log containing all the events of the smart contract. It facilitates debugging and gives a quick overview on all Send_Data function calls and their corresponding transactions together with age and block height.

The second function of the smart contract is called *Get_Last_Value* (line 45) and can be called by anyone. It returns the last value that was added to the array, if existent.

During its deployment, the smart contract is assigned an address which is calculated once again via RLP encoding, hashing and cutting the leftmost bytes away. The address of the presented contract is "0x5e2a561439602495e2DDdF0Ed59f247A6CE01E23" [5]. In the MCU's config, the RECEIVING_ADDRESS parameter has to be set to this address. In order to send valid transactions to the contract, it is not necessary to send any funds along with the transaction, which is why the transaction's "value" field can be left empty. What is important for a function call though is the transaction's data field. It has to contain an encoding of the contract's desired function as well as all the required parameter values. The encoding works as follows [Sol18]:

Function Selector Encoding

The beginning of each transaction data field for calling smart contract functions has to consist of a 4 byte long encoding of the desired function. The 4 bytes are obtained by taking the four leftmost bytes of the Keccak-256 hash of the function's signature. The signature contains the name of the function as well as the types of each of its arguments separated with commas and written inside parentheses. As no function with the same name and same list of arguments can appear twice in the same smart contract, this encoding gives an unambiguous identifier. The IDs for the functions of the smart contract given in Listing 6.1 are:

Encoding(Send_Data(uint16)) = "0dbd5e81"

Encoding(Get_Last_Value()) = "29bee342"

Encoding(valueArray(uint256)) = "858090d5"

Though valueArray is not a function, Soliditiy automatically generates getter functions for all public variables. The unsigned 256 bit integer is used here to index the array.

[5] The smart contract can be accessed via Etherscan under the following link: `https://ropsten.etherscan.io/address/0x5e2a561439602495e2dddf0ed59f247a6ce01e23`

```
Connecting to WiFi:
espaccesspoint
.......................
WiFi connected.
IP address is:
192.168.43.97

Connecting to the host address:api-ropsten.etherscan.io
Request sent.
Headers received.
Valid response received.
Response object is:
====================
{"jsonrpc":"2.0","result":"0x63","id":1}
====================
The connection is closed now.
```

Figure 6.1: Serial output of the setup phase

Encoding of the Arguments

Not only the function which is to be called must be encoded, but also its parameters. All elementary types like integers, boolean values or addresses are encoded as 32 byte long hexadecimal values, padded with zeros on the left. More complex types are not needed in the exemplary smart contract and are therefore omitted here.

The integer value 123 for example would first be transformed in a hexadecimal value, resulting in 0x7b. The padded argument that is appended to the function selector would then be:

“00000000000000000000000000000000000000000000000000000000000000007b”.

Serial Output

The concatenation of the function selector and the encoding of all arguments gives the required data field of the transaction. The MCU config therefore has to be modified so that TRANSACTION_DATA consists of the function selector encoding followed by a dummy which is replaced with the current sensor value at runtime.

The prototype is now able to not only issue basic transactions, but also transactions that enable an interaction with smart contracts. This allows for the storage and possibly further processing of obtained measurement data.

The serial monitor output that is generated by the MCU after its power on is shown in Figure 6.1 and Figure 6.2.

As a first step seen in Figure 6.1, a WiFi connection is established. If this is successful, the MCU sends an HTTPS request to retrieve the account’s current nonce. The JSON object shows that the current nonce is 0x63. Everything that is executed here is part of the setup() function.

```
 START OF TRANSACTION BUILDING

Sensor value for transaction is:
0x0369

RLP of unsigned transaction:
f84863843b9aca00830186a0945e2a561439602495e2dddf0ed59f247a6ce01e2380a40dbd5e810000000000000000000000000000 ...

Result of Keccak execution as hex values is:
115c0c877f7a475602a8ba7cfeca873fc2a23cdc7665347af612a709e8dc2e04

Now signing the transaction starts.
Signature was INVALID! - repeat
Signature was VALID!
Signing was SUCCESSFUL
Recovery bit is:
0

r of ECDSA is:
0x14984ea58dfef6f9bed06e7554064bd3c281d6ed8152bb5faa34235f195cf860

s of ECDSA is:
0x602b18561c5471ff4216d7b397f0b4bb89779a827e2bf69ed044db776a6f0efc

-------------------------------------------------------
Final transaction is:
0xf88863843b9aca00830186a0945e2a561439602495e2dddf0ed59f247a6ce01e2380a40dbd5e810000000000000000000000000000 ...

Connecting to the host address:api-ropsten.etherscan.io
Request sent.
Headers received.
Valid response received.
Response object is:
====================
{"jsonrpc":"2.0","result":"0xaa9fab49f865d77b2f1b96e8322b522f81f4643fc3432336b78bd08cc79fb5ee","id":1}
====================
The connection is closed now.
```

