A Distributed IoT Microservice using a Custom Ethereum Based Security Protocol

Mattias Kallman

Degree Thesis
Information Technology
2021

DEGREE THESIS	
Arcada University of Applied Sciences	
Degree Programme:	Information Technology
Identification number:	7782
Author:	Mattias Kallman
Title:	A Distributed IoT Microservice using a Custom Ethereum Based Security Protocol
Supervisor (Arcada):	Magnus Westerlund, DSc
Commissioned by:	Arcada University of Applied Sciences

Abstract:

With the mass expansion of Internet of Things (IoT) in industry and consumer life, IoT security has become a focal point of research and development. New technologies are enabling unprecedented methods of developing and securing IoT devices. This thesis focuses on studying and applying Web 3.0 technologies in an IoT device and service context while addressing IoT security vulnerabilities through the use of good security design practices. Through the application of Web 3.0 technologies this thesis illustrates the advantages and disadvantages that these technologies offer. The practical implementation utilizes a custom Ethereum based security protocol that enables an IoT device to use a decentralized data network as its dedicated backend infrastructure. The results of the implementation will be analyzed through the lens of security and practicality.

Keywords:	IoT Security, Distributed Security, Web3.0, Ethereum, Blockchain Technology, Decentralized Platform
Number of pages:	64
Language:	English
Date of acceptance:	

EXAMENSARBETE	
Arcada	
Utbildningsprogram:	Informationsteknik
Identifikationsnummer:	7782
Författare:	Mattias Kallman
Arbetets namn:	En distribuerad IoT mikrotjänst som använder ett originellt Ethereum baserat säkerhetsprotokoll
Handledare (Arcada):	Magnus Westerlund, DSc
Uppdragsgivare:	Arcada University of Applied Sciences

Sammandrag:
Det kraftiga anammandet av sakernas internet (IoT), inom industri och hos konsumenter, har lett till att forskning och utveckling av IoT-säkerhet har blivit ett viktigt ämne. Nya teknologier gör det möjligt att på nya sätt säkra och utveckla IoT-apparater. Det här examensarbetet fokuserar på att studera och tillämpa webb 3.0-teknologier i ett IoT-sammanhang. Samtidigt kommer god designpraxis att användas för att bearbeta säkerhetsbrister. Genom tillämpningen av webb 3.0-teknologier kommer detta examensarbete att illustrera för- och nackdelarna med dessa teknologier. Den praktiska tillämpningen i examensarbetet använder ett originellt utvecklat säkerhetsprotokoll som tillämpar Ethereum. Detta protokoll möjliggör att IoT-apparater kan använda ett decentraliserat nätverk som sin underliggande infrastruktur. Resultaten från examensarbetet analyseras utgående från säkerhet och användbarhet.

Nyckelord:	IoT-säkerhet, Distribuerad säkerhet, Web3.0, Ethereum, Blockkedja, Decentraliserad plattform
Sidantal:	64
Språk:	Engelska
Datum för godkännande:	

CONTENTS

1 Introduction . . . 6

1.1 Background . . . 6
1.2 Objective and Purpose . . . 7
1.3 Limitations . . . 8

2 Web 3.0 . . . 10

2.1 Edge Computing . . . 11
2.2 Network Architectures . . . 12
2.2.1 Centralized Networks . . . 12
2.2.2 Decentralized Networks . . . 14
2.2.3 Distributed Networks . . . 15
2.3 Artificial Intelligence . . . 16

3 IoT Security . . . 18

3.1 The State of IoT Security . . . 18
3.2 Vulnerabilities . . . 19
3.3 Best Practices . . . 21
3.4 The Impact of Web 3.0 on IoT Security . . . 24
3.4.1 Edge Computing Security Implications . . . 24
3.4.2 Decentralized Data Network Security Implications . . . 25
3.4.3 AI Security Implications . . . 26

4 Software Design and Architecture . . . 27

4.1 High Level Overview . . . 27
4.2 Utilizing the Custom Ethereum Security Protocol . . . 29
4.3 The IoT Microservice . . . 30
4.4 The Decentralized Application . . . 31

5 Development and Implementation . . . 33

5.1 Hardware Specification . . . 33
5.1.1 Raspberry Pi 2B . . . 33
5.1.2 Sensirion SPS30 Particulate Matter Sensor . . . 33
5.2 SPS30 Python Driver . . . 34
5.2.1 SPS30 Functional Overview . . . 35
5.2.2 SPS30 Sensor Measurement Output Formats . . . 36
5.2.3 SHDLC Protocol . . . 37
5.2.4 Development and Implementation of SPS30 Driver . . . 38
5.3 IoT Microservice . . . 39
5.3.1 Cron Job Sensor Scripts . . . 39
5.3.2 Database Manager . . . 40
5.3.3 Encryption and Encoding Manager . . . 41
5.3.4 Service Manager . . . 43
5.3.5 Installation and Setup . . . 44

5.4 dApp Development . . . 45
5.4.1 dApp Usage . . . 46

6 Results . . . 48

6.1 IoT Microservice Performance . . . 48
6.2 Validation of Sensor Data Quality . . . 49
6.3 IoT Microservice Security Implementations . . . 50

7 Conclusions . . . 53

7.1 Further development and research . . . 54

References . . . 55

Appendix 1 Swedish Summary . . . 59

FIGURES

Figure 1. Centralized, decentralized and distributed network architectures. 12
Figure 2. High Level Overview of how the software components relate and interact. 28
Figure 3. CESP functional overview. 29
Figure 4. Functional Overview of the Middleware and IoT-Microservice. 31
Figure 5. Functional Overview of the Decentralized Application. 32
Figure 6. UML Diagram of the SPS30 Driver library. 35
Figure 7. Diagram of Operating Modes of the Sps30 Sensor. 35
Figure 8. MOSI and MISO Frame Structure 38
Figure 9. UML Diagram of the database manager module. 40
Figure 10. Hybrid encryption model using symmetric and asymmetric encryption. 41
Figure 11. UML Diagram of Encryptor and Decryptor Classes. 42
Figure 12. The UML diagram of the service manager functionality. 43
Figure 13. The GUI of the dApp. 46
Figure 14. SPS30 data measurements: comparing mean values to unaggregated values. 50

TABLES

Table 1. The project source code. 8
Table 2. OWASP Security Team Top 10 Vulnerabilities of 2018 19
Table 3. IoT Security Foundation’s 14 areas to consider in IoT design. 21
Table 4. SPS30 sensor output formats. (Sensirion 2020 p. 5) 37

1 INTRODUCTION

1.1 Background

The Internet of Things (IoT) is becoming ever more integrated into human society. IoT has already improved how individuals work and is contributing to their overall well-being. Some illustrative examples of these smart devices could be a smart watch that helps track a person's health or smart medical devices like an insulin pump that monitors a person's blood sugar levels and administers doses accordingly. The mass adaptation of IoT has also helped companies improve their businesses by providing insight into different processes where IoT devices have been applied. (Gillis 2020)

The amount of data gathered via IoT is on a monumental new scale and has continued to grow steadily. At the end of 2020 there was estimated to be around 20 to 31 billion IoT devices globally and the number is projected to rise to around 75 billion by the end of 2025 (Vega 2021). New IoT use cases are developed as IoT devices grow in power and costs are reduced. Likewise emerging new technologies are also increasing the use cases for IoT devices. Maayan (2020) groups IoT technology in to five general categories: Consumer IoT, Commercial IoT, Industrial IoT, Infrastructure IoT and Military IoT. This grouping emphasizes the broad application of IoT.

There is a growing concern for the state of IoT security and IoT privacy. As IoT is taken into greater use in many key sectors and application of IoT is adopted in key infrastructure, there is a need for hardened security and better handling of data (Gillis 2020). The OWASP IoT Security Team (OWASP 2018) published a Top 10 vulnerabilities of IoT as of 2018. The case presented by OWASP against current IoT security practice is severe and modern technical solutions to these security problems need to be addressed. With the advent of Web 3.0 technologies, Mersch & Muirhead (2019) have presented new ways of developing IoT services as well as introduced new ways of tackling different security issues related to IoT.

1.2 Objective and Purpose

There are three main goals with this dissertation. Firstly (1), it is to develop a distributed IoT microservice and a decentralized application (dApp) using some of the core concepts from Web 3.0, namely, edge computing and decentralized data networks. The IoT microservice will use in its backend a custom second layer protocol that operates on the Ethereum blockchain. The practical implementation of this dissertation will utilize the protocol developed by Wickström (2020) and as no name has been given to the custom second layer protocol, it will be referenced as the Custom Ethereum Security Protocol (CESP) throughout this thesis. The decentralized application's basic premise is to fetch data that is stored on the decentralized data network and present the data to the user. The data will be visualized through the use of graphs and can also be downloaded for further processing.

Secondly (2), this thesis will analyze how Web 3.0 technologies impact the security aspects of the IoT microservice. The focus will be on edge computing and decentralized data networks as these were the two main technologies explored in the practical implementation. Artificial intelligence (AI), while a key part of Web 3.0 technologies (Mersch & Muirhead 2019), will only be presented in the theoretical sections as no AI was applied to the practical implementation.

The Third (3) and final objective of this thesis aims to evaluate the impact of security through design and its implications for IoT devices.

The source code for the practical implementation can be found at the following GitHub repositories found in table 1.

Project Source Code	
IoT-Microservice:	https://github.com/kallmanm/IoT-Microservice
Decentralized Application:	https://github.com/kallmanm/IoT-Dapp
Custom Ethereum Security Protocol	
Smart contract backend:	https://github.com/wickstjo/oracle-manager
IoT middleware:	https://github.com/wickstjo/iot-manager
Frontend application:	https://github.com/wickstjo/distributed-task-manager

Table 1. The project source code.

1.3 Limitations

This thesis was made at the request of Arcada University of Applied Sciences' Department of Business and Analytics. For this reason, there are key limiting factors in the practical implementation of this thesis, specifically the use of certain hardware and software. All hardware was provided by Arcada resulting in the need to find a technical solution with the provided resources. The IoT hardware provided were a Raspberry Pi 2B and a Sensirion SPS30 Particulate Matter Sensor. They will be presented in detail in section 5.1 of this thesis.

The software requirement for the practical implementation was the use of the CESP as the infrastructure for the IoT microservice. The software developed by Wickström will be presented in detail in section 4.2. The IoT microservice and the dApp were both developed in Python 3.8 and the use of base Python libraries was prioritized to minimize the strain on device resources.

The theoretical focus of this thesis will be on IoT security, Web 3.0 technologies and how they can improve IoT device security, specifically edge computing and decentralized data networks were applied in the practical implementation. Artificial Intelligence will be presented in the theoretical sections, but no implementation of AI was applied in the practical implementation to avoid scope creep. This focus on IoT security and Web 3.0

technologies impacted the overall development for the IoT microservice and dApp.

2 WEB 3.0

Web 1.0 is the first iteration of the World Wide Web and is described by Gettings (2007) as only granting access to searching and reading information from the Web. It was a very primitive version of the internet compared to what is available today in terms of user interaction and functionality. One of the more revolutionizing aspects of Web 1.0 was the introduction of online shopping. It enabled businesses to reach further than what was previously geographically possible. This technology enabled the advent of e-commerce. (Gettings 2007)

Web 2.0 has been described by Tim Berners-Lee as the 'read-write' Web. It transformed the Web and was defined by innovations like the mobile internet, social networks and cloud services. The adaptation of the mobile internet has broadened accessibility to the Web and has greatly increased the total amount of users. The emergence of social networks became possible with lower internet access costs and greater accessibility due in part to the mobile internet. As the Web grew and its use increased, more actors needed websites leading to the development of new technologies. This led to the emergence of cloud services, web hosting and web application hosting. Software as a Service (SaaS) models emerged making it easier to set up web services. So far, each iteration of the Web has been revolutionizing in its own way having a profound impact on human society and the Web. (Mersch & Muirhead 2019)

This leads us to the current iteration of the Web, that is Web 3.0. Web 3.0 is a term coined by Tim Berners-Lee who in fact is the inventor of the World Wide Web. The term Web 3.0 describes the next phase of the evolution of the Web and has been summarized as the "read-write-execute" Web. It aims to make the Web open, trustless and permissionless. **Open** in the sense that software used in networks are developed from open-source material and accessible to all. This in turn grants an open and transparent view into software source code. Anyone using an open-source application can access the source code to study how the application is built and how it uses personal data.

Networks will be **trustless** allowing participants to act publicly or privately on the net-

work depending on their needs. A trustless network is also built on the idea that there will be no need for a trusted third-party to govern the network. Currently the owners of networks dictate the terms of service for users on the networks. The last aspect of Web 3.0, **permissionless**, stipulates that both users and suppliers can participate in networks without the need for authorization from a governing third-party. (Mersch & Muirhead 2019)

Mersch & Muirhead (2019) highlight that three key technologies are the foundation on which Web 3.0 will be built. They are edge computing, decentralized data networks and artificial intelligence.

