University of Kaiserslautern

Department of Electrical Engineering and Information Technology

Microelectronic Systems Design Research Group

Bachelor Thesis

Design and Implementation of a Blockchain-Based Smart Outlet Concept

Entwurf und Implementierung eines Blockchain-basierten Smart Outlets Konzept

---

Presented:	May 24, 2019
Author:	Daniel Gretzke (392488)
Research Group Chief:	Prof. Dr.-Ing. N. Wehn
Tutor:	M.Sc. Frederik Lauer

Statement

I declare that this thesis was written solely by myself and exclusively with help of the cited resources.

Kaiserslautern, May 24, 2019

Daniel Gretzke

Abstract

Abstract English here.

Zusammenfassung

Zusammenfassung Deutsch hier.

Contents

1 Introduction

> Whereas most technologies tend to automate workers on the periphery doing menial tasks, blockchains automate away the center. Instead of putting the taxi driver out of a job, blockchain puts Uber out of a job and lets the taxi drivers work with the customer directly.
> — *Vitalik Buterin, co-founder of Ethereum*

The blockchain was first implemented in 2009 by Satoshi Nakamoto (pseudonym) and was called Bitcoin. Since then, it has steadily gained importance every year. Meanwhile thousands of cryptocurrencies and tokens were built on this technology. The hype in the year 2017 called the attention of many companies to blockchain and even last year, when prices fell as far as 95%, the interest in this field didn't drop.

Compared to traditional payment methods like Visa, Banks and PayPal, cryptocurrencies are built decentralized, meaning that there is no central organization that controls transactions, the issuance of new money, et cetera. The validity of the blockchain *P2P* (peer to peer) network is secured through cryptographic protocols. This brings some benefits. Traditional payment methods usually go with high transactions costs, most commonly in the amount of a few percent. On the contrary the cost of a single transaction on a blockchain averages out at just a few cents[1]. Some cryptocurrencies even work without any fees.

Because of this they are suited for micro transactions really well. There are some disadvantages through. The blockchain technology is still at an early stage and really immature. Compared to traditional electronic payments it only manages to achieve very few *TPS* (transactions per second) and has long transaction times. E.g. Bitcoin manages 4-5 TPS[2] as opposed to Visa, which manages to process almost 4,000 TPS on average[3].

But, coming back to the quote from Vitalik Buterin, the key strength of blockchain and cryptocurrencies is the decentralization aspect. For many, it will reshape various markets we know today, potentially revolutionize the financial industry and even disrupt monopolies in the future.

Another trend regarding the future is the electric car. It's expected that in a few years most cars on the road and almost all cars sold will be electric. Often these need to be charged overnight. Unfortunately, most city residents are familiar with the problem that they rarely park in front of their own house, let alone own a garage. It's foreseeable that recharging your car might bring difficulties.

This bachelor's thesis devotes itself to this problem. It examines whether a smart

electric socket, which is placed outside the house by a homeowner, can be used to efficiently sell electricity and which payment method is suited best for this task. Afterwards a prototype is to be developed that implements the previously worked out concept. Additionally it will serve as an example on how to implement *M2M* (machine to machine) payments on a micro controller level.

2 Theory

The purpose of this chapter is to explain the most important terms about blockchain and the key underlying components of the technology. Most explanations will be kept superficial, as the technological implementation is too complex to deal with in a few sentences.

Hash

The cryptographic hash function is a mathematical function that usually fulfills the following requirements[4]:

1. *Determinism*: the same input always produces the same output
2. *Preimage resistance*: it's nearly impossible to get the input from the output
3. *Second-preimage resistance*: it's nearly impossible to find a second input that produces the same output as the first input

The output of this function is called hash. Bitcoin uses the SHA-256[5], Ethereum uses the Keccak-256[6] hashing algorithm which produces an output hash with a length of 32 bytes.

Public Key Cryptography

Public key cryptography[7] is the backbone of the blockchain technology and consists of a key pair – a private and a public key. As the name suggests the private key needs to be secret and the public key can be shared with anyone. A message can be signed using the private key resulting in a signature. This signature can then be verified using the public key to prove that the message was, in fact, signed by the corresponding private key. Bitcoin and Ethereum use the *ECDSA* (Elliptic Curve Digital Signature Algorithm) for signatures. They are used to prove that transactions were really sent by a specific account and not manipulated.

Wallet

Everything that is needed to send and/or receive cryptocurrencies is a private key[5], which is a wallet in its simplest form. More complex forms of a wallet encrypt a private private key with a password or store it on a dedicated hardware device.

Address

An address is used to identify participants on the blockchain. Usually it's derived from a public key belonging to the private key of an account, e.g. Ethereum uses the right

most 20 bytes of the Keccak-256 hash of the public key as the address[6].

Ledger
The equivalent of a traditional record of all addresses and their balances. Every participant of the network has a copy of this ledger.

Node
A node is a participant on the blockchain. When a new node joins the network, it downloads the ledger, currently accepted by the majority. A so-called full node has the entire history of the ledger – from the first block to the latest.

Transaction
A transaction updates the ledger, e.g. reducing the balance of an address by a specific amount and adding it to the balance of another account.

Blockchain
A list of transactions is bundled together into a block. Also included in this block is the hash of the previous block, which contains the hash of the previous block and so on. This creates a chain of blocks[5] which always point to their predecessor, thus the name blockchain. This is also the reason why a blockchain is considered immutable: modifying any information in a block would automatically change the hashes of every block after it and would be rejected by the network.

Miner / Validator
A participant of the network whose task it is to secure the network and validate / mine new transactions and blocks.

Proof of Work
There are different approaches to reach consensus over which block should be added next to the network, PoW (Proof of Work) is one of them and is used by most cryptocurrencies. A *nonce* (number used once) is included in the block which is adjusted by a validator. Every different nonce produces a different block hash. The goal is to find a nonce that is numerically lower than a certain threshold value (also called difficulty). A validator who finds this number can submit their block to the network. All miners race to find the nonce as fast as possible, so their block will be accepted by all other

participants in the network and considered valid. The fastest submission gets a so-called block reward, a reward minted by the protocol to the fastest miner and all transaction fees from that block.

Block Time

The time it takes to add a new block to the network is called block time. On the Bitcoin network the block time is 10 minutes[5], Ethereum wants to archive a block time of 12 seconds[8], realistically it's averaging at about 15 seconds[9]. The difficulty is adjusted automatically to keep the block time roughly the same, no matter how much computing power is currently mining.

Confirmation Time

The time it takes until a transaction is mined. The higher the transaction fee the faster the transaction is confirmed.

Ethereum Virtual Machine

The *EVM* (Ethereum Virtual Machine) is a turing complete virtual machine that can execute computer code on the Ethereum Blockchain. As a result, the Ethereum network acts like a decentralized computer with all the features of a blockchain: the storage is entirely public and every computation or code execution is recorded on the blockchain.

Smart Contract

Smart contracts are written in special programming languages, the most popular being called "Solidity", and are compiled into bytecode afterwards. This bytecode is then publicly stored on the blockchain and can be executed by everyone. The advantage of smart contracts is, that the code is public to everyone and immutable, thus can theoretically be used as a binding, programmable contract that can control monetary value on its own. As a drawback, every computational step and byte stored on the blockchain costs money in form of transaction cost.

DApp / Web3

DApp stands for decentralized application. Web3 is also called the decentralized web, where websites are powered by smart contracts and DApps.

Account

There are two types of accounts on the Ethereum Blockchain: An *EOA* (externally owned account) and a contract account. Each of those accounts has a transaction count and a balance[6] associated with them. An EOA is controlled by a private key, whereas a contract account has code and storage associated with it.

Off-chain Transactions / Payment Channels

For every transaction on the blockchain fees have to be paid. One way to save transaction costs is to have multiple transactions that are not recorded on the blockchain, so-called off-chain transactions, and settle them on the blockchain, once the payment process is finished.

Ether - Gwei - Wei

Ether is the currency used on the Ethereum network. It can be divided into smaller fractions. The most important are Wei and GWei. Wei is the smallest unit: 1 Ether equals 10^{18} Wei[6]. Gwei is primarily used for calculations of transaction costs: 1 Ether equals 10^9 Gwei, 1 Gwei equals 10^9 Wei. Ether is always used in Wei format in Smart Contracts.

Mainnet / Testnet

The main network of a blockchain is called the Mainnet. Testnets are blockchain networks that behave the same way as the Mainnet. This allows developers to develop smart contracts without the risk of losing any monetary value. Additionaly, improvements to the Mainnet are often tested on a Testnet prior to their official implementation.

3 Analysis of different payment methods

To revisit the concept from the introduction, the prerequisites have to be specified first:

1. A potential customer Alice, who owns an electric car and wants to purchase electricity to charge it overnight.
2. A supplier Bob, who owns a smart electrical socket, wants to sell electricity.
3. Alice and Bob do not meet, Alice does not trust Bob to supply the electricity she paid for, Bob does not trust Alice to not wrongfully revert the payment after the electricity has been supplied. It has to be assumed that there are bad actors, who want to steal from the other party (in the form of monetary value or electricity).
4. For all calculations throughout this thesis the following values are assumed:
   a) Charging power: 3.7 kW
   b) Electricity price: 0.3 €/kWh
   c) Total transferred energy: 40 kWh
   d) Charging duration: 11 hours
   e) Electricity cost: 12€
   f) Price for the customer: 18€ (profit margin of 50%)

The objective of this chapter is to evaluate which payment method is suited best for the described use case. Next, the goals of the M2M payment system have to be defined:

1. Value has to be transferred from Alice to Bob.
2. Electricity has to be supplied in return for a payment, whereby the risk or impact of not getting electricity in return has to be minimal.
3. Alice needs to be able to stop paying for electricity when it's not longer needed, e.g. when the battery is fully charged, without overpaying.
4. Bob should only start supplying electricity after a valid payment was received and thus the risk of losing electricity is minimized.
5. Value should be transferred from Alice to Bob at least once per minute, to continuously pay for electricity to reduce risk, but transaction costs should stay at a minimum at the same time. For this thesis a payment interval of 15 seconds is assumed resulting in 0.007€ per payment.

Now that the requirements are defined it can be discussed, which payment method should be implemented on the prototype. In the following paragraph the advantages and disadvantages of traditional payment methods, established electronic and online payments and cryptocurrencies are compared.

The simplest form of monetary value is cash, it could be imagined that money could be paid through a coin slot and electricity would be returned as a result. Unfortunately it's not suitable for this use case, as the risk of not receiving any electricity in return for the customer and the risk of theft for the supplier, e.g. someone violently stealing the cash, is too high.

Traditional electronic or online payments like VISA or PayPal pose a low risk of theft for the buyer because of customer protection methods, unfortunately the risk for the seller is non-negligible, e.g. in the form of credit card fraud or customer protection exploitation. Another disadvantage are high fees, PayPal, for example, has a pricing of 0.10€ + 10% for micro-transactions[10] and wouldn't be economically feasible for the predefined goals. The key strength are fast, near-instant transaction times.

Cryptocurrency payments have some major advantages over the previously analyzed payment methods. Compared to electronic and online payments, fees are low because they are fixed, not variable. Some cryptocurrencies even work without any fees at all. Payments can be broken down into fractions, as mentioned above with payment intervals of a minute or less. Because all payments are immutable the supplier has no risk, the customer risks just losing a less than one cent for a first payment. One of the biggest disadvantages of cryptocurrencies for M2M payments are long transaction times, but as the technology is still in its infancy, this might change in the future.

Over a thousand different cryptocurrencies exist and every one of them has their own rules that can drastically differ from the next one. Each underlying blockchain technology has their benefits and drawbacks. In the next paragraph some of these digital currencies will be evaluated whether or not they are suitable to reach the set goals.

Because Bitcoin is the earliest cryptocurrency, it suffers from problems other cryptocurrencies could improve upon. As demonstrated in figure 3.1, scalability is one of these issues. The block time is 10 minutes[5] and during peak times the average confirmation time can take more than two hours. Thus bitcoin should not be considered for this use case.

Ethereum has a block time of 15 seconds. At the time of writing the fee to be included in the next block is 0.01€[1]. In this case the transaction would be fast enough to meet the goal of 4 transactions per minute, but the transaction cost would exceed the payment, raising the total price as much as 143%.