Figure 6.2: Serial output of one loop iteration

Once the setup phase ends, the MCU starts to cyclically build and issue transactions. Figure 6.2 begins with outputting the obtained sensor value and subsequently creates an unsigned transaction around it. The serial monitor shows the RLP encoding of the transaction and the Keccak-256 hash of said encoding. This is followed by the signature generation algorithm, which has to be repeated once on this case. As a next step, the signature values that were created are given as an output, namely the recovery bit, r and s.

With all this information, the signed transaction can be built and once again the RLP encoding of the now signed transaction can be done. The string of the final transaction is printed to the serial output.

The last step is sending the transaction. We can see that the MCU is establishing an HTTPS connection to the Etherscan API and sends an eth_sendRawTransaction request. If the response is valid, the received JSON object is output on the serial monitor. The result string corresponds to the transaction ID.

Now, the waiting period between two transactions begins, and when it is over, the next transaction will be constructed and sent the same way.

6.1.2 Code Execution Time Analysis

The time that the MCU requires in order to successfully issue a transaction is a crucial factor in the evaluation of the design. If building a transaction takes too long and cannot be further accelerated, the prototypical implementation and with it the whole concept fails. That is why an analysis of the execution time for the program loop's main functions was performed. The time that different steps of the code execution take was measured for ten consecutive transactions. For each transaction, a new sensor value was obtained and sent to the smart contract introduced in Section 6.1.1. Table 6.1 shows the results.

Table 6.1: Duration of the single steps in the transaction creation process in milliseconds

Step						
Acquire Data and Build Unsigned Transaction RLP	**Hash**	**Sign**	**Build Signed Transaction RLP**	**Establish Connection to Server**	**Positive Server Response**	**Total**
9	10	731 (1 Trial)	41	453	391	1635
9	10	2064 (3 Trials)	41	465	323	2911
9	9	707 (1 Trial)	41	420	271	1457
8	10	716 (1 Trial)	41	438	359	1572
9	9	620 (1 Trial)	40	448	434	1560
8	10	3460 (5 Trials)	40	620	400	4538
9	10	3766 (5 Trials)	41	490	527	4843
8	10	617 (1 Trial)	40	462	296	1433
9	9	617 (1 Trial)	41	444	442	1562
8	10	620 (1 Trial)	41	477	518	1674
8.6	**9.7**	**1391.8**	**40.7**	**471.7**	**396.1**	**2316.6**

We see that obtaining a sensor value, putting it in an unsigned transaction and hashing said transaction takes less than 20 milliseconds. The majority of time needed comes from the signature generation. Each time that the ECDSA algorithm is executed takes 600-750 ms. As described in Section 4.2, the hashing algorithm has to be repeated with a chance of 50%. This means that

on average, the algorithm has to be performed twice. This takes roughly 1.4 seconds then. If the algorithm should not produce a valid signature within ten trials, the transaction creation process gets aborted and a new transaction will be generated. However, this happens in less than 1 out of 1000 transactions, see Section 4.2. If necessary, the limit of 10 transaction can also be increased. The idea behind it is to limit the overall maximum execution time.

Constructing the signed transaction RLP takes longer than building the unsigned one, as all steps have to be repeated, and additional ones have to be performed.

A timing component that can barely be influenced are the last two steps. Both establishing the HTTPS connection to the server and receiving a positive response for a sent request depend heavily on factors outside of the prototype's reach. The server's performance varies over time and also with the amount of API users. Furthermore, the performance of the network that the MCU is connected to influences the transmission times. Yet, after an average waiting time of 0.9 seconds, a connection to the server was established and a valid response was received.

The last column, the overall time for issuing one transaction, is measured independently from the single steps and may therefore not always be the sum of the six individual steps. It is the time from the beginning until the end of one transaction sending loop iteration. The measured average value for issuing a successful transaction was 2.3 seconds. The big deviation of some samples from the mean arise mostly from the amount of signature generation repetitions.

6.2 Retrospect of Requirements

For a profound evaluation of both the chosen design and the resulting prototype, a comparison with the requirements defined beforehand has to be made. Only if the important requirements are met by the implementation, we have constructed a viable prototype.