2.1 Edge Computing

Edge computing can be described as distributed computation and data storage. It has the ability to enable computation and data storage closer to where it is created. In a large data network where traffic and data usage are continuously growing, the ability to offload computational tasks and aggregate data while lowering bandwidth usage is appealing. This has the added benefit of lowering latency in the network as data access and storage is happening on site in the nodes of the network rather than in a centralized data processing center. Edge computing thus reduces the overall data transferred to the main servers within the network and reduces the need for centralized computation. (IEEE 2019)

IEEE (2019) also emphasizes that edge computing has great potential at reducing bandwidth usage and directly improving reliability for IoT devices with limited network connectivity. This is in part due to the lowered cost of hardware and the increased processing power (Leonard 2019).

Utilizing edge computing within a system comes at a cost. As machines in the network become more powerful and are able to perform more complex tasks, the complexity of the underlying system grows. This in turn results in that system maintenance cost go up, configuration time is increased and deployment can become more demanding. Other than just the complexity increasing, the security of a network applying edge computing needs to be addressed simultaneously. Edge computing by its design leads towards a

more distributed computing architecture. This can create more advanced entry points to the network leading to more substantial attack vectors. Edge computing security, IoT security aspects and device hardening will be discussed in detail in section 3. The benefits and disadvantages of edge computing must therefore be weighed in each individual use case to determine if the application of edge computing is suitable. (IEEE 2019)

2.2 Network Architectures

There are three main types of network architectures that make up the majority of the internet. They consist of centralized, decentralized and distributed networks. This section will present what their defining features are, what limitations they have, what are the advantages of using a certain type of network as well as what inherent disadvantages they present. Figure 1 illustrates the general idea of how the different network models are structured and how the relationship between server(s) and clients in the different networks are organized.

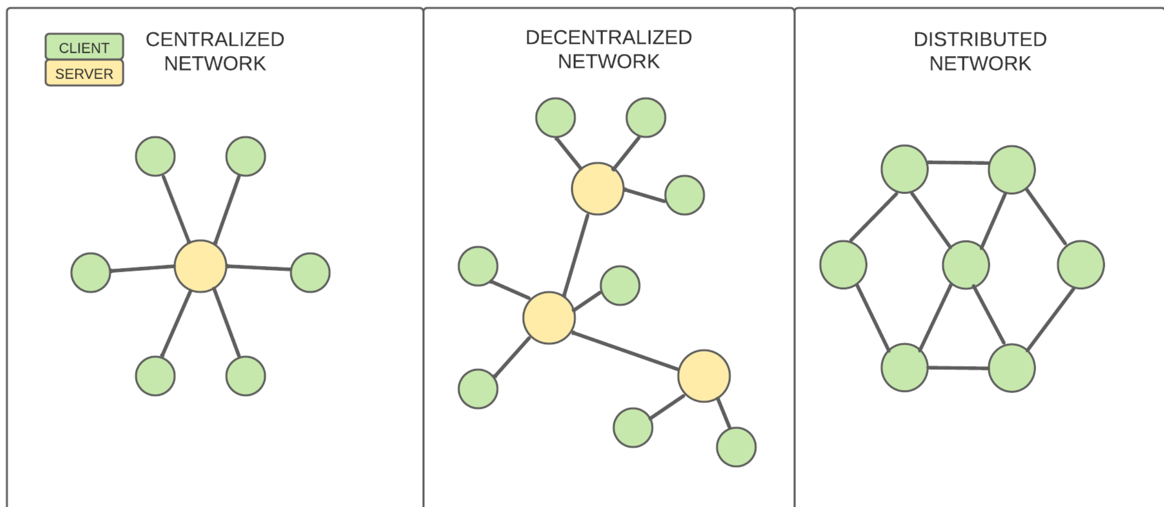

Figure 1. Centralized, decentralized and distributed network architectures.

2.2.1 Centralized Networks

Centralized networks are based around the server-client architecture where one or more clients are connected to one main server. The server functions as the sole provider of data to the network and it is the central authority that delegates and controls the system. Clients

that are connected to the network can consume data and services by issuing requests to the server. (Hooda 2019)

Some key defining characteristics of centralized networks are that there is only one central node that coordinates all activity on the network. Centralized networks have one master clock that all clients within the network sync to, this enables coordinated communication. Defining limitations with centralized networks are that they can only scale vertically. Vertical scaling implies that the core server's hardware is upgraded to increase performance. Examples of this would be upgrading the hard drive, processor or other key component in the server computer. An example of horizontal scaling would be installing a new server unit to work in tandem with the original server. This, however, is not possible in traditional centralized data networks as doing so would change the architecture from centralized to a distributed network. (Hooda 2019)

The advantages that centralized networks offer are many. Seal (2020) argues that the primary benefit with centralized networks is the efficiency they offer compared to other network types. It is easier to maintain and keep the network updated when it comes to hardware and software updates, as the network server is in one physical location. This also has the added benefit of making it easier to secure the network from physical intrusion (Hooda 2019). Hooda (2019) also points out that there is a lower initial cost in setting up a centralized network compared to its decentralized and distributed counterparts.

There are naturally disadvantages with using centralized networks. Seal (2020) and Hooda (2019) both emphasize that centralized networks are prone to forming bottlenecks when traffic spikes or when the server is under-dimensioned. Another risk with centralized networks is that they are highly dependent on network connectivity. If the connection is lost, the server's ability to answer requests is completely lost until connectivity is restored. Seal (2020) also points out that servers of centralized networks are prime targets for hackers as it is the best way to gain access to the whole network and its nodes. This type of breach compromises all applications on the network.

2.2.2 Decentralized Networks

Decentralized networks are vastly different from their centralized counterparts. As the name implies, there is no central server that dictates the network rules of engagement, rather it is the nodes within the network that contribute their local machine resources to manage and power the network. Each node has autonomy within the network, meaning that they can themselves choose exactly how much they contribute towards data storage and processing. (Seal 2020)

One interesting and defining aspect of decentralized networks is that they can scale both vertically and horizontally. Vertically by upgrading the hardware on the nodes and horizontally by adding more nodes to the network (Hooda 2019). Vermaak (2020) points out that decentralization is not a binary state but a spectrum. Decentralized networks can vary in how decentralized they are depending on the specific network architecture. Popular examples of decentralized networks are Bitcoin and Ethereum. These networks are governed by users, but the users adhere to the same 'rule of law' ie. the consesnsus algorithms.

Since decentralized networks have such a completely different architecture than its centralized counterparts, there are some interesting advantages and disadvantages that arise due to their intrinsic design. Firstly, a decentralized network is extremely robust. Its robust nature stems from the fact that the network is comprised of many nodes. If a node is taken offline it will not have any adverse effects on the network assuming that the network is comprised of enough nodes (Seal 2020). The base architecture of the network also makes scaling easier as it can always scale horizontally. The same aspect that makes scaling easier also severely reduces the risk of bottlenecks arising since network traffic is handled by a multitude of participating nodes (Seal 2020). Decentralized networks are also resilient to attacks that attempt to compromise its shared network. (Vermaak 2020)

For every novel feature available in decentralized networks, there tends to be an equivalent disadvantage. Managing and maintaining the network is more complex than in a

centralized network as the inherent complexity of decentralized networks makes the network as a whole slower and more energy intensive (Hooda 2019). The slowness comes from verification processes that utilize algorithmic computation to maintain and validate the network's legitimacy. This payoff in turn is the price for security and fairness in decentralized networks (Vermaak 2020). As Vermaak (2020) and Hooda (2019) point out, a decentralized network is only robust if the network has reached critical mass. This illustrates one of the key drawbacks for decentralized networks, if the network is not big enough it will lack in security and computational power.

2.2.3 Distributed Networks

The last network presented is the distributed network. While distributed and decentralized networks may seem similar, there are subtle differences between them. According to Eagar (2017) the key difference is that distributed networks have a centralized data hierarchy similar to that of a centralized network, but the processes and workloads within the network are distributed between the different nodes of the network. A distributed network has complete system information, where a decentralized network does not. Melendez (2019) points out that a distributed network may seem like one coherent system to an outsider looking in, but is in fact many small systems working together to maintain the system.

There are key advantages to using a distributed network architecture. Some of the primary benefits are the redundancy, resiliency and content distribution that it offers. This is achieved by having many nodes offer the same service in the network and distribute the load accordingly. The distributed nature of the network ensures redundancy measurements and guarantees that services have both high availability and speed. (Melendez 2018)

A concern with distributed networks is that they are complex in design. Just as with decentralized networks, this has security implications that similarly makes it harder to secure the system as a whole. This added complexity also has the downside of increasing network maintenance. (Melendez 2018)

2.3 Artificial Intelligence

The scientific field of artificial intelligence studies machine intelligence and its attempt to emulate human intelligence in different machine-driven contexts. The application of AI has exploded in recent years as people and corporations want to utilize the benefits of machine-driven intelligence. The application of AI is interesting as it shows promise and benefits in improving performance and security for IoT devices. The six different branches of AI are: Machine Learning (ML), Neural Networks (NN), Expert Systems (ES), Robotics, Natural Language Processing (NLP) and Fuzzy Logic (FL). (Tyagi 2021)

Machine Learning is the process where a program learns to do a task by applying an algorithm and simulating the task many times over. As the program simulates a task, the algorithm model is updated to improve performance between simulations. Tasks can often be simulated millions of times to achieved optimal results and model retraining is standard practice in ML as algorithms are reworked and reapplied. Different algorithms are applied depending on what type of data is available. Supervised Learning (SL) is applied in situations where available data is labelled and defined. SL then tries to find correlations within the data and make decisions accordingly. Unsupervised Learning (UL) uses unlabeled data to try and find correlation within the data. UL often tries to group data into meaningful contexts to draw relevant correlations. Reinforcement Learning (RL) is the process in which a program is trained to perform a task with clearly define rules and parameters. An example application of RL is training a program to play a game like Chess or Go.

A **Neural Network** attempts to apply cognitive science to a machine context through the use of different algorithms. By passing data through the NN model, it aims to identifying the relational context of the data it is trained on. NN models achieve this by emulating the brain's functions by creating its own artificial network of neurons. In a NN context a neuron is made up of a mathematical function. Weather forecasting or financial forecasting are two practical examples of where NN is applied.

Expert Systems Try to emulate the decision-making process of a human expert. This is done through the use of a knowledge database that the ES has access to. The ES ability is directly correlated to the quality of the knowledge database. Word's spellchecker is a good example of this. The more language references the spellchecker has, the better suggestions it is able to provide.

Robotics is an interdisciplinary field that applies AI, electrical and mechanical engineering and many other computer science disciplines that aim to make robots autonomous. Examples of robotics use cases are assembly-line robots and work situations too dangerous or laborious for humans.

Natural Language Processing is the area of AI that tries to teach machines to understand human speech and text. This is done by analyzing and deriving information from text data in the aim to extract meaningful information. Example use cases are speech recognition and sentiment analysis.

Fuzzy Logic is an AI technique that tries to determine a condition's correctness on a spectrum rather than by true or false. Often in real-world applications it can be hard to determine the validity of a condition, by applying a probability spectrum, FL tries to determine the likelihood that something is correct. One application area of FL is in ML as it can try to determine the correctness of an action.

3 IOT SECURITY

Shea & Wigmore (2018) describe IoT security as the process and act of securing and safeguarding IoT devices and the networks they are connected to. IoT devices can be found across a wide spectrum ranging from extremely simple to vastly complex. Simple IoT devices usually consist of a one chip microprocessor or microcontroller that perform simple processes and tasks. At the other end of the spectrum we have high powered IoT machines working in clusters performing complex tasks (IoT Security Foundation 2019 p. 4). Another aspect contributing to the complexity of IoT security is that different industries and sectors have different security needs. Together these factors have made it difficult to establish a best practice standard for IoT devices.

3.1 The State of IoT Security

The proliferation of IoT devices has led to data collection and processing on a monumental new scale. IoT security has been pushed into the limelight for this reason as current practices of IoT development do not adhere to software security standards. If current trends hold, the rate of IoT adaptation will only increase emphasizing the need for better standards and practices as more IoT devices connect to the internet (Vega 2021).

Poor design of IoT devices can have dire consequences to consumers and vendors. Malicious parties take advantage of poor design that leaves the device open for attacks. The consequences of these attacks vary greatly depending on the specific use case of the IoT device, ranging from simple inconveniences to severe risks to national security. This reiterates the importance of proper design from the beginning of the development process as all too often security aspects are first only considered once development is completed. Therefore, there needs to be a shift in IoT development processes that tackle these fundamental flaws. (IoT Security Foundation 2019 p. 4)

3.2 Vulnerabilities

As IoT devices are riddled with different vulnerabilities, identifying the problem areas is the first step in improving security. The Open Web Application Security Project (OWASP) is a community of developers focused on improving software security with the use of open-source software and standards. Their main tools for achieving these goals are through information, education and development of best practices. OWASP (2018) warns of severe security risks affecting IoT and have presented their findings in their report 'OWASP Top 10 Internet of Things 2018'. The list of security risks can be seen in table 2.