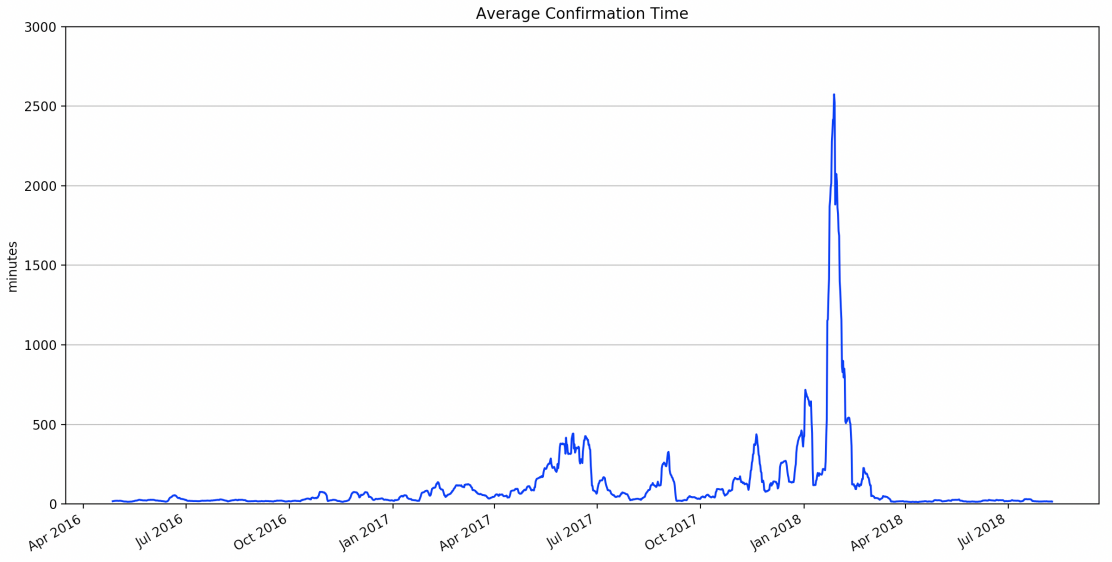

Figure 3.1: Average confirmation time of a transaction on the Bitcoin blockchain[11]

As mentioned above there are some cryptocurrencies that work without any fees that are worth to be considered. One of these currencies is IOTA. It was designed especially with M2M payments in mind and the underlying technology, called Tangle, differs from traditional blockchains. The way it works is that before someones transaction can be validated, they need to validate other transactions first[12]. In theory, this makes the network scale especially well — the more transactions are broadcasted, the shorter the transaction times get. In practice, transaction times are at 2.6 minutes at the time of writing[13].

Another feeless cryptocurrency is called Nano, formerly known as RaiBlocks. It's built upon a technology called block-lettuce. Each account on the network is an independent blockchain which can update itself asynchronously from the rest of the accounts. This makes Nano not only have zero transaction costs but also allows it to have transaction times of less than a second. It can also handle way more TPS than Bitcoin and Ethereum, namely around 100 TPS over a longer period of time with peaks up to 300 TPS[14].
Unfortunately, at the time of development of the prototype, the Nano network faced some issues which resulted in transaction times in up to 20 seconds, as seen in figure 3.2. After the V18 update these problems were solved and the transaction times returned back to less than a second[15].

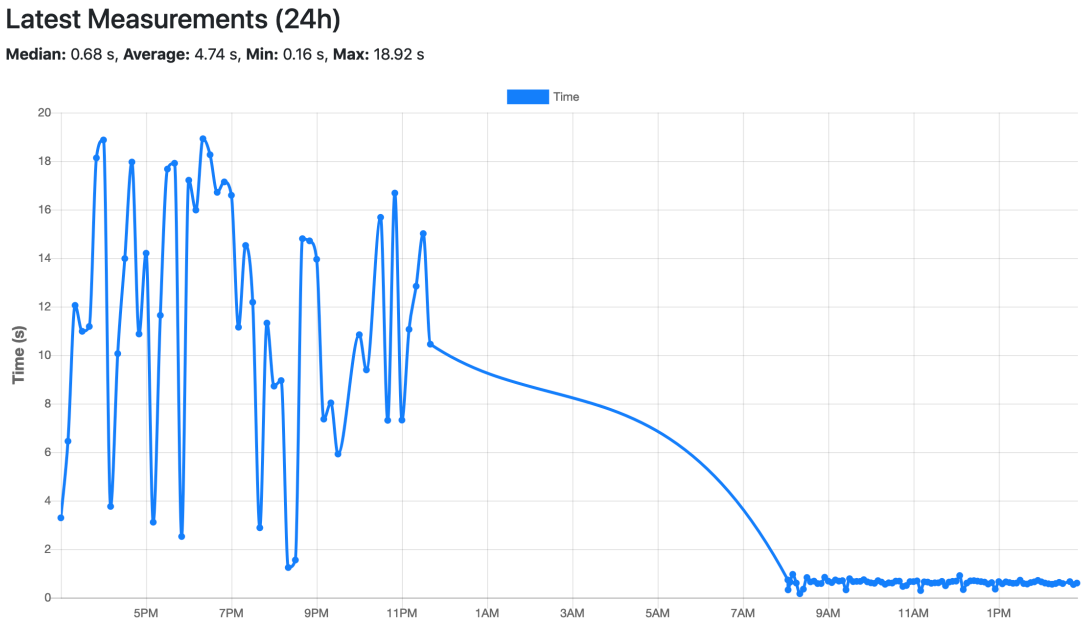

Figure 3.2: Improvements of transaction times after the update to V18 on February 22nd, 2019

Although increased transaction times did play a role, the main reason Nano was not chosen for the implementation is that it wasn't built with M2M transactions in mind. To prevent spamming, i.e. attacking the network through congesting it with transactions, PoW needs to be generated in order to send or receive transactions. According to the Nano white paper[16] an Intel Core i7-4790K processor with 4.00 GHz can handle up to 0.33 TPS. A microcontroller would not be able to generate the PoW required for the transactions in a reasonable amount of time. Instead, a dedicated server would be required for each supplier and customer, which would store the private keys and handle the transmission and reception of transactions. The micro controllers would merely communicate with said servers.

One way to cut transaction costs is the use of off-chain transactions. As previously mentioned, these work very similarly to on-chain transactions: information like a beneficiary and value can be signed by a sender, so it can be proven where the transaction originates from. They differ from each other, as off-chain transactions, as the name suggests, are not recorded on the blockchain and therefore are feeless. A number of these off-chain transactions can be bundled in a payment channel. In the end, these transactions have to be settled on the blockchain, actually modifying the ledger and updating the balances of all participants. Thus the 2,460 transactions required during a charging period of 11 hours can be bundled into a few transactions on the blockchain, while still meeting all the requirements listed above.
Ethereum was chosen for the implementation of this prototype, as it is Turing complete,

therefore a payment channel can be implemented using smart contracts. The following steps briefly explain how this concept works:

1. The customer places a deposit (max. purchase value) into the Smart Contract and initializes the payment channel.
2. The customer signs an off-chain transaction and sends it to the supplier.
3. Once the supplier receives the transaction, the delivery of electricity begins.
4. Steps 2 & 3 are repeated until the customer or the supplier want to discontinue the exchange.
5. As soon as any party wants to close the payment channel, the supplier submits the offline transactions to the Smart Contract. Each party is now able to withdraw their share from the Smart Contract.

With payment channels, the risk of the supplier is minimal, as electricity must only be provided when a valid payment was received and the customer only risks losing one transaction for the initialization of the payment channel and one off-chain transaction if no electricity is provided in return (with the values above this adds up to 0.017€).
In total, only 4 transactions have to be made. One for the payment channel initialization, one for the settlement, and one by each party to withdraw the balances totaling all transaction costs at a couple cents.

4 State of the art

A few companies already started researching and experimenting with blockchain technologies and even going as far as combining it with e-mobility. A company called accessec GmbH built a prototype of a car wallet that works with the IOTA cryptocurrency and integrates a point of sale allowing to conduct transactions with vendors seamlessly[17]. Bosch teamed up with energy supplier EnBW to build a prototype a blockchain based charging station[18]. The project that came closest to the topic of this bachelor's is called Share&Charge[19], formerly known as Blockcharge[20], which was founded by Innogy, a subsidiary of the energy company RWE[21]. It launched at the end of April 2017 with close to 1,500 charging stations, but the project was closed merely a year later[22]. It claimed to be a P2P charging network calling itself the "AirBnB of Charging Stations". It was running on the Ethereum Mainnet where anyone could become a charging station owner by purchasing a smart electrical socket to sell electricity. At first glance it seemed like the solution this bachelor's thesis was trying to achieve, but upon further investigation the decentralization aspect had to be questioned. The electrical socket was communicating with a smartphone app to manage the purchase of electricity. This app had to be preloaded with fiat currency via PayPal, etc., the transaction was conducted in fiat and a charging station owner could only withdraw fiat currency as well. There is no evidence that the monetary transaction itself was handled on the blockchain and not just the record aspect of it. Additionally it was claimed that the system worked without a middleman but the fee that had to be paid to Share&Charge with every charging process contradicts that claim. All this information led to the conclusion that the private keys probably were not handled by the users themselves which is a crucial point in building decentralized applications.
To summarize, all projects combining blockchain with e-mobility seem to be bringing existing centralized business models to the blockchain and could work just as well with traditional payment methods. Furthermore no technical information, let alone source code, could be found that this bachelor's thesis could use or improve upon.

Looking at the implementation of the interaction with the Ethereum network on a microcontroller level it was discovered that it is very far from production ready, as it is on high level programming languages. Although utility libraries exist, many of them were not compatible with the microcontroller and the ones that were, often had to have bugs fixed or additional functions implemented. Additionally, Smart Contracts mainly work with 256 bit unsigned integers for numbers which have to be passed as hexadecimal strings, which makes running the necessary code on hardware with very limited memory all the more challenging. All in all, Ethereum development on microcontrollers has a long way to go before it becomes easy to implement prototypes on it.

Lastly, payment channels were implemented in this bachelor's thesis. They are a subset of state channels, which currently are a heavily researched topic on their own[23].

As far as research went, every state channel implementation was still under development and not a single implementation of a payment channel on a microcontroller level could be found.

5 Concept, Setup & Implementation

5.1 Concept

To build a functioning prototype, which implements the payment channel described in the previous chapter, the following components are required:

1. A supplier in form of a socket
2. A customer in form of a plug
3. A server to run a node that connects to the Ethereum blockchain
4. A Smart Contract on the Ethereum blockchain

All components will communicate with each other over a Wifi connection and TCP. The following section will summarize all requirements each component of the prototype has to meet. An in depth explanation of the setup and technical implementation will follow in the next section.

Socket
An AC electrical socket is required. A microcontroller, which can be placed between the electrical circuit and the socket, needs two additional components: a WiFi module to communicate with the web and the plug and a relay to switch the current on and off.

Plug
An AC electrical plug is required, e.g. a short extension cord, that can be connected to any electronic device. A hall effect-based current meter can be attached to the hot wire to measure the current. This current meter needs to be attached to a microcontroller which also has a WiFi module to communicate with the socket and make http requests to the server.

Server
The server will run an Ethereum node. This node has to be reachable from the outside, so the microcontrollers can send http requests to it.

Smart Contract
The Smart Contract acts as a trustee, managing the money during the exchange. It has to be programmed in a way that guarantees that no party can steal from the other.

5.2 Setup

This section will focus on the technical setup of the hardware components and the installation and setup of all required software.

5.2.1 Socket

The Sonoff S20 smart socket was used for the technical implementation of the concept, as it meets all requirements listed in the section above. A microcontroller is automatically powered by the socket it's plugged into. The S20 also has a WiFi module and a relay, which can be switched on and off by said microcontroller. Lastly the microcontroller can be reprogrammed via a serial port. To program the Sonoff S20, the screws, as seen in the image below, have to be unscrewed first, revealing the logic board with the relay and the serial port.