Each requirement stated in Section 3.1 will briefly be repeated and its fulfillment is checked.

Functional Requirements

- Sending Data to the Blockchain ✓

  The prototype is able to send its own transactions into the Ethereum blockchain. Therefore, this requirement is met.

- Permanent Storage ✓

  By sending transactions that are accepted by the Ethereum network, the data within these transactions is permanently stored. In order to ameliorate the quality of the data storage, a smart contract was deployed in the blockchain. This smart contract keeps track of all data that was sent with the intention of storing it permanently. It also allows to access the collected values in a structured, yet easy fashion.

- Retrieving Blockchain Data ✓

  Through the use of the remote node's JSON-RPC, all kinds of data about the blockchain, its individual blocks or general meta information can be retrieved. In the current prototype implementation, the nonce of the sending account is queried each time the MCU powers up. If the data field of the transaction gets adapted, the sensor values stored within the smart contract's array can be obtained again. Generally, every information that is accessible via the JSON-RPC could potentially be retrieved by the MCU with the help of little adaptions.

- Minimize Time Between Transactions (10 seconds) ✓

  An efficient way for building a valid transaction was found. On average, this task takes 2.3 seconds and is therefore much faster than the initially demanded 10 seconds. Even the worst case execution time lies beyond the required limit.

- Real World Interaction ✓

  With the brightness sensor, a means to interact with the real world was implemented. This light value could be measured in the surrounding of a production machine or come from the machine itself. By encapsulating the data collection into an own component, it can also be interchanged with another component measuring different data. With the MCU supporting a variety of analog interfaces, a direct communication with a production machine could also easily be implemented.

Functional Requirements

- Minimize Security Risks ✓

  The presented design minimizes risks for the integrity of obtained sensor data. Once the measured value leaves the MCU, it is already put into a signed transaction, prohibiting an attacker to change data without the signature becoming invalid. That way, only valid data gets immutably stored, which was one of the main motivations for this work. The MCU is programmed only for this single and special purpose and opens no unnecessary attack vectors to the outside. Even if the receiving Ethereum node was corrupt, it would still not be possible to store manipulated data in the name of the honest sender.

- Measure Degree of Trust ✓

  In addition to developing a concept and implementing it, a way of measuring the certainty that a transaction will become part of a block was to be found. This was done by discussing and defining a metric that takes into account different factors directly or indirectly influencing the probability for the success of a transaction.

General Requirements

- Use Appropriate Blockchain Implementation ✓

  The used blockchain implementation, Ethereum, has allowed for the fulfillment of all requirements that were initially demanded. After a discussion of the most promising platforms, Ethereum was chosen because of the generality of its design. Through the existence of many similar blockchain implementations, the basic concepts can be adapted and prototypes for other blockchains can be implemented following this design.

- Hardware-near Implementation ✓

  The implementation was done on a MCU, enabling both the interaction with the real world and the blockchain.

- Direct Sensor Connection ✓

  With the chosen MCU board, the NodeMCU, sensor data can directly be obtained without an intermediary. This is currently done for a brightness value. In the future, more sophisticated protocols for the interaction with actual production machines can be realized based on this prototype.

6.3 Conclusion

The presented design along with its prototypical implementation meets all the requirements that were posed to it. The performed proof of concept allows machines or sensors an easy and yet secure connection to a blockchain. The only prerequisite is a WiFi access point. Running an own blockchain node will likely improve the performance by speeding up the HTTPS calls, but it is not required. The low price of the MCU makes it a viable candidate for substituting gateway solutions running on general purpose computers. Additionally, the design increases the integrity of the production data. Instead of being sent from the machine to a gateway computer and being signed there, the transaction gets directly signed on the MCU itself. Furthermore, the MCU can also issue the transaction, making it difficult to alter any measured value once it left the MCU, as the signature then gets invalid. Transactions with invalid signatures are rejected by the receiving Ethereum node, preventing manipulated data from getting stored.

As already mentioned, using a blockchain as a data storage for production data only makes sense if the integrity of the immutably stored data is given. The presented design is a big step towards this goal.

Also, the transaction generation is fast and enables the transmission of data in a brief cycle time. Other models apart from the cyclic issuance of transactions can also be implemented easily. One exemplary scenario could be sending a transaction to indicate that a certain threshold was surpassed. As the MCU can also process the real world events that it measures, it could issue such a transaction and additionally operate actuators, e.g. to shut down the production machine.

7 Summary & Future Work

This chapter briefly wraps up the results of the work and gives an outlook on possible future work resulting from the developed concept.