#	**Top 10 IoT Vulnerabilities:**
1	Weak, Guessable, or Hard-coded Passwords
2	Insecure Network Services
3	Insecure Ecosystem Interfaces
4	Lack of Secure Update Mechanism
5	Use of Insecure or Outdated Components
6	Insufficient Privacy Protection
7	Insecure Data Transfer and Storage
8	Lack of Device Management
9	Insecure Default Settings
10	Lack of Physical Hardening

Table 2. OWASP Security Team Top 10 Vulnerabilities of 2018

Weak or guessable passwords are easy to brute-force indicating that they lack complexity as a machine can guess them relatively quickly by applying a process of elimination. Often passwords are unchanged since production and producers use publicly available password lists to secure the device. **Hard-coded passwords** implies saving password in "plain text" in the source code rather than encrypting them. The risk here is if the source code becomes available to the public it can easily be discovered.

Insecure network service aspects relates to all other services running on the device that

are non-essential for its purpose. Non-essential services that are exposed to the internet are extremely detrimental as the can compromise information communications on the devices leaving room for unauthorized access locally and remotely.

Insecure ecosystem interfaces relates to outside factors that could compromise a device's security. This could be, for example, an insecure web interface, API, cloud or mobile interface that communicates with the device. Improper authentication and authorization, weak or no encryption and insufficient input/output filtering are some of the more common problems with these interfaces.

Lack of secure update mechanism describes the device's inability to securely update, rollback or validate different software/firmware updates due to insecure delivery processes.

Use of insecure or outdated components highlights the need for proper software management on the device. The use of deprecated or insecure software can lead to the device becoming compromised. All layers of software are vulnerable if not properly maintained. This includes BIOS, firmware, operating systems, applications and third-party software and hardware.

Insufficient privacy protection addresses the problem of mismanagement of user information in the device ecosystem. Types of mismanagement of data include insecurely handled data, improperly handled data or data handled without permission. The main infringement in **insecure data transfer and storage** is the lack of encryption or access control to sensitive data within the device ecosystem.

Lack of device management calls attention to how all aspects of the device's life cycle is managed. This includes production deployment, device update and asset management, device system monitoring and device dismantling.

Devices installed with **insecure default settings** are vulnerable to supply-chain attacks as default settings would be available to the attacker. The inability to modify device configurations also relates to this topic as the ability to modify configurations adds a

degree of security to the system.

Lack of physical hardening pertains to how best to secure the device physically. This could be for example removing unnecessary access ports to the device, making the device hard to access or creating a robust case for protecting the device from physical access or tampering.

3.3 Best Practices

The IoT Security Foundation (2019) has addressed these security issues by developing a best practices guide that highlights 14 key areas to consider in IoT design. This guide directly addresses the security concerns presented by OWASP (2018). Table 3 presents the 14 areas provided by IoT Security Foundation.

#	Key Areas in IoT Design:
1	Classification of Data
2	Physical Security
3	Device Secure Boot
4	Secure Operating System
5	Application Security
6	Credential Management
7	Encryption
8	Network Connections
9	Securing Software Updates
10	Logging
11	Software Update Policy
12	Assessing a Secure Boot Process
13	Software Image and Update Signing
14	Side-Channel Attacks

Table 3. IoT Security Foundation's 14 areas to consider in IoT design.

The area of **data classification** tackles the security aspect of creating good data structures.

This has the added benefit of making it easier to manage device and application data. Good practices dictate that the data used in an IoT system should be defined by a schema and documented. All data should be evaluated and ranked according to sensitivity and security implications.

Design aspects of **physical security** try to negate tampering and access to the IoT device through physical means. For example, this includes considering the location of device deployment, what kind of access is needed after development and securing the device through protective casing.

Having a **secure boot process** and **operating system (OS)** on an IoT device is an important security aspect to consider. An unmonitored boot process can lead to rogue software being run unnoticed on device start-up. Monitoring the boot process adds a needed layer of protection and should thus be considered in the device design process. Operating systems can have inherent weaknesses and strengths. It is therefore prudent to use the latest versions that provide the best security settings. An IoT device OS should only have the minimum amount of software components installed that are needed for it to function according to its purpose. OS access rights and ports should be minimized and disabled according to what serves the device best. This limits access points into the system.

Application security is paramount in securing the device system as the main application of the device is one of the main access points of the system and key data is collected and processed in it. Therefore, good software design practices that considers security from the beginning is important. Documenting what and how security is built into the software helps managing and maintaining device security.

Credential management is the process of controlling 'who' has access to the device system and 'what' privileges they have. Avoiding factory default users is paramount and managing password, keys and certificates properly helps protect against infiltration. Compromised credentials are one of the easiest ways to gain access to a device.

By using **encryption**, device data can be protected as it is communicated over the networks. Encrypting sensitive data adds a much needed layer of protection making it harder

to eavesdrop on network communication and protects the data if it were to come into the wrong hands. The appropriate level of encryption should be considered for each use case.

Network connections is how the IoT device communicates with the world. This is often done through different network interfaces. Protecting these network connections ensures a higher degree of security as well as minimizing the amount of different network connections.

Securing the software update processes ensures that the device can safely update software. This has traditionally been a problem as IoT devices are often remotely located and do not have remote updating capabilities. By designing secure updating software processes that allow remote updating of software, firmware and OS, IoT devices can maintain their security.

Logging enables diagnostics of device system and security. Managing good logging practices and securing logging processes is an important design aspect to consider. Logging only what is important is a good design practice as IoT devices often have limited storage. What the device writes to the logs is of equal importance as this may have legal implications on the device depending on in what geographical region it is active in.

The processes and mechanisms that perform updates on IoT devices is its **software update policy**. These need to be robust, secure and reliable as devices that do not update have an increased risk of vulnerabilities. Limiting factors to this process are often the hardware and location of a device, making it impractical or not possible to updated remotely. Therefore, the design process of the device and its software can take these aspects into consideration from the outset.

Assessing a secure boot process is a key component to loading up the IoT device system. The probability of malicious attacks can be reduced by having a secure process that validates step-by-step the boot process.

Using cryptographic signatures for software updates establishes the authenticity and in-

tegrity of the update. This is one way of improving the **software image and update signing** process. This process helps ensure that installed updates are secure and from a trusted source.

A **side-channel attack** is when a third-party observes and eavesdrops on the changes in the state of an IoT device system. By eavesdropping and studying device telemetry, the attackers deduce information from the changes in the system in hopes of exploiting it. Side-channel attacks are often complex and hard to fully mitigate, it is therefore important to properly evaluate the impact of such a breach and prevent them when possible.

OWASP and the IoT Security Foundation have presented an excellent starting point for improving IoT security as they together present 'what is wrong with IoT' and 'what to consider' in IoT development. Both sources will be used as a baseline to discuss and analyze if Web 3.0 technologies can provide a solution for the applicable items in either list. A final remark to remember is that many devices are not able to satisfy every requirement due to their real-world constraints, but should try to the best of their ability to give a reasonable amount of security for their situations.

3.4 The Impact of Web 3.0 on IoT Security

Edge computing, decentralized data networks and AI do not improve or worsen IoT security by themselves. They are tools that enable new possibilities and features and thus bring advantages and disadvantages. How and when these technologies are applied should be determined by the use case as each technology might have unforeseen consequences to the application.

3.4.1 Edge Computing Security Implications

The application of edge computing creates new challenges for IoT security. Edge computing nodes are empowered with high computational power and have direct access to their networks. This creates additional security implications as IoT devices are already vulnerable. Xiao et al. (2019) state that 82 percent of all attacks on edge computing devices fall into the following categories: distributed denial of service attacks, side-channel

attacks, malware injection attacks, and authentication and authorization attacks. As side-channel attacks and authentication and authorization attacks already are a severe threat to IoT devices, this overlap creates an even bigger security concern. Xiao et al. goes on to describe and discuss how best to tackle these security issues through the use of good software design and security practices that correlates to device hardening, the proper securing of device configuration and proper securing of device applications. (Xiao et al. 2019)

3.4.2 Decentralized Data Network Security Implications

Decentralized data networks present new possibilities for IoT device security as the technology is so vastly different compared to centralized networks. Wickström et al. (2021) have presented a novel solution that tackles many IoT security problems through the use of Ethereum smart contracts and applying them as a dedicated backend system. Their protocol enforces a security model that helps maintain distributed IoT networks. Wickström et al.'s solution addresses the following items from the OWASP top 10 Internet of Things 2018: insecure network services, insecure ecosystem interface, insufficient privacy protection, insecure data transfer and storage and lack of device management. (Wickström et al. 2021)

The smart contracts in Wickström et al. protocol are executed on the Ethereum Virtual Machine (EVM). Users and devices are authenticated and authorized by the EVM and any transaction written to these contracts must first be evaluated by the EVM. As smart contracts are immutable by design, this creates an immutable activity log for the protocol keeping track of all activity. Through the use of the protocol, services registered to the protocol can safely be updated and patched if the need arises. The design choice to use the EVM to validate all activity address the issues of insecure network services, insecure ecosystem interface and lack of device management. (Wickström et al. 2021)

The privacy in instantiated network devices is maintained by not storing geographical or physical device information in its contracts. All transmitted data is end-to-end encrypted guarding against eavesdropping. In the event that the data is intercepted by a third-party, encryption protects the data from being read. These steps help guard against insufficient

privacy protection and insecure data transfer. (Wickström et al. 2021)

The above-mentioned implementation illustrates how the protocol through the use of a decentralized data network addresses some key security issues with IoT devices. Many of the other design choices of the protocol also improve IoT security through good software and system design, but are not directly related to decentralized data networks. (Wickström et al. 2021)

3.4.3 AI Security Implications

The impact of artificial intelligence on IoT device security is profound and can be applied for either good or nefarious purposes. AI's ability to recognized patterns and find correlations make it an excellent tool for detecting device security weaknesses. By analyzing IoT device activity, different AI techniques and models can be used to detect behavior that might have otherwise gone unnoticed. This same ability to analyze large sets of data make it an excellent tool for performing types of espionage as the AI can learn and probe for system weaknesses. (Sciforce 2020)

Most IoT device telemetry can be quantified and aggregated. AI techniques and models can use this aggregated data as the base for its training data (Raj et al. 2020). After an AI model is trained, it can be used to monitor the IoT device telemetry for malicious activity. Machine learning techniques including supervised learning, unsupervised learning and reinforcement learning have shown promise in helping combat different security issues relating to authentication, access control and malware detection. Likewise, application of neural network models have shown promise in similar areas in detecting malicious activity on IoT devices (Xiao et al. 2018). Comparing the findings of Xiao et al. to the OWASP Top 10 Internet of Things 2018 list, AI techniques can directly respond to the security concerns relating to insecure network services, lack of device management and update mechanisms.

4 SOFTWARE DESIGN AND ARCHITECTURE

4.1 High Level Overview

The practical implementation of this thesis consists of three main software components. Together they make up a software ecosystem that illustrates how a distributed IoT microservice can be created, maintained and consumed. The main software components of the system are:

1. The Custom Ethereum Security Protocol (CESP)
2. The middleware executing the IoT microservice
3. The decentralized application

The CESP functions as the foundation of this implementation. It could be described as the core of this software ecosystem and is the software component that truly makes this a decentralized system. The main sub-components of the CESP are its frontend interface, the interlinked smart contracts that function as the backend system, the middleware that translates requests into actions on the device and a custom program that perform a certain task (Wickström 2020 p. 25). The middleware communicates with the interlinked smart contracts of the CESP and performs assigned tasks on-demand. The CESP will be presented in detail in section 4.2.

The IoT microservice is an application that provides aggregated particulate matter data from the location of the IoT device through the use of a sensor. The IoT microservice functions as an extension of the middleware. Section 4.3 presents the IoT microservice in detail.

The decentralized application (dApp) is a stateless application that accesses and displays data that is fetched from smart contracts. The dApp is presented in detail in section 4.4.

Each component will be expanded on in their respective subsection of this chapter. Figure 2 illustrates a high-level overview of the system architecture.

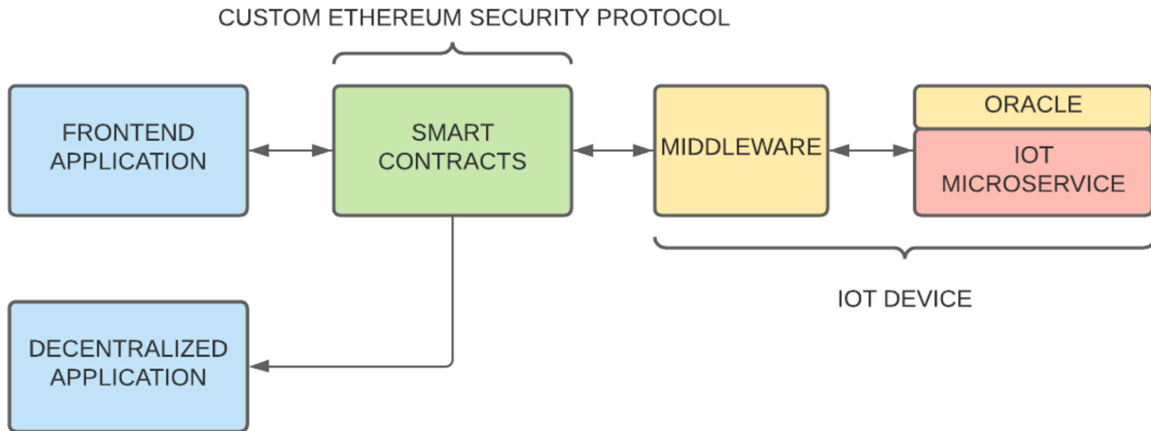

Figure 2. High Level Overview of how the software components relate and interact.