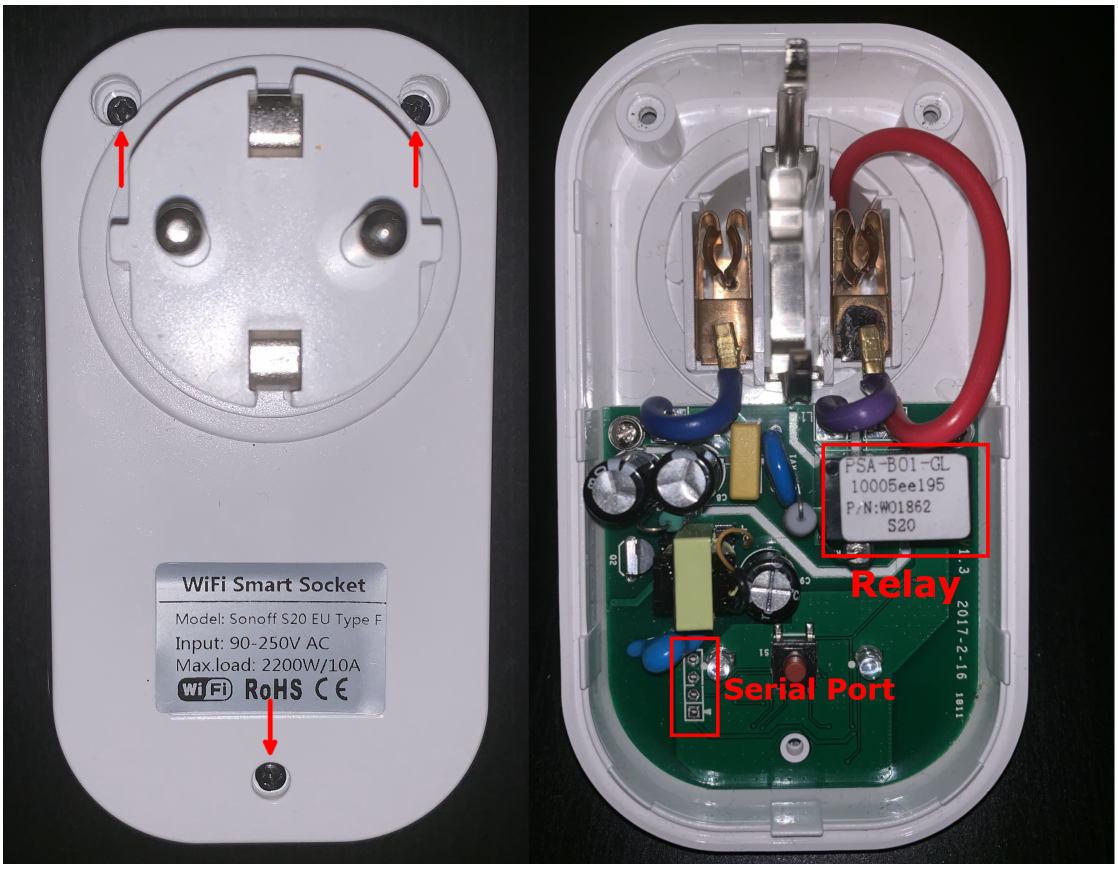

Figure 5.1: Sonoff S20

To program the microcontroller with a computer a FTDI USB to serial converter is required. The converter has to be plugged into the S20 as follows:

FTDI Converter	Sonoff S20
GND	GND
TX	RX
RX	TX
3.3V	3.3V

It's important to notice that the FTDI converter must operate at 3.3V entirely. Caution: some converters only switch the TX and RX pin to 3.3V while the VCC remains at 5V. This can fry the internals of the S20. To program the microcontroller, the button has to be pressed before plugging the pins into the serial port to put it in programming mode. After the pins have been inserted, the button can be released shortly after.

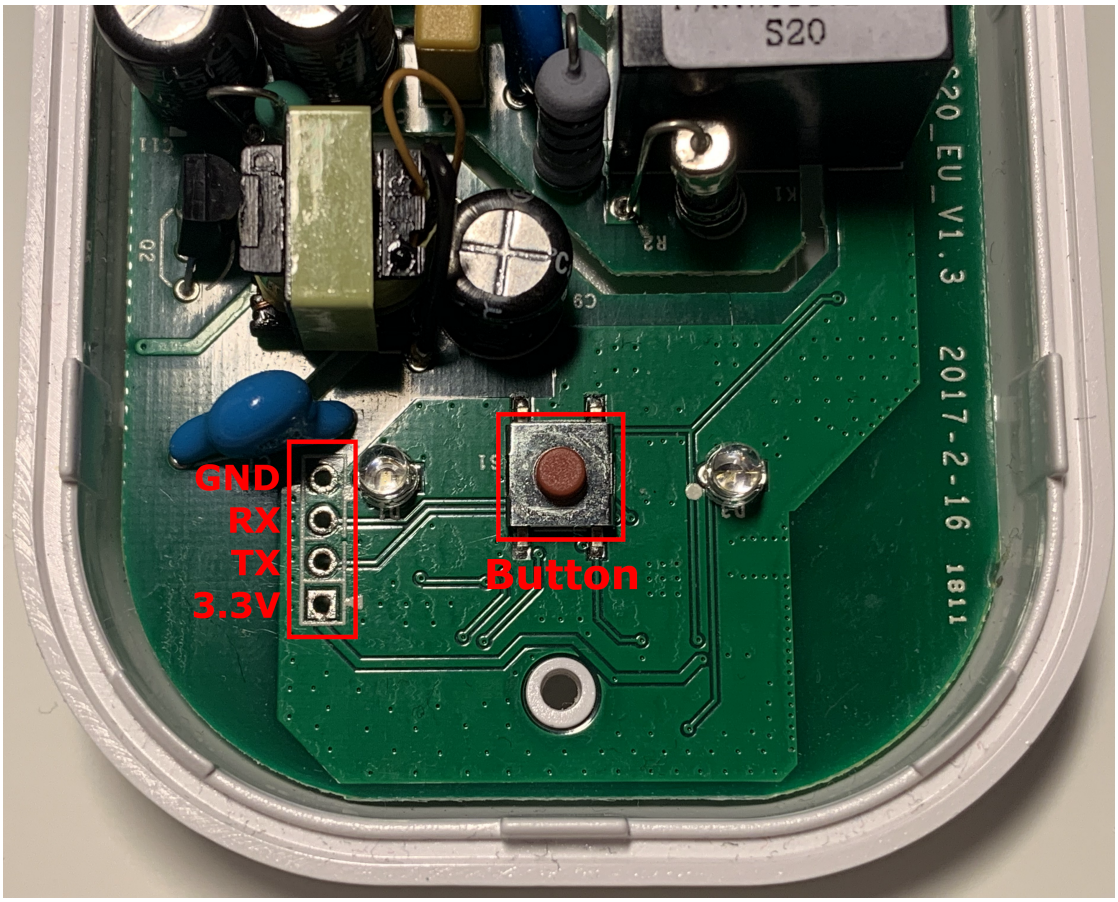

Figure 5.2: Serial ports of the Sonoff S20

Both, the socket and the plug, will be programmed using the Arduino IDE. The following steps need to be followed to install the ESP8266 Board, which the S20 is based on:

- Inside the Arduino IDE open "Preferences"
- Enter `http://arduino.esp8266.com/stable/package_esp8266com_index.json` under "Additional Boards Manager URLs"
- Open Tools → Board → Boards Manager
- Search and install "esp8266" by "ESP8266 Community"

After connecting the FTDI converter to the computer, it should appear under Tools → Port. To successfully flash code to the S20 the following settings have to be set:

- *Board*: "Generic ESP8266 Module"
- *CPU Frequency*: "80 MHz"
- *Flash Size*: "1M (no SPIFFS)"

5.2.2 Plug

The device to control the measurement of current in the plug and handle the communication with the socket is the Heltec WiFi Kit 8, which is based on an ESP8266 as well and has a 0.91 inch OLED display.

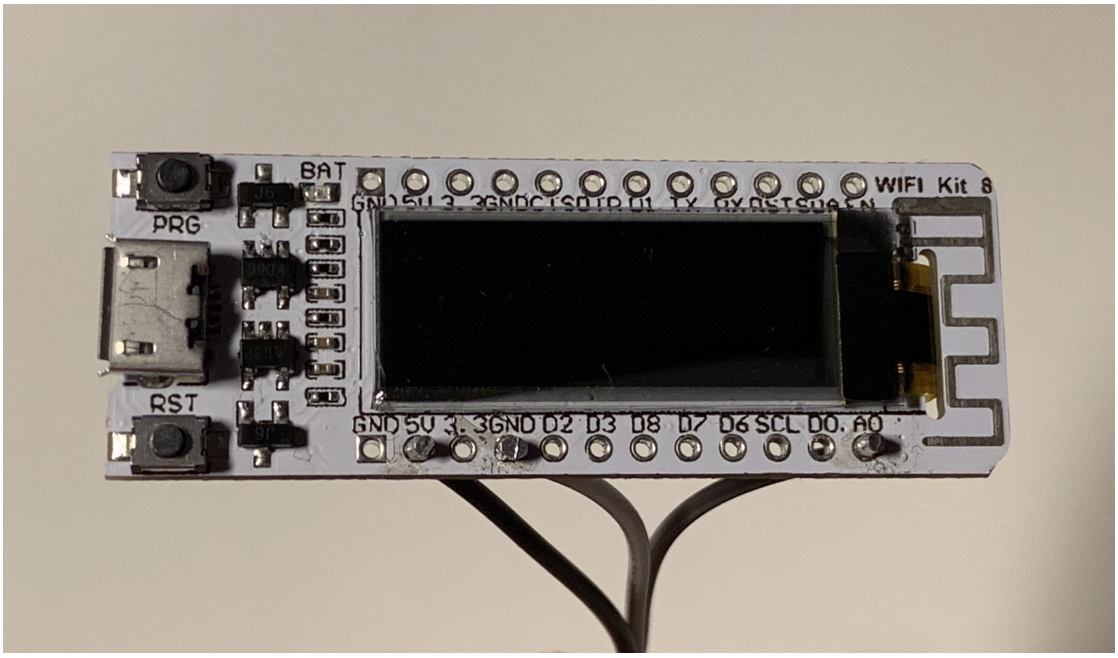

Figure 5.3: Heltec WiFi Kit 8

The ACS712 20A current meter is used to measure the current. It's hall effect-based and provides galvanic isolation up to a minimum of 2.1 kV (RMS)[24]. To connect the current meter, a part of the hot wire leading to the plug has to be cut and stripped. Both ends have to be inserted into the screw terminal of the ACS712. The current from the plug will now be redirected underneath the hall sensor.

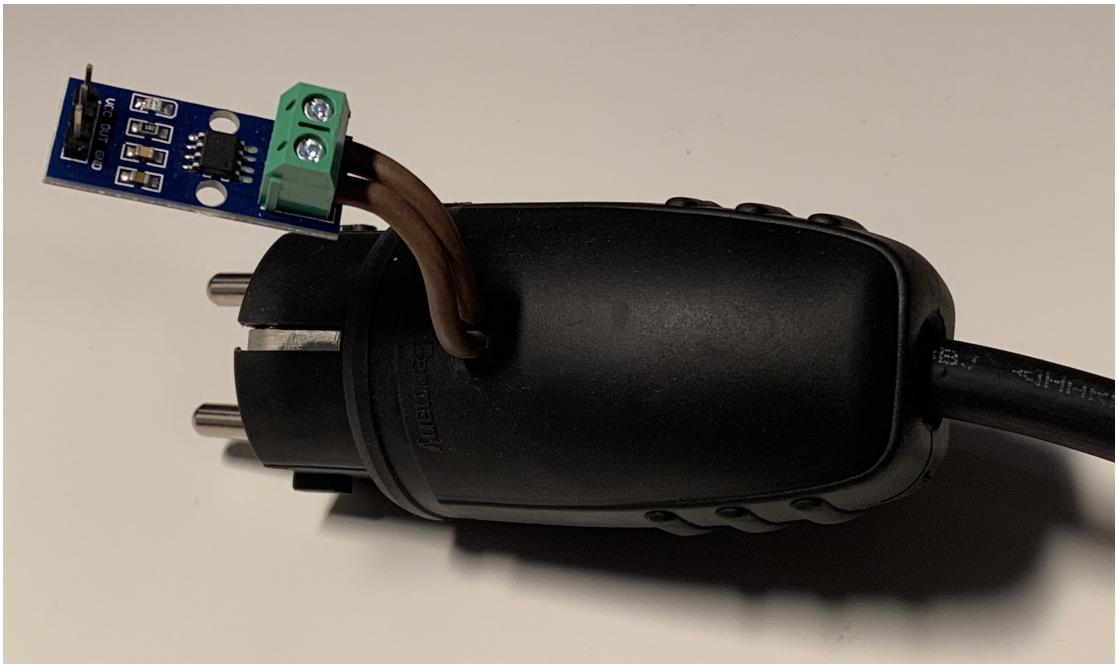

Figure 5.4: ACS712 current meter connected to plug

The pins of the current meter have to be soldered to the Heltec board as follows:

ACS712	Heltec WiFi Kit 8
VCC	5V
OUT	A0
GND	GND

To program the Heltec WiFi Kit 8, USB to UART drivers need to be installed first. The download link can be found under "References"[25]. Next, if the ESP8266 was not installed inside the Arduino IDE yet, the instructions found in the section above can be used to install the board.
After a successful driver installation, the board should be found under Tools → Port when the device is connected to the computer.