7.1 Summary

In this thesis, an approach to enable communication with a blockchain apart from using a general purpose computer as a gateway was designed. By making it possible for a machine to post transactions with the help of a more primitive hardware, security risks can be reduced. Machine data that has to keep its integrity is no longer sent to a blockchain node or a gateway computer before being part of a signed transaction. Instead, the developed MCU is building signed transactions directly after obtaining data. Once a signed transaction is sent to a blockchain node, the measured data within it can no longer be altered without the transaction becoming invalid, resulting in its rejection by the network.

As a first step, this work presented the blockchain technology itself as well as different existing implementations of it. The usage of blockchains for industrial applications was described as well as problems resulting from current gateway solutions.

Afterwards, requirements for a concept overcoming the identified problems were introduced. Based on these requirements, a concept for a different way of blockchain interaction was developed. As a blockchain implementation, the Ethereum test net Ropsten was chosen. The MCU of choice for the developed prototype was the NodeMCU. A possible concept as well as the subsequent steps for generating a valid Ethereum transaction were investigated. This resulted in the development of two different system architecture approaches, of which the one giving more security was chosen.

The single components of the prototypical implementation and their collaboration were then presented. The different possibilities for configuring the MCU were explained and an overview on the steps of code execution was given. The modular design allows for the exchange of single components.

Apart from concept and implementation, there was another focus laid on blockchain security aspects. Therefore, different attacks on blockchains were presented. Parameters for estimating how certain it is for a transaction to be successful were identified and a metric including such parameters was developed step by step.

For the evaluation of the design and the prototype, a smart contract collecting and storing sensor values was deployed. This showed how data sent from the prototype can indeed be stored and accessed in a useful way. Thereby, the concept got validated as well. In order to further substantiate

the practicality of the developed prototype, a timing measurement was performed. The result of this measurement was a mean time between two transactions of roughly 2.3 seconds. Additionally, the different aspects leading to this overall execution time were discussed.

The prototypical implementation in combination with the smart contract receiving and storing the single measured values prove that the design is fulfilling its intended goal: measured data can be durably stored in a blockchain while at the same time the risk of tampered data becoming part of a transaction is reduced.

It is not even necessary to run an own blockchain node. The developed prototype can be operated anywhere, as long as it has access to a WiFi. By changing few of the configuration parameters, the transactions can be sent to any Ethereum network instance, not only the Ropsten test net.

Additionally, an approach to measure the degree of trust that transactions are indeed durably stored within a blockchain was suggested. Therefore, different possible parameters influencing the chances of a transaction being added into a block were discussed. It was also shown how transaction finality is estimated in the Bitcoin network mathematically. Resulting from this discussion, a metric was developed to estimate how likely it is for a transaction to succeed for any PoW-based blockchain implementation.

All the requirements that were posed prior to the design were met. This comprises the general requirements as well as functional and non-functional requirements for the concept and its implementation.

7.2 Future Work

As this work is only the beginning of research with MCUs for blockchain communication, many further aspects can be investigated in the future.

By interchanging the data collection component of the design, different interfaces for different machines could be implemented. As the MCU supports multiple serial interfaces, such an adaption could be made for various kinds of targeted machines.

As a proof of concept, it was enough to show that the proposed design works for public Ethereum instances. As a next step, one could investigate private instances that are not accessible by anyone. If sensitive production data is to be stored, only trustworthy parties should be able to read and write it. Therefore, setting up such a private network and configuring an own node for JSON-RPC access is required.

It would also be interesting to change the blockchain implementation specific components of the design in order to send transactions to other blockchains apart from Ethereum. Therefore, the transaction building detail of another implementation has to be investigated and implemented on the MCU. As the client component sends regular HTTP(S) calls, the endpoint of such a call does not have to be a JSON-RPC. Any interface accepting HTTP(S) calls is sufficient.

Also, working together with companies testing the design in a real production environment could help with finding further possible improvements.

The data necessary to check the quality of the designed transaction trust metric is not easily obtainable. This makes an evaluation for the metric rather difficult. In the future, the proposed ways of getting data to verify the metric could be tested. The metric could then be further refined or additional parameters could be found an added to the calculation.

Bibliography

[ABB+18] E. Androulaki, A. Barger, V. Bortnikov, C. Cachin, K. Christidis, A. D. Caro, D. Enyeart, C. Ferris, G. Laventman, Y. Manevich, S. Muralidharan, C. Murthy, B. Nguyen, M. Sethi, G. Singh, K. Smith, A. Sorniotti, C. Stathakopoulou, M. Vukolic, S. W. Cocco, J. Yellick. "Hyperledger Fabric: A Distributed Operating System for Permissioned Blockchains". In: *CoRR* abs/1801.10228 (2018) (cit. on p. 30).