The general process of figure 2 is as follows:

1. A user can register IoT devices of their ownership to the CESP network via the use of the frontend application. Registered users can then register a service via the same frontend application or they can start consuming other existing services. Once a service has been registered to the smart contracts it is possible for other users to consume that registered service by requesting the device to perform it. Only the owner of a service is able to modify it.

2. When a user requests a service, a task request is created and securely assigned to the IoT device. The user petitioning the service also stakes the cost of the service at this point.

3. When the middleware then checks the state status of its smart contracts and detects a change, it processes the requested task and places a stake to guarantee that the service will be fulfilled. If the task is completed the middleware writes the results to its tasks smart contract confirming the transaction. At this point the monetary transaction is completed and payment is processed. If the task is not completed within the agreed upon time, the IoT service loses their stake to the customer requesting the service.

4. When data has been registered to the customer smart contract, it is possible for the user to use the decentralized application to access their information on the smart contract.

4.2 Utilizing the Custom Ethereum Security Protocol

Smart contract and blockchain technology are at their best complex and hard to understand. The CESP is an attempt to streamline their usage and incorporation (Wickström 2020 p. 31). As the CESP consists of many subsystems that utilize smart contracts and blockchain technology, simplifying the user experience makes the concept as a whole more appealing and user friendly.

The CESP has three main components:

1. The backend system
2. The middleware software
3. The frontend interface

Figure 3 illustrates how the three components of the CESP relate and function together. (Wickström 2020 p. 31)

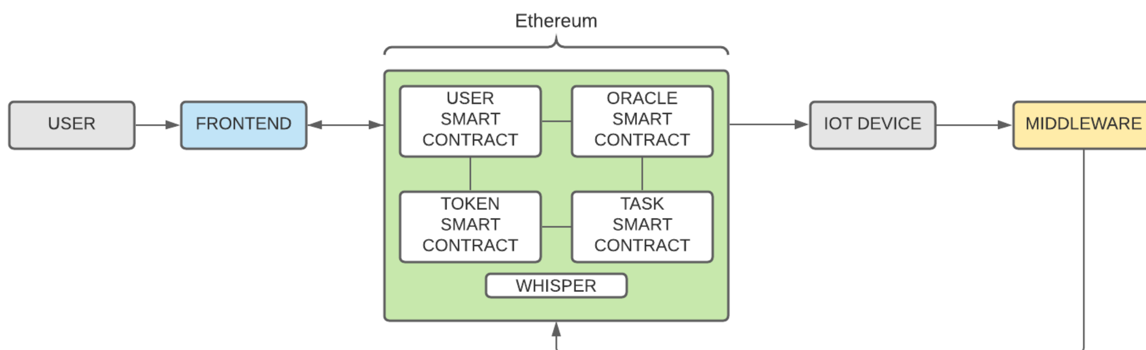

Figure 3. CESP functional overview.

The backend system is made up of four interconnected smart contracts that together constitute the whole backend architecture. The interlinked smart contracts consist of a User Manager, Oracle Manager, Token Manager and Task Manager contract (Wickström 2020

p. 34). The system is autonomous, meaning that the contracts only query each other for validation and it is not possible for a person to interfere with the audit process. The backend system enables user registration, publishing and monetization of oracle services. Users and services need to be registered before they are made publicly available. Ownership of accounts and services is automatically and immutably assigned to the person who paid for their creation through the use of their Ethereum wallet. Monetization is done through a two-way token staking model that demands both user and supplier of the service stake value before a service task is processed. (Wickström 2020 p. 31)

The middleware software's primary purpose is to interpret events on its smart contract. A user can update the contract parameters and the middleware interprets these events into an action and recalibrates itself. The device contract contains a backlog of tasks that the task manager adds tasks to. The middleware is programmed to perform and submit a result for tasks that are added to its backlog. (Wickström 2020 p. 31)

All smart contract functionality is made available through the frontend application. It simplifies the whole process making it easier for users to utilize the CESP.

4.3 The IoT Microservice

The IoT microservice is designed around three main concepts: data collection, data storage and data delivery. Sensor data is periodically collected, aggregated and saved to the IoT device while data requests to the IoT microservice are managed by the middleware software. The microservice parses the parameters passed to it from the middleware to execute a database query and then returns the data to the middleware. Before data is returned to the middleware, the data is encrypted and encoded to ensure security and integrity. The encryption feature and its process are detailed in section 5.3.3. Once the data is returned to the middleware, it proceeds to write the results to its smart contract and the process is completed.

Figure 4 illustrates the functional overview of the middleware software and the IoT microservice. The IoT microservice software implementation will be presented in detail in section 5.3.

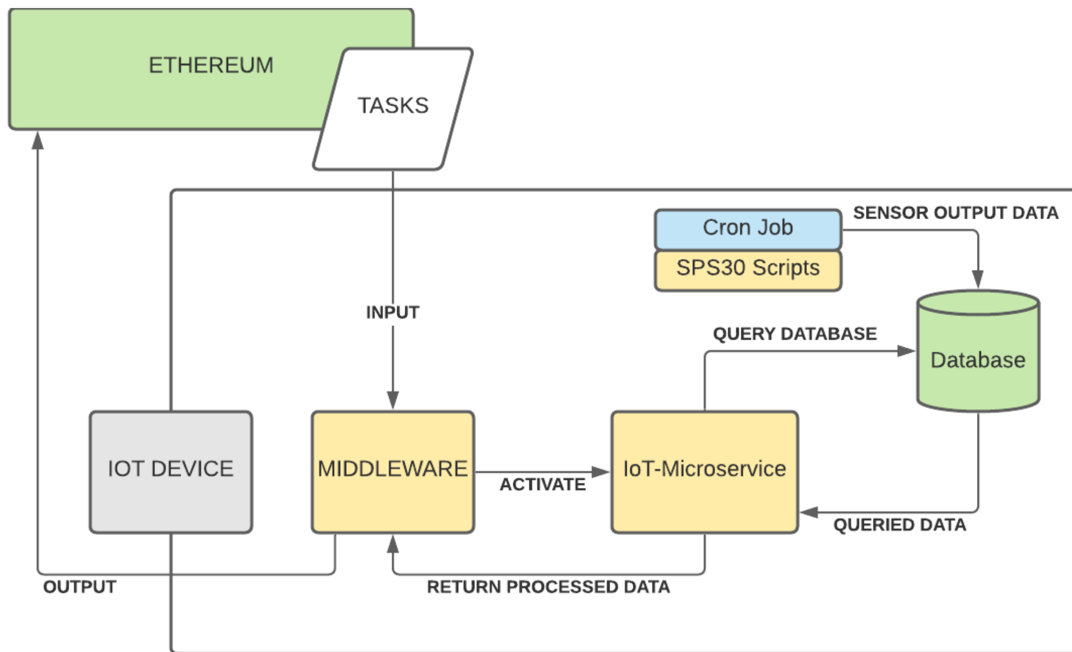

Figure 4. Functional Overview of the Middleware and IoT-Microservice.

4.4 The Decentralized Application

The dApp is a simple graphical user interface (GUI) that is launched via an executable file and aims to show proof-of-concept rather than be a fully developed application. Baseline functionality was developed to show case stateless design and decentralized architecture that enables users to fetch their data from the device smart contract. Once the data is retrieved, the user is able to view the data output in graphs or download it. Stateless design implies that the application has no dedicated backend, all code is executed locally on the device and no data is retained once it is terminated.

The application uses the Ethereum Application Binary Interface (ABI). The ABI is the functional interface that allows read and write functionality to Ethereum smart contracts. For the application to execute correctly and interact with the smart contracts, the ABI address needs to be provided by the user. If the data stored on the smart contract is encrypted, a valid private key must be provided for successful decryption. Not having a valid private key will result in the inability to decrypt the data. The dApp decrypts the data via the functionality presented in section 5.3.3. Figure 5 illustrates the functional

overview of the dApp.

Figure 5. Functional Overview of the Decentralized Application.

5 DEVELOPMENT AND IMPLEMENTATION

5.1 Hardware Specification

The Sensirion SPS30 Particulate Matter Sensor and Raspberry Pi 2B were the two main pieces of hardware provided for this project and both pieces of hardware are relatively inexpensive. Using inexpensive hardware illustrates that a dynamic high quality microservice can be built for an IoT use case. At the time of writing for this paper the price for a Raspberry Pi 2B in Finland was around 45 euros (Verkkokauppa.com n.d) and the sensor could be bought for 36 euros (mouser.fi 2021). All hardware was provided by Arcada University of Applied Sciences' Department of Business and Analytics.

5.1.1 Raspberry Pi 2B

The Raspberry Pi 2B (RPI2B) was released in 2015 and is the second-generation of the Raspberry Pi. It is a single-board computer that comes pre-installed with the operating system Raspbian, which is a type of Linux distribution (Raspberry Pi Foundation n.d.). Raspberry Pis are often used for different IoT projects.

5.1.2 Sensirion SPS30 Particulate Matter Sensor

The SPS30 Particulate Matter Sensor is a small and robust optical sensor for measuring particulate matter. Its measurements are 41 x 41 x 12 mm^3. Sensirion (2018) guarantees that the SPS30 will last at least 10 years with constant usage. Its measurement principles are based on laser scattering technology. This technology grants a high resolution for different types of particulate matter that is in the air (Sensirion 2020 p. 1). Laser scattering (laser diffraction) uses the diffraction pattern that the laser produces when particles pass through the laser beam. The patterns produced by the laser allow software to calculate the size of the particles in the laser beam.

5.2 SPS30 Python Driver

As there was no sensor specific Python driver provided by Sensirion for the SPS30 sensor, I opted to construct my own for the practical implementation of this thesis. This had the added benefit of ensuring source code integrity. The Python library PySerial was used to enable a Universal Asynchronous Receiver-Transmitter (UART) interface between the SPS30 and the Raspberry Pi while the Python library Struct was used to handle the binary data conversion from the sensor data output.

The SPS30 Python Driver library developed for this project was written in Python version 3.8. The library enables programmers to interact with the sensor's built-in functionality. It was designed to match the official SPS30 data sheet so that all functionality mentioned in the data sheet can be accessed via the driver's class methods. The naming convention is one-to-one between the data sheet and the library making it easy to compare the documentation (Sensirion 2020 p. 10). Figure 6 presents the UML diagram of the SPS30 class.

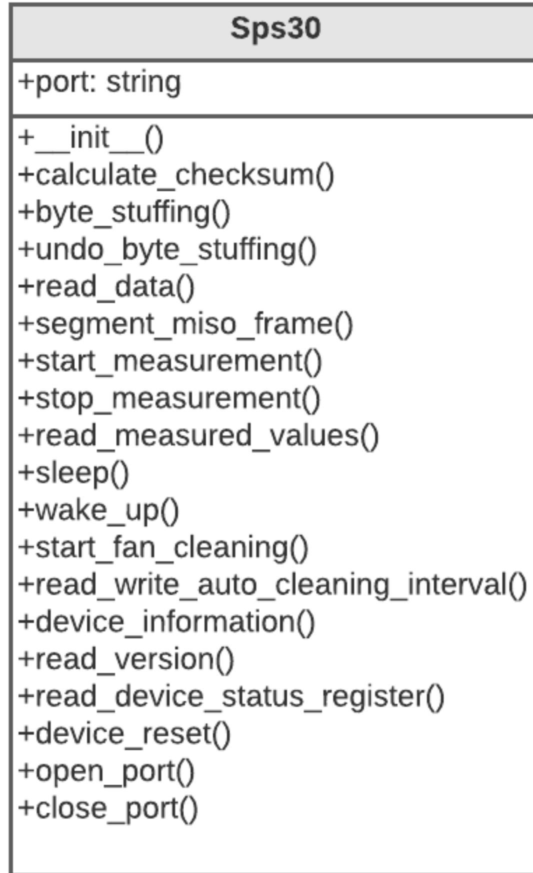

Figure 6. UML Diagram of the SPS30 Driver library.

5.2.1 SPS30 Functional Overview

The main operational modes of the sensor are: measurement, idle and sleep. The diagram in Figure 7 illustrates the modes' transitional states and how each state can be activated through internal sensor commands.

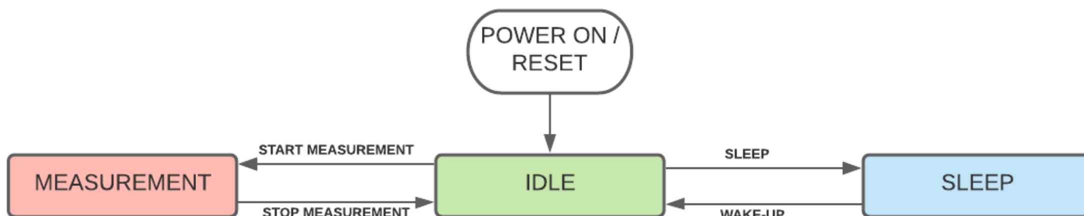

Figure 7. Diagram of Operating Modes of the Sps30 Sensor.