The following settings are required to ensure a successful flash of the Heltec WiFi Kit:

- *Board*: "NodeMCU 1.0 (ESP-12E Module)"
- *CPU Frequency*: "160 MHz"
- *Flash Size*: "4M (3M SPIFFS)"

5.2.3 Libraries

The prototype relies on some external libraries that have to be installed additionally:

Display
To use the display of the Heltec WiFi Kit a special library needs to be installed. Under Sketch → Include Library → Manage Library search for and install the "U8g2" library by "oliver".

WebSocket server
The communication between the plug and the socket relies on WebSockets. Download the library as a zip file from the git repository[26].
Install it via Sketch → Include Library → Add .ZIP Library.

ECDSA Library
Ethereum relies on the Elliptic Curve Digital Signature Algorithm, although there are some differences to the standard implementation of that algorithm, which will be explained later. The zip library is provided with the source code of this prototype and extends the "micro-ecc" library by Kenneth MacKay[27].

5.2.4 Server

A server is mandatory to act as a gateway to the Ethereum blockchain. Geth is the official "Golang implementation of the Ethereum protocol"[28] and is used to run a full node. After the installation it will be used to send transactions and make Smart Contract calls.

A *VPS* (Virtual Private Server) was used for this implementation running Ubuntu 18.04. The server has a six core CPU, 16 GB of RAM, 400GB SSD and 400 MBit/s unlimited traffic. The following instructions can be used to install the program under Ubuntu. For other environments the link to the instructions can be found under "References"[29].

To install geth add the repository first:

```
$ sudo add-apt-repository -y ppa:ethereum/ethereum
```

Next, install geth:

```
$ sudo apt-get update
$ sudo apt-get install ethereum
```

To interact with the Ethereum Blockchain the entire chain history has to be downloaded first. This can take up to 40 GB of disk space and will take several hours of synchronizing. Geth will be started via this console command:

```
$ geth console --rinkeby --rpc --rpcapi="db,eth,net,web3,
personal,txpool" --rpcaddr X.X.X.X --rpcport 8545 --cache=1024
```

The launch options have the following purposes[30]:

- *rinkeby*: synchronizes the Rinkeby Testnet
- *rpc*: enables the HTTP-RPC server, allows to receive JSON RPC requests
- *rpcapi*: exposed APIs, listing of all APIs can be found under "References"[31][32]
- *rpcaddr*: IP address of RPC interface, replace "X.X.X.X" with the IP address of the server, defaults to "localhost". Exposing the RPC interface without any restrictions is not advisable, especially on the Mainnet, as it's a severe security concern.
- *rpcport*: listening port of the RPC server
- *cache*: memory allocated in MB, a minimum of 1024 MB is advisable for a faster synchronization

The console parameter starts a JavaScript console, allowing to interact with the blockchain using the web3 library. Geth currently comes with web3 version 0.20.1[33] but might be upgraded to version 1.0[34] soon. The links to the documentation of both versions can be found under "References".
The version of the web3 library can be checked using:

```
> web3.version
```

The output of the synchronization could hamper the ability to properly read the output of the JavaScript console. The verbosity can be set via the following command:

```
> debug.verbosity(x)
```

Replace x with 0 for silent, 1 for error, 2 for warn, 3 for info, 4 for debug and 5 for detail. The verbosity defaults to info[30].

```
> eth.blockNumber
```

returns the block number of the latest synchronized block, which is the amount of all previous blocks. The first block, the also called the genesis block, starts with the block number 0. The current block number can be checked through so-called block explorers, e.g. rinkeby.etherscan.io.

```
> eth.syncing
```

returns the current block number and the highest block number. As soon as the client is synchronized it returns "false"[33].

5.2.5 Smart Contract

5.2.5.1 Wallet The first thing needed to start programming Smart Contracts is an Ethereum wallet. MetaMask is a browser extension for Chrome, Firefox and Opera, that not only allows to manage multiple accounts on multiple test chains, it also injects the web3.js library into websites allowing to interact with the Ethereum blockchain and Smart Contracts on web pages. Visit the MetaMask[35] website and download the browser extension. A new account will be generated using a mnemonic phrase, defined in the BIP39 (Bitcoin improvement proposal)[36]. It usually consists of 12 words which represent a private key, essentially creating an easy way to remember / write down private keys. An example for a mnemonic phrase is:

```
short heavy hidden anger nephew tragic fade dad renew finger among tiny
```

This phrase translates to the following seed:

```
b7b36d9ca1e105045344ecb7ca7b9449bfc0889139c9719876d03cf7b5814861
37e905b9e94e50c03ca22871937ae3c754dea1427eede8198c6774d90fc1a1f4
```

Using the BIP44[37] standard an unlimited amount of private keys can be derived from the seed. For example the first private key derived from this seed would be:

```
0xfb8502c03ea336344dc44b66b1a3c01e2917138e92bfa93c54725166394cd46b
```

with the corresponding address

```
0x4d43c1E254a9333fB0D8A50BD3f01b6787ee8895
```

The second derived private key is:

```
0x64b45c024041178aff2f9ed7b7026fff6890c871818c39c1c7bd826e6aa33773
```

with the corresponding address

```
0xd38F7dc2d9B6F6D9d5CB6C8813e213D5DC541458
```

and so on. This means that the single mnemonic phrase will act as a backup phrase for all accounts that will be created inside the MetaMask wallet. After MetaMask was set up create a second account and set the network to Rinkeby.

5.2.5.2 Getting Ether The next step would be to get Ethers on the Rinkeby network to interact with the Blockchain, via the website faucet.rinkeby.io. A public Facebook post or Twitter tweet containing the desired destination address has to be provided to receive the Ethers.

5.2.5.3 IDE The Smart Contract was developed using the Remix. It's an online IDE including a compiler for the language Solidity, various debugging and testing tools and can be found under remix.ethereum.org.
First a compiler has to be set inside the "Compile" tab. Because the programming language was designed especially for Ethereum, it is still under very heavy development with frequent updates coming out. The documentation for each specific version can be found under
`https://solidity.readthedocs.io/en/v0.5.8/`
while replacing "0.5.8" (the latest stable version at the time of writing) with the desired compiler version.

Inside the "Run" tab the connection to the blockchain can be chosen under "Environment". The most important options are:

- *JavaScript VM*: A personal Ethereum blockchain implemented in JavaScript that runs locally. It comes with 5 accounts which are preloaded with 100 Ether. It's suited for early development stages, unit testing, and all in all quick tests, as there are no transaction times.
- *Injected Web3*: As mentioned before, MetaMask injects Web3 into websites. When choosing this option, the currently active account and the chosen network in MetaMask are used for development.

Underneath, the Smart Contract can be chosen and deployed to the blockchain. Next to the deploy button, constructor arguments can be passed. Optionally an existing Smart Contract can be loaded from an address.

After a Contract has been deployed or loaded, all functions and public variables will be listed under the "Deployed Contracts" section. This will be the main way to interact

with the Smart Contract.

5.2.5.4 Solidity This paragraph will explain the basics and key features of the Solidity programming language. A solidity source file has the ".sol" file extension. The language has a C++ style syntax and works very similarly to object-oriented programming and is also called contract-oriented[38]. Contracts can be viewed as classes and also work with interfaces and inheritance.
See 1 for an example of a minimalistic Smart Contract, which demonstrates the general syntax as well as some of the features listed below.

General notice
There are a few important aspects in how certain things behave during Smart Contract programming and execution:

- Solidity does not implement floats at the time of writing. This is especially important for calculations using Ether. Therefore it's important to remember that Ether is always assumed as Wei by Smart Contract functions.
- When a function throws, all changes to the state that were made up to this point are reverted and the transaction is marked as failed.
- *undefined* and *null* does not exist in Solidity, variables rather have a default type, e.g. 0 for integers[39].

Types
The most important data types in Solidity are[39]:

- *bool*: The possible values are "true" and "false".
- *integer*: There are signed (*int*) and unsigned (*uint*) integers in Solidity. *uint* is an alias for *uint256*. The smallest size for an integer is 8 bit (e.g. *uint8*), the sizes grow by 8 up to 256 bit. The same applies to signed integers as well.
- *address*: The address variable stores 20 bytes.
- *address payable*: The address payable variable stores 20 bytes as well. Additionally it holds the members *transfer* and *send* to send Ether to that address.
- *bytes32*: Holds 32 bytes of data.
- *mapping(keyType => valueType)*: Mappings work similarly to hash tables. The key can be of any elementary type, the value can be of any type, even another mapping. An example for a frequently used mapping would be the balance variable: *mapping(address => uint256) balances;*
  Every address points to a uint256 which represents a balance.

- *bytes[]*: Dynamically-sized byte array.
- *string*: Dynamically-sized UTF-8 encoded string.

Pragma

The first line of a solidity file defines the compiler version[40]:

```
pragma solidity ^0.5.4;
```

The ^ symbol means that the code can be compiled by a compiler with the versions 0.5.4 and above, but below 0.6.0. Without the ^ symbol only the compiler version 0.5.4 can compile the source code.

Important variables and functions Solidity features some global units, variables and functions which can be very useful, if not necessary for Smart Contract programming:

- *ether*: The ether unit multiplies the current value by 10^{18}, e.g. 3 ether equals 3,000,000,000,000,000,000.
- *time units*: The following time units are available: "seconds", "minutes", "hours", "days", "weeks". 1 seconds equals 1, 1 minutes equals 60 seconds or 60, 1 hours equals 60 minutes or 3600 and so on.
- *now*: An alias for *block.timestamp*, a *uint256* variable as seconds since unix epoch. The variable is set by the miner during validation so it should not be used for random number generation as the number can be varied by up to ± 15 seconds, because the timestamp of a block has to be higher than the one of the previous block.
- *msg.sender*: Of type *address payable* and contains the sender of the current message. If a function is called directly by an EOA and not by a Smart Contract, *msg.sender* will contain the sender of the transaction.
- *msg.value*: Of type *uint256* and contains the number of Wei sent with the current message.
- *<address payable>.transfer(uint256 value)/<address payable>.send(uint256 value)*: Both functions send "value" in Wei to the payable address. Transfer throws, send returns false on failure.
- *function ()*: A function declared without any name is also called the fallback function. This function is called, when no matching function identifier is provided. Usually this function is triggered, when a simple transaction is sent to the Smart Contract, that's why the fallback function can often be seen with the *payable* modifier.

- *require(bool condition, string memory message)*: The require function throws if the condition is not met and provides the custom error message.
- *keccak256(bytes memory) returns (bytes32)*: Computes the Keccak-256 hash of an input.
- *ecrecover(bytes32 hash, uint8 v, bytes32 r, bytes32 s) returns (address)*: For a given message and r, s, v values of a ECDSA signature the function returns the address associated with the public key from the signature.

Constructor
A constructor function is optional and can be used to execute code directly when the Smart Contract is written to the blockchain[41].

Modifiers
A function can have multiple different modifiers, that make it behave in different ways[42]. First, there is the visibility modifier, which manages the access to functions and variables:

- *public*: The public modifier makes a function visible from inside and outside the Smart Contract. Adding the modifier to a variable automatically generates a getter function with the same name as the variable.
- *external*: The external modifier is only applicable for functions. It makes them only visible from outside the Smart Contract. To call the function from inside the Smart Contract it has to be called via "this.func()" instead of "func()".
- *internal*: The internal modifier makes a function or variable only visible internally. This means they can only be accessed from the contract and all derived contracts.
- *private*: The private modifier makes a function or variable only visible from the contract itself. It's important to notice that although a variable might be private, it still can be read, since all data stored on the blockchain is public.

Reading from the blockchain does not require mining or involve transaction fees. Therefore functions that do not write to storage can be marked as such via a modifier:

- *view*: If a function has a view modifier, it cannot write to storage, only read from it.
- *pure*: If a function has a pure modifier, it cannot modify storage. Additionally it cannot read state, e.g. read from variables. It's primarily used for computations.

The *payable* modifier allows a function to receive Ether. If Ether is included in a function call that doesn't have the *payable* modifier, it throws.

It's also possible to write custom modifiers. The example 1 will show the function of a custom modifier through a commonly used *onlyOwner* modifier, which only allows the owner of a Smart Contract to call the function.

Events

Events are an important part of Smart Contract programming for 2 key reasons:

1. With a complex Smart Contract handling thousands of transactions, it can get confusing to keep track of all changes made to the state of the Smart Contract. Events are a good way to sort these changes into different categories and make them searchable by specific filters.

2. Some time passes until a transaction was mined, therefore the return value of a function is not returned to the sender of the transaction. Events can be used to act as a return value. A computer system or a frontend can then scan for these events and act accordingly, e.g. show updates to the user.