[And96] R. J. Anderson. "The Eternity Service". In: *Proceedings of the First International Conference on Theory and Applications of Cryptology*. Prague, Czech Republic. CTU Publishing House, 1996 (cit. on p. 19).

[Ant14] A. M. Antonopoulos. *Mastering Bitcoin: Unlocking Digital Cryptocurrencies*. Sebastopol, CA. O'Reilly Media, 2014 (cit. on pp. 22, 25, 26, 73).

[Ard18] Arduino. *Arduino Language Reference*. 2018. URL: https://www.arduino.cc/reference/en/ (cit. on p. 41).

[AZV17] M. Apostolaki, A. Zohar, L. Vanbever. *Hijacking Bitcoin: Routing Attacks on Cryptocurrencies*. 2017. URL: https://btc-hijack.ethz.ch/files/btc_hijack.pdf (cit. on p. 75).

[BDPA11] G. Bertoni, J. Daemen, M. Peeters, G. V. Assche. *The KECCAK reference*. 2011. URL: https://keccak.team/files/Keccak-reference-3.0.pdf (cit. on p. 51).

[BG04] I. F. Blake, T. Garefalakis. "On the Complexity of the Discrete Logarithm and Diffie-Hellman Problems". In: *J. Complex.* 20.2-3 (Apr. 2004), pp. 148–170 (cit. on p. 51).

[Bit15] BitFury Group. *Public versus Private Blockchains*. 2015. URL: http://bitfury.com/content/downloads/public-vs-private-pt1-1.pdf (cit. on p. 23).

[Bit17] Bitcoin Wiki. *OP_RETURN*. 2017. URL: https://en.bitcoin.it/wiki/OP_RETURN (cit. on p. 27).

[Bit18a] Bitcoin Wiki. *Address*. 2018. URL: https://en.bitcoin.it/wiki/Address (cit. on p. 25).

[Bit18b] Bitcoin Wiki. *Satoshi Client Node Discovery*. 2018. URL: https://en.bitcoin.it/wiki/Satoshi_Client_Node_Discovery (cit. on p. 75).

[Bit18c] Bitnodes. *Global Bitcoin Nodes Distribution*. 2018. URL: https://bitnodes.earn.com (cit. on p. 25).

[BLMR14] I. Bentov, C. Lee, A. Mizrahi, M. Rosenfeld. "Proof of Activity: Extending Bitcoin's Proof of Work via Proof of Stake". In: *SIGMETRICS Perform. Eval. Rev.* 42.3 (Dec. 2014), pp. 34–37 (cit. on p. 24).

[BMC+15] J. Bonneau, A. Miller, J. Clark, A. Narayanan, J. A. Kroll, E. W. Felten. "SoK: Research Perspectives and Challenges for Bitcoin and Cryptocurrencies". In: *IACR Cryptology ePrint Archive* (2015) (cit. on p. 24).

[BN84] A. D. Birrell, B. J. Nelson. "Implementing Remote Procedure Calls". In: *ACM Trans. Comput. Syst.* 2.1 (Feb. 1984), pp. 39–59 (cit. on p. 45).

[Bra14] T. Bray. *The JavaScript Object Notation (JSON) Data Interchange Format*. 2014. URL: https://www.rfc-editor.org/rfc/pdfrfc/rfc7159.txt.pdf (cit. on p. 46).

[BS16] B. Blechschmidt, C. Stöker. *How Blockchain Can Slash the Manufacturing "Trust Tax"*. 2016. URL: http://www.genesisofthings.com/pdf/whitepaper.pdf (cit. on p. 36).

[But14] V. Buterin. *A Next-Generation Smart Contract and Decentralized Application Platform*. 2014. URL: https://github.com/ethereum/wiki/wiki/White-Paper (cit. on pp. 27, 43).

[Cho17] U. Chohan. *The Decentralized Autonomous Organization and Governance Issues*. 2017. URL: https://ssrn.com/abstract=3082055 (cit. on p. 37).

[CKLR17] M. Conti, S. Kumar, C. Lal, S. Ruj. "A Survey on Security and Privacy Issues of Bitcoin". In: *CoRR* abs/1706.00916 (2017) (cit. on p. 73).

[Esp18] Espressif Systems. *ESP8266*. 2018. URL: https://www.espressif.com/en/products/hardware/esp8266ex/overview (cit. on p. 41).

