A Novel Zero-Trust Framework to Secure IoT Communications

By

Sairath Bhattacharjya

B. Tech., Computer Science and Engineering
West Bengal University of Technology (WBUT)

Submitted to the graduate degree program in Department of Electrical Engineering and Computer Science and the Graduate Faculty School of the University of Kansas in partial fulfillment of the requirements for the degree of Master of Science.

Chair: Hossein Saiedian, Ph.D.
Professor and Thesis Adviser

Alex Bardas, Ph.D.
Assistant Professor

Fengjun Li, Ph.D.
Associate Professor

Date Defended: 5 June 2020

The Thesis Committee for Sairath Bhattacharjya certifies that this is the approved version of the following thesis:

A Novel Zero-Trust Framework to Secure IoT Communications

Chair: Hossein Saiedian, Ph.D.
Professor and Thesis Adviser

Date Approved: June 5, 2020

Abstract

The phenomenal growth of the Internet of Things (IoT) has highlighted the security and privacy concerns associated with these devices. The research literature on the security architectures of IoT makes evident that we need to define and formalize a framework to secure the communications among these devices. To do so, it is important to focus on a zero-trust framework that will work on the principle premise of “trust no one, verify everyone” for every request and response.

In this thesis, we emphasize the need for such a framework and propose a zero-trust communication model that addresses security and privacy concerns of devices with no operating system or with a real-time operating system. The framework provides an end-to-end security framework for users and devices to communicate with each other privately. A common concern is how to implement high-end encryption algorithm within the limited resources of an IoT device. We demonstrated that by offloading the data and process heavy operation like audit management to the gateway we were able to overcome this limitation. We built a temperature and humidity sensor and were able to implement the framework and successfully evaluate and document its efficient operations. We defined four areas for evaluation and validation, namely, security of communications, memory utilization of the device, response time of operations, and cost of its implementation, and for each, we defined a threshold to evaluate and validate our findings. The results are satisfactory and are documented.

Acknowledgement

First and foremost, I would like to express my deepest gratitude to my advisor, Dr. Hossein Saiedian for his guidance and patience in reviewing and correcting my writing throughout the research. I appreciate you giving me the freedom to pursue my interest and helping me in every step of the way. I would like to convey my gratitude to the committee members, Dr. Alex Bardas and Dr. Fenjung Li for giving their valuable time in reviewing my thesis.

I would like to thank Prof. Blake Bryant for helping me with the idea of the thesis and guiding me throughout the process of implementation. He has been a constant support and has always been available for any questions that I had. I would thank Mr. Felix Mercader and Mr. Alex Arntson from the KU IT team for helping me with technology and resolving issues with the lab setups. I would also thank Mr. Michael Gurwell for taking time to help me test the resilience of the device. This thesis would have been impossible without the support and guidance of each and everyone of them. I also want to thank KU library for providing me the required journals for my research and the writing center to help me formalize the ideas in words.

Lastly, I would like to thank my parents, Mr. Sailaja Nanda Bhattacharjya and Mrs. Krishna Bhattacharjya for their endless love and care. I would also thank my wife Mrs. Aparna Dasgupta for her constant care and mental support throughout the process of the thesis. My family provided the environment for me at home to carry out my experiments and supporting me in every possible way.

Contents

1 Internet of Things (IoT) **1**

1.1 The question of privacy . . . 2

1.2 Trust issues in heterogeneous environment . . . 3

1.3 The world without trust . . . 5

1.4 Validating zero-trust . . . 7

1.5 Thesis organization . . . 10

2 Security of IoT Devices **11**

2.1 The unprecedented growth . . . 11

2.2 Attack on IoT devices . . . 14

2.2.1 Malware . . . 15

2.2.2 DDoS and IoT Botnets . . . 17

2.2.3 Recent IoT based attacks . . . 19

2.3 Security concerns . . . 21

2.4 Proposed frameworks to secure the gaps . . . 23

2.4.1 IoT security . . . 24

2.4.2 IoT authentication . . . 26

2.5 Summary . . . 28

3 Zero-Trust Framework **31**

3.1 Understanding zero-trust . . . 32

3.2 IoT architecture 33
3.3 Data structure 36
3.3.1 Data stored in gateway 37
3.3.2 Data stored in devices 38
3.3.3 Data stored in users and delegates 39
3.4 P3 connection model 41
3.4.1 Connecting user and device 42
3.4.2 Connecting delegate and device 44
3.5 Communication over untrusted medium 47
3.5.1 Heartbeat communication 48
3.5.2 Sending command to device 51
3.6 Patch management 55
3.7 Addressing other security concerns 57
3.8 Summary 60

4 Implementing Zero-Trust for IoT Security 62

4.1 Building the environment 63
4.1.1 Setting up the device 63
4.1.2 Mobile app for the user and delegate 70
4.1.3 Configuring the gateway 76
4.1.4 Creating the AWS resources 81
4.2 Implementing the P3 connection model 83
4.3 Communicating with the device 100
4.3.1 Implementing the heartbeat 101
4.3.2 Executing command on the device 107
4.4 Summary 116

