University of Zurich[UZH]

BACHELOR THESIS – Communication Systems Group, Prof. Dr. Burkhard Stiller

Design and Prototypical Implementation of an IoT Identification Platform based on Blockchains and Physical Unclonable Functions (PUF)

Benjamin Jeffrey
Zurich, Switzerland
Student ID: 14-921-530

Supervisor: Sina Rafati, Dr. Eryk Schiller
Date of Submission: April 28, 2020

University of Zurich
Department of Informatics (IFI)
Binzmühlestrasse 14, CH-8050 Zürich, Switzerland

ifi

Bachelor Thesis
Communication Systems Group (CSG)
Department of Informatics (IFI)
University of Zurich
Binzmühlestrasse 14, CH-8050 Zürich, Switzerland
URL: http://www.csg.uzh.ch/

Abstract

Deutsch

Mit dem explosiven Wachstum des Internet of Things in jüngster Zeit wird viel Forschung betrieben, wie die enormen Mengen an Daten, welche die verbundenen Geräte generieren, am besten verarbeitet werden könnten. Eine der anderen aufkommenden Technologien der vergangenen Jahre, Blockchain, wird dabei oft vorgeschlagen, um in Verbindung mit dem Internet of Things, Ziele wie das Überwachen von Lieferketten oder das Anbieten eines Marktplatzes für die gesammelten Daten, auf eine besser skalierbare und dezentralisierte Weise zu erreichen. Es ist deshalb Essenziel, die vielen individuellen Geräte, aus welchen diese Netzwerke bestehen, zuverlässig identifizieren zu können, da kein Vertrauen zwischen teilnehmenden Parteien durch einen Drittanbieter gesichert wird. Diese Arbeit schlägt ein Design für eine Plattform vor, welche IoT verbundene Geräte auf der Ethereum Blockchain identifiziert. Um eine zuverlässige Identifizierung von diesen Geräten zu ermöglichen, wird eine Physical Unclonable Function verwendet, welche Abweichungen im Herstellungsprozess des im Gerät integrierten SRAM Chips nutzt, um einen einzigartigen, identifizierbaren Schlüssel abzuleiten. Dieser Identifikationsmechanismus wird mit dem ERC 734 / 735 Ethereum Identifikationsstandard kombiniert, um jedes Gerät auf der Blockchain zu vertreten. Das Resultat dieser Arbeit ist ein funktionsfähiger Prototyp dieser Plattform, welcher eine Benutzeroberfläche enthält, die mit der Blockchain interagiert. Gerätebesitzer können diese Plattform nutzen um ihre Geräte zu registrieren und deren Identitäten durch die Plattform verifizieren zu lassen.

English

With the explosive growth of the Internet of Things in recent years, there is much research going into how to best handle the vast amounts of data all these connected devices produce. One of the other upcoming technologies of recent years, blockchains, are often proposed to be used in conjunction with the Internet of Things to achieve things like monitoring supply chains or providing a marketplace for gathered data in a more scalable and decentralized way. It is essential, therefore, to be able to reliably identify the many individual devices that make up these networks, as trust between participating parties is not guaranteed by a controlling third party. This thesis proposes a design for a platform, which identifies IoT-connected devices on the Ethereum blockchain. To achieve a reliable identification of these devices, a Physical Unclonable Function is used, which takes advantage of the manufacturing variability of the device's SRAM chip to derive a unique identifying key

for each device. This identification mechanism is combined with the ERC 734 / 735 Ethereum identity standard to get each device represented on the blockchain. The result of this thesis is a functional prototype implementation of this platform, which includes a user interface that interacts with the blockchain. Device owners can use this platform interface to register their devices and have their identities verified by the platform.

Acknowledgements

I would like to thank Sina Rafati for supervising this thesis and always being available on short notice to have discussions concerning the project.

Furthermore, I would like to thank Vaidya Girish Bhagwan and Dr. T V Prabhakar from the Indian Institute of Science, Bangalore, for offering valuable guidance on Physical Unclonable Functions.

I would also like to thank Prof. Dr. Burkhard Stiller for letting me write this thesis under his Communication Systems Group at the Department of Informatics.

Lastly, I would like to express my gratitude towards Dr. Eryk Schiller, for helping me out with occasional problems concerning the hardware used for this project.

Contents

Chapter 1

Introduction

1.1 Motivation

The Internet of Things (IoT) market has been consistently growing in recent years and is starting to see wide-range adoption by both businesses and private costumers, with the number of IoT-connected devices expected to triple between 2018 and 2023. The growth of the IoT is projected to be accelerated further by the continued advancements in computing power, sensor technology, the upcoming of 5G technology and the growing interest of investors in the IoT [1]. Many of the use-cases for these devices require the transfer of data from remote locations. Due to IoT-connected devices being increasingly mobile and distributed over vast distances, as well as the increased importance of data security, a secure way of identifying these devices is vital.

The transfer of data is typically encrypted using a secret key, that is often recorded in non-volatile storage. However, this storage solution is susceptible to numerous possible attacks, where an unauthorized retrieval of the secret key could occur [2]. The concept of Physical Unclonable Functions (PUFs) promises a possible solution to this problem. As a side effect of the manufacturing process, small variations in the characteristics of Integrated Circuits (ICs) occur. These variations are utilized by PUFs to derive unique keys based on the individual characteristics of the IC, thus enabling secret key generation without the need for storage in non-volatile memory [3].

With the rise of big data, machine learning and artificial intelligence, the value of data has skyrocketed in recent years. Abundantly deployed IoT-connected devices can provide valuable sensor data, that can be sold to third parties for further processing. In order to secure the necessary trust for transactions between these parties, a mechanism is required that guarantees the sold data is not coming from imposter devices. Similarly to the registration of customers in a Know Your Customer (KYC) database which verifies the identity of individuals, a Know Your Device (KYD) platform for the identification of IoT-connected devices could provide the sought after trust.

1.2 Description of Work

In this thesis, we will design and implement a KYD platform for the identification of IoT-connected devices using a PUF. As mentioned in the previous section, PUFs can derive a unique identifier from the physical characteristics of an IC. To understand PUFs, it is helpful to imagine a PUF as a black box. It takes some form of an IC's characteristics as input and produces a key that is unique to the given IC, as shown in Figure 1.1. The function must produce the same key when using the same IC with high probability and must, within reason, produce a different key for each new IC.

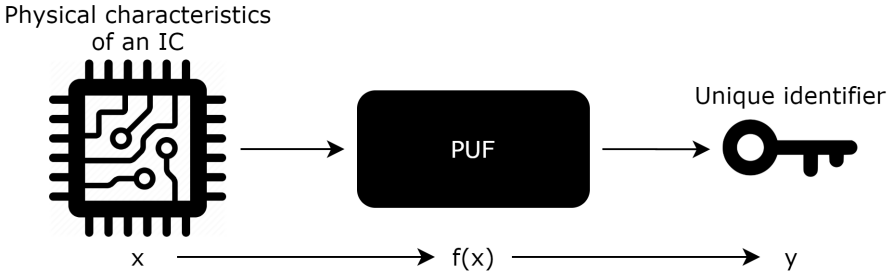

Figure 1.1: The structure of a Physical Unclonable Function (PUF)

It is helpful to think of a black box because there are many different PUFs for various kinds of input data. In fact, any function that derives a unique identifier from provided physical characteristics of a semiconductor device is called a PUF. The different kinds of input data that could be requested from the IC are also often referred to as challenges because they are used to test the IC and see if the response is as expected. The combination of the challenge and the expected response is known as a Challenge-Response Pair (CRP). For this thesis, we will be looking into the development of a static random-access memory (SRAM)-based PUF, since many IoT devices, like the Arduino Mega2560 used in this thesis, have a sufficient amount of SRAM built in. The manufacturing variability of SRAM chips causes small mismatches between N-channel and P-channel transistors, which result in individual SRAM cells having a bias towards 0 or 1 when powering on the chip. Since these cell's values are consistent for each powering on, we can view them as a physical characteristic for the chip. We can, therefore, use the start-up values of SRAM bits as the input for a PUF [2].

However, not all SRAM cells will consistently provide a stable start-up value. These unstable bits introduce noise that makes it impossible to use the raw SRAM reading for a unique key since any collection of SRAM cells will have some variation on each power cycle. We must, therefore, introduce some error correction to be able to retrieve the same key on every power-up of the IoT-connected device. We will be utilizing a fuzzy extractor for this purpose in this thesis. A fuzzy extractor is a primitive used for the generation of strong keys from noisy data, often used for biometrics like extracting a unique key from fingerprints [4].

For the registration and verification of devices, a KYD platform will be developed. We want this platform to integrate with the Ethereum blockchain so that future projects can

expand on the design proposed in this thesis. The end goal is to implement a decentralized marketplace for data collected by IoT-connected devices. For this to be viable, the IoT devices need to be reliably identifiable. We will implement an Ethereum identity standard, that ensures each device is represented on the blockchain by a smart contract (SC). We will then have the KYD platform issue a certification to these smart contracts that verifies their identity. We will also link the devices to a smart contract that represents the device owner and show how he could also be verified by a KYC platform. The trust between a potential buyer on the decentralized marketplace and the device he is transacting with could be significantly increased this way. This is because the device would be a verified entity that is associated with a real-world person that is far more accountable for any wrongdoings than an anonymous Ethereum account would be.

The KYD platform will include a web interface to register users and subsequently, their devices. It will also deploy the smart contracts to the Ethereum blockchain. The user data, as well as the device data, including its PUF-generated unique key, will be stored in a database connected to a server. The server will interface with the web application and process requests to verify devices using the implemented PUF.

1.3 Thesis Outline

In order to better understand the system proposed in this thesis, Chapter 2 provides some background on the implemented technologies. It further presents some related work to this thesis.

In Chapter 3, we go over the design of the prototype implemented for this thesis. Design choices and considerations are explained in this chapter, as well as the requirements for each component of the system. The data flow for the various processes included in the design is also explained using a series of sequence diagrams.

The details of the implementation, including the most critical lines of code, are then covered in Chapter 4.

Chapter 5 contains an evaluation of the implemented prototype. Its overall strengths are covered as well as the possibilities to improve the system. Some of the future work that could be undertaken based on the discoveries of this thesis are also mentioned in this chapter.

The project is then summarized in Chapter 6, bringing this thesis to its conclusion.

Chapter 2

Related Work and Background

This chapter explains the most important concepts and technologies used in this thesis. Parallel to this, the most relevant related work is mentioned, that could be used to deepen the readers understanding of the discussed concepts.

2.1 Physical Unclonable Functions (PUFs)

The concept of PUFs, originally termed Silicon Physical Random Functions, was first proposed in 2002 by [3]. They described several possible implementations of circuits to identify and authenticate individual ICs, using Field Programmable Gate Arrays (FPGAs). Their thesis was that there is sufficient variability in the manufacturing process of IC's to produce a unique set of CRPs for each circuit, thus enabling both identification and authentication of ICs. They then showed that by taking advantage of statistical delay variation for equivalent wires and devices, they could generate these unique sets of CRPs for each IC [3].

Since then, there have been numerous designs for PUFs, that have been put forward. [5] proposed using the power-up state of SRAM as an identifying fingerprint in 2008. [2] investigated using this approach as a PUF for chip identification in 2013. The manufacturing process of SRAM chips causes mismatches between the P-channel and N-channel transistors that result in most individual cells of the chip having a bias towards either 0 or 1 during powering up. [2] showed that the microcontroller they were testing, the ATMega 1284p contained enough entropy to uniquely identify each chip based on its power-up state. They also showed, however, that the error rates, meaning the rate of the SRAM cells that behaved completely randomly, of SRAM PUFs are too high to be used for applications such as key generation and authentication, without utilizing some form of error correction [2].

2.2 Fuzzy Extractors

Fuzzy Extractors were proposed in 2008 by [4], as a primitive to extract nearly uniform randomness from noisy data in order to securely authenticate biometric data. The primitive can be applied to any information that is not reproducible precisely and is not distributed uniformly. The output of fuzzy extractors is error-tolerant, meaning the same output is produced for input data that is slightly variant, as long as the differences remain within some margin. Two separate scannings of the same fingerprint would, therefore, produce the same key, if the scannings match closely enough to assume they came from the same finger. This can be applied to many different kinds of input data and thus can be used to generate a unique key from power-up states of SRAM chips if the error rate is low enough. The process behind fuzzy extractors is outlined in Figure 2.1. The result is the extraction of a uniform random string R from some input w. Additionally, an input w' that is close to w can reproduce an identical R. This is achieved using two separate procedures, the generation procedure (GEN) and the reproduction procedure (REP). During the generation procedure, the fuzzy extractor outputs a non-secret helper P alongside R. During the reproduction procedure, P is then used together with w' to reproduce R, as long as the distance between w and w' is below a threshold t. This will only work with both w' and P, as neither provides adequate information on R on its own [4].

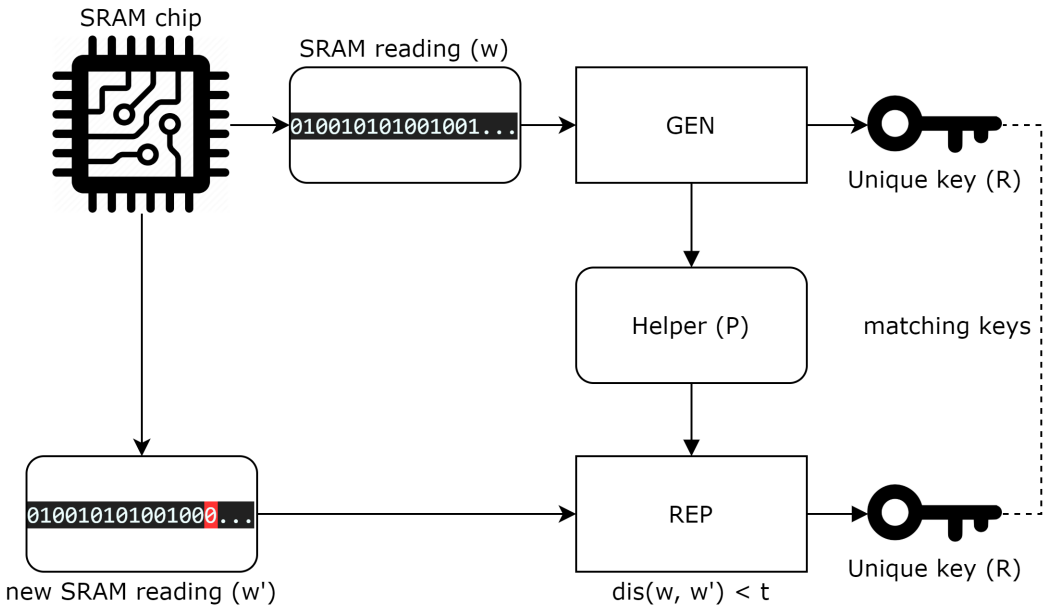

Figure 2.1: Fuzzy Extractor using power-up SRAM state as input data

2.3 Ethereum Identity Standards

Handling identities is essential in many of today's interactions. Without proof of their identity, individuals cannot travel, vote, drive a car, open a bank account or transact money. In short, it is nearly impossible to participate in society. Traditionally it is

governments or other centralized authorities that issue proofs of an individual's identity through distributing passports or setting up bank accounts. This poses some problems in a decentralized, digital environment, like the Ethereum blockchain, however, as relying too heavily on centralized authorities negates many of the benefits of blockchain technology and keeping track of physical artefacts like passports is highly inefficient. Handling digital identities is a broad research subject on its own with much hope being put into blockchain technology to solve this, due to its immutable nature [6]. On Ethereum specifically, there are many standards to handle the identity of users and devices that are being developed, as none of them has managed to establish itself as the leading standard yet. Proposals for standards like these on Ethereum are submitted through Ethereum Requests for Comments (ERCs) or Ethereum Improvement Proposals (EIPs). The ongoing Developments for identity standards on Ethereum can be viewed on [7].

The Identity Standard used in this thesis is ERC 725 v1, later moved into the key-manager standard ERC 734 [8] and is extended with claim holder functionality as described in ERC 735 [9].

In short, ERC 734 describes standard functions for uniquely identifiable proxy smart contracts that can be used by other accounts or even other smart contracts. The contracts can describe anything from humans to groups or devices, and act as an identity proxy on the blockchain. The described identity smart contract has a key-storage that controls the degree to which other parties can interact with the contract. E.g. upon creation of the contract, a management key is added to the key-storage based on the creator's account address to ensure certain functionality can only be executed by the smart contract creator. The contract also features an execute function to run arbitrary contract calls, which enables it to act as a proxy for whatever instance it is representing [10].

ERC 735 is an associated standard to ERC 734 that extends its functionality to add and remove claims. The idea is that similarly to the real world, where a government will vouch for an individual's claim of being a citizen of their country by issuing a passport, signed claims can be issued by a trusted authority on the blockchain and added to the identity-smart contracts. The claims will be hashed and signed by an identity contract of this trusted third party. In order to issue claims, this contract must first add a claim key to its key-storage, as it will be used to sign the claim and will be later checked by anyone wishing to check if the claim is valid [9].

2.4 Blockchain based PUFs for the IoT

There have been several research efforts to combine PUFs, blockchains and IoT thus far, most of which were conducted in the last 2-3 years, which highlights the novelty of this kind of approach to IoT identification. This section will cover some of the most noteworthy related works which combine these technologies. We will then also show some of the similarities the proposed designs have as well as the ways in which this thesis differs from the presented work.

Using PUFs to identify or authenticate IoT-connected devices was introduced in 2017 by [11] when they proposed an identity-based cryptosystem to enable secure authentication

and message encryption between IoT-connected devices using PUFs to generate the required keys. They argued that due to the resource-constrained nature of IoT-connected devices and their tendency to be deployed in potentially hostile environments, traditional cryptographic algorithms were not feasible for the IoT use case. PUFs being a naturally lightweight alternative to produce unique keys to use for encryption of messages and identification of devices promised a better solution. The proposed protocol first has devices in an IoT environment go through an enrolment procedure. Each data node, e.g. an IoT-connected device, saves its PUF-derived CRP data on a database hosted by a server node in the IoT network. Then when two devices in the network wish to communicate with each other, the server node first authenticates them by checking the newly generated PUF data matches with the CRPs stored in the database. If the devices pass the authentication, the server node will aid in creating public and private keys for the devices, as well as securely distributing them to the other device. From then on, the devices can communicate securely using their key to encrypt and decrypt each other's messages [11]. Note that this approach does not utilize blockchain technology. A similar approach that combines these principals with the Ethereum blockchain will be covered shortly.