[Eth16] Ethereum Homestead. *Mining*. 2016. URL: https://ethereum-homestead.readthedocs.io/en/latest/mining.html (cit. on p. 29).

[Eth18a] Ethereum Community. *Ethereum Clients*. 2018. URL: http://ethdocs.org/en/latest/ethereum-clients/index.html (cit. on p. 44).

[Eth18b] Ethereum Homestead. *Account Types, Gas, and Transactions*. 2018. URL: http://ethdocs.org/en/latest/contracts-and-transactions/account-types-gas-and-transactions.html (cit. on p. 48).

[Eth18c] Etherscan. *Ethereum Average BlockSize Chart*. 2018. URL: https://etherscan.io/chart/blocksize (cit. on p. 45).

[Eth18d] Etherscan. *Ethereum Developer APIs*. 2018. URL: https://ropsten.etherscan.io/apis (cit. on p. 66).

[Fra15] P. Franco. *Understanding Bitcoin: Cryptography, Engineering, and Economics*. Chichester, West Sussex, United Kingdom. Wiley, 2015 (cit. on p. 25).

[Git16] GitHub. *Ethereum EIP 2*. 2016. URL: https://github.com/ethereum/EIPs/blob/master/EIPS/eip-2.md (cit. on p. 65).

[Git17a] GitHub. *Ethash*. 2017. URL: https://github.com/ethereum/wiki/wiki/Ethash (cit. on p. 29).

[Git17b] GitHub. *Light Ethereum Subprotocol (LES)*. 2017. URL: https://github.com/zsfelfoldi/go-ethereum/wiki/Light-Ethereum-Subprotocol-(LES) (cit. on p. 45).

[Git18a] GitHub. *Decentralized apps (dapps)*. 2018. URL: https://github.com/ethereum/wiki/wiki/Decentralized-apps-(dapps) (cit. on p. 28).

[Git18b] GitHub. *Ethereum EIP 155*. 2018. URL: https://github.com/ethereum/EIPs/blob/master/EIPS/eip-155.md (cit. on p. 52).

[Git18c] GitHub. *Getting Started with Indy*. 2018. URL: https://github.com/hyperledger/indy-node/blob/stable/getting-started.md (cit. on p. 30).

[Git18d] GitHub. *JSON RPC API*. 2018. URL: https://github.com/ethereum/wiki/wiki/JSON-RPC (cit. on pp. 48, 58).

[Git18e] GitHub. *Mining*. 2018. URL: https://github.com/ethereum/wiki/wiki/Mining (cit. on p. 29).

[Git18f] GitHub. *Proof of Stake FAQs*. 2018. URL: https://github.com/ethereum/wiki/wiki/Proof-of-Stake-FAQs (cit. on p. 29).

[Git18g] GitHub. *RLP*. 2018. URL: https://github.com/ethereum/wiki/wiki/RLP (cit. on p. 49).

[GPM17] A. Gantait, J. Patra, A. Mukherjee. *Integrate device data with smart contracts in IBM Blockchain*. 2017. URL: https://www.ibm.com/developerworks/cloud/library/cl-blockchain-for-cognitive-iot-apps-trs/index.html (cit. on p. 15).

[Gro18] I. Grokhotkov. *ESP8266 Arduino Core*. 2018. URL: http://arduino-esp8266.readthedocs.io/en/latest/index.html (cit. on p. 41).

[HS91] S. Haber, W. S. Stornetta. "How to Time-stamp a Digital Document". In: *Journal of Cryptology* 3 (1991), pp. 99–111 (cit. on p. 19).

[Hyp17] Hyperledger Architecture Working Group. *Hyperledger Architecture, Volume 1*. 2017. URL: https://www.hyperledger.org/wp-content/uploads/2017/08/Hyperledger_Arch_WG_Paper_1_Consensus.pdf (cit. on p. 29).

[Hyp18a] Hyperledger. *Hyperledger Fabric*. 2018. URL: http://hyperledger-fabric.readthedocs.io/en/latest/ (cit. on p. 30).

[Hyp18b] Hyperledger Fabric. *Transaction Flow*. 2018. URL: https://hyperledger-fabric.readthedocs.io/en/release-1.1/txflow.html (cit. on p. 31).

[IOT18] IOTA Foundation. *Frequently Asked Questions*. 2018. URL: https://www.iota.org/get-started/faqs (cit. on p. 34).

[JM99] D. Johnson, A. Menezes. *The Elliptic Curve Digital Signature Algorithm (ECDSA)*. Tech. rep. 1999 (cit. on p. 20).