When the sensor is powered on or reset it defaults to idle mode. Idle mode is a low power mode that greatly reduces the power consumption of the device. From this mode the sensor is ready to receive and process commands. Measurement mode is the state in which all internal sensor components are turned on and it is continuously consuming maximum power. In measurement mode the sensor is continuously processing measurement data at a rate of one measurement per second. The sensor has built-in fan cleaning functionality that triggers once per week if the sensor has been continuously in measurement mode. By default, this occurs once per week, but this interval can be customized. The sensor can only perform the fan cleaning function while in measurement mode during which all other sensor actions are disabled. If the sensor goes into idle mode the timer for the fan cleaning will be reset. Sleep mode is the lowest power mode available for the sensor. Its primary purpose and use case are for when the sensor is used in a battery dependent context. To wake up the sensor a wake-up sequence needs to be sent to the sensor.

(Sensirion 2020 p. 5)

5.2.2 SPS30 Sensor Measurement Output Formats

Sensor measurement values are returned as either big-endian float IEEE754 or big-endian unsigned 16-bit integer. The format of return values must be specified when measurement mode is initiated. Table 4 presents the sensor output formats and what particulate matter information that is collected with every measurement. When a sensor measurement is performed, all 10 values are collected and returned in the chosen format. (Sensirion 2020 p. 5)

Sensor Output Formats	
Datatype	**Output**
big-endian float IEEE754	Mass Concentration PM1.0 [$\mu g/m^3$]
	Mass Concentration PM2.5 [$\mu g/m^3$]
	Mass Concentration PM4.0 [$\mu g/m^3$]
	Mass Concentration PM10 [$\mu g/m^3$]
	Number Concentration PM0.5 [$\#/cm^3$]
	Number Concentration PM1.0 [$\#/cm^3$]
	Number Concentration PM2.5 [$\#/cm^3$]
	Number Concentration PM4.0 [$\#/cm^3$]
	Number Concentration PM10 [$\#/cm^3$]
	Type Particle Size [nm]
big-endian unsigned 16-bit integer	Mass Concentration PM1.0 [$\mu g/m^3$]
	Mass Concentration PM2.5 [$\mu g/m^3$]
	Mass Concentration PM4.0 [$\mu g/m^3$]
	Mass Concentration PM10 [$\mu g/m^3$]
	Number Concentration PM0.5 [$\#/cm^3$]
	Number Concentration PM1.0 [$\#/cm^3$]
	Number Concentration PM2.5 [$\#/cm^3$]
	Number Concentration PM4.0 [$\#/cm^3$]
	Number Concentration PM10 [$\#/cm^3$]
	Type Particle Size [nm]

Table 4. SPS30 sensor output formats. (Sensirion 2020 p. 5)

5.2.3 SHDLC Protocol

The sensirion High-Level Data Link Control (SHDLC) protocol utilizes the UART interface. SHDLC uses a byte array format to communicate with the sensor through a master-slave communication protocol. This is done by sending a Master Out Slave In (MOSI) frame to the sensor that then responds with a Master In Slave Out (MISO) frame. Each frame contains many different byte elements communicating key information that

can be seen in figure 8. The start and stop bytes indicate when the frame content begins and ends. The address byte is the identifier for the slave device (the SPS30 sensor). In the MOSI frame, the command byte describes to the sensor what command is to be executed, whereas in the MISO frame the command byte identifies from what command the incoming MISO frame is responding to. The state byte, which is unique to the MISO frame, returns a byte that communicates sensor execution status. The length byte indicates the length of the TX and RX data bytes. The main data package consists of the transmit (TX) and the receive (RX) data bytes. Its content and length depend on the command that is being issued or received. The checksum byte is a mathematical calculation based on the content of a MOSI or MISO frame. For a frame to be valid, the checksum needs to be correctly calculated, otherwise the SPS30 sensor will not process the frame and return an error message. (Sensirion 2020 p. 8-10)

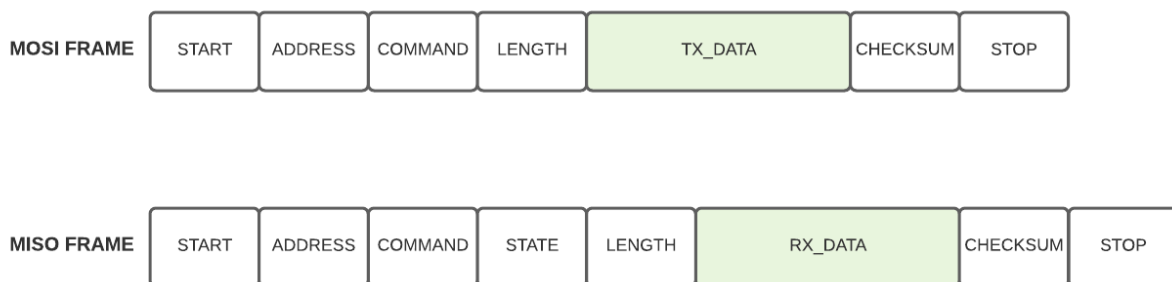

Figure 8. MOSI and MISO Frame Structure

5.2.4 Development and Implementation of SPS30 Driver

The sensor communicates to the master device via the I^2C protocol or the UART protocol. If the cable connecting the SPS30 is longer than 20 cm, it is better to use the UART interface, due to its innate robustness against electromagnetic interference. For this reason, the UART interface became the mandatory choice as the length of the cable in the project setup was about 100 cm. (Sensirion 2020 p. 4)

The SPS30 driver was designed as an independent open-source Python library that can be utilized outside this project. The library can be found at: https://github.com/kallmanm/Sps30-python-driver

5.3 IoT Microservice

The IoT microservice was designed around an IoT device context meaning that device resource management was strongly considered due to limiting factors such as power consumption, bandwidth limitations and hardware restraints. As stated in section 4.3, the IoT microservice was designed around the concepts of sensor data collection, data storage and data delivery. While at the same time trying to implement good design practices to improve the security of the application and IoT device. This led to the development of software components that manage specific parts of the microservice independent of one another. This decoupling of the internal modules improves the robustness of the microservice as data gathering and data delivery are independent of one another. The four main software components of the microservice are: the cron job sensor scripts, the database manager, the encryption and encoding manager and the service manager.

5.3.1 Cron Job Sensor Scripts

Cron job is a time-based task scheduler that comes standard on all Linux operating systems. Tasks are defined in a file called crontab, cron job then proceeds to process the tasks in the file according to their defined intervals. By using cron it was easy to manage all sensor functionality through the use of well define scripts. A data gathering script and a maintenance script were made to manage all needed sensor functionality. The cron job sensor scripts utilized the SPS30 driver presented in section 5.2.

The data gathering script was set to run at a 15-minute interval where 30 measurements are taken per script execution. The mean value of the 30 measurements is then written to the microservice's database through the use of the database manager module. A mean value is used to protect the dataset from outlier data measurements becoming overrepresented.

The maintenance script manages the cleaning process of the sensor keeping its instruments in optimal condition. To maintain optimal performance the instruments need to be cleaned at a minimum once per week. As the sensor is not continuously in measurement mode, the fan cleaning functionality will not trigger automatically, demonstrating

the need to have the maintenance script trigger through cron job.

5.3.2 Database Manager

The database manager module controls all database functionality of the microservice. As the microservice only needed basic read/write functionality, this led to the use of SQLite as the database engine. SQLite is an embedded relational database system that is part of the base Python package. This was a perfect fit for the microservice as it was aiming to use simple processes that save on computational power. SQLite's simple setup and configuration was another added benefit. If the need arises, the database can easily be transferred into another database engine.

The UML diagram of the database manager is presented in Figure 9. The database manager does not use a class structure, but rather is made up of four independent functions. The function add_entry adds entries to the database and also instantiates a database if none is found in the microservice's directory. While the remaining get functions fetch database rows based on the provided parameters. Variables d1–d10 in function 'add_entry' correlate to the sensor measurement outputs seen in Table 4.

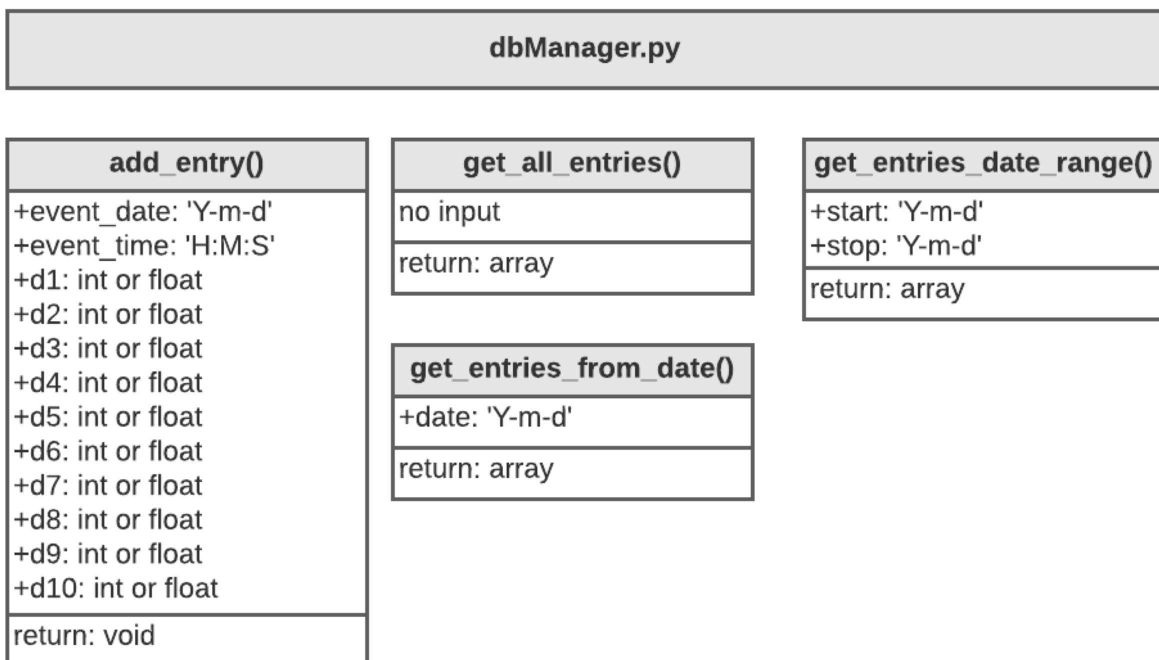

Figure 9. UML Diagram of the database manager module.

5.3.3 Encryption and Encoding Manager

The ability to utilize cryptography and encoding were key features of this project. The microservice had to be able to securely encrypt sensor data as well as guarantee data integrity while in transit. Likewise, the dApp needed to be able to decrypt the device data.

A hybrid encryption model was selected as the feasible method for protecting the microservice's output data. A hybrid encryption model utilizes both symmetric and asymmetric encryption. The reason for using a hybrid model was that the microservice data queries exceeded the size limitations of asymmetric encryption. At the same time, asymmetric encryption was needed to allow the microservice users to pass an encryption key securely, yet publicly to the microservice. Symmetric encryption does not have the size limitations that are inherent in asymmetric encryption. Figure 10 illustrates how the hybrid encryption model functions.

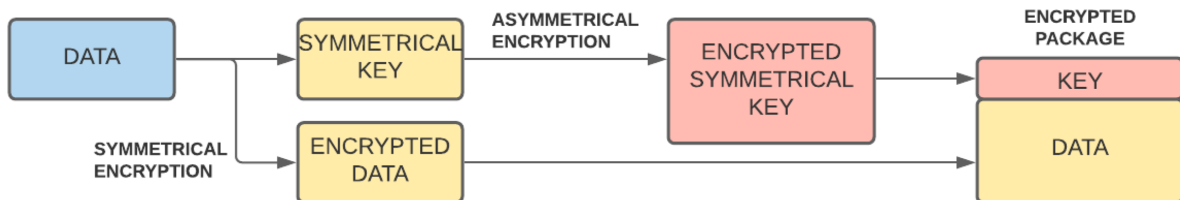

Figure 10. Hybrid encryption model using symmetric and asymmetric encryption.

The hybrid model uses symmetric encryption to encrypt the data package. This produces a symmetric key that can be used to decrypt the now encrypted data package. The process then continues to asymmetrically encrypt the symmetric key with the provided public key. The data package containing the encrypted data and the encrypted symmetric key is finally encoded into base64 and is ready for transit. Base64 is a binary-to-text process that encodes binary data into ASCII text ensuring no data loss or modification happens when binary data is sent between different computer systems. Therefore, the hybrid model is able to send data securely utilizing the benefits of both encryption models while avoiding the disadvantages. To decrypt the data package the matching private key is used to decrypt the encrypted symmetric key. Thereafter, the symmetric key can decrypt the main data

package.