Blockchains have been gaining in significance similarly to the IoT ever since the infamous Bitcoin whitepaper was published in 2008 [12]. Blockchains are essentially digital ledgers that are maintained simultaneously by a distributed network of computer nodes. Information gets added to the chain in blocks, that are linked to their preceding block through hashed data. The blockchain is kept in sync throughout the network using a consensus mechanism. Ethereum, for example, uses a consensus mechanism called Proof-of-Work. It requires the nodes wishing to add a block to the chain to compete on solving a mathematical puzzle. To add a block that does not abide by the consensus rules, a malicious party would need to control nodes more computationally powerful than 51% of the network. As this is practically impossible if the network is sufficiently large and distributed, blockchains are generally regarded as immutable. Many applications have been brought up besides cryptocurrencies to provide solutions that use these immutable ledgers to ensure trust between two parties, without the need for a trusted third party. Using this technology in IoT devices is therefore subject to much research, as it allows IoT-connected devices to transact with each other without going through third parties [13]. In 2018, [14] proposed a concept to uniquely link physical devices to logical addresses on the blockchain in order to better protect IoT transactions. The approach used Identification Random-Access Memory (IDRAM) as a replacement chip for the Random-Access Memory (RAM) built into devices to facilitate physical chip identification. It is not stated how exactly the IDRAM generates unique keys, but its functionality seems to be similar to PUFs. The unique keys are then used with public-key encryption for secure communication on the blockchain. This makes [14] one of the first approaches using blockchains to validate transactions in the IoT.

One of the earliest research papers to combine all three aspects was [15]. They proposed an authentication scheme in 2018 that utilized SRAM PUFs to generate digital fingerprints. Their proposed scheme saw both a global blockchain being used for tracing and authenticating devices, as well as a locally permissioned blockchain to authenticate devices within a local network periodically. The goal of the second blockchain is to address threats based on devices becoming compromised after some time of being registered within the IoT infrastructure. E.g. the scenario of a malicious party within the network

replacing an authentic device with a cloned imposter device. As part of their work [15] also proposed a secure communication protocol to facilitate the regular authentication of devices within the IoT infrastructure. The second blockchain, which interacts more on a local level, exceeds the scope of our project, but the approach taken with the global blockchain has several similarities with the work presented in this thesis. Although no specific blockchain is prescribed by the authors, it is mentioned that Ethereum is one of the possibilities that could be used for the global blockchain, as it provides decentralized data storage. Furthermore, the proposed authentication scheme has the manufacturers register the devices before selling them, comparable to the design proposed in this thesis. While their scheme also uses SRAM PUFs to generate a unique identifier for the device, the way the data is integrated into the blockchain, as well as the subsequent authentication process, differ from our design. Our design has the manufacturer verify the device's identity after it was sold and then issuing a KYD claim to the device's identity-contract according to the ERC 734 / 735 standard above. [15], however, uses the blockchain just as data storage to upload the CRP as well as a hash of the generated identifier. It is then up to interacting parties to authenticate the device by giving the device the same challenge and then checking if the identifier hashed with the same algorithm matches the one stored on the blockchain.

Also in 2018, [13] proposed a framework that combined PUFs with Ethereum smart contracts to ensure data provenance and data integrity in IoT environments. Their approach focuses more on the authentication of transactions within the network and thus focuses more on device-authentication, while the design for this thesis is more focused on the reliable identification of devices. These authentications work as follows in their proposed design. Authentications are handled by deployed smart contracts that act as trusted servers. Devices then publish the CRP from their PUF implementation to this contract to register themselves. Whenever authentication is necessary, due to transactions made by the device, the server contract initiates a verification process. The server encrypts a generated nonce using the PUF key for the device and sends it to the device. The device then proves its identity by decrypting the nonce and returning it alongside more hashed data. The hash is subsequently sent back to the server which verifies it and approves the transaction or declines it. In short, the proposed design uses smart contracts and PUFs to authenticate transactions based on traditional public-key cryptography techniques. Notice that this approach is executed entirely by the device owner himself, while our approach has a trusted third-party, the KYD platform, verify the device's identity. As will be mentioned later, however, the design proposed in [13] could complement our design nicely to add PUF-based authentications to the identification-focused design.

[16] went further with the concepts of using PUFs and blockchains in 2019 when they proposed a system to track separate components along the supply chain until the final assembly. The design is based on a smart contract that tracks each component while it moves through the supply chain, updating the current owners of the components on the way. The manufacturers of a component first use a PUF to register it in the smart contract. This way, each component that will be assembled into an IoT-connected device receives a unique id, based on its physical characteristics. The fully assembled device is then given an id based on the hash of all its component's ids. This approach is therefore focused on ensuring device integrity before the devices end up in the hands of users, to combat a growing problem of counterfeit parts being used during the assembly [16]. This

design goes further than our own design in that it uses PUFs to verify the identities of every component in an IoT-connected device. As these components will vary greatly, many different PUF implementations would have to be used, whereas our design only incorporates SRAM PUFs. Both designs are based on Ethereum smart contracts, however.

The design for the IoT identification platform proposed in this thesis is thoroughly covered in Chapter 3. As was mentioned above, it has some similarities with the related work covered in this section. Table 2.1 shows a broad overview of the covered designs, as well as the different ways PUFs and blockchains were incorporated. The work done in this thesis mainly sets itself apart in two areas. Firstly, this thesis implements a working prototype and therefore is more implementation focused than the more theoretical works covered above. Secondly, the chosen blockchain integration implements the ERC 734 / 735 Ethereum standard, which has a trusted third party on the blockchain issue KYD claims and includes an identity proxy in the form of a smart contract. As far as we are aware, this is the first implementation to combine PUF-based device identification with the issuing of ERC 734 / 735-based claims.

Table 2.1: Broad overview of the related works and the proposed design

Design	**PUF**	**Blockchain**	**Use-case Focus**
[11]	Not specified	None	Authentication & message encryption
[14]	None (IDRAM)	Not specified	Authentication & message encryption
[15]	SRAM PUF	Not specified	Periodic Identification
[13]	Not specified	Ethereum	Authentication & message encryption
[16]	Various	Ethereum	Device Integrity & Supply chain tracking of components
Proposed Design	SRAM PUF	Ethereum	Identification & ERC 734 / 735 proxy

Chapter 3

Design

This chapter first outlines the overall design architecture that was conceived for the implementation of this project. We then go over the specific design choices that were made, as well as the requirements that were set, for each component of the system. This chapter also includes sequence diagrams for the most critical processes to aid in the understanding of the data flow of the system.

3.1 Architecture Overview

During the course of this thesis, two differing designs for the general architecture were conceived. The following sections will first introduce these approaches and then discuss their advantages and disadvantages, as well as provide justification for the design that was ultimately chosen.

3.1.1 One-time Authentication Approach

This section describes the architecture for the case that we require the IoT devices to be registered and authenticated only once or rarely, as the process is not fully automated and requires the user's interaction. Figure 3.1 shows an overview of the proposed system with one-time authentication.

The system consists of two layers. The bottom layer, of primary concern to the device owner, is made up of his IoT-connected devices, which gather data using sensors or other means. In this model all communication with the IoT devices concerning the authentication process goes through the device owner.

The top layer of the architecture consists of the KYD platform, which has multiple components interacting with each other. The central component of the platform's architecture is the Python-based server, which provides the functionality to manage and authenticate devices. To showcase the integration of a KYC system into the general architecture the

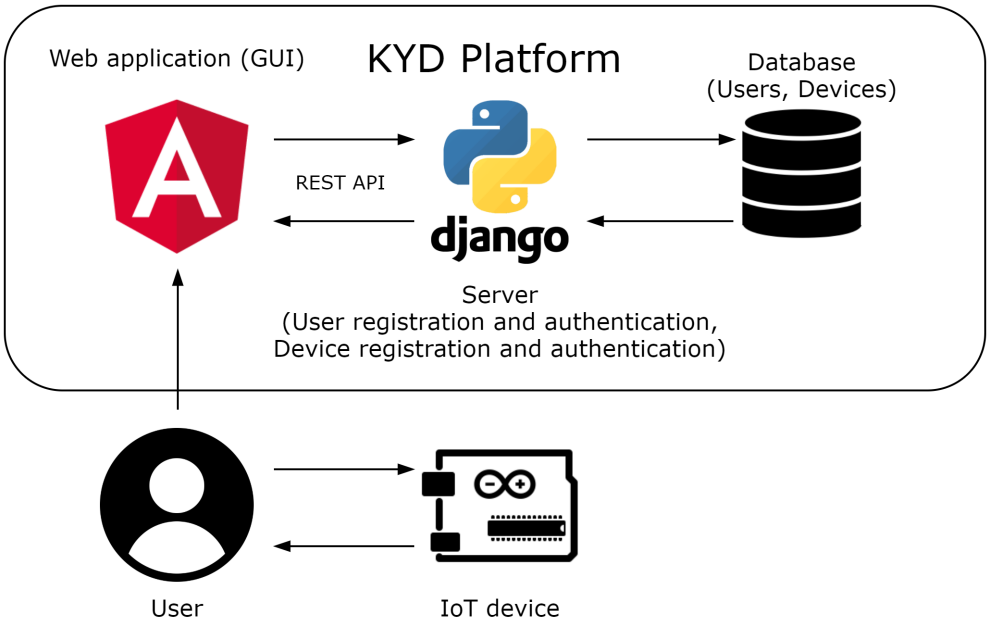

Figure 3.1: System architecture for the one-time authentication approach

server also supports the registration of users (more on this in Chapter 3.4.1). The server is connected to a database where all the necessary data for user and device identification is stored. The server mainly has two responsibilities. Firstly, it facilitates the registration of devices and users, as well as keeping track of these and providing basic configuration tools. Secondly, it contains the authentication logic for the devices. The last component of the KYD platform is the Angular-based web application. It provides a graphical user interface (GUI) for the device owners to register themselves and their devices.

3.1.2 Periodical Authentication Approach

This section describes the architecture for the case that we require the IoT devices to be authenticated regularly. As the previously discussed approach requires too much user interaction, some adjustments are necessary for this design in order to automate the authentication process. Figure 3.2 shows an overview of the proposed system with periodical authentication.

Many IoT connected devices cannot turn themselves off and subsequently power back up again on their own, including the Arduino Mega2560 primarily used in our testbed. Since power-cycling is a requirement for our PUF implementation, we require a controller device that acts as an intermediary between then KYD platform and the IoT-connected devices, as a replacement for the device owner. It is responsible for both communicating with the KYD platform and power cycling the IoT devices. This device has to be able to connect to the KYD platform's server in a secure and automated way, so a secure shell (SSH) connection will have to be set up for remote command execution from the KYD platform.

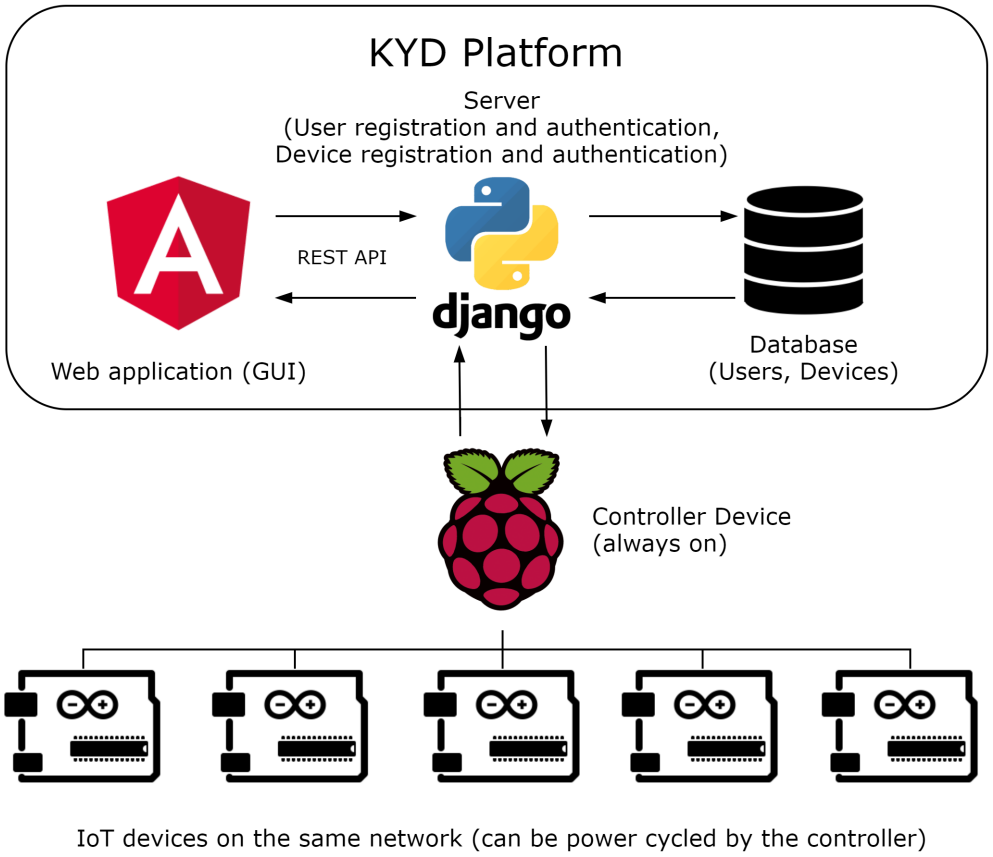

Figure 3.2: System architecture for the periodical authentication approach

3.1.3 Comparison

Both the periodical and the one-time authentication designs have their strength and weaknesses but differ mostly in the use case they are designed for. The one-time authentication approach is focused more on device-identification, where the KYD platform acts like a certification service, that issues a certificate of authenticity after verifying a device's identity. In the periodical authentication approach, however, the KYD platform acts more like an oversight entity that continually makes sure the registered devices are still whom they claimed to be. This makes the periodical approach more focused on continuous device authentication than the one-time approach.

The periodical authentication design has the benefit of being more automated, requiring less active participation of the device owner after the set-up process. Furthermore, it is the more secure approach as it requires the devices to be re-verified periodically and locks out the user better from the authentication process.

The one-time authentication design has the advantage, that it is a more lightweight solution, which makes it easier to implement and less complicated for the device owner, as minimal set-up is required on his part. The design is also more flexible and expandable, as it is less tied down by the additional restrictions that accompany the utilization of a separate controller device and the need for a secure automated connection to the server.

Device support can be expanded better because of this as well, as different PUF solutions for IoT devices that do not have built-in SRAM could be added on the server's side rather effortlessly. Furthermore, the device owner is unlikely to want to give access and control over his devices to the KYD platform all the time, due to privacy concerns and the probability that it will disrupt his usage of the devices. SRAM PUFs are also considered a weak PUF, and the authentication process is not tamper-proof, especially in the case where the user himself is malicious. The periodical authentication approach would, therefore, be more suited to a stronger PUF design, with more emphasis being laid on security aspects for the entire design, which exceeds the scope of this thesis. Finally, we are using the ERC 735 standard for the issuing of KYD claims on the Ethereum blockchain. This standard does not currently accommodate the expiration of claims or any other mechanism to force periodic re-authentications. We will discuss this topic in more depth in Chapter 5.3. However, to stay in line with the current standards, it makes little sense to introduce periodical authentications on the KYD platform without a way to incorporate this within the blockchain.

Due to the advantages of the one-time authentication design laid out above and the better compatibility with the chosen integration of blockchain technology, this design was, therefore, chosen for the implementation of the KYD system. The emphasis of this thesis is the identification of devices, making the more rigorous periodical authentication model excessive. However, the devices could still utilize the same PUF implementation to generate their private key on the fly and use it for traditional asymmetric cryptography schemes, without having to store any keys in unreliable and insecure non-volatile storage.

3.2 IoT-connected Devices

In the context of our proposed system, the crucial functionality that the IoT-connected devices must provide is the reading of its SRAM cells and the transfer of that data to either the device owner or the controller device. The goal was to keep the system as open as possible for different kinds of microcontroller boards. However, due to the design of our PUF relying on SRAM readings, only boards that have integrated SRAM chips will be compatible with this system. For our testbed, we used the Arduino Mega2560 as the primary microcontroller board for our testing. It lacks internet connectivity without the appropriate add-ons, so it cannot be described as an IoT-connected device in this configuration. However, it features 8 KB of built-in SRAM and can be extended with an Ethernet- or Wifi-shield to allow for internet connectivity. The device has to be connected to the user's computer or the controller device via USB, as this gives us the ability to power cycle the boards by enabling and disabling the USB ports or unplugging the device. For a true IoT setting and a continuous authentication approach, a different solution for power cycling would have to be explored, or each device would need a separate controller device. In Table 3.1 the high-level requirements for the data collection devices are listed. The table also has a column to specify the priority of the requirement, where low, medium, high and critical signify how dependent the system is on the successful implementation of this requirement. Furthermore, the table includes a column that indicates the likelihood that this requirement will not be met, from low to high. Consequently, requirements

with both high risk and high priority received the highest degree of attention during the implementation of the system. All requirements tables from this point forth will be using this structure.

Table 3.1: Requirements for the data collection devices

Requirement	**Priority**	**Risk**
The device must feature a built-in SRAM chip, that cannot be easily transferred to a different device.	critical	low
The device must feature at least 32 B of free SRAM that remains uninitialized through the start-up sequence.	critical	low
The functionality to read the start-up values of the SRAM must be available.	critical	medium
The device must be able to connect to the controller device and transfer the SRAM data.	critical	low
The transfer of the data must be secure.	medium	medium
The SRAM data must be unalterable by the device owner.	low	high

3.3 Controller Device

The component described here is only needed in the periodical authentication approach and is as such irrelevant for the final implementation. The information provided does, however, help with the understanding of the different designs. The controller device acts as a communication middle layer between the IoT-connected devices and the KYD platform. For our testbed, we initially choose to use a Raspberry Pi 4 for this job, because it features built-in Wi-Fi and Ethernet to connect to the KYD platform and has 4 USB ports. We can utilize these to connect to the IoT devices and power cycle them by enabling and disabling the ports. Raspbian, the operating system typically used with Raspberry Pis, is also based on Linux and offers out of the box support for Python. This could be useful since it allows us to run the fuzzy extractor code directly on the Raspberry Pi, to generate unique keys for the IoT-connected devices. The device is set-up for remote command execution from the KYD platform via an SSH-connection to limit any potential interference of the device owner as much as possible during the authentication process. Table 3.2 lists the high-level requirements for the controller device.

Table 3.2: Requirements for the controller device

Requirement	**Priority**	**Risk**
The device must provide secure connection capabilities to the IoT devices.	critical	low
The device must provide the capability to connect to the KYD platform via SSH and allow remote command executions.	critical	medium
The device be able to control the power delivery to the IoT-connected devices, if the USB approach to power cycling is used.	high	low
The device must be always-on and ready to process incoming requests continuously.	high	low
The device must be able to run Python commands.	low	low

3.4 KYD Server and Database

The KYD server is the central processing component in the system. We decided to write the server code in Python, using the Django framework [17]. We chose Python, as this allows us to utilize a pre-existing fuzzy extractor library to generate the unique key for the IoT-connected devices. Using the Django framework allows us to efficiently manage users on the platform, the database integration, as well as providing a representational state transfer (REST) application programming interface (API) [18]. We used SQLite for our database architecture, though any database compatible with Django would work. Table 3.3 lists the high-level requirements for the KYD server and the connected database.