[JMV01] D. Johnson, A. Menezes, S. Vanstone. "The Elliptic Curve Digital Signature Algorithm (ECDSA)". In: *Int. J. Inf. Secur.* 1.1 (Aug. 2001), pp. 36–63 (cit. on p. 51).

[JSO13] JSON-RPC Working Group. *JSON-RPC 2.0 Specification*. 2013. URL: http://www.jsonrpc.org/specification (cit. on p. 47).

[JSO18] JSON. *Introducing JSON*. 2018. URL: http://www.json.org/ (cit. on p. 46).

[KAC12] G. O. Karame, E. Androulaki, S. Capkun. "Double-spending Fast Payments in Bitcoin". In: *Proceedings of the 2012 ACM Conference on Computer and Communications Security*. New York, NY, USA: ACM, 2012, pp. 906–917 (cit. on p. 73).

[KDF13] J. A. Kroll, I. C. Davey, E. W. Felten. "The Economics of Bitcoin Mining, or Bitcoin in the Presence of Adversaries". In: *Proceedings of the Twelfth Workshop on the Economics of Information Security (WEIS)*. 2013, p. 11 (cit. on p. 74).

[KN12] S. King, S. Nadal. *PPCoin: Peer-to-Peer Crypto-Currency with Proof-of-Stake*. 2012. URL: https://peercoin.net/assets/paper/peercoin-paper.pdf (cit. on p. 23).

[Kob87] N. Koblitz. "Elliptic Curve Cryptosystems". In: *Mathematics of Computation* 48.177 (Jan. 1987), pp. 203–209 (cit. on p. 52).

[Lin15] Linux Foundation. *Linux Foundation Unites Industry Leaders to Advance Blockchain Technology*. 2015. URL: https://www.linuxfoundation.org/press-release/linux-foundation-unites-industry-leaders-to-advance-blockchain-technology/ (cit. on p. 29).

[LSP82] L. Lamport, R. Shostak, M. Pease. "The Byzantine Generals Problem". In: *ACM Trans. Program. Lang. Syst.* 4.3 (July 1982), pp. 382–401. ISSN: 0164-0925 (cit. on p. 23).

[Mat16] J. Mattila. *The Blockchain Phenomenon*. 2016. URL: http://www.brie.berkeley.edu/wp-content/uploads/2015/02/Juri-Mattila-.pdf (cit. on p. 19).

[MKL+02] D. S. Milojicic, V. Kalogeraki, R. Lukose, K. Nagaraja, J. Pruyne, B. Richard, S. Rollins, Z. Xu. *Peer-to-Peer Computing*. 2002. URL: http://www.cs.ucsb.edu/~almeroth/classes/F02.276/papers/p2p.pdf (cit. on p. 23).

[Nak08] S. Nakamoto. *Bitcoin: A peer-to-peer electronic cash system*. 2008. URL: http://bitcoin.org/bitcoin.pdf (cit. on pp. 19, 20, 25, 76).

[Nat13] National Institute of Standards and Technology. *FIPS PUB 186-4: Digital Signature Standard (DSS)*. 2013. URL: https://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.186-4.pdf (cit. on p. 20).

[Nod18a] NodeMCU. *NodeMCU Documentation*. 2018. URL: https://nodemcu.readthedocs.io/en/master/ (cit. on p. 41).

[Nod18b] NodeMcu Team. *NodeMcu Connect Things EASY*. 2018. URL: http://www.nodemcu.com/index_en.html (cit. on p. 41).

[Nxt14] Nxt community. *Nxt Whitepaper*. 2014 (cit. on p. 23).

[Par18] Parity Ethereum Documentation. *Proof-of-Authority Chains - Wiki*. 2018. URL: https://wiki.parity.io/Proof-of-Authority-Chains (cit. on p. 24).

[Pop17] S. Popov. *The Tangle*. 2017. URL: http://iotatoken.com/IOTA_Whitepaper.pdf (cit. on p. 33).

[PP15] G. W. Peters, E. Panayi. “Understanding Modern Banking Ledgers through Blockchain Technologies: Future of Transaction Processing and Smart Contracts on the Internet of Money”. In: *Computing Research Repository* abs/1511.05740 (2015) (cit. on p. 23).

[Sah17] S. S. Sahoo. *ESP8266 Node MCU Pinout*. 2017. URL: https://myelectronicslab.com/esp32-pinout-development-board-2/ (cit. on p. 42).