See 1 for an example of an event.

```
1  pragma solidity 0.5.8;
2
3  contract ModifierExample{
4      // variable to store the owner of the Smart Contract
5      address owner;
6      // number to demonstrate the view function
7      uint256 public num;
8
9      // definition of an event
10     // the indexed keyword allows to use the parameter as a filter
11     event ownerChanged(address indexed oldOwner, address indexed newOwner);
12
13     // the constructor gets called after contract creation
14     // arguments can be passed to the constructor
15     constructor(uint256 _num) public {
16         // set owner to the sender of the transaction
17         owner = msg.sender;
18         // set the number to the passed argument
19         num = _num;
20     }
21
22     // custom modifier
23     modifier onlyOwner() {
24         // throws if sender of call is unequal to value stored in owner variable
25         require(owner == msg.sender, "sender is not owner");
26         // _; defines where the code of the function, the modifier is used on, runs
27         _;
```

```
28      }
29
30      // function to change the address of the Smart Contract
31      function changeOwner(address _owner) external onlyOwner {
32          // emits the ownerChanged event
33          emit ownerChanged(owner, _owner);
34          // sets the owner variable to the passed argument
35          owner = _owner;
36      }
37
38      // the function will execute the code and return the calculated number
39      // because of the view modifier no state is modified and no transaction has to be
            sent
40      function calculatedNum() public view returns(uint256) {
41          return num * 2;
42      }
43
44  }
```

Listing 1: Basic structure of a Smart Contract

5.3 Implementation

5.3.1 Microcontrollers

5.3.1.1 Transactions As both, the plug and the socket, implement transactions, the following section will explain the generation of on- and off-chain transactions on a microcontroller. The process of sending a transaction, no matter of what kind, can be broken down into the following steps:

1. Gather data
2. Encode data
3. Hash data
4. Sign hash
5. Submit signature

JSON RPC

As previously mentioned, the node was set up to accept and handle http requests, which follow the JSON-RPC 2.0 specification[43]. The implementation heavily relies on the API, as it is used to not only fetch data from the blockchain, but also to send transactions and make Smart Contract calls[31][32].

1. Gather Data

The following information is necessary to send an on-chain transaction on the Ethereum Blockchain:

1. *to*: Receiving address of the transaction.
2. *value*: Value of the transaction in Wei.
3. *data*: Hexadezimal data passed with the transaction.
4. *nonce*: The *nonce* (number used once) is important, as it protects a user against replay attacks. The nonce starts at zero and increments by one after a transaction was sent. This means the nonce always equals to the total amount of transactions sent by an account. Without said nonce an attacker could resubmit a transaction, that was already mined, over and over again to drain the balance of a victim.
5. *gas price*: The fee of a transaction that gets paid out to a miner in Wei. The higher the fee, the faster the transaction will be mined. The RPC-API method "eth_gasPrice" returns the current gas price. Websites like ethgasstation.info can be used for a more accurate estimation of the gas price, as multiple past blocks are taken into consideration and even average confirmation times for different prices

can be estimated. The Rinkeby Testnet uses a *POA* (proof of authority) mining algorithm, which only allows a few accounts to act as validators, as in a test environment without monetary incentives stability is more important than decentralization. Therefore there are no competing miners and high transaction costs are not really relevant. A transaction with a gas price of 1 GWei will be usually included in the next block resulting in a confirmation time of < 15 seconds.

6. *gas limit*: The gas limit specifies how much code can be executed with a transaction. The base limit of any transaction, whether it's a Smart Contract call or not, is 21,000 gas – it covers the storage of the transaction on the blockchain as well as the elliptic curve operation to recover the sender of the transaction[44]. Each additional computational step costs more gas, e.g. an addition (opcode: ADD) costs 3 gas, loading a variable from storage (opcode: SLOAD) costs 200 gas and even goes as far as 20,000 gas if a storage value (variable stored on the blockchain) is set from zero to non-zero. As it can be seen, storing data on the blockchain is one of the most expensive operations to disencourage storing vast amounts of information on-chain. If the gas limit is set too low and there is not enough gas to finish the Smart Contract execution, the transaction is unsuccessful and reverts any changes made. If the limit was set too high, any unused gas is refunded to the sender. Using the "eth_estimateGas" RPC-API method the gas usage of a Smart Contract call is returned. It's important to note that the node then simulates the Smart Contract call against its own copy of the ledger at the current time. Thus side effects through global variables or different senders can alter the amount of gas that a function requires to execute. The total transaction fee is calculated by multiplying the gas price with the gas limit.

7. *chain ID*: Each Ethereum Blockchain has a unique chain ID to differentiate different chains. For example the Mainnet uses the ID 1 and the Rinkeby Testnet uses the ID 4. With the Ethereum Improvement Proposal 155 (EIP-155)[45] the chain ID should be included into a transaction to prevent so-called cross chain replay attacks, where a transaction on one Ethereum chain can be resubmitted to another Ethereum chain.

The off-chain transactions require the following data – the current value of the entire transaction, a nonce and the address of the Smart Contract that is managing the payment channel. The reasoning why these informations were chosen for the off-chain transactions will be explained later in detail.

2. Encode Data

On-chain transactions are encoded with the *RLP* (Recursive Length Prefix)[6]. Before the data listed above can be hashed, it's converted into hexadecimal and therefore byte-

arrays. RLP was especially designed for Ethereum to efficiently encode these values with the following rules:

- If a single byte is in the range of 0x00 and 0x7f, the byte is its own encoding.
- If a byte array is 0 to 55 bytes long, the byte array is prefixed with a single byte with the value 0x80 plus the length of the byte array.
- If a byte array is longer than 55 bytes, its encoded by a first byte that has the value 0xf7 plus the length of the length prefix of the byte array, followed by the bytes describing the length of the byte array and finally the byte array itself.
- A list of byte arrays can be encoded as well, as it's required for the transaction data. All items are encoded individually first and then encoded again as a total byte array. This allows to encode data, no matter how nested the information is.

For the implementation a RLP C++ library was used, written by Takahiro Okada[46]. It had to be modified to make it work for the prototype though:

- The library was rewritten, to implement encoding of transactions into one utility library.
- The library now supports the EIP-155[45], when encoding transactions with the chain ID.
- A bug was fixed that would incorrectly encode byte arrays that are longer than 55 bytes.
- A bug was fixed that would throw an out of memory exception.

The modified library is included with the source code of this bachelor's thesis.

Before hashing data in a Smart Contract, which is required for validating off-chain transactions, it has to be encoded in so-called packed mode[47]. To ensure consistency, the microcontrollers implement the same encoding. The rules are a whole lot simpler than RLP encoding:

- all variables are converted to bytes and concatenated together
- "types shorter than 32 bytes are neither zero padded nor sign extended"[47]
- "dynamic types are encoded in-place and without the length"[47]

3. Hash data

The encoded data of both, on- and off-chain transactions, is hashed using the Keccak-256 hashing algorithm. The implementation relies on a library that was written by Aleksey Kravchenko and was found in the source code of the firefly DIY hardware wallet[48].

The library is also included with the source code of this bachelor's thesis.

Before signing an Ethereum specific signature, i.e. an off-chain transaction, the data has to be prefixed with an Ethereum specific prefix[49]. The data is hashed first, then prefixed according to the following rules:
"\x19Ethereum Signed Message:\n" + len(message) + message
As a hash is the message that is signed, the length of the prefix will always be 32 bytes. Finally the prefixed message is hashed again. The reasoning behind prefixing a signed message is to protect users unknowingly signing a transaction, when they actually thought they would sign a message.

4. Sign data
Although Ethereum relies on ECDSA elliptic curve signatures, there are some additional rules and features that differ the signature process from its standard implementation:

- It takes up to 2 guesses to recover the public key / address from the r & s values of an ECDSA signature. Additionally a third value v, also called the *recovery ID* is calculated. According to the Ethereum yellow paper[6] "the recovery identifier is a 1 byte value specifying the parity and finiteness of the coordinates of the curve point for which r is the x-value". Therefore the recovery ID is used to recover the address of the sender from a signature and is determined during the signature process by looking at the y value on the elliptic curve. The recovery ID is determined by the following formula, which is implemented for off-chain transactions:
  $v = 27 + (y\%2)$
  i.e. if y is even the recovery ID equals to 27, if it's odd, it equals to 28. The implementation of the aforementioned EIP-155, which secures transactions against cross-chain replay attacks, is optional and uses the following formula, which is implemented for on-chain transactions:
  $v = chain\_id * 2 + 35 + (y\%2)$
  This means that for transactions on the Rinkeby network the recovery ID is either 43 or 44.

- According to the Ethereum yellow paper[6] the s part of the signature has to be less or equal than half of the numeric value of the elliptic curve n, which equals to
  $0x7FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF$
  $FFFF5D576E7357A4501DDFE92F46681B20A0$

As previously mentioned the micro-ecc ECDSA library by Kenneth MacKay was used for the implementation. As the library only covers the standard application of the ECDSA algorithm, the library had to be extended, implementing the Ethereum specific signatures. This extended library is included in the source code of this bachelor's thesis.

5. Submit signature

To submit an on-chain transaction, the transaction data and the signature has to be RLP encoded again. The resulting byte array can then be submitted to the Ethereum network using the "eth_sendRawTransaction" RPC-API method.

Both, the plug and the socket, know all transaction data, therefore when the plug is submitting an off-chain transaction, it simply sends the signature to the socket via the WebSockets connection.

5.3.1.2 State Machine

The microcontrollers were implemented as finite state machines. The following states with the according IDs were implemented:

0: disconnected

1: connected_P

2: connected_S

3: initialized_P

4: initialized_S

5: active_P

6: active_S

7: closed

According to the different states, different code is executed – the suffix *_P* means that the plug is currently executing code and the socket is expecting a message from the plug, *_S* means the opposite. The following paragraph will explain the entire payment execution with the help of some flowcharts visualizing the state machine.

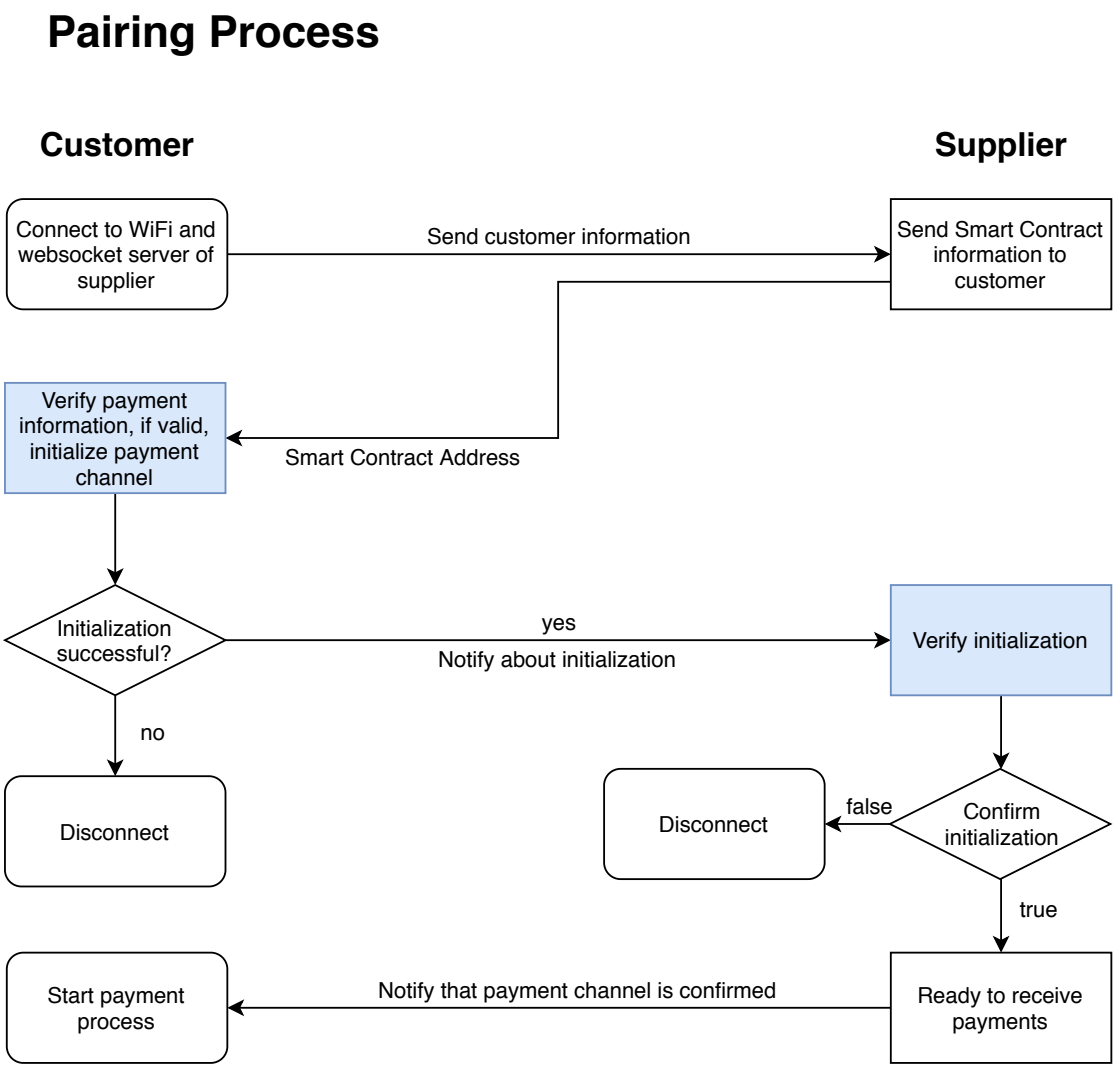

Figure 5.5: Pairing process

0: Both, the customer in form of the plug and the supplier in form of the socket start in the *disconnected* state.