Table 3.3: Requirements for the KYD server and its database

Requirement	**Priority**	**Risk**
The server must provide functionality to register an IoT-connected device.	critical	low
The server must be able to generate a unique key from provided SRAM readings.	critical	medium
The server must be able to reproduce the unique key from a similar SRAM reading.	critical	medium
The server must be able to verify devices autonomously by checking if the original key and the reproduced key match.	critical	medium
The database must include a table to store device data.	critical	low

3.4.1 User Handling

During the course of this thesis, two differing designs for the handling of users on the KYD platform were conceived. The following sections will first introduce these approaches and then discuss their advantages and disadvantages, as well as provide justification for the design that was ultimately chosen.

Custom Implementation using JSON Web Tokens (JWTs)

Django features easy integration of JavaScript Object Notation (JSON) Web Tokens (JWTs) into its user authentication system [19]. For this approach, we generate JWTs on the server for each user that is signing in and distribute them through the REST API to the web application. Subsequently, a valid JWT is required for all requests to the server other than user creation and sign in.

To clarify the sequence of events during the user registration process with the JWT approach, Figure 3.3 shows a sequence diagram of this process. The web application stores the user's inputs in a form which is continuously validated. The data from this validated form is then sent from the web application to the KYD server through the REST API. The server stores this data in the database and generates the access and refresh tokens needed for JWT authentication. These tokens are then returned to the web application as part of the response body of the initial request. The web application stores these tokens in local storage as long as the user does not sign out. Before each request to the server, the web application checks if the access token is not expired, as it is only valid for 15 minutes. If the access token is expired, the refresh token which is valid for 24 hours is sent to the KYD server as part of a token refresh request. The server uses the refresh token to check if the user is valid and creates a new access token. This token is returned to the web application which will then store it in local storage once more. The access token is attached to the header of all requests to the server requiring authentication. E.g. in the case where the web application wants to display a list of devices registered to the user, a request is sent to the server to retrieve the list with the access token attached. The server checks the validity of the access token and only retrieves the list of devices from the database if it is valid. The server then either returns the device list to the web application or throws an authentication error.

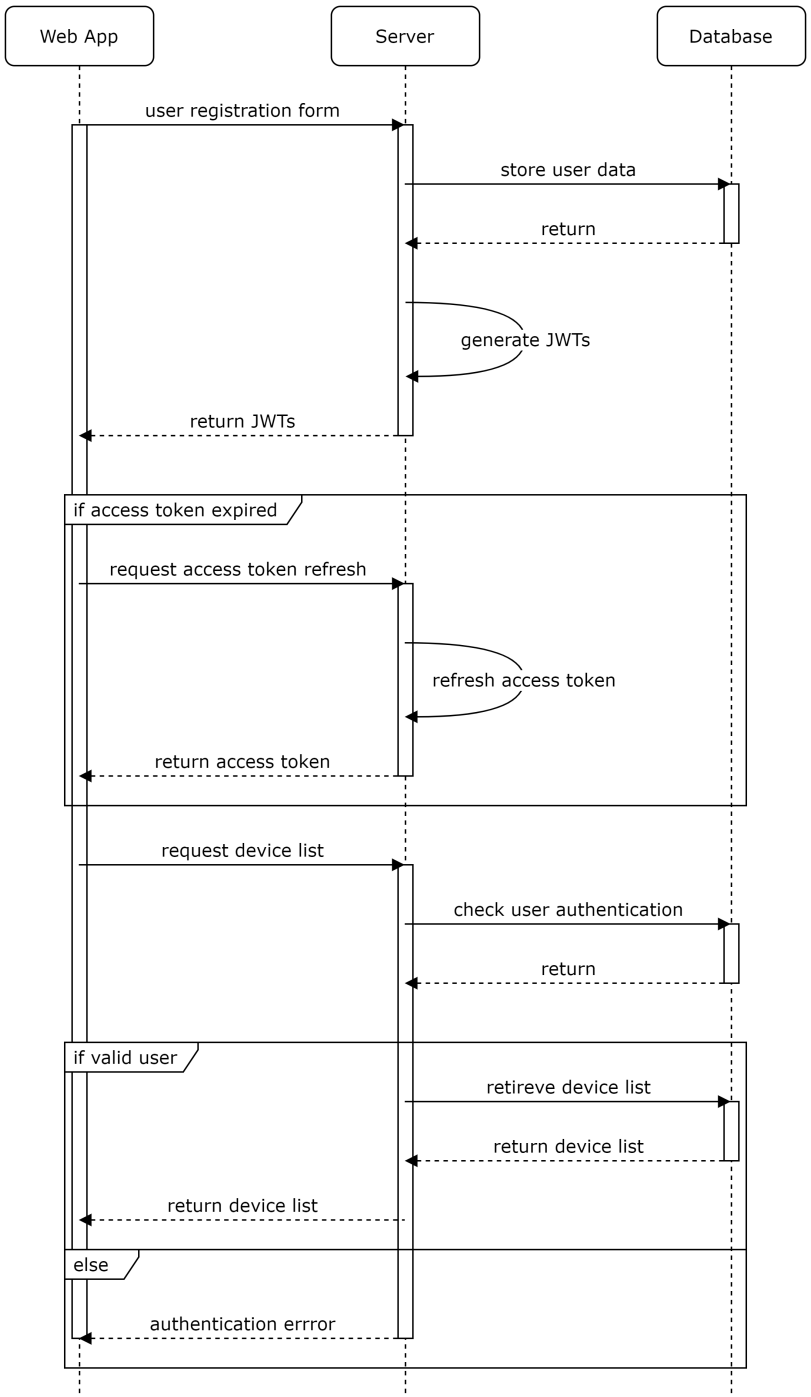

Figure 3.3: Sequence diagram of the user registration process and a subsequent request to retrieve all registered devices using JWTs to handle users

For this approach there are a few extra requirements for the KYD server. These are listed in Table 3.4.

Table 3.4: Requirements for handling users with JWTs

Requirement	**Priority**	**Risk**
The server must provide functionality to register a user. This process must also generate and return a JWT.	critical	medium
The server must provide functionality to sign in a user, by generating and returning JWTs if the provided email address and password match the values stored in the database.	critical	low
The server must provide functionality to verify JWTs and deny or approve requests based on the authenticity of the token.	critical	low
The server must provide functionality to refresh access tokens when a valid refresh token is provided.	critical	low
The database must include a table to store user data.	critical	low

MetaMask Implementation

Another approach is handling users through the MetaMask browser extension [20]. MetaMask allows users to run decentralized Applications (dApps) on the Ethereum network and is a requirement for the blockchain integration of this project. It further acts as an Ethereum wallet, which means it is linked to an Ethereum account, which is also a unique identifier on the network. Anyone wishing to interact with the Ethereum network needs one of these accounts with some Ethereum stored in them. Because the use of an Ethereum account is required for the blockchain integration of the KYD platform and it uniquely identifies the user owning the account, we can use the account address to handle users on the platform.

To clarify the sequence of events during the user registration process with the MetaMask approach, Figure 3.4 shows a sequence diagram of this process. Firstly, the user will have to sign in to his MetaMask account so that the web application can extract the Ethereum account address. This address will then be forwarded to the KYD server to check if there is any user data stored on the server. If no user data can be found the GUI will redirect the user to a registration page. The web application subsequently stores the user's inputs in a form which is continuously validated. The data from this form is then sent from the web application to the KYD server through the REST API. The server stores this data in the database. We collect this additional data on the user to issue KYC claims alongside the KYD claims later during the blockchain integration (more on this in Chapter 3.6). In the case where the web application wants to display a list of devices registered to the user, for instance, a request is sent to the server to retrieve the list with the account address included in the query parameters. The server retrieves the list of devices associated with this user from the database and returns it to the web application. The list will be empty if no devices were found.

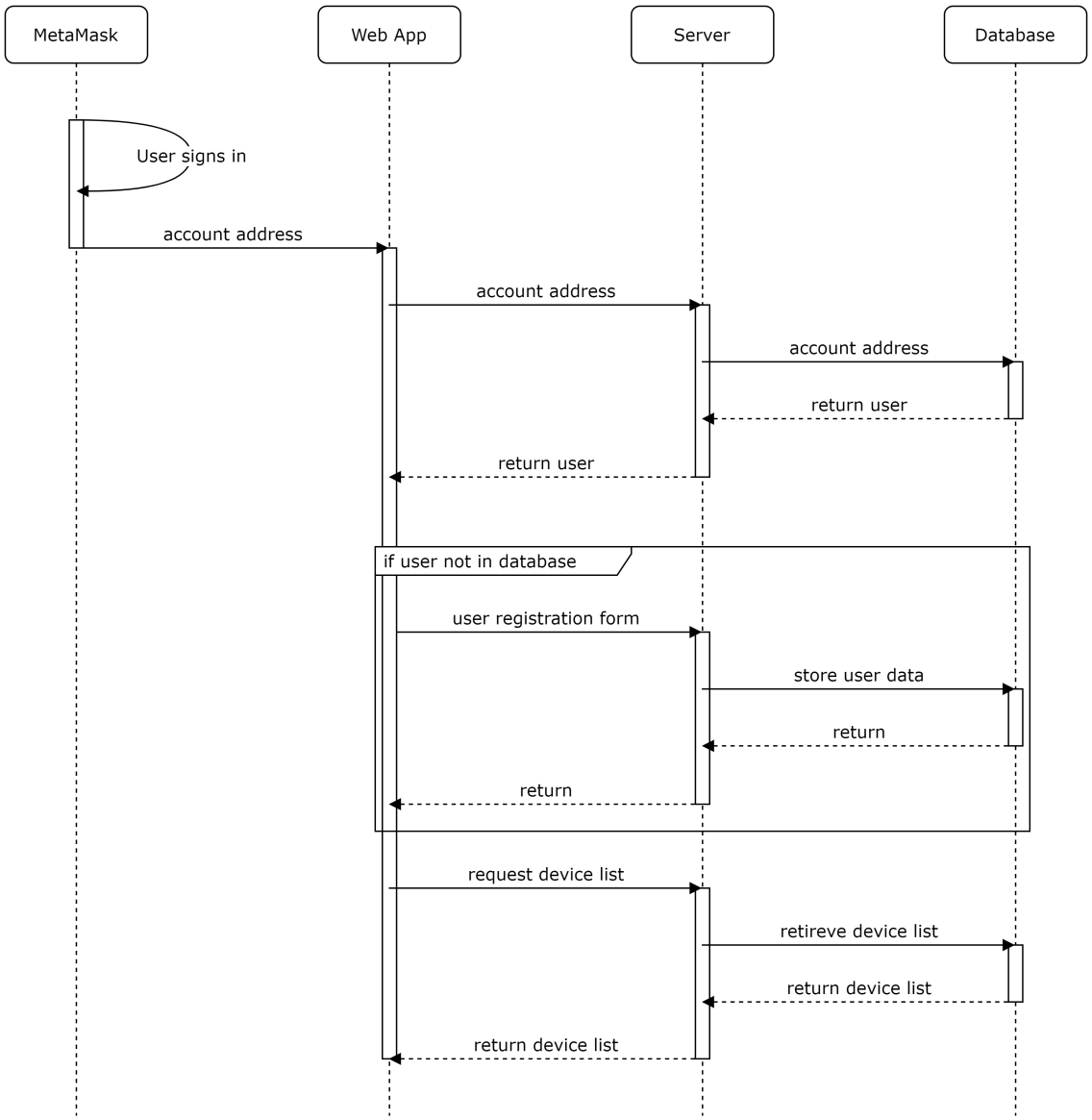

Figure 3.4: Sequence diagram of the user registration process and a subsequent request to retrieve all registered devices using MetaMask to handle users

For this approach there are a few extra requirements for the KYD server. These are listed in Table 3.5.

Table 3.5: Requirements for handling users with MetaMask

Requirement	**Priority**	**Risk**
The server must provide functionality to register a user.	critical	medium
The database must include a table to store user data.	critical	low

Comparison

The most crucial difference between these two user handling options is their compatibility with the Ethereum blockchain. While the JWT approach is perfectly suitable for traditional web applications, it makes sense to use the MetaMask approach if interacting with the Ethereum network is required. Since the integration of MetaMask and Ethereum accounts is necessary in that case, it is easier to also use it for handling users in the application. It requires less work to implement this approach, as many features like handling login data and resetting passwords are outsourced to the MetaMask extension. It is also a more user-friendly approach, as only one user account has to be set up to use the KYD platform.

As integration with the Ethereum network is an integral part of this thesis, and due to the advantages outlined above, the MetaMask approach was chosen to be used in the final version. However, since the blockchain integration was one of the last aspects to be implemented, the JWT approach was developed and used for a significant portion of this project.

3.4.2 Device Verification

The process of verifying a device on the server is comprised of two stages. The KYD platform registers the device themselves first before the user gets access to the device. Subsequently, the user can register the device on the KYD platform when he wishes.

To further illustrate, let us consider a realistic scenario of how this might work. We will assume that the KYD platform is operated by the manufacturer of the devices. As he has access to the devices before they are sold, it makes the most sense to trust the manufacturer with verifying the identities of the devices being registered by the device owner. Theoretically, the platform could also be operated by a third-party organization focused solely on device verification. In which case the devices would have to be sent to this third party before verification can occur, as physical access to the device is necessary for the registration in the database.

To clarify the sequence of events during the first stage, where the manufacturer registers the device on its database, Figure 3.5 shows a sequence diagram of this process. After manufacturing is completed, the device is connected to a computer in order to provide power and access the serial bus of the device. The required start-up values for our PUF implementation are then extracted from the device's SRAM chip. This data is subsequently forwarded to the KYD server, alongside other metadata, through a web application. On the server, a fuzzy extractor is then used to generate a unique key for the device based on the SRAM values, as well as the helper that is necessary for the reproduction function of fuzzy extractors. At this point, a non-secret unique identifier is also generated to keep track of the device on the database. All this data is then stored on the database, at which point the device will be ready to be sold. The unique identifier will have to be included in the shipment of the device somehow, as the device buyer will need it to register and verify his device on the KYD platform.

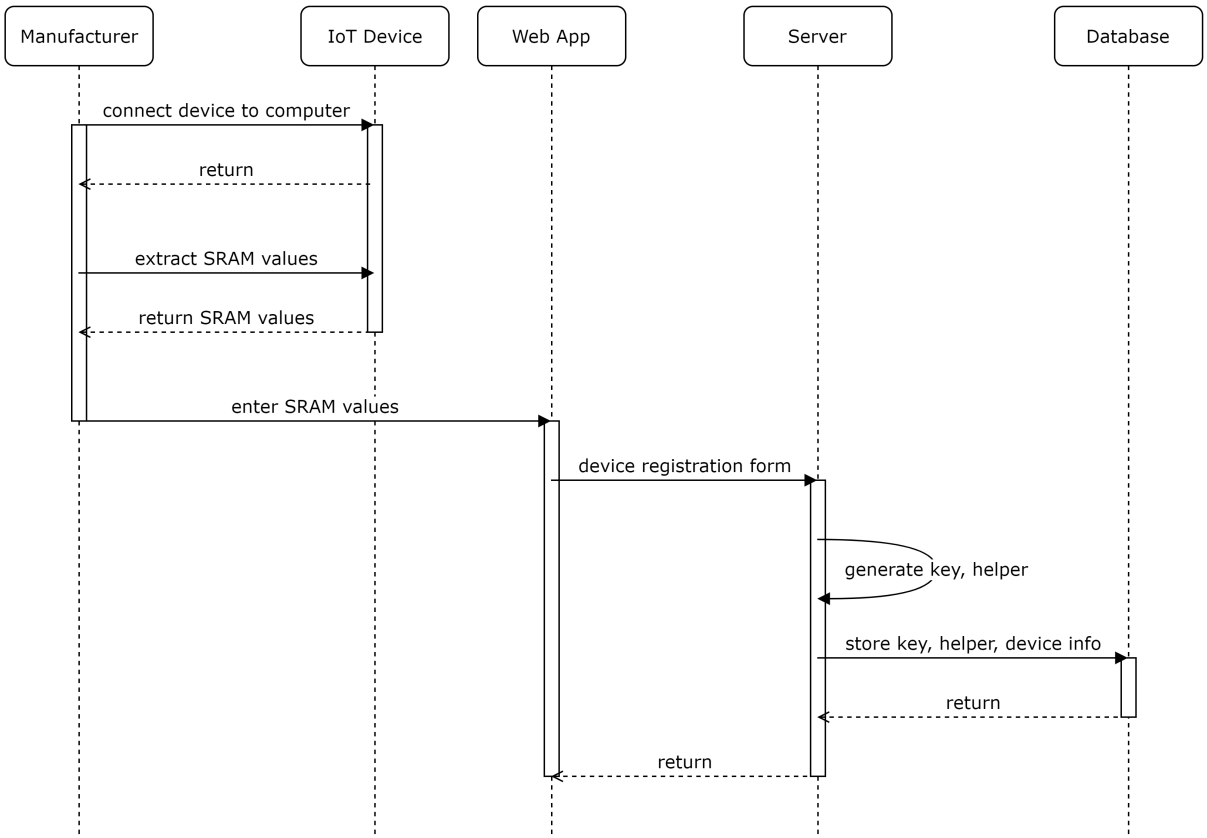

Figure 3.5: Sequence diagram of the device registration process

The next step would be for the device owner to register his device on the KYD platform using the unique identifier he received with the purchase of his device. Once successfully registered, the user can proceed to verify his device. To clarify the sequence of events during this second stage, Figure 3.6 shows a sequence diagram of this process. Similarly to the registration process of the manufacturer, the user first connects the device to his computer and extracts the required PUF data. He then enters this data in the GUI of the web application and sends it to the server. The server first retrieves the stored key and helper from its database. The helper is then used alongside the PUF data in a fuzzy extractor to reproduce the device's key. This reproduced key is then compared to the key stored in the database by the manufacturer. If the keys match precisely the device can be considered verified, and a corresponding message is returned to the web application. If the fuzzy extractor was unable to reproduce the same key from the PUF data, the device is considered to be invalid as the physical characteristics of the SRAM chip likely do not match the ones recorded by the manufacturer before the sale.