[Sch17] K. Schwab. *The Fourth Industrial Revolution*. New York, NY, USA: Crown Publishing Group, 2017. ISBN: 1524758868, 9781524758868 (cit. on p. 15).

[SHK17] J. J. Sikorski, J. Haughton, M. Kraft. “Blockchain technology in the chemical industry: Machine-to-machine electricity market”. In: 195 (2017), pp. 234–246 (cit. on p. 35).

[Sob17] M. Sobotka. “Gateway Administration im Zeitalter der Blockchain”. In: 1 (2017), p. 17 (cit. on p. 37).

[Sol16] Solidity. *Solidity*. 2016. URL: https://solidity.readthedocs.io/en/latest/ (cit. on p. 28).

[Sol18] Solidity. *Application Binary Interface Specification*. 2018. URL: http://solidity.readthedocs.io/en/latest/abi-spec.html (cit. on p. 83).

[Sor18] Soramitsu Co. *Iroha*. 2018. URL: https://iroha.readthedocs.io/en/latest/ (cit. on p. 30).

[SSUF16] V. Schlatt, A. Schweizer, N. Urbach, G. Fridgen. *Blockchain: Grundlagen, Anwendungen und Potenziale*. Projektgruppe Wirtschaftsinformatik des Fraunhofer-Instituts für Angewandte Informationstechnik FIT. 2016. URL: http://www.fim-rc.de/wp-content/uploads/Blockchain_WhitePaper_Fraunhofer_FIT_2016.pdf (cit. on p. 22).

[Ste17] T. Stein. *Faizod.Blockchain-Gateway*. 2017. URL: https://faizod.com/wp-content/downloads/faizod.Blockchain-Gateway-en.pdf (cit. on p. 37).

[Swa15] M. Swan. *Blockchain: Blueprint for a New Economy*. 1st. O’Reilly Media, Inc., 2015. ISBN: 1491920491, 9781491920497 (cit. on p. 15).

[Sza96] N. Szabo. *Smart Contracts: Building Blocks for Digital Markets*. 1996. URL: http://www.fon.hum.uva.nl/rob/Courses/InformationInSpeech/CDROM/Literature/LOTwinterschool2006/szabo.best.vwh.net/smart_contracts_2.html (cit. on p. 28).

[Tec18] Techopedia Inc. *Microcontroller*. 2018. URL: https://www.techopedia.com/definition/3641/microcontroller (cit. on p. 41).

[The18a] The Genesis of Things Project. *Secure Online Platform for Industrial 3D Printing*. 2018. URL: http://www.genesisofthings.com/ (cit. on p. 36).

[The18b] The IOTA Foundation. *The IOTA Foundation*. 2018. URL: https://www.iota.org/the-foundation/the-iota-foundation (cit. on p. 32).

[The18c] TheTangle.org. *TheTangle.org Live*. 2018. URL: https://thetangle.org/live (cit. on p. 44).

[TS16] F. Tschorsch, B. Scheuermann. “Bitcoin and Beyond: A Technical Survey on Decentralized Digital Currencies”. In: *IEEE Communications Surveys & Tutorials* 18.3 (Feb. 2016), pp. 2084–2123. URL: http://dx.doi.org/10.1109/comst.2016.2535718 (cit. on p. 24).

[Tu18] P. S. Tu. *Data structure in Ethereum. Episode 1: Recursive Length Prefix (RLP) Encoding/Decoding*. 2018. URL: https://medium.com/coinmonks/data-structure-in-ethereum-episode-1-recursive-length-prefix-rlp-encoding-decoding-d1016832f919 (cit. on p. 50).

[Woo18] G. Wood. *Ethereum: A Secure Decentralised Generalised Transaction Ledger*. 2018. URL: https://ethereum.github.io/yellowpaper/paper.pdf (cit. on pp. 51, 52).

[WWWS17] J. Wang, P. Wu, X. Wang, W. Shou. “The outlook of blockchain technology for construction engineering management”. In: *Frontiers of Engineering Management* 4.1, 67 (2017), p. 67 (cit. on p. 35).

[Zoh15] A. Zohar. “Bitcoin: Under the Hood”. In: *Commun. ACM* 58.9 (Aug. 2015), pp. 104–113 (cit. on p. 23).

All links were last followed on July 4, 2018.

Declaration

I hereby declare that the work presented in this thesis is entirely my own and that I did not use any other sources and references than the listed ones. I have marked all direct or indirect statements from other sources contained therein as quotations. Neither this work nor significant parts of it were part of another examination procedure. I have not published this work in whole or in part before. The electronic copy is consistent with all submitted copies.

---

place, date, signature