The implementation of this hybrid encryption model was achieved through the use of the Python library 'pyca/cryptography'. The library has built-in functionality that supports both symmetric and asymmetric encryption standards. This led to the development of the Encryptor and Decryptor classes that can be seen in Figure 11. The encryption algorithms used were Secure Hash Algorithm-2 (SHA256) for the asymmetric process and the default Fernet class settings for the symmetric process. The Fernet object is the symmetrical encryption class of the cryptography library. Its default settings are Advanced Encryption Standard (AES128) in Cipher Block Chaining (CBC) Mode using a SHA256 keyed-hash message authentication code (HMAC). These algorithms were chosen to show proof-of-concept and do not claim to be the best or most practical standards for encryption but in the context of the microservice they were more than adequate.

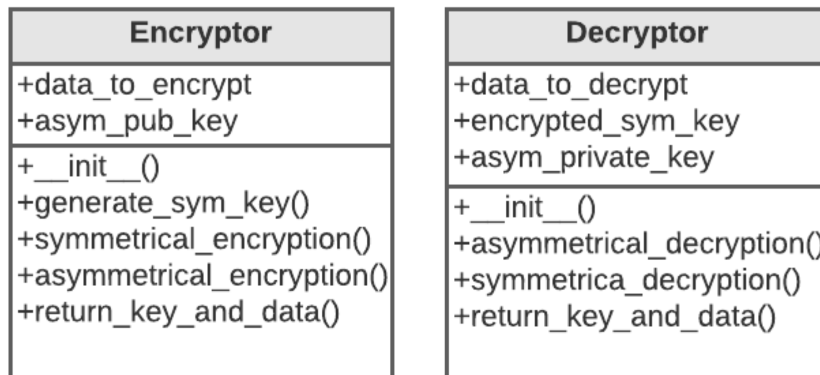

Figure 11. UML Diagram of Encryptor and Decryptor Classes.

The Encryptor class is an instance object designed with the purpose of encrypting a data package according to the hybrid encryption model mention earlier. It needs two parameters to function:

1. data_to_encrypt: The data to encrypt.
2. asym_pub_key: The public key used for encrypting the data.

On instantiation the Encryptor class will encrypt the data that has been provided. The

results of the Encryptor instance object can be fetched by running the class method 'return_key_and_data' that returns the asymmetrically encrypted key and the symmetrically encrypted data. Each instance of the Encryptor generates a unique symmetric key.

The Decryptor class is structured and functions the same way as the Encryptor class while its purpose is to decrypt rather than encrypt. The Decryptor needs three parameters to function:

1. data_to_decrypt: The data to decrypt.

2. encrypted_sym_key: The encrypted symmetrical key used to decrypt the encrypted data.

3. asym_private_key: The private key need to decrypt the encrypted symmetrical key.

Once the Decryptor has been instantiated the user is able to extract the decrypted data by calling the class method 'return_key_and_data'. This function will return the decrypted symmetrical key and the decrypted data.

5.3.4 Service Manager

The service manager module is the main software interface for the IoT microservice that allows users and programs to interact with the microservice. The main purpose of the service manager is to function as the interface that connects the IoT microservice to the CESP middleware. The software module UML can be seen in Figure 12.

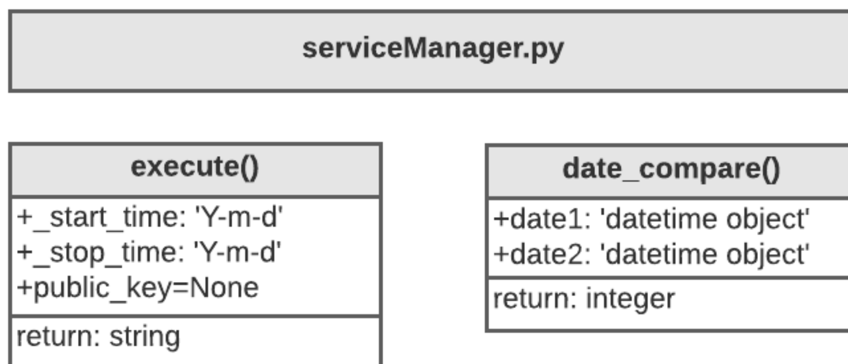

Figure 12. The UML diagram of the service manager functionality.

The service manager consists of one main function named 'execute' and a support function named 'date_compare'. **Date_compare** requires two parameters that consist of two datetime objects. The function calculates and verifies the delta of the two datetime parameters and returns the difference in days (integer). The function **execute** manages all logic relating to database queries, encryption and encoding through the use of the database manager and encryption and encoding manager modules. It requires three parameters to function. '_start_time' and '_stop_time' indicate between what date interval the function should query the database. If the 'public_key' parameter is provided the function will encrypt the data.

5.3.5 Installation and Setup

The installation and setup of the IoT microservice and CESP middleware is achieved via a manual process. It's assumed that a Raspberry Pi has been setup and that a Sensirion SPS30 sensor is connected via USB port. The Raspberry Pi will be referred to as the IoT device through the remainder of this section.

Start by downloading the source code of the Iot microservice to the IoT device. The source code contains all needed software to install and setup the microservice and the middleware. Once the source code has been downloaded, continue the installation process by setting up a Python virtual environment. When the virtual environment has been installed, proceed to activate it and install all Python library dependencies through the use of the Python Package Index (pip). This can easily be done by having pip read the 'requirements.txt' file that contains all dependency requirements.

When all dependencies have been installed, continue the setup process by updating the path of the shebang in all IoT microservice python files to point at the IoT microservice's virtual environment Python installation. By doing this all python files utilize the virtual environment's Python installation without having to activate the virtual environment.

Update the IoT device's crontab file to set sensor scripts according to desired intervals. At this point the microservice will start aggregating and collecting data. This can be verified

by running the data collection script manually. Upon successful execution the script will instantiate a database object named 'sensor_data.db' in the microservice directory. By executing 'dbManager.py' in the terminal you can confirm that data has been written to the database.

The final steps in the setup process requires the configuration of the middleware. The microservice directory has a folder named resources, it contains three files pertaining to the middleware setup. The file 'identifier.yaml' contains all information relating to the IoT microservice. In our case this is just mock data, but in a live environment it should contain factual information about your service. The 'settings.yaml' file contains all Ethereum gateway, key and whisper parameters. The 'latest.json' file contains the Application Binary Interface (ABI) files that are required for the middleware to interact with the corresponding smart contracts on the blockchain. When all middleware files have been updated the middleware can be activated by executing the 'launcher.py' file found in the project directory. The IoT microservice should now be live and can receive tasks assuming the CESP has also been activated.

5.4 dApp Development

As the IoT microservice developed for this project was of a decentralized nature, it was only suiting to develop a stateless dApp that utilizes the data of the IoT microservice. The stateless design gave the application an overall simpler design as no dedicated backend and data storage was needed as presented in section 4.4. To streamline development of this application, smart contract functionality from the middleware and cryptography functionality from the IoT microservice were reused in its design. This greatly reduced the time to develop the application.

The dApp was built using the following Python libraries: Web3, Tkinter, Matplotlib.pyplot, Numpy, Pandas and PyInstaller. Like the middleware, the dApp uses the web3 library to manage interactions with the Ethereum smart contracts. Tkinter was used to develop the GUI of the dApp. Numpy and Pandas were used to treat and clean the data while Matplotlib was used to build the data visualizations. The dApp utilizes the Decryptor class,

that was presented in section 5.3.3, to decrypt data stored on the Ethereum blockchain. The last component needed for the dApp was PyInstaller that is a packaging tool that helps ease the process of making executable files that require no installation of dependencies.

5.4.1 dApp Usage

To run the dApp, start the program via the executable file or download the source code and activate the dApp with the command 'python main.py'. For demonstrative purposes, the dApp can read in data locally if no Ethereum test network is activated. Figure 13 shows the layout of the dApp.

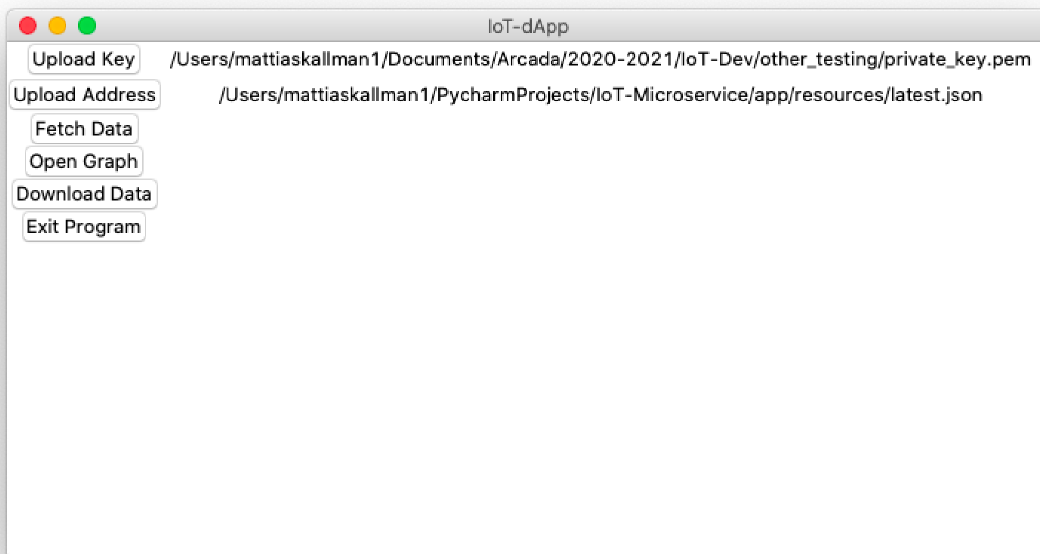

Figure 13. The GUI of the dApp.

On application activation the user is brought to the main GUI interface window as seen in Figure 13. From here the user needs to add their private key and ABI file via the use of the Upload Key button and the Upload Address button. Once the key and ABI file have been uploaded into the dApp the user can fetch their data by pressing the Fetch Data button. From here the user can either view their data or download it as a csv file. This is done through the use of the Open Graph button or the Download button. The program can be

terminated by pressing the Exit Program button.

6 RESULTS

The results of the practical implementation were encouraging and have been broken down into three subsections. They will present the performance of the IoT microservice as a whole, the validity and quality of the sensor data and how design aspects impacted IoT security.

6.1 IoT Microservice Performance

The IoT microservice performed as expected when testing it in a local in-memory Ethereum test environment. A user was first registered to the network and afterwards this user was able to register the IoT microservice to the CESP network. The IoT microservice was able to process service requests and return the sensor data to the network but due to transactions exceeding the gas values of the local test environment, the IoT microservice return data had to be modified and not return the whole data request as it exceeded the blockchain's gas policy. Gas refers to the fees or cost of conducting a transaction or executing a smart contract on Ethereum. This illustrated proof-of-concept that the IoT microservice works.

As data delivery, collection and storage were decoupled in the IoT microservice, this led to a robust service that could continue to operate at a reduced capacity even if data delivery or data collection were temporarily unavailable as each component can perform its process independent of one another. The critical point of the IoT microservice is its database. Without the database the IoT microservice collapses as a concept and cannot deliver or collect data. On a functional level the microservice can be queried but no data will be returned and no data would be collected and aggregated. As data is the primary purpose of this microservice and its ability to collect and aggregate data is key, there was no way of removing the need for a database without drastically changing the nature of the service. This could theoretically be achieved by providing on-demand sensor data that is directly served forgoing the use of a database, but there are some severe drawbacks to this implementation. The first disadvantage of this approach is that it locks the application process and has to finish one task before starting another as the sensor cannot process

multiple requests simultaneously. Depending on how much data is requested this could substantially slow down the microservice and possibly lock the service indefinitely if no safeguards were implemented. The second big factor invalidating this model is that by only collecting and delivering data on request there will be gaps in the data that the microservice can provide. The usefulness on this microservice model is questionable at best compared to the current implementation that can both serve data on-demand and collect data simultaneously and independently of one another.

6.2 Validation of Sensor Data Quality

The IoT microservice reliably and periodically measured and collected particulate matter data. In the practical implementation the IoT microservice was set to collect data every 15 minutes which was a reasonable time interval to detect trend changes in the particulate matter situation. During data collection, the IoT microservice takes 30 measurements within a 30 second time frame. After the measurements are completed the microservice then continues to aggregate and calculate the mean value of these 30 measurements. The reasoning for this data collection model is to avoid outlier data poisoning the database giving a false picture of the actual situation. This can be seen in Figure 14 where the blue line shows every data point from a 10-minute testing period while the red line shows the mean value of the same testing period.