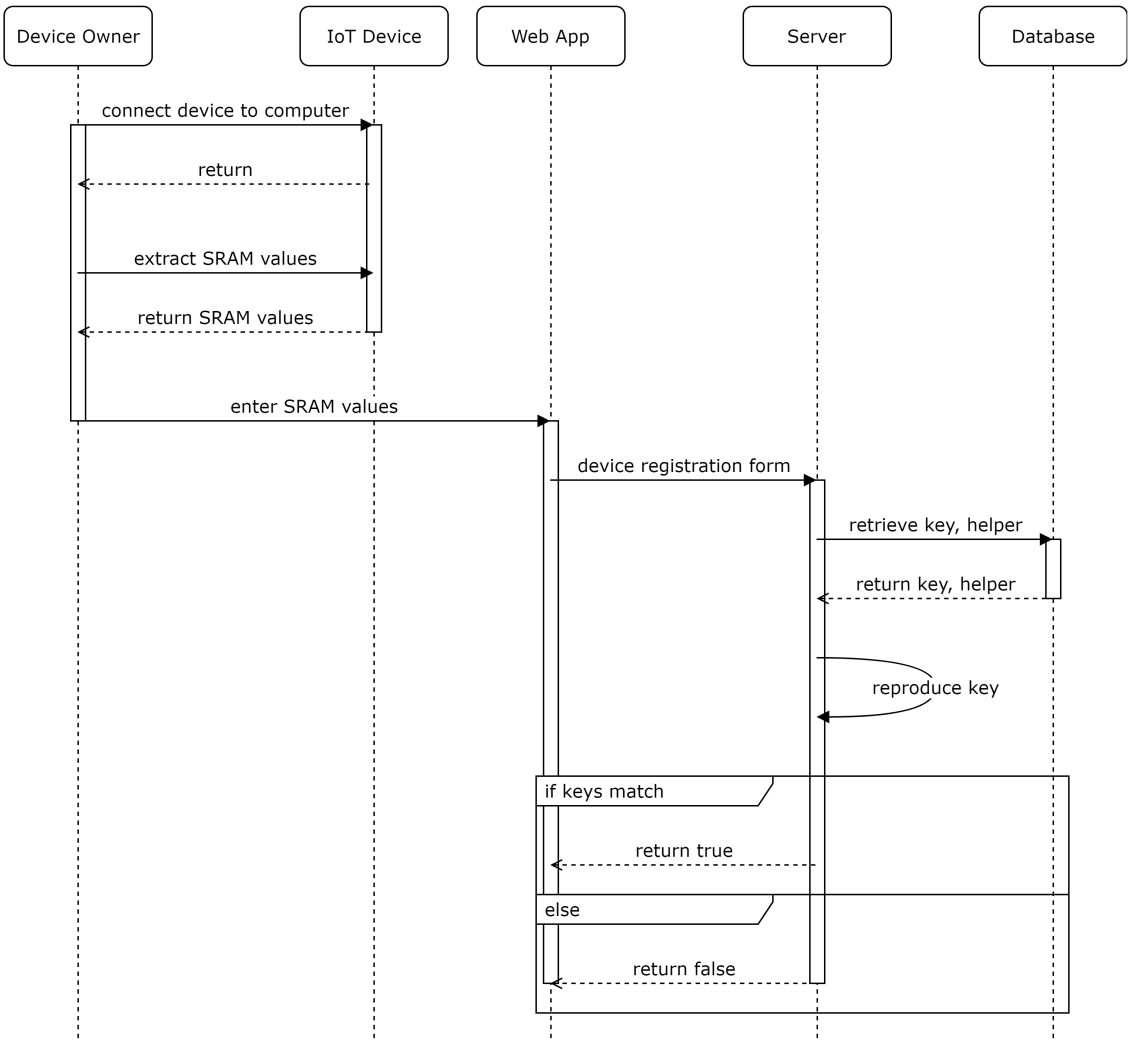

Figure 3.6: Sequence diagram of the device verification process

3.4.3 REST API

The server communicates with the web application, or other future clients and services, through a REST API. The possible requests that can be made to the server are listed in the REST API specification in Table 3.6. The table also shows the required parameters for the request body of post and patch requests, as well as any return values the requests will produce.

Table 3.6: REST API specification for the KYD server

Mapping	**Method**	**Parameters**	**Return**	**Description**
/user/:id	GET		account address, contract address, first name, last name, birthdate, email, mobile number	Retrieves the user corresponding to the id from the database.
/users/register	POST	account address, contract address, first name, last name, birthdate, street, postal code, city, country, email, mobile number		Creates a user in the database.
/device/:id	GET		id, model, name, contract address	Retrieves the device corresponding to the id from the database.
/devices/user/:id	GET		List<devices>: id, model, name, contract address	Retrieves all the devices associated with the account address of a user.
/devices/create	POST	model, PUF data		Creates a device in the database.
/devices/register	PATCH	id, name, account address, contract address		Updates the device's data and associates it with the user's account address.
/devices/verify	POST	id, PUF data	true/false	Verifies the device.

3.5 KYD Web Application

The KYD Web Application acts as a GUI for the device owners to register themselves and their devices. We decided to write the application in Angular since this is the framework we were most familiar with. Angular offers many useful properties for a project like this, like built-in form handling and validation, support for routing and a HyperText Transfer Protocol (HTTP) client to interface with the REST API. Table 3.7 lists the high-level requirements for the KYD Web Application.

Table 3.7: Requirements for the KYD Web Application

Requirement	**Priority**	**Risk**
The web application must be able to interface with MetaMask and retrieve the active account address.	critical	low
The web application must provide a GUI to register a user. All the required fields listed in the REST API specification must be included in the form.	critical	low
The user registration form must be validated to ensure no unwanted inputs can be entered and no subsequent server errors will occur.	high	medium
The web application must include a page where the active user and all the user's owned devices are listed, including all the details that are returned by the server on the relevant GET requests.	critical	medium
The web application must provide a GUI to register a device. All the required fields listed in the REST API specification must be included in the form.	critical	low
The device registration form must be validated to ensure no unwanted inputs can be entered and no subsequent server errors will occur.	high	medium
The web application must include a page where instructions are given on all the steps required for the verification of devices outside of the KYD platform.	critical	low
The web application must provide a GUI to verify a device. All the required fields listed in the REST API specification must be included in the form.	critical	low
The device verification form must be validated to ensure no unwanted inputs can be entered and no subsequent server errors will occur.	high	medium

3.6 Blockchain Integration

The desired end goal for this project is to have a verified presence of an IoT device on the Ethereum blockchain. The idea is that this presence can then interact with other accounts and smart contracts on the network and that those parties can trust the device's identity. The device should further be linked to a device owner that also has a verified presence on the blockchain. This will open up many possibilities, like the selling of data collected by an IoT device over the Ethereum network, with the buyer being able to verify the identity of the data supplying device as well as its owner. To achieve this, we decided to follow the ERC 734 combined with the ERC 735 standard of handling identities on the Ethereum blockchain.

The standard is based on having proxy smart contracts deployed on the blockchain to represent the users, devices, groups, and so forth. A trusted party can then verify the identities of these instances. To register a successful identification on the blockchain, the trusted third party's proxy contract will issue a signed claim stating the verification of a specific identity. This claim can then be added to the corresponding proxy contract of the device, which consequently clearly states to any account or contract interacting with the proxy, that its identity was verified by the trusted third party. This claim system can further be used for many types of claims exceeding identification.

There are three parties to consider from the creation of a user to the successful verification of a device by the KYD platform, and each of them will have their separate identity contract. These are the device owner, the device that we want to get verified, and the KYD platform. We will assume the KYD platform already has a deployed identity contract with the relevant keys added. The relevant keys would be a management key, derived from the account address that deployed the contract, as per ERC 734 standard, and a claim key to sign claims.

Figure 3.7 shows a sequence diagram of the steps required to get all contracts deployed and set-up, including getting the user verified. Firstly, the user deploys an identity smart contract to the Ethereum blockchain. The required management key is automatically added upon the creation of the contract. Next, we want this proxy contract of the user to have his identity verified before adding any devices. We added simple user registration to the KYD platform so that we can justify having the KYD platform also signing a KYC claim. This was done for demonstration purposes mostly, and a far more sophisticated approach to identity verification is recommended. In practice, a different third party focused solely on the issuing of KYC claims could handle this user verification and let the KYD platform focus on KYD claims. The user sends the required data to get verified to the KYC service provider, which will then process the verification request. If successful, it will have its identity contract sign a KYC claim. This claim then gets added to the user's deployed identity contract, at which point the user's identity will be considered verified.

If the user now chooses to add some devices, he must first add a claim key to his identity contract, as this will be required to sign ownership claims for his devices, effectively linking the devices to the device owner. Subsequently, the user deploys an identity smart contract for his device, signs an ownership claim and adds it to the newly created proxy contract.

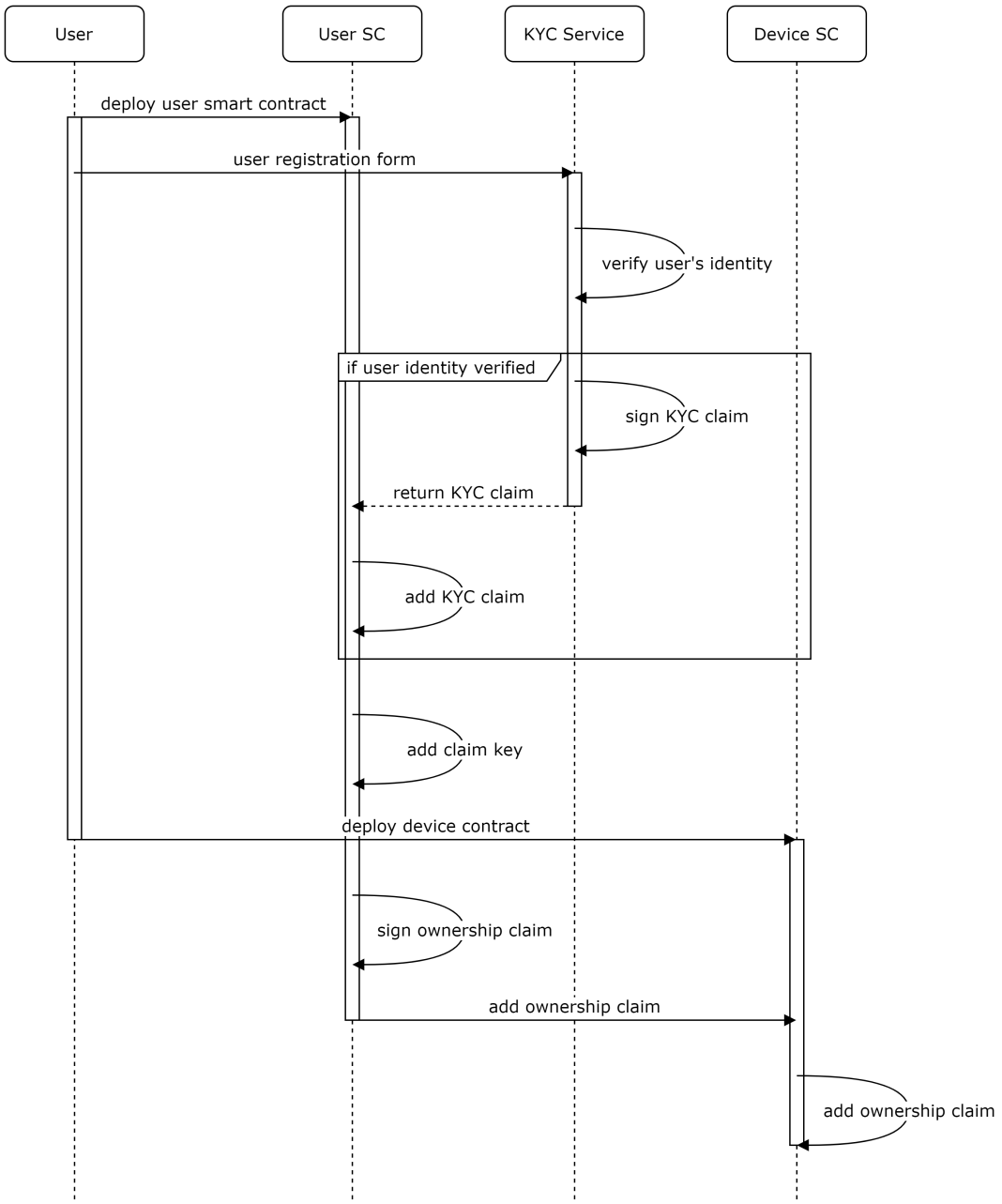

Figure 3.7: Sequence diagram showing the deployment and set-up of identity contracts for a user and one of his devices

At this point, we have successfully added proxy instances of the user and his device to the blockchain, as well as verified the user's identity.

To reach our set end goal, we must also verify the identities of the added devices. Figure 3.8 outlines the steps for this process. In order to be verified by the KYD platform, the device most first go through the verification process that was detailed in Chapter 3.4.2. If the KYD platform deems the device to be valid, it has its proxy smart contract sign a KYD claim. As with the other claims, the device's identity contract can then add this claim and show that it was verified by the KYD platform.

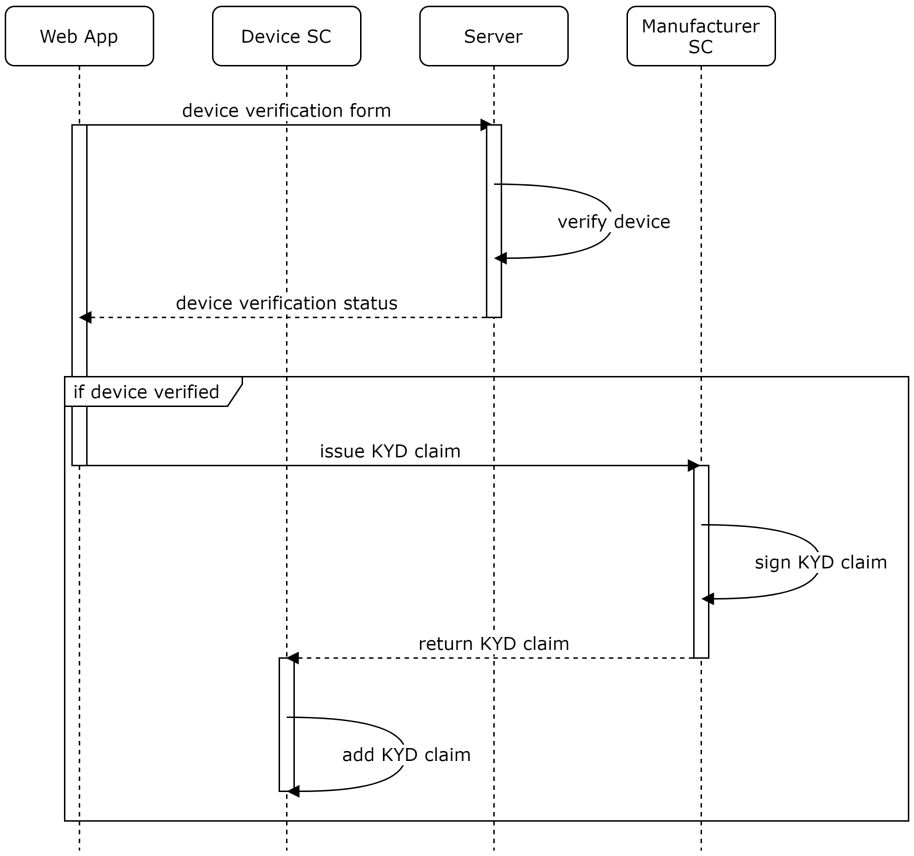

Figure 3.8: Sequence diagram showing the issuing of a KYD claim

If other contracts or accounts on the Ethereum blockchain want to make sure the identity of this device is verified, they can simply check if its identity contract includes a signed KYD claim. The claim also provides information on the issuer and other information to follow up on the claim if necessary.

Interaction with the Ethereum blockchain is handled by the KYD Web Application. It uses the provider injected by the MetaMask extension to interact with the blockchain on the user's behalf, as well as having its separate identity contract for the KYD platform to sign the KYC and KYD claims. The blockchain integration discussed in this chapter, therefore, leads to additional requirements for the web application, which are listed in Table 3.8.

Table 3.8: Requirements for the blockchain integration within the web application

Requirement	**Priority**	**Risk**
The web application must have a deployed identity smart contract representing the KYD platform.	critical	low
The web application must be able to interact with the Ethereum blockchain on behalf of the user using the MetaMask provider.	critical	low
The web application must be able to deploy an identity contract to the blockchain.	critical	low
The web application must be able to add keys to the deployed identity contracts.	critical	low
The web application must be able to add claims to the deployed identity contracts.	critical	low
The web application must be able to retrieve information about keys and claims from the deployed identity contracts.	high	low
The web application must be able to sign claims using the claim key of an identity contract.	critical	medium

Chapter 4

Implementation

This chapter will cover the details for the implementation of the design described in Chapter 3. First, the handling of users on the KYD platform is covered, including their registration process. Then we will look at the registration and subsequent verification of devices while outlining all relevant implementation details. This chapter covers details on all the components of the architecture, like the GUI, smart contracts and the server code.

4.1 User Handling

This section outlines how users are detected with MetaMask and maintained on the KYD server. It further shows the process of registering a new user, the deployment of a corresponding smart contract as well as the issuing of a KYC claim by the KYD platform.

4.1.1 MetaMask Integration and Sign in

Users on the KYD platform are primarily handled through their Ethereum account addresses using the MetaMask browser extension. MetaMask automatically injects an instance of Web3 when the user is logged in. Web3.js is the most popular JavaScript library to interact with the Ethereum blockchain. Using this Web3 instance, we can deploy smart contracts, interact with contracts, and retrieve the active account address selected in the MetaMask extension.

The first thing that happens when the KYD web application is opened, therefore, is checking if a valid instance of Web3 was injected into the web app and that an account address can be found. This is achieved through an Angular route guard placed on the main Uniform Resource Locator (URL) of the platform, which automatically redirects the user to the login page if the user's account address can not be retrieved.

The sign-in page automatically launches a MetaMask pop-up where the user can sign in and subsequently approve account access for the KYD platform, as shown in Figure 4.1. If no MetaMask is detected a download link is displayed on the sign-in page.

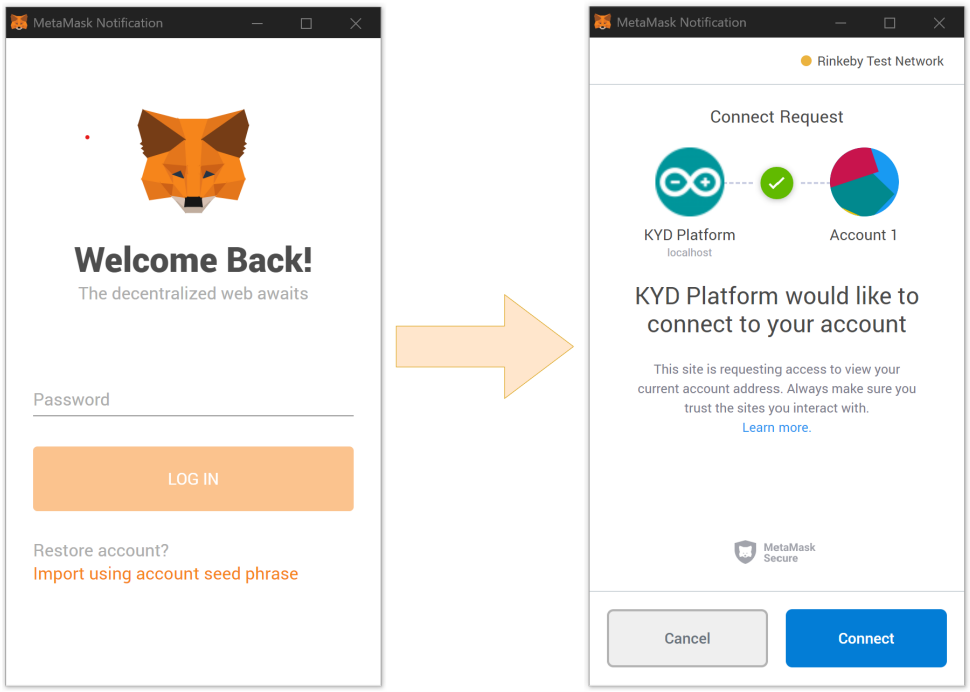

Figure 4.1: Sign-in prompt and connection request through MetaMask on the web app

If the user signs in for the first time, his account address will not yet be registered on the server's database. In this case, the user gets redirected to the user registration page by another route guard. The data requested by the user here is for demonstration purposes in order to justify issuing KYC claims from the KYD platform. As discussed earlier, the issuing of these claims could be carried out by a separate KYC service provider in which case this extra step would be redundant. The KYD platform only requires an account address to operate. However, gathering more information allows us to display more details on the user on the main page as well.