1: When the plug connects to the websocket server of the supplier, both devices enter the *connected_P* state. The customer then sends information about itself to the socket in form of its Ethereum address and changes the state to *connected_S*.

2: As soon as the socket receives the customer information from the socket, it changes the state to *connected_S* as well. The supplier can now optionally validate the received data, e.g. check the address against black- or whitelists. If the customer is accepted, the socket sends the address of the Smart Contract, which manages the payment channel, to the plug and changes the state to *initialized_P*.

3: As soon as the plug receives the Smart Contract address, it changes the state to *initialized_P* as well. The customer will now fetch data from the Smart Contract, verify information and initialize the payment channel. These steps will be explained

in detail in the next section. If the initialization of the payment channel was successful, the socket is notified and the state is changed to *initialized_S*.

4: As soon as the socket receives the notification from the plug, it changes the state to *initialized_S* and verifies the initialization of the payment channel. Again, the verification will be explained later. If the initialization is confirmed, the socket is ready to receive payments, changes its state to *active_P* and notifies the plug.

5: As soon as the plug receives the notification that the socket is ready to receive payments, it changes its state to *active_P* and the payment process begins.

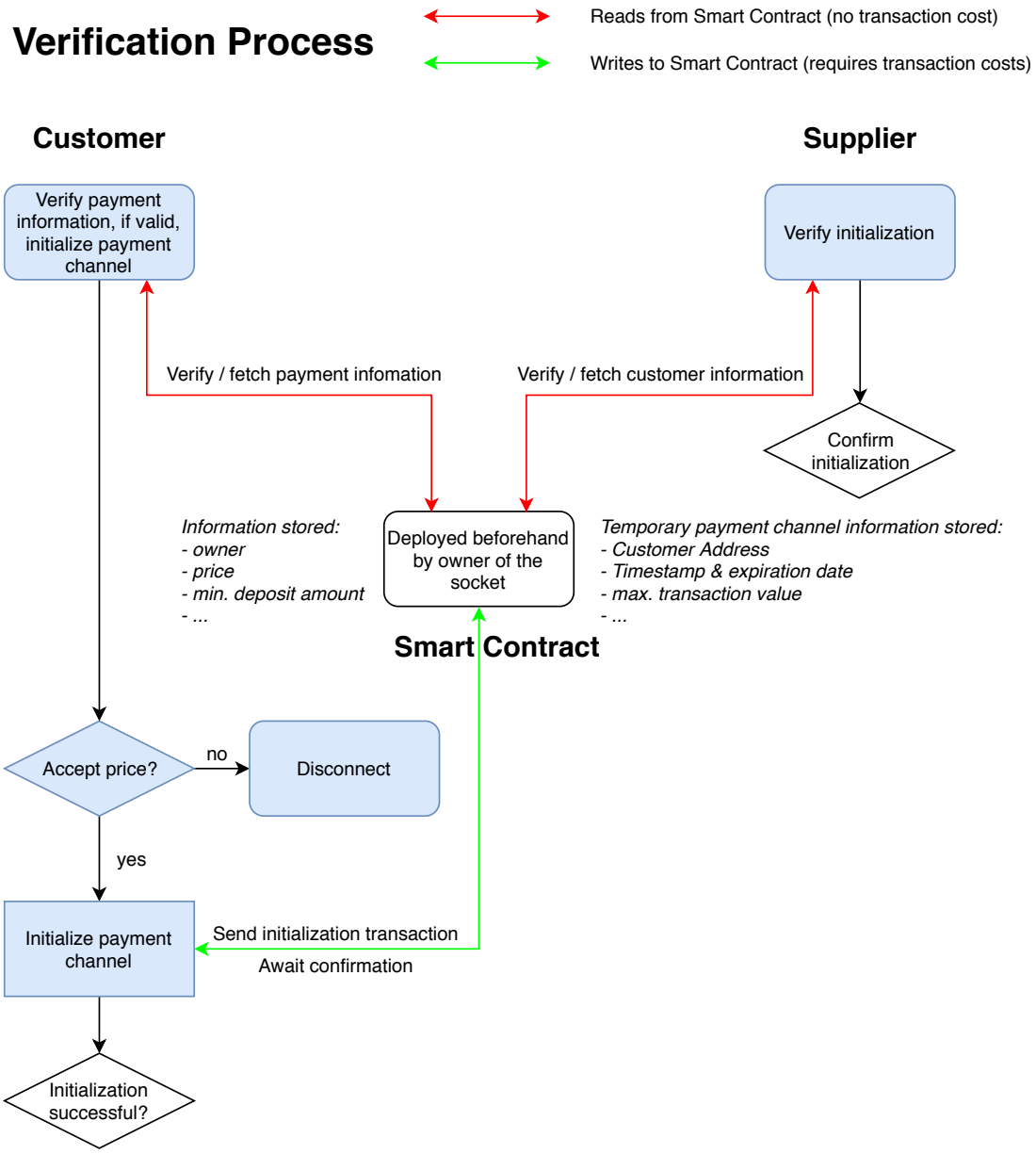

Figure 5.6: Verification process

This section will describe the verification and initialization processes that were skipped in the previous section. The flowchart 5.6 can be inserted into the blue sections of the flowchart of the pairing process 5.5.

Verification – Customer
The payment information is publicly available on the Smart Contract. After the plug received the address of said Smart Contract, it can fetch all data e.g. the address of the owner (supplier), the price per second, a minimum deposit amount, et cetera. The customer can then either accept or decline the price of the electricity. If the price is accepted, the payment channel can be initialized, through a Smart Contract call. A transaction, containing the maximum value the customer is willing to spend, is sent to the Smart Contract calling the *initializePaymentChannel()* function. The Smart Contract acts as a trustee managing the value and making sure no party can scam the other. After the transaction was sent, the plug has to wait until the transaction was mined, which will take approximately 15 seconds on the Rinkeby Testnet. The RPC-API method "eth_getTransactionReceipt" can be used to check whether a transaction was mined yet. If the payment channel was set up successfully, the Smart Contract will emit the *InitializedPaymentChannel* event.

Verification – Supplier
After the plug notified the socket about the successful payment channel initialization, the supplier should verify the initialization, e.g. fetch the maximum value of the transaction and make sure the customer address stored on the Smart Contract actually matches the customer address transmitted during *connected_P*.

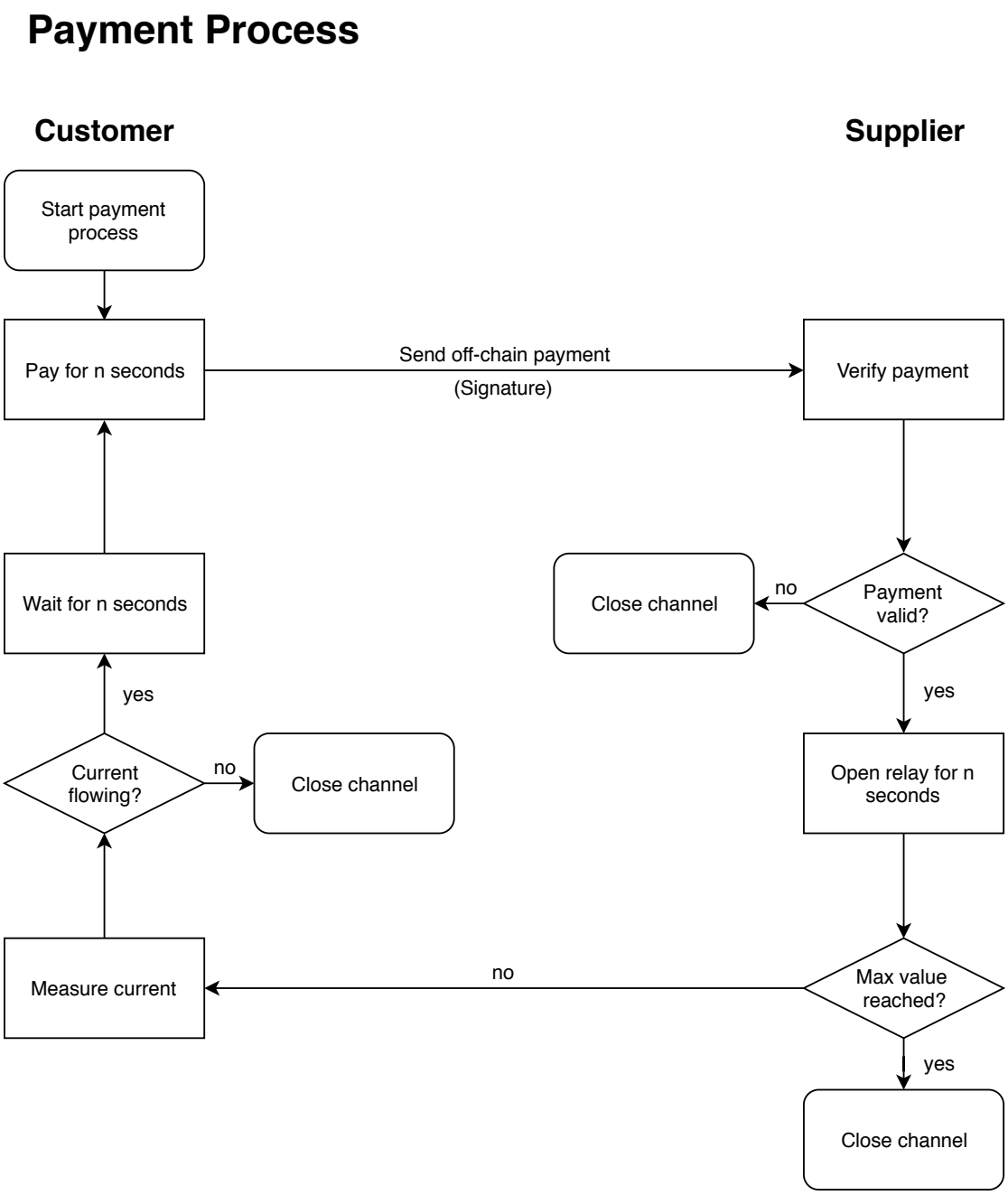

Figure 5.7: Payment process

The payment process starts when the customer sends the first signature, i.e. off-chain transaction to the supplier. The value of the transaction is calculated as follows:
$value = number\_of\_transactions * price\_per\_second * seconds\_between\_transactions$
With the assumed values from chapter 3 (a total price of 18€ over a duration of 11 hours) roughly 0.007€ are transmitted with each off-chain transaction. With the formula above, this means that the value hashed and signed is 0.007€ for the first transaction, 0.014€ for the second, and so on. This means that each time a new transaction is sent it contains the entire value and the last transaction becomes invalid. As the Smart

Contract operates on Ether and not €, the values are converted to Wei first. When the socket receives an off-chain transaction, it should verify that the payment is actually valid. The Smart Contract implements a helper function *verifySignature(uint256 _value, bytes memory _signature)* which takes the signature from the off-chain transaction and the value as a parameter and returns true, if the sender of the transaction equals to the channel customer stored in the Smart Contract, in other words the transaction is valid. If the transaction is valid, the socket opens the relay for a specific number of seconds, in this case 15 s.

To ensure that the socket doesn't deliver more electricity than the plug can pay for, the supplier should now check whether the maximum value, that was deposited into the Smart Contract, was reached. In that case the channel is closed, as described in the next section.

After the plug sends a transaction that pays for n seconds, it should start measuring the current and as soon as it is detected, start a timer and wait an appropriate amount of time to send the next transaction. In the implementation, a new transaction is sent to the socket when there is just 5 seconds of paid electricity left to ensure a steady flow without interruptions. On the other hand if the socket closes the relay and no current is measured, the socket can immediately interrupt the payment process, by closing the payment channel, without losing another payment.