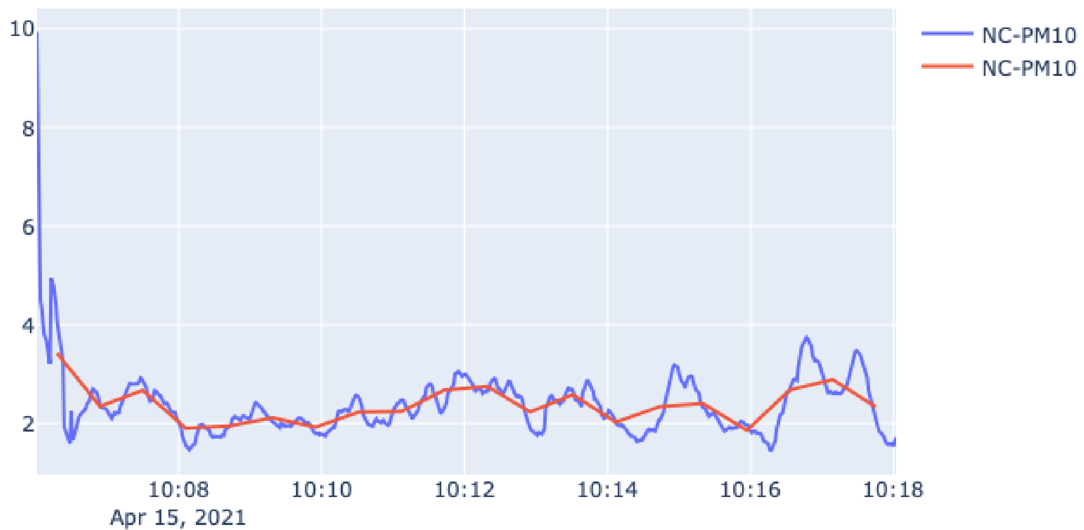

Figure 14. SPS30 data measurements: comparing mean values to unaggregated values.

Another action taken to guarantee the validity of the sensor data was the use of a sensor instrument cleaning script. The cleaning script guaranteed that the microservice would perform under as optimal conditions as possible. Accumulated dust particles can distort the sensor data results emphasizing the importance of maintaining clean sensor instruments. In the practical implementation the cleaning program was run once per week during a time window that would not collide with the normal data collection process.

6.3 IoT Microservice Security Implementations

The security implementations of this project can be divided into two categories. Security implementations derived from the CESP, which are detailed in section 3.4.2, and security implementations from the design of the IoT microservice. The latter will be presented in this section.

The IoT microservice attempted to apply good security design as presented in section 3.3. The IoT microservice only stores sensor measurement data in its database and all CESP

related data is stored in configuration files that the IoT microservice does not have access to. This precaution ensures that no configuration data can be leaked or accessed by the use of the IoT microservice, illustrating the benefits of applying good **data classification** that contributes to understanding the security implications of the service's internal and external data structures.

Decoupling the IoT microservice's internal modules and relying on as few as possible third-party libraries helped contribute towards better **application security**. Decoupling software modules made it easier to manage the IoT microservice software as key concepts and logic were grouped into their respective modules. Python comes with many standard libraries that are monitored and maintained by the Python Software Foundation (PSF). These libraries are inherently more secure than unofficially supported libraries as the PSF's reputation depends on the quality of the libraries they include in the Python software language. Libraries that are included in Python package are also validated according to industries standards and best practices. By using these standard libraries when possible ensured that the IoT microservice was built on reliable software.

Another layer of application security was achieved by building the custom sensor driver for the IoT microservice. By doing this, the IoT microservice can only use the sensor according to the driver's design pattern. The driver only allows sensor actions through the use of predefined functions that disallow direct byte array inputs as this could lead to nefarious use of the sensor.

Encryption was successfully implemented through the use of the encryption and encoding manager (eeManager) presented in section 5.3.3. The eeManager allows the use of asymmetric and symmetric encryption, securing data delivery of the IoT microservice. Asymmetric encryption protects against eavesdropping while symmetric encryption ensures that the whole data payload can be safely encrypted and transported.

While not addressed in detail in the theoretical sections of this thesis, software is only as good as the platform it runs on. There is therefore an assumption here that the following items have been dealt with to ensure optimal security for the IoT device. If the OS and IoT device are not correctly and safely configured the device will most likely be plagued

with security issues. All default and hard-coded passwords should be changed. Likewise, configurations and device firewall settings need to be updated to match the needs of the IoT device. Physical hardening and security should also be addressed according to the sensitivity of the IoT device.

7 CONCLUSIONS

The development of the IoT microservice was able to successfully incorporate Web 3.0 technologies and improve IoT device security by using the theoretical knowledge presented in sections 2 and 3. By Using the CESP, an Ethereum based blockchain was used as the dedicated backend fulfilling the criteria of utilizing a Web 3.0 technology. The IoT microservice also utilized edge computing paradigms by applying data collection, treatment and storage on the IoT device. The data treatment process and storage illustrated how shifting computational tasks to the IoT device can utilize the full potential of the IoT device lowering bandwidth usage and reducing the cost of using the Ethereum blockchain.

IoT device security can be improved by introducing the paradigm of 'security by design', meaning that security concerns should be considered through the whole software design and development process. A real challenge in IoT security design is understanding what a suitable amount of security for a specific IoT device is, since devices vary greatly in functionality and purpose. All too often the security implications are not properly understood leading to an insufficient amount of security. There is often an increased cost in increasing security and should therefore be proportionate to the security needs of the IoT device.

The theoretical study into Web 3.0 technologies presented in section 3, revealed that all three technologies have an impact on the future of IoT security to varying degrees. They can solve many security concerns if correctly applied and at the same time create new security concerns as some of these technologies can also be used nefariously. When designing software and IoT devices, the solution needs to consider the security aspects from the beginning. Likewise, the key technologies used need to be evaluated to determine their respective security impact.

7.1 Further development and research

The results and implementation of this thesis led to some interesting results that are worth exploring. While testing the practical implementation, the need for a proper data transportation solution arose due to the amount of data the IoT microservice was trying to write to the smart contracts. It is impractical and expensive to set up services that write all end user data to smart contracts as writing data to the Ethereum blockchain has a cost that is proportional to the amount of data written. Finding a solution how to deliver the IoT microservice data to the end user needs to be explored and solved, preferably with a technology that supports a decentralized data structure.

Another item worth exploring would be how best to orchestrate microservice deployment via the use of containerized orchestration software such as Docker. Running the IoT microservice in a container would add a degree of security as the containerized application creates another layer of abstraction protecting the OS from direct interaction. Integrating containerization functionality into the CESP could help manage software versioning and deployment.

The theoretical study of Web 3.0 technologies hinted that there are interesting applications of AI for device security. A higher degree of security could be achieved by integrating AI into an IoT device and analyzing device telemetry. Designing and creating a device monitoring algorithm could be a future area worth exploring.

REFERENCES

Eagar, MaRi. 2017, *What is the difference between decentralized and distributed systems?*, [online]. Available at: https://medium.com/distributed-economy/what-is-the-difference-between-decentralized-and-distributed-systems-f4190a5c6462, Accessed: 2.3.2021.

Gettings, Brian. 2007, *Basic Definitions: Web 1.0, Web. 2.0, Web 3.0*, [online]. Available at: https://www.practicalecommerce.com/Basic-Definitions-Web-1-0-Web-2-0-Web-3-0, Accessed: 17.2.2021.

Gillis, Alexander. 2020, *internet of things (IoT)*, [online]. Available at: https://internetofthingsagenda.techtarget.com/definition/Internet-of-Things-IoT, Accessed: 16.2.2021.

Hooda, Parikshit. 2019, *Comparison – Centralized, Decentralized and Distributed Systems*, [online]. Available at: https://www.geeksforgeeks.org/comparison-centralized-decentralized-and-distributed-systems/, Accessed: 25.2.2021.

IEEE. 2019, *Why Does Edge Computing Matter?*, [online]. Available at: https://innovationatwork.ieee.org/why-does-edge-computing-matter/, Accessed: 18.2.2021.

IoT Security Foundation. 2019, *Secure Design Best Practice Guides*, [online]. Available at: https://www.iotsecurityfoundation.org/wp-content/uploads/2019/12/Best-Practice-Guides-Release-2_Digitalv3.pdf, Accessed: 5.3.2021.

Leonard, Matt. 2019, *Declining price of IoT sensors means greater use in manufacturing*, [online]. Available at: https://www.supplychaindive.com/news/declining-price-iot-sensors-manufacturing/564980/, Accessed: 5.3.2021.

Maayan, David. 2020, *The IoT Rundown For 2020: Stats, Risks, and Solutions*, [online]. Available at: https://securitytoday.com/Articles/2020/01/13/The-IoT-Rundown-for-2020.aspx, Accessed: 16.2.2021.

Melendez, Steven. 2018, *Advantages & Disadvantages of Distributed Systems*, [online]. Available at: https://www.techwalla.com/articles/advantages-disadvantages-of-distributed-systems, Accessed: 25.2.2021.

Melendez, Steven. 2019, *Advantages & Disadvantages of Distributed Systems*, [online]. Available at: https://www.techwalla.com/articles/advantages-disadvantages-of-distributed-systems, Accessed: 11.3.2021.

Mersch, M. & Muirhead, R. 2019, *What Is Web 3.0 & Why It Matters*, [online]. Available at: https://medium.com/fabric-ventures/what-is-web-3-0-why-it-matters-934eb07f3d2b, Accessed: 16.2.2021.

mouser.fi. 2021, *SPS30*, [online]. Available at: https://www.mouser.fi/ProductDetail/Sensirion/SPS30/?qs=lc2O%252BfHJPVbEPY0RBeZmPA%3D%3D, Accessed: 5.3.2021.

OWASP. 2018, *OWASP Top 10 Internet of Things 2018*, [online]. Available at: https://owasp.org/www-pdf-archive/OWASP-IoT-Top-10-2018-final.pdf, Accessed: 16.2.2021.

Raj, Emmanuel; Westerlund, Magnus & Espinosa-Leal, Leonardo. 2020, Reliable Fleet Analytics for Edge IoT Solutions, *Cloud Computing 2020: The Eleventh International Conference on Cloud Computing, GRIDs, and Virtualization*, p. 55.

Raspberry Pi Foundation. n.d., *Raspberry Pi 2 Model B*, [online]. Available at: https://www.raspberrypi.org/products/raspberry-pi-2-model-b/, Accessed: 5.3.2021.

Sciforce. 2020, *Artificial Intelligence for Cyber-Security: A Double-Edge Sword*, [online]. Available at: https://medium.com/sciforce/artificial-intelligence-for-cyber-security-a-double-edge-sword-6724e7a31425, Accessed: 16.4.2021.

Seal, Alan. 2020, *Centralized vs Decentralized Network: Which One Do You Need?*, [online]. Available at: https://www.vxchnge.com/blog/centralized-decentralized-network, Accessed: 18.2.2021.

Sensirion. 2018, *Particulate Matter Sensor SPS30*, [online]. Available at: https://www.sensirion.com/en/environmental-sensors/particulate-matter-sensors-pm25/, Accessed: 18.2.2021.

Sensirion. 2020, *Datasheet SPS30*, [online]. Available at: https://www.sensirion.com/en/download-center/, Accessed: 18.2.2021.

Shea, Sharon & Wigmore, Ivy. 2018, *IoT security (internet of things security)*, [online]. Available at: https://internetofthingsagenda.techtarget.com/definition/IoT-security-Internet-of-Things-security, Accessed: 5.3.2021.

Tyagi, Neelam. 2021, *6 Major Branches of Artificial Intelligence (AI)*, [online]. Available at: https://www.analyticssteps.com/blogs/6-major-branches-artificial-intelligence-ai, Accessed: 16.4.2021.

Vega, Malvina. 2021, *Internet of Things Statistics, Facts & Predictions [2020's Update]*, [online]. Available at: https://review42.com/resources/internet-of-things-stats/, Accessed: 16.2.2021.

Verkkokauppa.com. n.d, *Raspberry Pi 2 model B - yhden piirilevyn tietokone*, [online]. Available at: https://www.verkkokauppa.com/fi/product/4657/fjxtn/Raspberry-Pi-2-model-B-yhden-piirilevyn-tietokone?list=OZCYkRh1CSb9Yi0gOJUg2N, Accessed: 5.3.2021.

Vermaak, Werner. 2020, *How Decentralized Are Decentralized Networks?*, [online]. Available at: https://coinmarketcap.com/alexandria/article/how-decentralized-are-decentralized-networks, Accessed: 2.3.2021.

Wickström, J.; Westerlund, M. & Pulkkis, G. 2021, Smart Contract based Distributed IoT Security: A Protocol for Autonomous Device Management, *In proceedings of 21st ACM/IEEE International Symposium on Cluster, Cloud and Grid Computing (CCGrid 2021) (forthcoming)*.

Wickström, John. 2020, *Distributed IoT Security using an Ethereum based Blockchain Infrastructure*, B.s. thesis, Arcada, pp. 1–44.

Xiao, Liang; Wan, Xiaoyue; Lu, Xiaozhen; Zhang, Yanyong & Wu, Di. 2018, IoT security techniques based on machine learning: How do IoT devices use AI to enhance security?, *IEEE Signal Processing Magazine*, vol. 35, no. 5, pp. 41–49.

Xiao, Yinhao; Jia, Yizhen; Liu, Chunchi; Cheng, Xiuzhen; Yu, Jiguo & Lv, Weifeng. 2019, Edge computing security: State of the art and challenges, *Proceedings of the IEEE*, vol. 107, no. 8, pp. 1608–1631.