If the user is already registered on the KYD platform, he will be redirected to the main page instead (See Chapter 4.4 for more on this).

4.1.2 User Registration

The user registration process includes interactions with the web application, the blockchain and the KYD server so we will look at the included steps in some detail in this section.

The first step requires the user to fill in the Angular reactive form displayed on the user registration page. It includes all the fields necessary for the user creation request to the KYD server, outlined in the REST specification in Table 3.6. The GUI of the user registration component is shown in Figure 4.2.

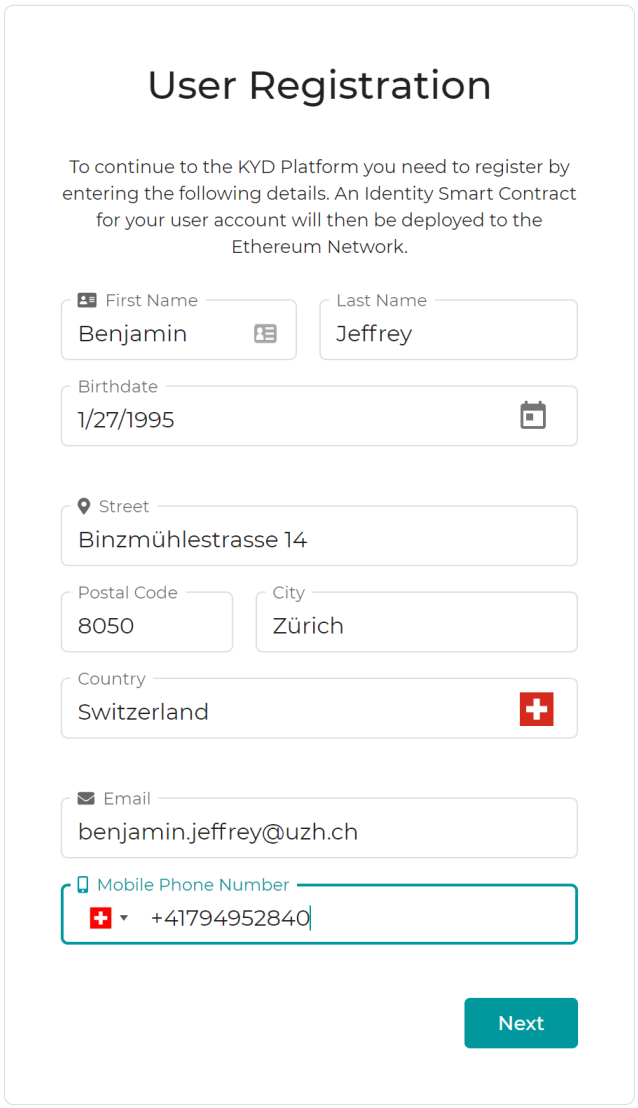

Figure 4.2: GUI for the user registration form

To avoid any server errors that could come up when trying to save the form data on the database, we must make sure the form is thoroughly validated. Proceeding with the *Next* button is therefore generally disabled unless the displayed form is valid. For the user registration component, all fields are required. Further validation requirements for each form field are outlined in Table 4.1. The *Min* and *Max* columns specify the minimum and maximum number of characters that can be saved to the database for each field, while the *Special* column specifies any further requirements for the validation.

Table 4.1: User registration form validation

Form field	**Special**	**Min**	**Max**
First Name			30
Last Name			150
Birthdate	Valid date format, User must be at least 18 years old		
Street			150
Postal Code			12
City			150
Country	Valid ISO 3166-1 alpha-3 country code	3	3
Email	Valid email format		150
Mobile Phone Number	Valid phone number format for the selected country		20

The next step in the user registration process is deploying a proxy smart contract representing the user on the Ethereum blockchain. To do this, the form data gathered in the previous step is passed to the *register* function of the `user.service` in the web application. There the injected Web3 instance is used to deploy a plain version of the `Identity.sol` contract to the blockchain. The code for this step is shown in Listing 4.1. The second block of code is the adding of a claim key to the newly created contract. The key is derived from the contract address using a hash function. We require this key for the issuing of ownership claims to the associated devices later.

```
1    // Deploys an identity smart contract from accounts[0].
2    const result = await new this.web3.eth.Contract(abi)
3      .deploy({ data: bytecode })
4      .send({ from: accounts[0], gas: 3000000 });
5
6    // Adds a claim key to the deployed SC.
7    await result.methods.addKey(
8      this.web3.utils.keccak256(result.options.address), 3, 1
9    ).send({ from: accounts[0], gas: 3000000 });
```

Listing 4.1: Smart contract deployment for a user within the user service of the web app

These interactions with the blockchain require the approval from the account owner, as they do come with a transaction cost. MetaMask will automatically show a pop-up screen asking the user if he wishes to approve the transactions, as is shown in Figure 4.3. These approval requests will pop up on all interactions with the blockchain that come with a transaction cost. Meaning all transactions that change data on the blockchain, as reading data is generally free.

As these transactions need to be mined first and are added to the blockchain in blocks, it takes some time for these requests to go through. Specifically, the longer the block time of the blockchain, meaning the average time between blocks being added, the longer the user will have to wait. On Ethereum the block time lies around 20 seconds.

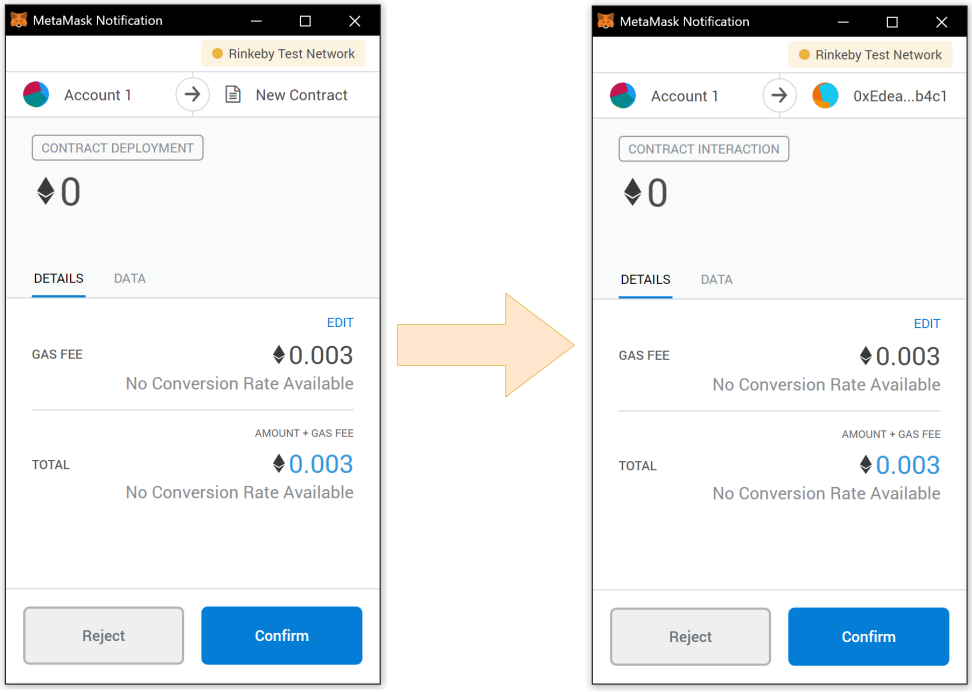

Figure 4.3: MetaMask prompt for the approval of contract deployment and adding a claim key

At this point, we have successfully deployed a proxy smart contract for the user on the blockchain and set it up for adding devices. As one of the benefits of linking the devices to a user is accountability, we also want the user to be a verified entity that could be held accountable for wrongdoings. To achieve this, a KYC claim is added to the smart contract if the user successfully identifies himself. In this implementation, this process is very rudimentary as only the filing out of the above registration form is required. The code executing the signing of the KYC claim and adding it to the user's identity contract is shown in Listing 4.2.

```
1    // Creates a signature from the contract's address.
2    const data = this.web3.utils.asciiToHex(
3      [result.options.address, 'verified']
4        .join(' ')
5    );
6    const hashedData = this.web3.utils.soliditySha3(
7      result.options.address, 7, data
8    );
9    const signature = await kydWeb3.eth.sign(
10     hashedData, environment.walletAddress
11   );
12
```

```
13      // Creates a KYC claim from the signature.
14      const addClaimABI = result.methods.addClaim(
15        7, 1, environment.contractAddress,
16        signature, data, 'https://www.arduino.cc/'
17      ).encodeABI();
18
19      // Adds the KYC claim to the identity smart contract.
20      await result.methods.execute(
21        result.options.address, 0, addClaimABI
22      ).send({
23        gas: 4612388, from: accounts[0]
24      });
```

Listing 4.2: Signing a KYC claim and adding it to the user's identity contract

The first block of code creates the data we want to include in the claim. Due to privacy reasons, we chose only to include the smart contract's address for this. This data is then hashed alongside the identity contract address and the designation of a KYC claim ($claimType = 7$). Finally, the hashed data is signed by the KYD platform. Notice that a different Web3 provider is used, as only the KYD platform can sign claims in its name. The signature is subsequently added to a *Claim* object alongside the issuers Ethereum address and URL. The entire *addClaim* method is then encoded into an Application Binary Interface (ABI) so that the identity contract can execute the method. The last step in getting the identity contract verified is, therefore, using the *execute* method on the smart contract, to add the KYC claim to the contract.

The user is now successfully represented by his proxy smart contract on the Ethereum blockchain. We do, however, also want to store the user data in the KYD platforms database. The form data collected by the web application is therefore sent to the server via the *register user* endpoint of the REST API. Listing 4.3 shows the server code handling this request.

```
1  class RegisterUser(APIView):
2      # Processes a user registration request.
3      def post(self, request):
4          # Creates user object from the data in the request body.
5          birth_date = dt.parse(request.data['birthDate'])\
6              .strftime("%Y-%m-%d")
7          User.objects.create_user(
8              username=request.data['account'],
9              contract=request.data['contract'],
10             first_name=request.data['firstName'],
11             last_name=request.data['lastName'],
12             birth_date=birth_date,
13             email=request.data['email'],
14             mobile_number=request.data['mobileNumber']
15         )
16
17         # Creates address object from the data in the request body
```

```
18          address_serializer = AddressSerializer(data={
19              "user": request.data['account'],
20              "street": request.data['street'],
21              "postal_code": request.data['postalCode'],
22              "city": request.data['city'],
23              "country": request.data['country'],
24          })
25
26          # Serializes the address object and saves it in the
27          # database.
28          if address_serializer.is_valid():
29              address_serializer.save()
30          else:
31              return Response(
32                  data=address_serializer.errors,
33                  status=status.HTTP_400_BAD_REQUEST
34              )
35
36          return Response(status=status.HTTP_201_CREATED)
```

Listing 4.3: User registration request on the KYD server

To store the user information in the database, we first create a user object with the data included in the request body. The primary key of the user table on KYD database is the username, which we associate with the user's Ethereum account address. Since we are also storing the addresses of the users, we next create an address object from the remaining data included in the request body. We include a foreign key to the user's account address to keep track which address is associated with which user. The address object is subsequently serialized and saved to the database in the corresponding separate address table. With the user now registered in the database and his associated smart contract being deployed, this brings the user registration process to an end.

4.2 Device Registration

This section outlines how devices are registered on the KYD platform. The process is comprised of multiple steps. In order to register a device, we need to extract some SRAM data from the device, as the data is used to construct a PUF for the verification of said device. Chapter 4.2.1 shows how this extraction process is handled. Devices are first registered in the KYD platform's database by the platform itself before the device owner receives his device, as is covered in Chapter 4.2.2. After the device owner receives his device, he can register it on the KYD platform using an identification number that was included with the device. Chapter 4.2.3 outlines this process along with the deployment of the identity contract representing the device on the Ethereum blockchain.

4.2.1 SRAM Data Extraction

The PUF we are using to authenticate the IoT-connected devices requires input data in the form of a collection of start-up values of SRAM cells. These values are extracted from the devices using the following process. The devices we are using are Arduino Mega2560s. Arduino boards generally run sketches written in C++, that control the behaviour of the device. We will a sketch to read out the SRAM values require for the PUF. Listing 4.4 shows the sketch used for this task.

```
// This gets executed on startup.
void setup() {
  Serial.begin(9600);

  readSRAM();
  overwriteSRAM();
}

// This is continuously looped while the device is powered on.
void loop() { }

// Reads the last 32 Bytes of the device's SRAM.
void readSRAM() {
  uint8_t volatile * pointer = (volatile uint8_t *)(8192-32);
  uint8_t i;

  for(i=0; i<32; i++) {
    Serial.print(*pointer < 16 ? "0" : "");
    Serial.print(*pointer, HEX);
    pointer++;
  }
}

// Overwrites the last 32 Bytes of the device's SRAM with 0s.
void overwriteSRAM() {
  uint8_t volatile * pointer = (volatile uint8_t *)(8192-32);
  uint8_t i;

  for(i=0; i<32; i++) {
    *(pointer + i) = 0;
  }
}
```

Listing 4.4: SRAM extraction sketch

Sketches always include two functions, one called *setup*, which is executed first on each start-up of the device and one called *loop*, which will be continuously executed over and over again while the device is powered on. Since we are interested in the SRAM start-up values, the setup function is where we execute our code. First, we call the *readSRAM* function, which reads the last 32 bytes from the end of the SRAM. We use the last

values of the SRAM because some values at the very beginning get written to during the start-up sequence before the *setup* function is executed. The read values then get printed to the serial bus of the device in hex representation. This can then be read from a connected computer as we will show in the device registration and verification sections. We subsequently execute the *overwriteSRAM* function, which overwrites all the values that were just printed to the serial bus with 0s. We do this to make sure the cells revert back to their biased start-up values following a power cycle and to make an unauthorized readout of the data slightly more difficult for potential malicious parties.

4.2.2 Device Initialization by the KYD Platform

In the case where the KYD platform is run by the manufacturer, the device first gets registered on the server before the device gets sent to the customer. This way, we can generate a unique key using the PUF and store it without the user having any influence on this process. When the user eventually wants to verify his device, we can then compare the reproduced key derived from the new SRAM data the user will provide to the original key stored in the database by the manufacturer. The most crucial component to the device initialization by the KYD platform is, therefore, the SRAM data extracted from the device.

In the previous section, we showed the sketch that is responsible for reading the SRAM values and outputting them in the serial bus of the device. Arduino sketches are written and uploaded to the device using the Arduino Create Editor [21]. The editor includes a serial monitor, as shown in Figure 4.4, that can be used to display data from the serial ports of the device. The SRAM data will show up here after the device is connected to the computer, where it can subsequently be copied to be used in the next step.

Figure 4.4: Serial Monitor of the Arduino Create Editor showing the extracted SRAM values

For this project, we implemented a simple GUI for the manufacturer to register the device on the database, which is shown in Figure 4.5. In a production setting, the manufacturer could, of course, implement his own system that interfaces with the REST API and is

more automated. The inputs required are the device model and the data copied from the serial monitor in the previous step. In terms of the form validation, the model input has set options to choose from, so no additional validation occurs. For the PUF data, the form is only considered valid if the input string has the correct length of 64 characters.

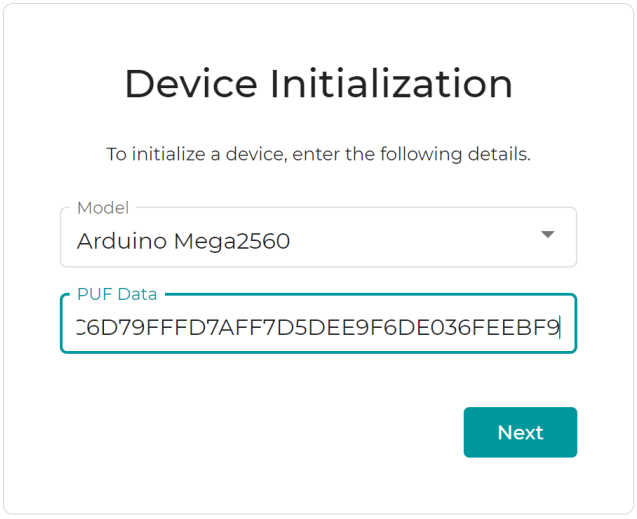

Figure 4.5: GUI for the device initialization form

The most crucial part of the device registration process is, of course, the generation of the unique key to identify and authenticate the device. This is handled by the KYD server utilizing the form inputs covered above enclosed within the request to the server. Listings 4.5 and 4.6 show the server code handling this request.

```
class CreateDevice(APIView):
    # Processes a device creation request.
    def post(self, request):
        try:
            # Generates a unique key and the corresponding helper.
            puf_data = bytes.fromhex(request.data['pufData'])
            extractor = FuzzyExtractor(32, 8)
            key, helper = extractor.generate(puf_data)
```

Listing 4.5: Device creation request on the KYD server (Part 1)

As we pointed out in Chapter 1.2, a proportion of the SRAM cells have no bias towards 0 or 1 and their start-up value is entirely random. This introduces noise into the SRAM reading, and consequently, we cannot use the raw SRAM data as a unique key. To use the SRAM data as a physical identifier in a PUF implementation, we, therefore, have to error correct the data first. As laid out in Chapter 2.2, fuzzy extractors are very well suited for this particular task. The first step for the device registration, therefore, is to use a fuzzy extractor to generate a unique key from the noisy SRAM data, as well as a helper, which we will need in order to reproduce the unique key in future verification processes. We are using the *fuzzy-extractor* Python library by Carter Yagemann for this

step [22]. To use the extractor, we create a *FuzzyExtractor* object, defining the length of the input in bytes as well as the number of noisy bits we want to allow for the reproduction of the key. We choose to use 32-Byte input strings, as the larger the inputs we allow, the larger the helper becomes, increasing storage costs and key reproduction time, which could negatively impact the efficiency of the system. We set the number of allowed noisy bits to 8 as this was sufficient to reproduce keys reliably in our testing, though this might have to be adjusted to allow for different devices.

If we let n = length of the input string and k = the number of matching bits, then the probability of getting exactly k matching bits is

$$P(k) = \binom{n}{k} 0.5^k * 0.5^{n-k} = \binom{n}{k} 0.5^n$$

Let q = number of allowed noisy bits, then the probability of getting at least the required $n - q$ matching bits for a successful key reproduction is

$$P(k \geq n - q) = \sum_{k=n-q}^{n} \binom{n}{k} 0.5^n$$

For our set parameters of $n = 256$ and $q = 8$, we therefore get a probability of randomly selecting enough matching bits for a successful key reproduction of

$$P(k \geq 248) = \sum_{k=248}^{256} \binom{256}{k} 0.5^{256} \approx 3.65 * 10^{-63}$$

This should be an adequately low probability to ensure the security of our PUF design.