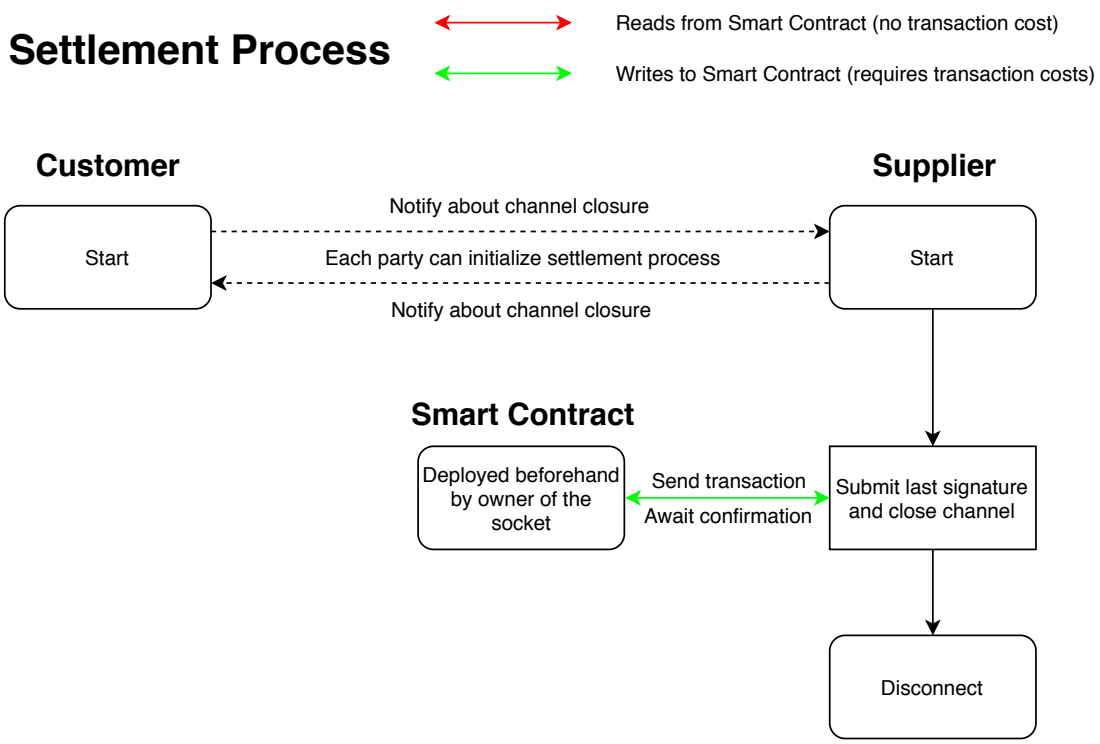

Figure 5.8: Settlement process

As soon as a party wants to stop the payment process, e.g. when the battery of the customer is fully charged or the socket received an invalid signature, it can notify the other party about it. The latest valid off-chain transaction, which contains the total value spent by the plug up to this point, is then submitted to the Smart Contract closing the payment channel, in other words settling the transaction on the blockchain. The *closeChannel(uint256 _value, bytes memory _signature)* function verifies the submitted signature and pays out the amounts accordingly.
As it is in the interest of the supplier to close the channel with the latest signature, containing the highest value, it's the sockets task to settle the transaction. The Smart Contract also protects the customer – if the supplier fails to submit a valid signature in a certain amount of time, the deposited amount can be withdrawn again.

5.3.2 Smart Contract

Security is extraordinarily important for Smart Contract development, as possibly large sums of money are handled and the code is immutable, i.e. can't be changed once the Smart Contract has been deployed to the Ethereum network. This section will go over the Smart Contract implementation and explain all key functions, some security best practices and design choices. The entire source code can be found under the appendix, listing 4 & 5.

5.3.2.1 SafeMath Library Solidity has no built in checks for integer under- and overflows, which can become a major security issue, especially when dealing with balances, allowing an attacker to drain a Smart Contract. A so-called SafeMath library, as the one found under appendix 4 can help in preventing integer under- and overflows. An example on how to use a library for a certain data type can be found in line 70 of the Smart Contract source code:

```
70 using SafeMath for uint256;
```

Listing 2: Using the SafeMath library

Here are some examples on how to add, subtract or multiply integers.

```
1  uint256 a = 5;
2  uint256 b = 7;
3
4  // add two integers
5  // c = a + b
6  uint256 c = a.add(b)
7
8  // subtract two integers
9  // d = b - a
10 // would result in an integer underflow, therefore the safemath library throws
11 uint256 d = b.sub(a)
12
13 // multiply two integers
14 // e = a * b
15 uint256 e = a.mul(b)
```

Listing 3: Examples for SafeMath calculations

5.3.2.2 Off-chain transaction As previously mentioned, there is no standard for off-chain transactions and it heavily relies on its implementation. At its core, they are nothing else than a few values that are hashed and signed. The first, most obvious part of the transaction is the value, an unsigned 256 bit integer. The second value is a nonce. It's necessary to protect the customer against replay attacks. Without the nonce, the supplier could just resubmit an off-chain transaction with a higher value from another

payment channel. The nonces are stored in a mapping variable defined in line 96 and initialized with zero for all customers. As soon as a payment channel is closed via the *closeChannel* function in line 148 or the channel times out via the *timeOutChannel* function in line 187, the nonce of the payment channel customer increments by one, rendering all off-chain transactions up to this point invalid for the future. The last value is the contract address, so the transaction is protected against replay attacks on other Smart Contracts.

The *verifySignature* function is used to verify the off-chain transaction on the Smart Contract. It takes the value and the signature of the transaction as a parameter. Then in lines 235-241 it takes the 3 values listed above, the value, nonce and contract address and recreates the message used for the signature. First the 3 values are encoded using the *abi.encodePacked()* function and hashed using the *keccak256()* function:

```
bytes32 message = keccak256(
  abi.encodePacked(
    _value,
    contractAddress,
    customerNonces[channelCustomer]
  )
);
```

The resulting hash is then prefixed by the Ethereum specific prefix and hashed a second time.

```
bytes32 prefixedMessage = keccak256(
  abi.encodePacked(
    "\x19Ethereum Signed Message:\n32",
    message
  )
);
```

This recreates the exact hash that the customer signed to generate the off-chain transaction. With this hash and the signature passed to the function, the sender of the off-chain transaction can be recovered using the *ecrecover* function in line 244 and if it matches the current channel customer stored inside the *channelCustomer* variable in line 91, the function returns true, accepting the off-chain transaction.

```
return ecrecover(prefixedMessage, v, r, s) == channelCustomer;
```

6 Conclusion

7 Appendix

```
/**
 * @title SafeMath
 * @dev Unsigned math operations with safety checks that revert on error
 * @notice https://github.com/OpenZeppelin/openzeppelin-solidity
 */
library SafeMath {
    /**
     * @dev Multiplies two unsigned integers, reverts on overflow.
     */
    function mul(uint256 a, uint256 b) internal pure returns (uint256) {
        // Gas optimization: this is cheaper than requiring 'a' not being zero, but the
        // benefit is lost if 'b' is also tested.
        // See: https://github.com/OpenZeppelin/openzeppelin-solidity/pull/522
        if (a == 0) {
            return 0;
        }

        uint256 c = a * b;
        require(c / a == b);

        return c;
    }

    /**
     * @dev Integer division of two unsigned integers truncating the quotient, reverts on
          division by zero.
     */
    function div(uint256 a, uint256 b) internal pure returns (uint256) {
        // Solidity only automatically asserts when dividing by 0
        require(b > 0);
        uint256 c = a / b;
        // assert(a == b * c + a % b); // There is no case in which this doesn't hold

        return c;
    }

    /**
     * @dev Subtracts two unsigned integers, reverts on overflow (i.e. if subtrahend is
          greater than minuend).
     */
    function sub(uint256 a, uint256 b) internal pure returns (uint256) {
        require(b <= a);
        uint256 c = a - b;

        return c;
    }

    /**
     * @dev Adds two unsigned integers, reverts on overflow.
```

```
     */
    function add(uint256 a, uint256 b) internal pure returns (uint256) {
        uint256 c = a + b;
        require(c >= a);

        return c;
    }

    /**
     * @dev Divides two unsigned integers and returns the remainder (unsigned integer
         modulo),
     * reverts when dividing by zero.
     */
    function mod(uint256 a, uint256 b) internal pure returns (uint256) {
        require(b != 0);
        return a % b;
    }
}
```

Listing 4: SafeMath library

```
pragma solidity ^0.5.0;

library SafeMath {
    // insert SafeMath library code from above
}

contract SocketPaymentChannel {
    // use the SafeMath library for calculations with uint256 to prevent integer over-
        and underflows
    using SafeMath for uint256;

    address public owner;
    uint256 public pricePerSecond;

    // stores the balances of all customers and the owner
    mapping(address => uint256) public balances;

    // duration after a payment channel expires in seconds
    uint256 public expirationDuration;
    // minimum required deposit for payment channel
    uint256 public minDeposit;

    // global payment channel variables
    // boolean whether the payment channel is currently active
    bool public channelActive;
    // timestamp when payment channel was initialized
    uint256 public creationTimeStamp;
    // timestamp when payment channel will expire
    uint256 public expirationDate;
    // address of current customer
    address public channelCustomer;
```

```
92      // value deposited into the smart contract
93      uint256 public maxValue;
94
95      // nonces to prevent replay attacks
96      mapping(address => uint256) public customerNonces;
97
98      // events
99      event InitializedPaymentChannel(address indexed customer, uint256 indexed start, uint256 indexed
            maxValue, uint256 end);
100     event ClosedPaymentChannel(address indexed sender, uint256 indexed value, bool indexed expired,
            uint256 duration);
101     event PriceChanged(uint256 indexed oldPrice, uint256 indexed newPrice);
102     event Withdrawal(address indexed sender, uint256 indexed amount);
103
104     // modifier that only allows the owner to execute a function
105     modifier onlyOwner() {
106         require(msg.sender == owner, "sender is not owner");
107         _;
108     }
109
110     constructor(uint256 _pricePerSecond, uint256 _expirationDuration, uint256 _minDeposit) public {
111         owner = msg.sender;
112         pricePerSecond = _pricePerSecond;
113         expirationDuration = _expirationDuration;
114         minDeposit = _minDeposit;
115         channelActive = false;
116     }
117
118     /// @notice function to initialize a payment channel
119     /// @return true on success, false on failure
120     function initializePaymentChannel() public payable returns (bool) {
121         // payment channel has to be inactive
122         require(!channelActive, "payment channel already active");
123         // value sent with the transaction has to be at least as much as the minimum
                required deposit
124         require(msg.value >= minDeposit, "minimum deposit value not reached");
125
126         // set global payment channel information
127         channelActive = true;
128         // set channel customer to the address of the caller of the transaction
129         channelCustomer = msg.sender;
130         // set the maximum transaction value to the deposited value
131         maxValue = msg.value;
132         // set the timestamp of the payment channel intialization
133         creationTimeStamp = now;
134         // calculate and set the expiration timestamp
135         expirationDate = now.add(expirationDuration);
136
137         // emit the initialization event
138         // It's cheaper in gas to use msg.sender instead of loading the channelCustomer
                variable, msg.value instead of maxValue, etc.
```

```
139         emit InitializedPaymentChannel(msg.sender, now, now.add(expirationDuration), msg.value);
140         return true;
141     }
142 
143     /// @notice function to close a payment channel and settle the transaction
144     /// @dev can only be called by owner
145     /// @param _value the total value of the payment channel
146     /// @param _signature the signature of the last off-chain transaction containing the
            total value
147     /// @return true on success, false on failure
148     function closeChannel(uint256 _value, bytes memory _signature) public onlyOwner returns (bool) {
149         // save value to a temporary variable, as it is reassigned later
150         uint256 value = _value;
151         // payment channel has to be active
152         require(channelActive, "payment channel not active");
153         // call verify signature function, if it returns false, the signature is invalid
                and the function throws
154         require(verifySignature(value, _signature), "signature not valid");
155 
156         // increase nonce after payment channel is closed
157         customerNonces[channelCustomer] = customerNonces[channelCustomer].add(1);
158 
159         // if maxValue was exceeded, set value to maxValue
160         if (value > maxValue) {
161             value = maxValue;
162             // value of payment channel equals the exact deposited amount
163             // credit owner the total value
164             balances[owner] = balances[owner].add(value);
165         } else {
166             // credit owner the value from the payment channel
167             balances[owner] = balances[owner].add(value);
168             // refund the remaining value to the customer
169             balances[channelCustomer] = balances[channelCustomer].add(maxValue.sub(value));
170         }
171 
172         // emit payment channel closure event
173         emit ClosedPaymentChannel(msg.sender, value, false, now - creationTimeStamp);
174 
175         // reset channel information
176         channelActive = false;
177         channelCustomer = address(0);
178         maxValue = 0;
179         expirationDate = 0;
180         creationTimeStamp = 0;
181 
182         return true;
183     }
184 
185     /// @notice function to timeout a payment channel, refunds entire deposited amount
            to customer
186     /// @return true on success, false on failure
```

```
187     function timeOutChannel() public returns (bool) {
188         // payment channel has to be active
189         require(channelActive, "payment channel not active");
190         // payment channel has to be expired
191         require(now > expirationDate, "payment channel not expired yet");
192 
193         // increase nonce after payment channel is closed
194         customerNonces[channelCustomer] = customerNonces[channelCustomer].add(1);
195 
196         // return funds to customer if channel was not closed before channel expiration
                date
197         balances[channelCustomer] = balances[channelCustomer].add(maxValue);
198 
199         // emit payment channel closure event
200         emit ClosedPaymentChannel(msg.sender, 0, true, now - creationTimeStamp);
201 
202         // reset channel information
203         channelActive = false;
204         channelCustomer = address(0);
205         maxValue = 0;
206         expirationDate = 0;
207         creationTimeStamp = 0;
208 
209         return true;
210     }
211 
212     /// @notice helper function to validate off-chain transactions
213     /// @param _value value of the off-chain transaction
214     /// @param _signature signature / off-chain transaction
215     /// @return true if the sender of the off-chain transaction is equal to the current
            customer, else false
216     function verifySignature(uint256 _value, bytes memory _signature) public view returns (bool) {
217 
218         // split signature into r,s,v values (https://programtheblockchain.com/posts
                /2018/02/17/signing-and-verifying-messages-in-ethereum/)
219         require(_signature.length == 65, "signature length is not 65 bytes");
220 
221         bytes32 r;
222         bytes32 s;
223         uint8 v;
224 
225         // split using inline assembly
226         assembly {
227             // first 32 bytes of message
228             r := mload(add(_signature, 32))
229             // second 32 bytes of message
230             s := mload(add(_signature, 64))
231             // first byte of the next 32 bytes
232             v := byte(0, mload(add(_signature, 96)))
233         }
234 
```

```
235         // to recover the address of the sender, the signed data has to be recreated
236         // variables that are included in the message: value, address of contract, nonce
                of customer
237         address contractAddress = address(this);
238         // hash variables
239         bytes32 message = keccak256(abi.encodePacked(_value, contractAddress, customerNonces[
                channelCustomer]));
240         // prefix message with ethereum specific prefix
241         bytes32 prefixedMessage = keccak256(abi.encodePacked("\x19Ethereum Signed Message:\n32
                ", message));
242         // ecrecover recovers the address from a signature and the signed data
243         // returns true if recovered address is equal to customer address
244         return ecrecover(prefixedMessage, v, r, s) == channelCustomer;
245     }
246
247     /// @notice function to withdraw funds from the smart contract
248     /// @return true on success, false on failure
249     function withdraw() public returns (bool) {
250         // save balance of sender to a variable
251         uint256 balance = balances[msg.sender];
252         // set balance of sender to zero
253         balances[msg.sender] = 0;
254         // send entire balance to sender
255         msg.sender.transfer(balance);
256         // emit withdrawal event
257         emit Withdrawal(msg.sender, balance);
258         return true;
259     }
260
261     /// @notice function to change the electricity price, only callable by the owner
262     function changePrice(uint256 _newPrice) public onlyOwner {
263         // emit price changed event
264         emit PriceChanged(pricePerSecond, _newPrice);
265         // update price per second
266         pricePerSecond = _newPrice;
267     }
268 }
```