APPENDIX 1 SWEDISH SUMMARY

Introduktion och syfte

Sakernas internet (engelskans Internet of Things, IoT) är en term för att beskriva apparater som använder internet för att lösa diverse uppgifter. Det har länge varit svårt att definiera IoT-apparater på grund av att de varierar kraftigt i kapacitet och syfte samt tillämpas i många olika industrier och områden. Smarta armbandsur och insulinpumpar som är kopplade till en mobiltelefon är två tydliga exempel på IoT-apparater. (Gillis 2020)

Statistik som Vega (2021) presenterar visar att tillämpningen av IoT-apparater växer kraftigt och kommer att fortsätta växa explosionsartat fram till 2025. Denna tillämpning av IoT-apparater drivs av att tillverkningskostnader har gått ner samt att kapaciteten på apparaterna ökat. Ett stort problem med IoT-apparater har varit att säkerhetsåtgärder inte har blivit adekvat åtgärdat i förhållande till deras ibruktagande (OWASP 2018). Samtidigt utvecklas nya och intressanta teknologier som kan användas för att förbättra säkerheten med IoT-apparater.

Det finns tre syften med detta examensarbete. Det första är att utveckla en distribuerad IoT-mikrotjänst och en decentraliserad applikation (dApp) som använder sig av kantdatorsystem (engelskans edge computing) och decentraliserat datanätverk. IoT-mikrotjänsten kommer att tillämpa ett specialtillverkat protokoll (Wickström 2020) som använder sig av Ethereum vilket är en typ av decentraliserat datanätverk. Det andra syftet är att analysera och studera hur webb 3.0-teknologier inverkar på säkerhet hos IoT-mikrotjänsten. Det tredje syftet är att evaluera hur IoT-apparater påverkas av ’säkerhet genom design’.

Webb 3.0

Webb 3.0 är ett samlingsbegrepp för att beskriva den senaste utvecklingen av webben. De tre teknologier som primärt definierar webb 3.0 är kantdatorsystem, decentraliserade datanätverk och artificiell intelligens (AI). (Mersch & Muirhead 2019)

Kantdatorsystem kan beskrivas som databearbetning och datalagring i nätverkets utkant.

Genom att avlasta den centrala servern kan man minska på den totala mängden data som skickas över nätverket vilket gör att systemet sparar på nätverksresurser. Dessutom är kantdatorsystem väldigt snabba på grund av sin struktur. En nackdel med systemet är dock den utökade säkerhetsrisken som kommer med att tillämpa kantdatorsystem. Kantdatorsystem använder sig av kraftiga maskiner i nätverkets utkant och dessa maskiner är ofta utsatta för olika sorters intrång och kapning. Det är därför viktigt att överväga om kantdatorsystem är lämpligt för en situation. (IEEE 2019)

Decentraliserade datanätverk har en sorts datahierarki som skiljer sig markant från centraliserade datanätverk som är den traditionella modellen. Centraliserade datanätverk består av en server-klient arkitektur där servern delegerar och styr nätverket. I decentraliserade datanätverk finns ingen central enhet som styr över nätverket, utan det är maskinerna som medverkar i datanätverket som bygger upp helheten. Varje maskin som deltar i datanätverket har självstyre och kan själv välja hur mycket av sina resurser den allokerar till datanätverket. En intressant egenskap som decentraliserade datanätverk har är att de kan expanderas och förstoras mycket enkelt jämfört med centraliserade datanätverk. Dess egenskap att vara fri från en central enhet som styr över datanätverket bidrar till att det är svårt att kapa ett decentraliserat datanätverk. (Seal 2020)

Artificiell intelligens är den sista teknologin som brukar grupperas med webb 3.0 och det är den delen av datavetenskap som studerar maskiners intelligens och dess försök att emulera mänsklig intelligens. AI är intressant med tanke på datasäkerhet för att AI kan användas för att analysera och hitta samband i stora mängder data. Det finns flera olika grenar av AI och var och en specialiserar sig på en viss typ av problemfrågeställning. (Tyagi 2021)

IoT-Säkerhet

IoT-säkerhet går ut på att säkra och skydda IoT-apparater vilket har visat sig vara utmanande på grund av att IoT-apparater varierar kraftigt i funktion och design. Den explosionsartade tillämpningen av IoT i industrier har bidragit till att säkerhetsaspekter inte har blivit åtgärdade till den grad som anses vara acceptabelt. (Vega 2021)

IoT Security Foundation (2019) har utvecklat en manual med bästa praxis för utvecklandet av IoT-apparater och system. Manualen ger svar på OWASPs (2018) åtgärdslista för att förbättra säkerheten hos IoT-system. På grund av IoT-apparaters mångfald så är det i praktiken inte möjligt att åtgärda alla punkter i listorna, men genom att välja de mest kritiska punkterna kan IoT-apparaters säkerhet förbättras.

Tillämpningen av webb 3.0-teknologier i IoT-apparater har en direkt inverkan på säkerheten. Xiao et al. (2019) beskriver i deras artikel de säkerhetsutmaningar som kantdatorsystem skapar i IoT-system. De framför en lösning till dessa säkerhetsproblem genom att tillämpa god designpraxis som beaktar de svagheter som kantdatorsystem medför.

Wickström et al. (2021) har skapat en originell lösning som tillämpar Ethereum smarta kontrakt och använder dem som infrastrukturen för ett decentraliserat datasystem. Det utvecklade protokollet använder sig av flera unika egenskaper hos Ethereum som gör att säkerheten i systemet förbättras. Alla datatransaktioner är autentiserade och auktoriserade med Ethereums protokoll vilket gör det praktiskt taget omöjligt att förfalska dataöverföring i systemet. Samtidigt är Ethereum smarta kontrakt oföränderliga vilket innebär att användare av Wickströms protokoll har en tydlig och oförfalskbar historik på deras aktivitet i nätverket. Dessa säkerhetsåtgärder har dessutom inverkan på andra delar av datasystemet som löser problem angående till osäker nätverksaktivitet och bristande apparatförvaltning. (Wickström et al. 2021)

AI-modeller kan tillämpas och användas i IoT-apparater genom att analysera apparatens loggar och övrig apparattelemetri. Genom att analysera informationen kan AI-modeller märka anomalier i systemet. En intressant egenskap med AI är att det går att använda AI-modeller för att försöka bryta sig in i system genom att analysera systemens telemetri. AI är alltså tveeggad i sin natur och så vis kan användas för att säkra eller bryta sig in i datasystem. (Xiao et al. 2018)

Mjukvaradesign och arkitektur

Den praktiska tillämpningen av examensarbete består av tre mjukvaror som tillsammans bygger ett fungerande mjukvaruekosystem som illustrerar hur man kan skapa, upprät-

hålla och konsumera en IoT-mikrotjänst. De tre mjukvaror som används och utvecklades är: det originella säkerhetsprotokollet utvecklat av Wickström (2020), IoT-mikrotjänsten och den decentraliserade applikationen. IoT-mikrotjänsten och den decentraliserade applikationen använder sig av säkerhetsprotokollet som fungerar som den underliggande infrastrukturen i systemet. Funktionen som IoT-mikrotjänsten erbjuder är att samla in och leverera partikelmassainformation. Partikelmassa används som mått för att mäta storleken och frekvensen av partiklar i luften vilket indikerar hur bra luftkvalitén är. Syftet med den decentraliserade applikationen är att kunna visualisera och använda den information som IoT-mikrotjänsten erbjuder.

IoT-mikrotjänsten och den decentraliserade applikationen kommer att tillämpa god designpraxis för att illustrera hur man kan förbättra en apparats säkerhet samt tillämpa kantdatorsystem koncept.

Implementering

IoT-mikrotjänsten byggdes och utvecklades för en Raspberry Pi 2B och en Senserion SPS30 partikelmassasensor. Programmeringsspråket som användes var Python och för att beakta hårdvarans begränsningar så användes till den grad det var möjligt den inbyggda funktionalitet som kommer med Python. Det fanns inga färdigt utvecklade Python drivrutiner för SPS30 sensorn vilket ledde till att en originell drivrutin utvecklades för IoT-mikrotjänsten. Mjukvaran av IoT-mikrotjänsten strukturerades i tydliga komponenter enligt deras funktionalitet. De fyra komponenterna delades in i följande grupper: skripten som tar hand om sensormätningarna och upprätthållandet av sensorn, databasförvaltarmjukvaran som sköter lagringen och hämtningen av sensordata, krypterings- och kodningsmjukvaran som säkrar databasinformationen före det sänds över nätverket och serviceförvaltarmjukvaran som möjliggör att IoT-mikrotjänsten kan kommunicera med det originella säkerhetsprotokollet.

Den decentraliserade applikationen utvecklades också med Python och använde sig av dess inbyggda funktionalitet så långt det gick vilket bidrog till att applikationen är sparsam med datorresurser. Programvarukomponenter från säkerhetsprotokollet samt IoT-mikrotjänsten återanvändes vilket ledde till att den totala utvecklingstiden för den decentraliserade app-

likationen förkortades.

Resultat

IoT-mikrotjänsten testades i en lokal testmiljö som simulerar Ethereum-nätverket och testandet var lyckat. När en användare skickade en förfrågan om sensordata till mikrotjänsten så lyckades den tolka förfrågningen och ge svar via det originella protokollet. En oväntad konsekvens i testandet visade att mängden sensordata var för stor för att skriva till Ethereum-testnätverk i förhållande till kostnaderna. Det antyder att det i praktiken skulle vara opraktiskt att skriva sensordata till det verkliga Ethereum-nätverket på grund av den inbyggda transaktionskostnaden på nätverket. Mikrotjänsten modifierades i testmiljö och skickade bara en del av sensordata för att visa att servicen fungerade konceptuellt.

Validiteten av sensordata var en viktig del av slutarbete och uppnåddes med två verktyg. Det första var att göra en medeltalsberäkning på sensordata då mikrotjänsten skulle göra mätningar. Sensorn var kalibrerad att gör mätningar var femtonde minut. Vid mätningstillfällen så gör mikrotjänsten 30 stycken mätningar på en 30 sekunders period varefter den räknar ut medeltalet på mätningarna. Denna åtgärd skyddar den insamlade datan från avvikande datapuntker som skulle kunna ge en felaktig bild av situationen. Den andra åtgärden för att garantera att sensorn hade optimala förhållanden var att använda sig av en städfunktion på sensorinstrumentet. Genom att blåsa ut sensorn en gång per vecka skyddar man den från att damm ska ackumuleras och påverka sensormätningarna.

Med användningen av god designpraxis och tillämpningen av webb 3.0-teknologier så lyckades utvecklingen av IoT-mikrotjänsten visa på förbättrad säkerhet. Datan som mikrotjänsten använder sig av och samlar in var klassificerat och definierat. Bara den allra nödvändigaste datan används för att driva mikrotjänsten och enbart sensordata lagras i databasen. Viktiga konfigurationsdata lagras i konfigurationsfiler som mikrotjänsten inte har tillgång till. Den här åtgärden garderar mikrotjänsten mot att någon skulle försöka komma åt konfigurationsfilerna via själva servicen. Att frikoppla de olika interna delarna av mikrotjänsten och att själv utveckla drivrutiner samt inte använda sig av tredje part programvara till den mån det var möjligt hjälper garantera säkerheten i mikrotjänsten. Den sista säkerhetsåtgärden implementerad i mikrotjänsten var användningen av kryptering. Genom

att kryptera den externa datan som mikrotjänsten levererar skyddar man hela ekosystemet från att någon tjuvlyssnar på kommunikationen.

Diskussion

Forskningen och tillämpningen av detta examensarbete var lyckat. Studien visade tydligt på inverkan av webb 3.0-teknologier på IoT-apparatsäkerhet. Den praktiska implementeringen lyckades tillämpa flera viktiga teknologier och illustrera olika sätt att förbättra IoT-apparatsäkerheten. Säkerhet genom design visade sig vara det kraftigaste verktyget för att förbättra säkerheten generellt för IoT-mikrotjänsten.

Resultaten och implementeringen uppenbarade några intressanta saker värda att utforska. Ett bra sätt att sända stora mängder av data i decentraliserade datanätverk behöver utforskas. Testandet av mikrotjänsten visade att det var opraktiskt att skicka och skriva data till Ethereum-nätverket redan på den nivå som mikrotjänsten verkade. Teknologier och metoder som stöder decentraliserad arkitektur bör utforskas i hopp om att lösa detta problem.

IoT-mikrotjänsten installerades manuellt i den praktiska implementeringen. Att studera och skapa en lösning som tillämpar programbehållarteknik (engelskans containerization) på mikrotjänsten skulle lösa en del säkerhetsproblem. Genom att tillämpa en programbehållarlösning med det originella protokollet skulle man kunna förbättra programvarahanteringen samt programvaradistributionen.

Den teoretiska studien i webb 3.0-teknologier antydde att det finns intressanta tillämpningsmöjligheter av AI i IoT-sammanhang. Att utveckla en AI-modell som bevakar IoT-apparater är värt att utforska.