We can then call the *generate* function of the fuzzy extractor and pass it the PUF data included in the request body to get the sought-after key and helper for the device. The next steps of handling a device creation request on the server are outlined in the code of Listing 4.6.

```
            # Creates device object from the data in the request
            # body.
            helper_string = json.dumps((
                hex_list(helper[0]),
                hex_list(helper[1]),
                hex_list(helper[2]),
            ))
            id_num = hex(zlib.crc32(key))
            device = {
                "id": id_num,
                "model": request.data['model'],
                "key": key.hex(),
                "helper": helper_string
            }

            # Serializes the device object and saves it in the
            # database.
```

```
18              serializer = DeviceSerializer(data=device)
19              if serializer.is_valid():
20                  serializer.save()
21                  return Response(status=status.HTTP_201_CREATED)
22              return Response(
23                  data=serializer.errors,
24                  status=status.HTTP_400_BAD_REQUEST
25              )
26          except:
27              return Response(
28                  data=traceback.format_exc(),
29                  status=status.HTTP_500_INTERNAL_SERVER_ERROR
30              )
```

Listing 4.6: Device creation request on the KYD server (Part 2)

To prepare for storage in the database, the helper is first converted to string representation. We also use a hashing function on the key to generate a ten-character unique identifier. This is the identifier that will have to be included with the device when the customer buys it and is used as the primary key for the device in the database. All of this data is then added to a *Device* object along with the device model included in the request body. The *Device* object is subsequently serialized and stored in the database if it is valid, at which point the registration process on the manufacturer's side is concluded.

4.2.3 Device Registration by the Owner

The device owner can choose to register his devices whenever he wishes using the KYD platform. To do so, he must first create an Ethereum account and go through the user registration process outlined in Chapter 4.1.2. When the user has successfully signed in and is on the main page of the platform, he is presented with the option to register devices. There are again several steps to this process, as we also need to deploy a smart contract representing the device, similarly to the user registration process.

The first step in this process is for the user to fill in the device registration form that is shown in Figure 4.6. The user can give the device a name to recognize it when multiple devices are registered. The second field that is required is for the unique identifier we generated in Chapter 4.2.2, which is shipped with the device when purchased. This device id was used as the primary key during the creation on the database and is needed to retrieve or manipulate that instance. The name input can be up to 255 characters long for the form to be valid while the device id must always be a 10-character string.

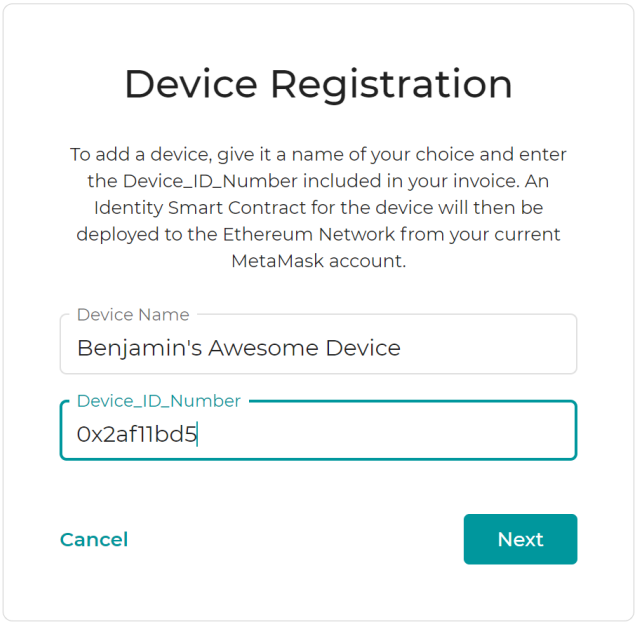

Figure 4.6: GUI for the device registration form

The next step is to deploy an identity smart contract to represent this device on the Ethereum blockchain. To do this, we deploy the same base `Identity.sol` smart contract used for the user's contract to the blockchain, as shown in Listing 4.7.

```
1    // Deploys an identity smart contract from accounts[0].
2    const result = await new this.web3.eth.Contract(abi)
3      .deploy({ data: bytecode })
4      .send({ from: accounts[0], gas: 3000000 });
```

Listing 4.7: Smart contract deployment for a device from the web app

We have previously stated that we wish the device to be linked to a user on the blockchain as well. This is one of the reasons we created a user smart contract in Chapter 4.1.2 and had the user's identity verified by adding a KYC claim to the contract. We also added a claim key so that the user contract can issue claims. We can utilize this functionality to issue ownership claims to the devices. This effectively links the devices to the user and will lead to having verified instances of all involved components represented on the blockchain once the devices are also verified by a KYD claim. The code responsible for issuing and adding the ownership claim can be seen in Listing 4.8.

```
1    // Creates a signature for the device id.
2    const data = this.web3.utils.asciiToHex(
3      [device.value.id, 'verified'].join(' ')
4    );
5    const hashedData = this.web3.utils.soliditySha3(
6      result.options.address, 9, data
7    );
```

```
    const signature = await this.web3.eth.sign(
      hashedData, accounts[0]
    );

    // Creates an ownership claim from the signature.
    const user = await this.userService.get();
    const addClaimABI = result.methods.addClaim(
      9, 1, user.contract, signature, data, ''
    ).encodeABI();

    // Adds the ownership claim to the identity smart contract.
    await result.methods.execute(
      result.options.address, 0, addClaimABI,
    ).send(
      { gas: 4612388, from: accounts[0] }
    );
```

Listing 4.8: Signing an ownership claim and adding it to the device's identity contract

The process is almost identical to the adding of the KYC claim to the user contract. However, this time it is the user's Ethereum account that will be signing the claim. As for the contract deployment and the following adding of the ownership claim, this signing must be approved by the device owner through the MetaMask extension. Furthermore, we use the device's unique identifier in the signed message and indicate the claim to be an ownership claim ($claimType = 9$). The steps for adding the claim differ only in the information given on the issuer, as the user's identity contract will be given this time, and no URL is provided. Once these transactions have executed, the device will be successfully represented on the blockchain by its proxy contract, which is also linked to the user's proxy contract through the ownership claim.

In order to keep track of which devices have been registered and belong to which user, we also want to update the data stored in the device table of the KYD database. For this reason, we send the device registration form along with the user's account address and the deployed identity contract's address to the KYD server through the *user registration* endpoint of the REST API. Listing 4.9 shows the server code that handles this request.

```
class RegisterDevice(APIView):
    # Processes a device registration request.
    def patch(self, request):
        try:
            # Retrieves the device with the id specified from the
            # database.
            device = Device.objects.get(id=request.data['id'])
            if not device:
                return Response(status=status.HTTP_204_NO_CONTENT)

            # Updates the empty device values and saves the
            # changes to the database.
            device.name = request.data['name']
```

```
14              device.account = request.data['account']
15              device.contract = request.data['contract']
16              device.save()
17
18              return Response(status=status.HTTP_200_OK)
19          except:
20              return Response(
21                  data=traceback.format_exc(),
22                  status=status.HTTP_500_INTERNAL_SERVER_ERROR)
```

Listing 4.9: Device registration request on the KYD server

Notice that this request is a patch request as it updates the already existing resource on the database rather than creating or retrieving one. The first step, therefore, is to retrieve the device instance stored on the database linked to the unique identifier we asked for in the registration form. Once the resource is retrieved from the server, we can update the values that were previously left empty. This includes the name chosen by the user, the device owner's Ethereum account address and the address of the previously deployed device smart contract. This allows us, among other things, to retrieve all devices of a user from the database by providing the user's account address, meaning the device is now also logically linked to the user on the database. This brings the device registration to a close. It is imperative to understand, however, that the device is currently only linked to the user and the ownership claim does not guarantee its identity yet. Handling this will be the subject of the next chapter where we will look into the verification of devices on the KYD platform.

4.3 Device Verification

Devices registered by the user are shown on the main page. From there, the user can view the device details. As long as the device is not verified by the KYD platform, meaning it does not have a signed KYD claim added to its contract, it will be marked as not verified. If this is the case, a button to verify the device is also displayed. To verify his device, the user will have to provide SRAM data collected from his device using the same Arduino sketch we introduced in Chapter 4.2.1.

The user will first be directed to a page containing detailed instructions of the process of obtaining the required data. The instructions include the following steps:

1. **Setting up the Arduino Create Editor**

   The user is asked to follow a get-started guide provided by Arduino, which walks him through setting up the Create Editor. The guide involves creating an account, setting up the online Integrated Development Environment (IDE) and installing the required plug-in to connect to Arduino devices. It also shows the basic steps of developing sketches as well as how to upload them to the devices. Further into the tutorial, there is also a section where interacting with the serial monitor is shown,

which we do require in a later step. Most users that bought an Arduino board will already be familiar with this, of course, and can skip this step.

2. **Downloading the data extraction sketch**

   A *Download* button is displayed that allows the user to download the required SRAM extraction sketch.

3. **Uploading the sketch to the device**

   The user is asked to import the sketch into his sketchbook using the *Import* button in the Create Editor. Subsequently, he should connect his Arduino, wait for a successful connection and upload the sketch.

4. **Copying the SRAM data from the Serial Monitor**

   The user is then asked to navigate to the serial monitor tab and copy the displayed data, as shown in Figure 4.4 previously.

Overall, a user familiar with Arduino boards will be able to do all this in under a minute, while a novice will probably require around 15 minutes on his first attempt. As this is one of the few things we require the user to do manually, it reinforces the assumption, that implementing a fully automated solution would have resulted in more work for the user, in addition to being far more complicated.

After following the above steps, the user can navigate to the next page where the device verification form is displayed. The corresponding GUI can be seen in Figure 4.7.

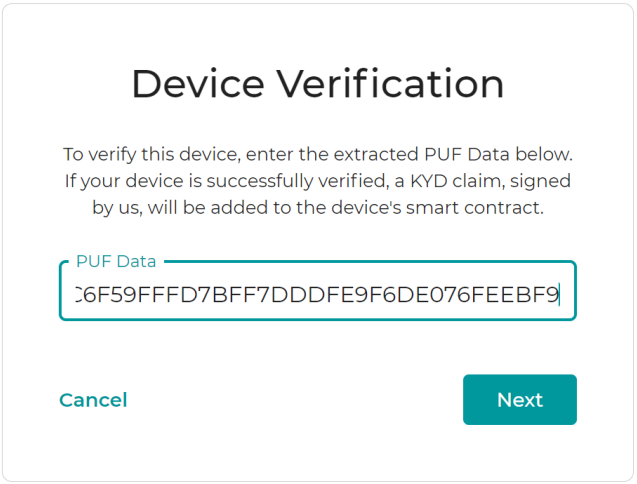

Figure 4.7: GUI for the device verification form

The GUI for this form asks only for the PUF data that was copied from the serial monitor in step 4, so all the user has to do is copy it into the field. In terms of validation, the form checks if the SRAM reading has the correct length of 64 characters. The SRAM values are printed to the serial monitor in hex representation so every Byte results in two characters.

Once the user verification form is submitted, the PUF data gets sent to the KYD server alongside the device's unique identifier. Listings 4.10 and 4.11 show the server code handling this device verification request.

```
class VerifyDevice(APIView):
    # Processes a device verification request.
    def post(self, request):
        try:
            # Retrieves the key and helper corresponding to the
            # device id from the database.
            try:
                key = Device.objects.get(
                    id=request.data['id']
                ).key
                helper_json = json.loads(
                    Device.objects.get(
                        id=request.data['id']
                    ).helper
                )
            except:
                return Response(
                    data=traceback.format_exc(),
                    status=status.HTTP_404_NOT_FOUND
                )

            # Recreates the helper from its string representation.
            helper = (
                np.array(
                    bytes_list(helper_json[0]), dtype='uint8'
                ),
                np.array(
                    bytes_list(helper_json[1]), dtype='uint8'
                ),
                np.array(
                    bytes_list(helper_json[2]), dtype='uint8'
                ),
            )
```

Listing 4.10: Device verification request on the KYD server (Part 1)

In order to reproduce the device's key and have something to compare it to, the first step is to retrieve the key and helper stored in the database corresponding to the device id included in the request body. We then need to recreate the NumPy matrix structure for the helper that it originally had before we converted it to string representation for storage in the database. At this point, we should have all components ready to attempt the key-reproduction, shown in Listing 4.11, to see if the device is valid.

```
            # Uses a FuzzyExtractor to reproduce the key using the
            # pufData from the request body and the helper.
            extractor = FuzzyExtractor(32, 8)
            puf_data = bytes.fromhex(request.data['pufData'])
            r_key = extractor.reproduce(puf_data, helper)

            # Checks if the key stored in the database matches the
            # reproduced key.
            if r_key and key == r_key.hex():
                return Response(
                    data='true',
                    status=status.HTTP_200_OK
                )
            return Response(
                data='false',
                status=status.HTTP_200_OK
            )
        except:
            return Response(
                data=traceback.format_exc(),
                status=status.HTTP_500_INTERNAL_SERVER_ERROR
            )
```

Listing 4.11: Device verification request on the KYD server (Part 2)

The reproduction procedure for the device's key is similar to the generation procedure in Listing 4.5. First, we create a *FuzzyExtractor* object using the same parameters used in the generation process. We then call the *reproduce* function on the fuzzy extractor and pass it the PUF data that was included in the request body and the helper we just recreated. If successful, we should receive the device's unique key. We then check if this reproduced key matches the key stored in the database since the device's first registration by the KYD platform. If they match, we return *true* to the web application and *false* otherwise.

In the case where the verification failed, the key could not be reproduced precisely by the KYD server. Consequently, the SRAM values provided by the user must not have been compatible with the device they tried to verify. Either a wrong device was connected, or the readout of the SRAM values introduced some errors. To combat these issues, the user is asked to check the device corresponds to the device id provided with his purchase and to unplug his device to repeat the SRAM extraction procedure. The user is provided with the option to try again using the newly copied PUF data in this case.

If the device was successfully verified by the KYD server, the web application can issue a KYD claim to the device's proxy contract. This process is very similar to the other two claims that were added in Chapter 4.1.2 and 4.2.3. However, it is important to understand the differences between these claims, so we will outline the key points here. Like in Listing 4.2 the KYD claim has to be signed by the KYD platform's smart contract. To do this, a separate Web3 provider is required that is linked to the platforms Ethereum account. This time, we wish to issue a KYD claim, so the *claimType* parameter in the signature

and the claim is set to 8. The address set inside the signature is that of the device's smart contract indicating the recipient the claim is issued for. The issuer parameter inside the *Claim* object, however, is set to the KYD platform's identity contract and the URL links to the KYD company's website.

The claim is then added by the device owner's Ethereum account to the device's smart contract. This transaction will also have to be approved through the MetaMask extension, as all transactions that change data on the contract must. When the transaction is approved and added to the blockchain, the device contract will successfully be verified by the KYD platform, as it holds a signed KYD claim. Subsequently, other contracts or accounts willing to interact with the device can check if the device is verified by viewing its KYD claim.

4.4 KYD Platform Main Page

A successfully registered user is automatically redirected to the KYD platform's main page upon sign-in. This page provides an overview of the deployed smart contracts of a user, as shown in Figure 4.8. The signed-in user is always shown at the top. If devices have been registered, they will be listed below the user. From this page, the user has the option to register as many devices as he wishes using the *Register new device* button. Devices that have been verified are marked with the green *Verified* badge. Devices that have been registered and thus have a deployed smart contract, but no KYD claim, are also listed but are marked with the red *Unverified* badge. Unverified devices also have a displayed button, *Verify device*, that redirects the user to the device verification page. The entries for the user and the devices can be expanded to reveal more details on the deployed smart contracts and the entities they represent.

The left side of this expanded view differs from the user to the devices but shows additional basic information. For the user, this includes the user's name, the contract's address on the blockchain and the Ethereum account address used to deploy the contract. For the devices, the name which was chosen by the user, the make and the model of the device are shown. Additionally, the contract's address and the owner's address are displayed, like with the user. The addresses also link to the corresponding account or contract on Etherscan [23]. Etherscan is a block explorer platform for Ethereum, that allows users to search the blockchain for anything from transactions to tokens and addresses. For our case specifically, it shows all interactions and transactions made by or with the contract or account specified.

In the middle section, all keys held by the identity contract are listed. In the user's case, this includes the management key added upon creation of the contract and the claim key we added subsequently. Devices only have the management key in our design, however, as they do not issue any claims.

The right side shows a list of the claims held by the identity contract. The user has a KYC claim that we added in Chapter 4.1.2. Furthermore, a link to the Etherscan page of the issuing contract is provided, meaning the identity contract of the KYD platform,

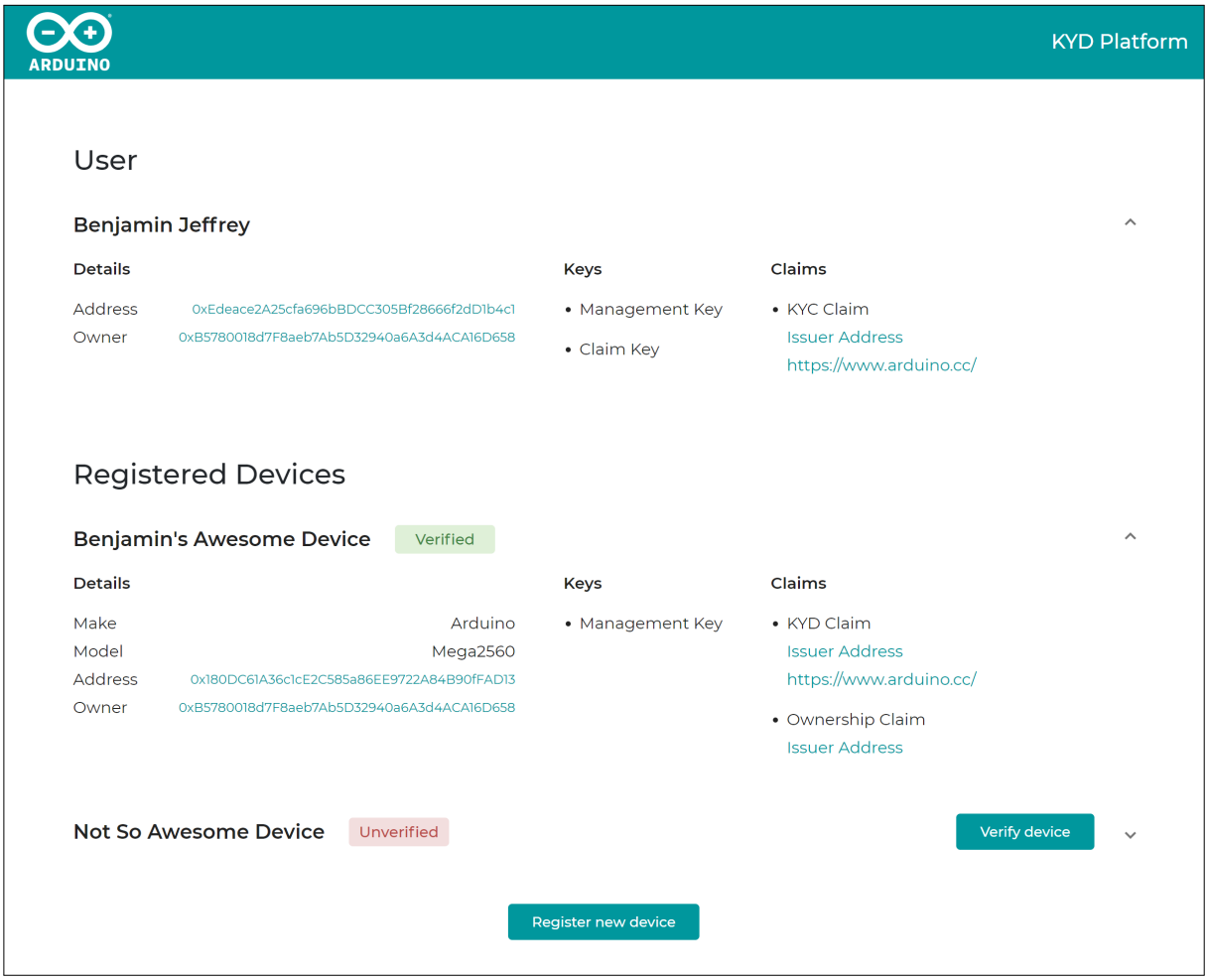

Figure 4.8: The main page of the KYD platform showing a successfully registered user

as well as the URL that was specified as part of the *Claim* object. The devices should always have an ownership claim, where the issuer address links to the Etherscan page corresponding to the user's smart contract. Verified devices will additionally have a KYD claim that is again linked to the KYD platform, and its identity contract.