Listing 5: Payment Channel Smart Contract

List of Figures

List of Tables

List of Listings

List of Abbreviations

References

[1] ETH Gas Station. Fees. https://ethgasstation.info/, 2019. [Online; accessed 2019-05-06].

[2] blockchain.com. 7 day average chart. https://www.blockchain.com/de/charts/transactions-per-second?daysAverageString=7. [Online; accessed: 2019-05-06].

[3] Visa. Annual report 2018. https://s1.q4cdn.com/050606653/files/doc_financials/annual/2018/Visa-2018-Annual-Report-FINAL.pdf. [Online; accessed: 2019-01-05].

[4] Shrimpton T. Rogaway P. Cryptographic hash-function basics: Definitions, implications, and separations for preimage resistance, second-preimage resistance, and collision resistance. *Roy B., Meier W. (eds) Fast Software Encryption. FSE 2004. Lecture Notes in Computer Science, vol 3017. Springer, Berlin, Heidelberg*, (2004).

[5] Satoshi Nakamoto. Bitcoin white paper. https://bitcoin.org/bitcoin.pdf, 2008. [Online; accessed 2019-05-06].

[6] Dr. Gavin Wood. Ethereum yellow paper. https://ethereum.github.io/yellowpaper/paper.pdf, 2019. [Online; accessed 2019-05-06].

[7] Scott Vansto Don Johnson, Alfred Menezes. The elliptic curve digital signature algorithm (ecdsa). https://web.archive.org/web/20170921160141/http://cs.ucsb.edu/~koc/ccs130h/notes/ecdsa-cert.pdf. [Online; accessed 2019-05-06].

[8] Vitalik Buterin. Toward a 12-second block time. https://blog.ethereum.org/2014/07/11/toward-a-12-second-block-time/, 2014. [Online; accessed 2019-05-06].

[9] etherscan.io. Ethereum average block time chart. https://etherscan.io/chart/blocktime, 2019. [Online; accessed 2019-05-06].

[10] Paypal. Fees for micro transactions. https://www.paypal.com/de/webapps/mpp/paypal-fees, 2019. [Online; accessed 2019-05-06].

[11] blockchain.com. Average confirmation time. https://www.blockchain.com/en/charts/avg-confirmation-time?timespan=all&daysAverageString=7. [Online; accessed: 2019-05-12].

[12] Serguei Popov. The tangle. https://assets.ctfassets.net/r1dr6vzfxhev/2t4uxvsIqk0EUau6g2sw0g/45eae33637ca92f85dd9f4a3a218e1ec/iota1_4_3.pdf, 2018. [Online; accessed 2019-05-06].

[13] The Tangle Monitor. Average confirmation time. https://tanglemonitor.com/, 2019. [Online; accessed 2019-05-06].

[14] Brian Pugh. Stress testing the raiblocks network: Part ii. https://medium.com/@bnp117/stress-testing-the-raiblocks-network-part-ii-def83653b21f, 2018. [Online; accessed 2019-05-06].

[15] Repnode.org. Block propagation and confirmation times. http://repnode.org/network/propagation-confirmation, 2019. [Online; accessed: 2019-02-22, 14:54].

[16] Colin LeMahieu. Nano: A feeless distributed cryptocurrency network. https://nano.org/en/whitepaper. [Online; accessed 2019-05-06].

[17] accessec GmbH. Car wallet. https://accessec.com/car-wallet/?fbclid=IwAR0bbgAra62V6FNWt_Ani5zi6v3xK4CXyNqefIPNH0ukJYZ-k3qdgUMffGQ. [Online; accessed: 2019-05-23].

[18] Bosch Presse. Dlt restores trust in the internet. https://www.bosch-presse.de/pressportal/de/en/dlt-restores-trust-in-the-internet-189824.html. [Online; accessed: 2019-05-23].

[19] Innogy. Share&charge. https://shareandcharge.com/,https://blog.slock.it/blockchain-energy-p2p-sharing-project-share-charge-going-into-live-beta-ad4e069e79d. [Online; accessed: 2019-05-23].

[20] Innogy. Blockcharge. https://www.youtube.com/watch?v=0A0LqJ9oYNg. [Online; accessed: 2019-05-23].

[21] Innogy. Share&charge: the future of charging. https://innovationhub.innogy.com/news-event/1AStXhfTXmG6EKSSQUm6mQ/share-charge-the-future-of-charging. [Online; accessed: 2019-05-23].

[22] Share&Charge. Blogpost regarding closure. http://snc-wordpress.hidora.com/information-for-users/. [Online; accessed: 2019-05-23].

[23] Ethereum Foundation. State channel research. https://ethresear.ch/c/state-channels. [Online; accessed: 2019-05-23].

[24] Allegro MicroSystems Inc. Fully integrated, hall effect-based linear current sensor. https://www.sparkfun.com/datasheets/BreakoutBoards/0712.pdf. [Online; accessed: 2019-05-11].

[25] Silicon Labs. Cp210x usb to uart bridge vcp drivers. https://www.silabs.com/products/development-tools/software/usb-to-uart-bridge-vcp-drivers. [Online; accessed: 2019-05-12].

[26] Links2004. Websocket server and client for arduino. https://github.com/Links2004/arduinoWebSockets. [Online; accessed: 2019-05-12].

[27] Kenneth MacKay. micro-ecc. https://github.com/kmackay/micro-ecc. [Online; accessed: 2019-01-15].

[28] Ethereum Foundation. Go ethereum. https://github.com/ethereum/go-ethereum. [Online; accessed: 2019-05-11].

[29] Ethereum Foundation. Installing geth. https://github.com/ethereum/go-ethereum/wiki/Installing-Geth. [Online; accessed: 2019-05-12].

[30] Ethereum Foundation. Command line options. https://github.com/ethereum/go-ethereum/wiki/Command-Line-Options. [Online; accessed: 2019-05-12].

[31] Ethereum Foundation. Json rpc api. https://github.com/ethereum/wiki/wiki/JSON-RPC. [Online; accessed: 2019-05-12].

[32] Ethereum Foundation. Management apis. https://github.com/ethereum/go-ethereum/wiki/Management-APIs. [Online; accessed: 2019-05-13].

[33] Ethereum Foundation. Javascript ethereum api 0.2x.x. https://github.com/ethereum/wiki/wiki/JavaScript-API. [Online; accessed: 2019-05-13].

[34] Ethereum Foundation. Javascript ethereum api 1.0. https://web3js.readthedocs.io/en/1.0/. [Online; accessed: 2019-05-13].

[35] Metamask. Wallet browser extension. https://metamask.io/. [Online; accessed: 2019-05-13].

[36] Bitcoin. Bip39 mnemonic phrases. https://github.com/bitcoin/bips/blob/master/bip-0039.mediawiki. [Online; accessed: 2019-05-13].

[37] Bitcoin. Private key derivation. https://github.com/bitcoin/bips/blob/master/bip-0044.mediawiki. [Online; accessed: 2019-05-13].

[38] Ethereum Foundation. The solidity contract-oriented programming language. https://solidity.readthedocs.io/en/v0.1.5/README.html. [Online; accessed: 2019-05-14].

[39] Ethereum Foundation. Types. https://solidity.readthedocs.io/en/v0.5.3/types.html. [Online; accessed: 2019-05-16].

[40] Ethereum Foundation. Layout of a solidity source file. https://solidity.readthedocs.io/en/v0.5.8/layout-of-source-files.html. [Online; accessed: 2019-05-14].

[41] Ethereum Foundation. Constructor. https://solidity.readthedocs.io/en/v0.5.8/contracts.html. [Online; accessed: 2019-05-15].

[42] Ethereum Foundation. Modifiers. https://solidity.readthedocs.io/en/v0.5.8/miscellaneous.html#function-visibility-specifiers. [Online; accessed: 2019-05-15].

[43] JSON RPC Google Group. Json rpc 2.0. https://www.jsonrpc.org/specification. [Online; accessed: 2019-05-19].

[44] Ethereum Foundation. Design rationale. https://github.com/ethereum/wiki/wiki/Design-Rationale. [Online; accessed: 2019-05-19].

[45] Ethereum Foundation. Eip 155. https://github.com/ethereum/EIPs/blob/master/EIPS/eip-155.md. [Online; accessed: 2019-05-19].

[46] Takahiro Okada. Rlp c++ implementation. https://github.com/kopanitsa/web3-arduino/blob/master/src/Util.cpp. [Online; accessed: 2019-05-21].

[47] Ethereum Foundation. Packed encoding. https://solidity.readthedocs.io/en/v0.4.24/abi-spec.html#abi-packed-mode. [Online; accessed: 2019-05-21].

[48] Aleksey Kravchenko. Keccak-256 c implementation. https://github.com/firefly/wallet. [Online; accessed: 2019-05-21].

[49] Ethereum Foundation. Signature prefix. https://github.com/ethereum/wiki/wiki/JSON-RPC#eth_sign. [Online; accessed: 2019-05-23].