Overall, this page is meant to display the progress made by the user in the process of getting his devices verified. It also helps to guide him through this process by displaying buttons that link to the other components of this system. It is meant to provide a visual overview by providing helpful information and links so that the user does not lose himself, adding a large number of devices. Furthermore, the user can log out of the KYD platform by signing out of his MetaMask account.

4.5 Identity Smart Contract

So far, we have talked a lot about all the processes to getting the user, and his devices represented and verified on the Ethereum blockchain. To clarify how these identity contracts

look like we will go over the essential details in this section.

Smart contracts are essentially a set of functions and conditions written in code that is committed to the blockchain and describes a contract that is guaranteed to be enforced. This protocol is of course publicly available on the blockchain. In practice, writing a smart contract is a bit like writing a class in an object-oriented programming language. A set of functions that the contract can execute, or other parties can use to interact with the contract are defined. There are also events in smart contracts that can automatically execute certain functionalities when triggered. As a contract deployed to the blockchain is immutable, in the way that these functions can not be changed, the conditions described in the contract will always be enforced [24].

As was stated in Chapter 2.3, the design of the identity smart contract in this thesis is based on the ERC 734 and ERC 735 standards. The implementation itself is an adjusted version of Fractal's design they put together for an article covering the standards [25] [26].

ERC 734 focuses on describing a key manager for the identity contract in order to control privileges on the contract and enable it to act as a blockchain proxy account for an entity in the real world. As such, the implementation of the interface described in the ERC revolves heavily around the handling of keys on the smart contract. Listing 4.12 shows such a *Key* object.

```
1     struct Key {
2         uint256 purpose; // e.g. MANAGEMENT = 1, CLAIM = 3
3         uint256 keyType; // e.g. ECDSA = 1, RSA = 2
4         bytes32 key;
5     }
```

Listing 4.12: Key Object in the ERC 734 Standard

Each key, therefore, also has a purpose and a type. We have touched on the purpose many times in this thesis, as this specifies what the key is used for, like the purposes *management* and *claim*. *keyType* refers to the Digital Signature Algorithm (DSA) used for this key. To follow the ERC 734 standard, a range of functions and events were also added to the identity contract to specify the protocol the contract should follow. The ERC 734 functions are listed in Table 4.2 alongside a basic description of their functionality. Furthermore, Table 4.3 shows the defined events that were added to the identity contract based on the ERC 734 standard.

Table 4.2: The ERC 734 functions of the identity smart contract

Function	**Description**
getKey	Returns the *Key* object of a given id, if the key is stored in the contract.
keyHasPurpose	Returns the *purpose* of a given key. E.g. $Management = 1$.

Function	Description
getKeysByPurpose	Returns a list of all the keys, stored in the contract, with the specified *purpose*.
addKey	Adds a *Key* object to the contract with the specified parameters. Triggers *KeyAdded* event. Requires payment to execute.
removeKey	Removes the *Key* object with the given id from the contract. Triggers *KeyRemoved* event. Requires payment to execute.
execute	Processes given execution instructions. First triggers *ExecutionRequested* event, then calls *approve* function. If approved, executes the request and triggers *Executed* event. Requires payment to execute.
approve	Approves executions and adding of claims, by checking if the sender has the correct management or action key. Triggers *Approved* event. Requires payment to execute.

Table 4.3: The ERC 734 events of the identity smart contract

Event	Description
KeyAdded	Signals a *Key* object was successfully added to the contract.
KeyRemoved	Signals a *Key* object was successfully removed from the contract.
ExecutionRequested	Signals that the *execute* function was called, but approval is required to progress further.
Executed	Signals an execution request was fulfilled.
Approved	Signals an execution request was approved.

ERC 735 focuses on describing functionality for handling claims on a smart contract. Listing 4.13 shows an example of a *Claim* object in the ERC 735 standard.

```
1    struct Claim {
2        uint256 claimType; // e.g. KYC = 7, KYD = 8
3        uint256 scheme; // e.g. ECDSA = 1, RSA = 2
4        address issuer;
5        bytes signature; // this.address + topic + data
6        bytes data;
7        string uri;
8    }
```

Listing 4.13: Claim Object in the ERC 735 Standard

Claims have several parameters. The *claimType* is another parameter we have mentioned repeatedly as it distinguishes what the claim signifies, like if it is a KYC or a KYD claim. *scheme* refers to the key types with which this claim should be verified. *issuer* refers to the issuer's identity contract address on the blockchain. The *signature* holds the critical part of the claim, the proof that the claim issuer issued this claim. It is, therefore, a message containing the claim data, the target address, and the *claimType*, signed by the issuer's account. The *data* is some form of information that will be included in the signature and the claim, like some device data, for instance. The *uri* specifies a URL to the location of the claim, most likely the issuer's website.

The functions that were implemented in the identity smart contract to conform with the ERC 735 standard are listed in Table 4.4, while the added events are listed in Table 4.5.

Table 4.4: The ERC 735 functions of the identity smart contract

Function	**Description**
getClaim	Returns the *Claim* object of a given id, if the claim is stored in the contract.
getClaimIdsByType	Returns a list of all the claims, stored in the contract, with the specified *claimType*.
addClaim	Adds a *Claim* object to the contract with the specified parameters. Triggers *ClaimAdded* event. Requires payment to execute.
removeClaim	Removes the *Claim* object with the given id from the contract. Triggers *ClaimRemoved* event. Requires payment to execute.

Table 4.5: The ERC 735 events of the identity smart contract

Event	**Description**
ClaimAdded	Signals a *Claim* object was successfully added to the contract.
ClaimRemoved	Signals a *Claim* object was successfully removed from the contract.

In combination, the ERC 734 and ERC 735 standard, therefore, provide a well-rounded package that allows us to represent physical entities on the blockchain very well. Through the ERC 735 standard, the identity smart contracts also facilitate the verification of any claim, like the identity claims, that are crucial for this thesis.

Chapter 5

Evaluation

In order to evaluate the implemented system, let us have a look at how well it fulfils the original goals we set in Chapter 1.2. The main objective of this thesis was to design and implement a prototype for the identification of IoT devices on the Ethereum blockchain. This is covered in entirety by the prototype we implemented for this thesis. However, there are many details concerning the system we should evaluate in some more detail, which we will do in this chapter. First, we will go over the positive and negative aspects of the prototype in a general sense. Subsequently, we will evaluate the security of the system and the blockchain integration used. Finally, we will provide pointers on what future work could be conducted based on the shortcomings of the implemented system discovered in the previous sections.

5.1 System Design

The prototype we implemented for this thesis is a practicality focused solution to the requirements set at the beginning of this thesis. By this, we mean that the system is in a mature state that would not require much more effort to become production-ready. In this sense, it exceeds the more theoretical implementations usually encountered in the current research, like the articles mentioned in Chapter 2.4. All requirements set in Chapter 3, which are relevant to the chosen design, are fulfilled by the implemented prototype. The system is robust in the functionalities it provides and has stood up to our limited testing very well. As the system is a prototype implementation, we did refrain from the kind of extensive testing that would be conducted for production systems, however.

Another area where the practicality focus of the design is noticeable is in the graphical user interface. The interface guides the user through all the required steps with detailed instructions reliably. Special care was put into making sure the user is always informed of what is going on and what is required of him. Furthermore, the GUI was designed to feel welcoming and familiar using Google's material design in most places and giving off a polished appearance in general. Essential for the robustness of the system is also the detailed validation of the user's inputs at each step to ensure he does not enter any

values that could break the system or do not make any sense. The user is further clearly informed which entry was not accepted in the case where the form validation discovers an error. This ensures that the user does not get frustrated with any of the processes of the system and is informed at all times what the KYD platform requires of him.

Overall, the design for this prototype is very modular. The server is addressed through the REST API, meaning it could be used with any client, other than the web application implemented for this thesis, without a problem. The blockchain integration could also be changed without too much effort by changing the user and device service within the web application. In fact, it is plausible to remove the blockchain integration entirely or even provide support for a different blockchain. Changes to the identity smart contract are straight forward as well, as it is all contained in one Solidity file. On a more detailed level, the web application is broken up into many Angular modules, which could be deleted, modified, or even imported into other projects freely with minimal adjustments required to keep the application intact. What this modularity generally allows for is to swap out or use different components reasonably easily without breaking the system. This leads to the system's design being very flexible and able to adapt to many different use cases with minimal effort required for the adaptation process.

One of the strengths of the modular nature of the implemented system is also its expandability. For instance, adding functionality to the web application is achieved by simply adding new modules and linking them up in the routing module of the project. On the blockchain side of things, there is even more room for expansion. Figure 5.1 shows some of the many possibilities that could be implemented to expand the ERC 734 / 735 system outlined in this thesis.

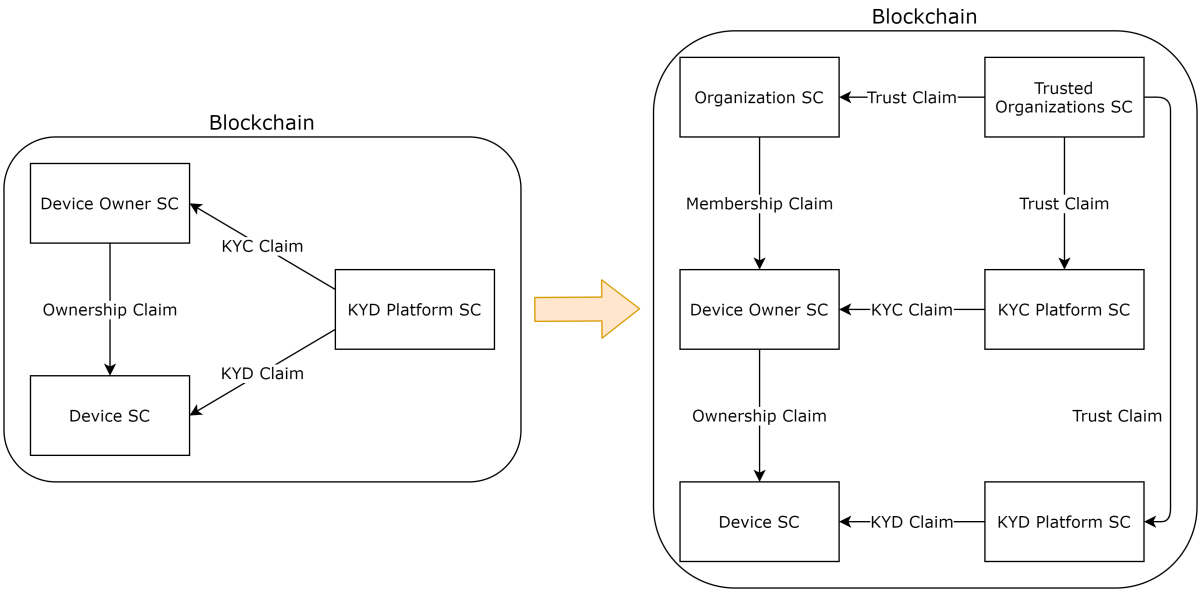

Figure 5.1: Possibilities of expanding the ERC 734 / 735 implementation

We have already covered the possibility to have a dedicated KYC service to issue KYC claims in this thesis. The reason being, that the user registration form we ask for on the KYD platform is not proof enough of an individual's identity. We implemented the

KYC mechanism mainly to illustrate how the adding of the KYC claims would work, as it makes little sense to have a KYD system without also linking the devices to a verified owner. In a production setting, we would want that separate KYC platform to check more details on the user including the submission of government-issued identity confirmation and probably having a live video chat with the user to confirm his identity. Another possibility to expand this system is to add new types of claims. For instance, the device owner could also be part of an organization that works together and has its own representation on the blockchain. The identity contract of the organization could then issue membership claims to all of its members. It could also use the key system to fine-tune how much control specific members get on the organization's identity contract, like who has the right to add new members. We will touch on the problem of the need for trust in the KYD platform in Chapter 5.3. However, what if there were an additional entity on the blockchain that issued trust claims to organizations it has deemed trustworthy? This approach would be similar to the way webshops often get a certification from a trusted entity stating they are a safe online store, or the way food is labelled to indicate it was produced following ethical guidelines. In any case, the problem of needing trust in a third party always remains and is just shifted upwards. These expansion ideas for the ERC 734 / 735 implementation are just a few of basically endless possibilities to expand this system.

A considerable limitation of the platform design proposed in this thesis is the limited list of supported devices. Since we are using an SRAM PUF design, the support of the KYD platform is limited to devices that have a sufficiently large SRAM chip built-in. This is unavoidable, however, due to the broad market of different types of IoT-connected devices, built for various use cases. PUFs are also a very tailored solution for identification as they are reliant on the physical characteristics of an IC and thus must be adjusted for different chips all the time. Not being easily scalable is, therefore, part of a PUFs nature. There simply is no one-PUF-fits-all solution. To expand the device support of the KYD platform, many different types of PUFs would, therefore, have to be added to support the various types of devices on the market, which exceeded the scope of the prototype implementation for this thesis.

5.2 Security

In terms of the security of the system, we have already mentioned some concessions that were made over the course of this project, most importantly the decision to move to a one-time authentication approach over a periodical implementation. The motivation for doing this is outlined in Chapter 3.1.3. However, in doing so, we have, of course, reduced the frequency of authentications drastically, which in turn makes it easier for malicious parties to circumvent the system. It is, for instance, a lot easier to use an imposter device, with the wrong identity smart contract after the verification process, meaning the correct device would only have to be used for the initial verification. This is, of course, the same in the centralized world, where a passport is generally valid for a long time without the owner needing to prove his identity to the authorities repeatedly. However, passports include a picture that allows quick, even if not entirely secure, verification that

the passport is used by its owner rather than a third party. Similarly, the identity contract of devices could be coupled with the public key of said device. Using traditional public-key cryptography schemes, the device could then provide quick verification of its identity by being the holder of the corresponding private key, thus being able to encrypt and decrypt public-key encrypted messages. This, of course, only works as long as the private key is kept secret.

However, this approach would not help in the case where the malicious party is the device owner himself. It is exceedingly hard to deal with a scenario where the device owner wishes to exploit the system. Throughout this thesis, many ideas were brainstormed to deal with this kind of issue, but we ultimately concluded that this was not feasible using an SRAM PUF implementation. The problems mainly arise from the device owner having physical access to the device. With the most straightforward implementation of just asking the user to use the Arduino sketch to read out the SRAM data, the device owner could just return any values he sees fit, regardless of what the sketch emits. In a more sophisticated design where the device owner gets blocked out as much as possible, through the establishment of a secure connection from the KYD platform to the device, we run into the same problem. Since it is the user that sets up this connection, it would be entirely possible to insert malware or imposter devices at any point in the communication chain. Unless a stronger PUF design is chosen, that is based on multiple different kinds of CRPs, that the malicious party can not predict, it is impossible to ensure fully secure authentication, if that party has physical access to the device. For instance, the device owner can read out the values of the entire SRAM chip, and no matter what challenge is set, as long as it is based on SRAM, the user will know the correct response even if he is verifying an imposter device. Splitting up the registration and verification process among multiple actors as we did in the design of this thesis helps somewhat but is not sufficient to guard against a malicious device owner that knows what he is doing.

The device owners are, of course, verified by a KYC provider in our design, meaning their blockchain representations are linked to their real-world identities. Considering this, malicious device owners could be held legally accountable for their actions. However, laws and regulations are often still unclear when it comes to blockchain technology and relying on the legal system of a state goes against the decentralized philosophy of most blockchain systems.

To summarize, our design requires some degree of trust in the device owner and is not impenetrable to tampering with physical access to the device. The same is also true for the KYD platform itself. From the standpoint of a business partner on the blockchain, if the platform acts maliciously, the claims issued by it are also worthless. This means trust in the KYD platform is also required for this design to work. We will touch on the issue of trust a bit more in the next chapter.

First, let us talk about some of the security aspects this system does well. The security of the implemented system is mostly dependent on how secure the MetaMask log-in is. Interactions with the blockchain that could lose the user money always have to be approved through the extension. As MetaMask invests heavily in its security, using it to manage the users on the platform almost certainly provides better protection than a custom solution implemented within the timeframe of this thesis would have. Since we did not choose

to further invest in the JWT-authentication approach, access to most functionalities of the server is dependent on the user's Ethereum account. For instance, retrieving all the registered devices is achieved by filtering the device list by the user's account address. However, the only data stored on the KYD platform that is sensitive and not published on the blockchain is the PUF key and perhaps the user's registration data. The PUF key never leaves the server environment, so that is not an issue. As for the user's data, this part of the process is not intended for a production environment, and a dedicated KYC system should protect the access points to retrieve a user's data more securely.

Privacy concerns were given special thought throughout this project. One of the goals was to ensure that no sensitive data gets published on the blockchain since it is accessible to anyone. The KYC and KYD verification processes are therefore run securely on the server. The data added to the signatures is intentionally kept to a minimum and never includes any data that could be regarded as sensitive. This way, only the platform has access to the proof provided during the verification process of any entity.

5.3 Blockchain Integration

The blockchain integration of the KYD platform is remarkably seamless from a user's perspective. The user does not need to know any of the details that go into deploying contracts or even what a smart contract is, in order to use the system described in this thesis. The processes of deploying and interacting with the smart contracts are always automatic with no user interaction required beyond hitting an approve button. This allows for a much broader user base for the system, as it is not restricted to blockchain experts.

As we have mentioned above, our system does require some trust from both the user and the KYD platform. The KYD platform is especially dependent on trust as it is an issuer of certifications based on data only the platform possesses. Blockchain systems traditionally do not like to be dependent on third parties to provide trust. The entire motivation for the creation of blockchain technology was to remove the requirement of a trusted third party in payment transactions. However, removing the requirement for trust in the KYD platform would require the whole verification process to run on the blockchain. That way, the verification would be handled by the protocol described in a smart contract and would be guaranteed to be executed identically for each device through the consensus mechanism of the blockchain. This approach was not feasible for two reasons. Firstly, the verification process outlined in this thesis is far too computationally complex to be run in a smart contract and would result, if at all possible, in extortionately high mining fees. Secondly, in one way or another, some sensitive data would have to be sent to the smart contract in order to verify a device, which would then be instantly accessible to anyone. There is no way around needing trust in the third-party KYD platform, therefore. As this is a centralized authority, the system proposed in this thesis is neither completely trustless nor fully decentralized.

One possible improvement to the ERC 734 / 735 implementation of this project was taken into consideration for this thesis. That is to implement an expiration date for claims issued

by the KYD platform. Doing this, we could introduce more periodical verification than the one-time approach while still preserving most of its benefits. Expiration possibilities are not covered in the ERC 735 standard, so introducing this would have to be discussed on the standards discussion thread or a new standard would have to be defined in order to ensure compatibility with other systems on the blockchain. The expiration date would have to be reasonably far in the future for the re-verification process not to be overly annoying to the device owner. Two approaches were thought of on how this might be implemented during this project.

The first possibility is to include an expiration date as part of the data parameter of the *Claim* object. This is easily implemented on the KYD platform's side. Any smart contracts or accounts interacting with the device's identity contract would then have to check the claim's data to verify its validity. This approach introduces more work on the interacting party's side, however, as merely viewing the claims held by a contract is inadequate to determine verification. More critically, however, the interacting party has no way of knowing he is supposed to check the expiration date of a claim, as this is not part of the ERC 735 standard.

The second possibility is to use events on the smart contract that trigger upon the expiration date of the claim and delete the claim from the contract. This is more challenging to implement, and we have not spent enough time with Ethereum smart contracts to definitively say this is feasible. However, it would be a clean solution that is more compatible with other parties using the ERC 735 standard than the previous approach. There is a possible issue of the device owner just extracting the claim before deletion and then adding it again himself, however, since the expiration date is not baked into the claim itself.

Since both these approaches come with drawbacks, and we wanted to stay as true to the standards as possible, we decided not to implement an expiration for claims in this thesis. This ensures the prototype is compatible with as many systems willing to use it as possible.

5.4 Future Work

Through some of the critical evaluations above, we can defer several avenues in which future work related to this thesis could head. Firstly, the prototype proposed in this system could be developed further and be brought to production-ready status. This would likely include expanding the device support substantially, as well as thoroughly testing all aspects and components of the system.

Implementing a KYC platform based on the same ERC 734 / 735 Ethereum standards is also a project that would help in getting the vision of this thesis to production. As we have mentioned multiple times, the KYC claims should be issued by a separate platform that is much more focused on the secure authentication of human entities. In the same way, there are infinite possibilities to expand the network of identity contracts to allow for new functionalities and different types of claims, as we have eluded to above.

To expand the authentication capabilities of the current design, implementing a system that allows for message encryption using the SRAM PUF would be desirable. This could be achieved based on the proposal of [13] mentioned in Chapter 2.4, by using the PUF to generate a private key for the device whenever a message needs to be encrypted or decrypted. The corresponding public key could be published in the identity smart contract.

Another avenue to explore could be to develop a new ERC to allow for expirations of claims or another solution to making identity verifications more periodical. This is likely desirable in the bigger vision of entire decentralized markets where any measure to increase the trust between transacting parties will surely be welcome.

A more theoretical area to further research would be to try and reduce the centralized aspect of the KYD platform proposed in this thesis and reduce or remove the need for trust in this external party. Achieving this would allow for a solution of identifying IoT-connected devices on the blockchain in a more trustless and decentralized way. This would, of course, be more in line with the general philosophy of blockchain systems wanting to decentralize processes as much as possible.

Chapter 6

Summary and Conclusions

The main goal of this thesis was to devise a design for a smart contract-based platform for the identification of IoT-connected devices. The identification of the devices was to be based on an SRAM-PUF, and a working prototype of the design was to be implemented.

On the way to achieving this task, the focus of certain aspects of the system changed somewhat. This can be best seen in the gradual move from a more authentication-centred approach to the identification-focused design presented in the preceding chapters. This was due to the complications of implementing continuous and secure authentications by a third-party platform and the difficulties in restricting the device owner's access to the physical characteristics that are essential for PUFs. Authentications of devices can still be handled with the traditional public-key encryption method, of course, if the device owner can be trusted and keeps the private keys to his devices secure. This shift in focus did not diminish the design's capability of achieving the defined goals, but pursuing that extra layer of security did consume considerable resources.

The implemented prototype that was developed for this thesis does work very well regardless however and achieves the set task in its entirety. As we have mentioned in the evaluation, the design has many positive aspects going for it and performs remarkably well overall.

In conclusion, this thesis describes a design that is capable of identifying IoT-connected devices using PUFs and smart contracts. It is unique in the way it combines these technologies into one functional prototype and implements industry standards to make it accessible and usable easily by any interacting parties.

Bibliography

[1] Dahlqvist Fredrik, Patel Mark, Rajko Alexander, and Shulman Jonathan. Growing opportunities in the Internet of Things. https://www.mckinsey.com/industries/private-equity-and-principal-investors/our-insights/growing-opportunities-in-the-internet-of-things, Jul 2019. Accessed: 09.02.2020.

[2] Mikhail Platonov, Josef Hlavác, and Róbert Lórencz. Using power-up SRAM state of Atmel ATmega1284P microcontrollers as physical unclonable function for key generation and chip identification. *Information Security Journal: A Global Perspective*, 22(5-6):244–250, Feb 2013.

[3] Blaise Gassend, Dwaine Clarke, Marten Van Dijk, and Srinivas Devadas. Silicon physical random functions. *Proceedings of the 9th ACM conference on Computer and communications security - CCS*, 2002.

[4] Yevgeniy Dodis, Rafail Ostrovsky, Leonid Reyzin, and Adam Smith. Fuzzy extractors: How to generate strong keys from biometrics and other noisy data. *SIAM Journal on Computing*, 38(1):97–139, 2008.

[5] D.e. Holcomb, W.p. Burleson, and K. Fu. Power-up SRAM state as an identifying fingerprint and source of true random numbers. *IEEE Transactions on Computers*, 58(9):1198–1210, 2009.

[6] Justine Humenansky. The impact of digital identity. https://medium.com/blockchain-at-berkeley/the-impact-of-digital-identity-9eed5b0c3016, Nov 2018. Accessed: 11.04.2020.

[7] EthHub. Identity standards. https://docs.ethhub.io/built-on-ethereum/identity/ERC-EIP/. Accessed: 11.04.2020.

[8] Fabian Vogelsteller. ERC: Key manager · issue #734 · ethereum/EIPs. https://github.com/ethereum/EIPs/issues/734, Oct 2017. Accessed: 11.04.2020.

[9] Fabian Vogelsteller. ERC: Claim holder · issue #735 · ethereum/EIPs. https://github.com/ethereum/EIPs/issues/735, Oct 2017. Accessed: 11.04.2020.

[10] ERC725 Alliance. ERC-725 ethereum identity standard. https://erc725alliance.org/. Accessed: 11.04.2020.

[11] Urbi Chatterjee, Rajat Subhra Chakraborty, and Debdeep Mukhopadhyay. A PUF-based secure communication protocol for IoT. *ACM Transactions on Embedded Computing Systems*, 16(3):1–25, Jul 2017.

[12] Satoshi Nakamoto. Bitcoin: A peer-to-peer electronic cash system. `https://bitcoin.org/bitcoin.pdf`, 2008. Accessed: 10.04.2020.

[13] Uzair Javaid, Muhammad Naveed Aman, and Biplab Sikdar. BlockPro: Blockchain based data provenance and integrity for secure IoT environments. *Proceedings of the 1st Workshop on Blockchain-enabled Networked Sensor Systems - BlockSys18*, 2018.

[14] Hiroshi Watanabe. Can blockchain protect Internet-of-Things?, Jul 2018.

[15] Ujjwal Guin, Pinchen Cui, and Anthony Skjellum. Ensuring proof-of-authenticity of IoT edge devices using blockchain technology. *2018 IEEE International Conference on Internet of Things (iThings) and IEEE Green Computing and Communications (GreenCom) and IEEE Cyber, Physical and Social Computing (CPSCom) and IEEE Smart Data (SmartData)*, 2018.

[16] Lydia Negka, Georgios Gketsios, Nikolaos A. Anagnostopoulos, Georgios Spathoulas, Athanasios Kakarountas, and Stefan Katzenbeisser. Employing blockchain and physical unclonable functions for counterfeit IoT devices detection. *Proceedings of the International Conference on Omni-Layer Intelligent Systems - COINS 19*, 2019.

[17] Django Software Foundation. Django. `https://pypi.org/project/Django/`. Accessed: 21.02.2020.

[18] Tom Christie. Django REST framework. `https://pypi.org/project/djangorestframework/`. Accessed: 21.02.2020.

[19] David Sanders. Django REST framework - simple JWT. `https://pypi.org/project/djangorestframework-simplejwt/`. Accessed: 21.02.2020.

[20] MetaMask. `https://metamask.io/`. Accessed: 14.04.2020.

[21] Arduino create editor. `https://create.arduino.cc/editor/`. Accessed: 17.04.2020.

[22] Carter Yagemann. fuzzy-extractor. `https://pypi.org/project/fuzzy-extractor/`. Accessed: 21.02.2020.

[23] Rinkeby etherscan. `https://rinkeby.etherscan.io/`. Accessed: 19.04.2020.

[24] Ameer Rosic. What are smart contracts? [Ultimate beginner's guide to smart contracts]. `https://blockgeeks.com/guides/smart-contracts/`, 2016. Accessed: 20.04.2020.

[25] Fractal. trustfractal/erc725. `https://github.com/trustfractal/erc725`. Accessed: 20.04.2020.

[26] Julio Santos. First impressions with ERC 725 and ERC 735 - identity and claims. https://hackernoon.com/first-impressions-with-erc-725-and-erc-735-identity-and-claims-4a87ff2509c9, Jun 2018. Accessed: 20.04.2020.

[27] IBM Knowledge Center. Identification and authentication. https://www.ibm.com/support/knowledgecenter/SSFKSJ_7.5.0/com.ibm.mq.sec.doc/q009740_.htm. Accessed: 28.04.2020.

[28] Ameer Rosic. What is blockchain technology? A step-by-step guide for beginners. https://blockgeeks.com/guides/what-is-blockchain-technology/, 2016. Accessed: 28.04.2020.

[29] Jen Clark. What is the Internet of Things? https://www.ibm.com/blogs/internet-of-things/what-is-the-iot/, Nov 2016. Accessed: 28.04.2020.

[30] LexisNexis. Kyc: What is know your customer?: A definition. https://internationalsales.lexisnexis.com/glossary/compliance/kyc-know-your-customer. Accessed: 28.04.2020.

Abbreviations

ABI	Application Binary Interface
API	Application Programming Interface
CRP	Challenge-Response Pair
dApp	decentralized Application
DSA	Digital Signature Algorithm
ECDSA	Elliptic Curve Digital Signature Algorithm
EIP	Ethereum Improvement Proposal
ERC	Ethereum Request for Comments
FPGA	Field Programmable Gate Array
GUI	Graphical User Interface
HTTP	HyperText Transfer Protocol
IC	Integrated Circuit
IDE	Integrated Development Environment
IDRAM	Identification Random-Access Memory
IoT	Internet of Things
JSON	JavaScript Object Notation
JWT	JSON Web Token
KYC	Know Your Customer
KYD	Know Your Device
PUF	Physical Unclonable Function
RAM	Random-Access Memory
REST	Representational State Transfer
RSA	Rivest–Shamir–Adleman
SC	Smart Contract
SRAM	Static Random-Access Memory
SSH	Secure Shell
URL	Uniform Resource Locator

Glossary

Authentication is the ability to prove that an entity is genuinely whom it claims to be [27]. E.g. in public-key cryptography, entities prove their identity by holding their private key and being able to decrypt messages that were encrypted with the corresponding published public key.

Blockchain refers to a time-stamped series of immutable records of data that is managed by a cluster of computers not owned by any single entity. Each of these blocks of data (i.e. block) is secured and bound to each other using cryptographic principles (i.e. chain) [28].

Identification is the ability to uniquely identify an entity within a system or an application that is running in the system [27]. E.g. a user can be identified if he provides the required login information to a system.

Internet of Things is the concept of connecting any device (so long as it has an on/off switch) to the Internet and to other connected devices. The IoT is a giant network of connected things and people – all of which collect and share data about the way they are used and about the environment around them [29].

Know Your Customer is the aspect of due diligence that deals with the precise identification of customers. It involves checking personal and business details in order to exclude negative hits such as sanctions lists, watch lists and PEP lists and to identify ownership relationships and links between companies [30].

Verification is the process of checking whether a particular claim is valid. E.g. border control checks if a person is whom they claim to be by viewing identity documentation issued by a trusted government.

List of Figures

List of Tables

List of Code Listings

Appendix A

Installation Guidelines

This installation guide is based on a Ubuntu Linux distribution, and the required steps may vary for other operating systems. Note, that at the time this thesis was written there was a compatibility problem for one of the essential Python packages with Windows and the project was therefore moved to Linux.

A.1 Requirements

Before we get started with the installation steps for the KYD platform, we need to ensure the development environment is set-up with the required programming languages and frameworks. Ubuntu includes many of these by default so installation may not be required for all of the following requirements.

1. **Python 3**

   If Python is not yet installed, please follow this guide to get Python 3.x installed on your system: `https://wiki.python.org/moin/BeginnersGuide/Download`

2. **venv**

   venv allows you to set up virtual environments using the command line, and will be used to set-up the KYD server environment. Install venv by entering the following command in your terminal.

   ```
   $ sudo apt-get install python3-venv
   ```

3. **Node.js**

   If Node.js is not yet installed on your system, please download it from here:
   `https://nodejs.org/en/download/`
   Node.js should also include the npm package manager, so check if it is installed by entering the following command in your terminal.

   ```
   $ npm -version
   ```

4. **Angular CLI**

   To run Angular applications the Angular CLI is required. It can be installed using the npm package manager by entering the following command in your terminal.

```
$ npm install -g @angular/cli
```

5. **Git**

   If Git is not yet installed, please follow this guide to get Git set up on your system: `https://git-scm.com/book/en/v2/Getting-Started-Installing-Git`

A.2 KYD Server

Setting up the KYD server environment is achieved using console commands in your terminal. Navigate to a directory of your choosing to hold the source files and execute the following steps.

1. **Cloning the GitHub Repository**

   Download the source files for the KYD server by cloning the repository from GitHub using the following console command.

```
$ git clone https://github.com/bjeffr/kyd_service.git
```

2. **Setting up a virtual environment**

   To set up a virtual environment, where all the required packages will be installed, first navigate to the `kyd_service` project folder. Then enter the following two commands in the terminal.

```
/kyd_service$ python3 -m venv venv
/kyd_service$ source venv/bin/activate
```

3. **Install the required Python packages**

```
/kyd_service$ pip install -r requirements.txt
```

4. **Create the Database**

   The Environment for the server should now be set up. Create the database for the KYD server by running the following console command.

```
/kyd_service$ python manage.py migrate
```

5. **Starting the server**

   The set-up for the KYD server is now complete. You can start the server by entering the following console command. Keep the server running while using the KYD platform.

```
/kyd_service$ python manage.py runserver
```

A.3 KYD Web Application

To get the web application set up correctly, some additional steps are required. Besides the console commands outlined below, accounts for MetaMask and Infura will have to be created. To get started, navigate back to the directory where you want your projects to be stored and execute the following steps.

1. **Cloning the GitHub Repository**

   Download the source files for the KYD web application by cloning the repository from GitHub using the following console command.

   ```
   $ git clone https://github.com/bjeffr/kyd-gui.git
   ```

2. **Installing the required packages**

   To install the required packages navigate to the `kyd-gui` project folder and run the following command.

   ```
   /kyd-gui$ npm install
   ```

3. **Infura: Ethereum API**

   To connect to the Ethereum network without hosting your own node, sign up with Infura by following the guide below. In step 3, make sure to select the Rinkeby test network unless you want to spend real money. Copy the displayed endpoint URL. `https://blog.infura.io/getting-started-with-infura-28e41844cc89/`

   Then head over to the KYD web application project folder and open the following file: `kyd-gui\src\environments\environment.ts`

   Under `provider` paste the copied endpoint URL.

4. **MetaMask Set-up**

   To set-up the MetaMask account for the KYD platform follow the guide below. Note that it would make sense to use a different account for using the platform to represent the device owner, so you will probably go through this process twice. Make sure to copy the 12-word seed phrase (mnemonic) as we will need it in the next step. `https://blog.infura.io/getting-started-with-infura-28e41844cc89/`

   In the same `environment.ts` file paste the seed phrase under `mnemonic`. Then copy the account address from MetaMask, but make sure the correct network is selected (e.g. Rinkeby Test Network). Then paste the account address under `walletAddress`.

   Save the changes made to the `environment.ts` file.

5. **Adding Ether to your Ethereum Account**

   Deploying contracts on the Ethereum network costs money in the form of Ether tokens. Approximately 0.006 Ether will be required for the following step, so make sure you have sufficient funds. For the Rinkeby Test Network you can add Ether using the faucet available here: `https://faucet.rinkeby.io/`

6. **Deploying the identity smart contract for the KYD platform**

   To deploy the identity smart contract for the KYD platform to the Ethereum blockchain, navigate to this folder: `kyd-gui\src\app\scripts`

   Then run the `deploy.js` script by entering the following console command in the terminal.

```
/kyd-gui/src/app/scripts$ node deploy.js
```

   This step can take a minute, but when finished you should see something similar to this:

```
Deploying from: 0xFc869DE19bEfb3DC9d71b4b65ad602990AC2831d
SC deployed to: 0xD2b3ed4dAf8B41730a0F33E13ABA8d15031723aF
```

   The first line shows your account address, where the smart contract was deployed from. The second line shows the smart contract's Ethereum address. Copy this address and paste it in the `environment.ts` file under `contractAddress`. Make sure to save the changes made to the file.

7. **Building and serving the web application**

   The set-up for the KYD web application is now complete. You can serve the web application by entering the following console command from the project folder. Keep the application running while using the KYD platform.

```
/kyd-gui$ ng serve
```

Appendix B

Contents of the CD

- Final thesis PDF
- LaTeX source code
- KYD server repository (*kyd_service*)
- KYD web application repository (*kyd-gui*)
- Arduino SRAM extraction sketch
- Abstract & Zusammenfassung
- Related work papers
- draw.io files for the created diagrams
- Midterm presentation