5 Analyzing the Framework **120**

5.1 Security in transit 121

5.1.1 Physical security 128

5.2 Memory utilization of device 132

5.3 Time to response 135

5.4 Cost of the device 138

5.5 Summary 139

6 Future of IoT Security **142**

6.1 Future research 145

6.2 Need for policy 146

Bibliography **148**

A Arduino program for blink **156**

B Create table script **157**

C Source code of the device **159**

D Template to deploy resources using SAM **170**

List of Figures

1.1 IoT architecture 5
1.2 IoT proposed internal architecture for temperature and humidity sensor . 9

2.1 Growth of IoT by industry 2017 – 2022 [16] 12
2.2 Adoption of IoT by 2030 [16] 14
2.3 Different types of malware attacks in IoT [70] 16
2.4 Distributed Denial of Service (DDoS) 17
2.5 Parts of US effected by Dyn DNS attack [21] 20
2.6 Reference IoT Architecture for E-commerce [61] 24
2.7 Trust assessment framework for cloud services [34] 26
2.8 Architecture of blockchain-based IIoT system for smart factory [23] 29

3.1 Generalized network communications for a IoT request 34
3.2 Proposed architecture for zero-trust framework 35
3.3 ER diagram of the gateway data structure 37
3.4 P3 connection between user and device 42
3.5 P3 connection between delegate and device 45
3.6 Heartbeat from device to gateway 49
3.7 Communication between user and device via gateway 54
3.8 Updating device firmware 57

4.1 The physical device 64

4.2 Adding ESP8266 board manager to the IDE 67
4.3 Installing ESP8266 board 67
4.4 Selecting the required setting for Arduino IDE 68
4.5 Pinout diagram for the device 69
4.6 Android Studio 3.5.3 72
4.7 Setting environment variable for React Native to work with Android Studio 73
4.8 Setting path variable for platform tools 74
4.9 Verify React Native CLI installation 74
4.10 Test app opened in Visual Studio Code 75
4.11 Mobile devices connected to the development machine 75
4.12 Deployment of the app in emulator 76
4.13 AWS console dashboard 77
4.14 AWS CLI installed in the development machine 79
4.15 Configuring AWS account for development machine 80
4.16 Buckets present in AWS S3 80
4.17 Roles present in IAM 82
4.18 App screen to show the current devices added 85
4.19 App screen to add a new device 86
4.20 App screen to pass Wi-Fi credentials to the device after successful connection 93
4.21 Setting up heartbeat checker in AWS 107
4.22 Data captured from the device on the user's app 117

5.1 App screen to show registration, confirmation and user login 122
5.2 Email send to user for verification 122
5.3 Wireshark logs showing use of TLS 123
5.4 Encrypted command send from the user to the device via gateway 127
5.5 Encrypted data sent from device to the user via gateway 127

5.6 Transaction audit at the gateway 128
5.7 Memory error caused by DDoS attack on device 130
5.8 Available memory of the device during the attack 131
5.9 Alert email send to the user when device is offline 131
5.10 Memory usage during P3 connection model 133
5.11 Memory utilization of device for heartbeat communication 134
5.12 Memory utilization of device for command execution 134
5.13 Time utilization of device for heartbeat communication 136
5.14 Command execution time 137

List of Tables

1.1 Contribution of ongoing European projects on IoT security [60] 4

3.1 Transactions of the communication between user and device 59

4.1 NodeMcu V3 specification [76] 64

4.2 Components of the IoT device 66

4.3 Pinout connection of the components with the microcontroller 70

5.1 Comparison of signing time between RSA and ECC [65] 125

5.2 Comparison of verification time between RSA and ECC [65] 125

5.3 Comparison of different symmetric encryption technique [2] 126

5.4 Operational time for each step in P3 connection model 135

5.5 Cost of building the device 138

Chapter 1

Internet of Things (IoT)

Oxford dictionary describes *Internet of Things (IoT)* as the interconnection, via the internet, of computing devices embedded in everyday objects, enabling them to send and receive data. Technically speaking, IoT is all about turning everyday objects into digital products and services, bringing new value and meaning to the lifeless things, making it more intelligent. Ronen and Shemir [52] define it as "network devices which bridge the physical and virtual world". Effectively, this means adding cloud connectivity and computing capabilities to the regular objects, along with adding backend services and web and mobile apps for viewing and analyzing data and communicating with those devices.

The concept of a network of smart devices was discussed as early as 1982, with the modified Coke vending machine at Carnegie Mellon University becoming the first internet connected appliance. Raji [48] published the first paper in 1994 to talk about automation in everyday appliances. Since then it has caught the attention of the research community to convert everyday things into smart objects. The term *internet of things (IoT)* was coined by Kevin Ashton of Procter & Gamble, though he prefers the phrase "internet for things". At that point, he viewed radio-frequency identification (RFID) as essential to IoT, which would allow computers to interact with individual

things [36].

The advancement of IoT into the modern world is phenomenal. This has the potential to bring the next revolution to the human society as well as devastate the very fabric of the society if not properly administered [33]. IoT is intrusive and has already extended into our everyday lives with wearable devices and home automation systems and eventually with become part of every city, town and country. It has the potential to become the most important innovation of the century and therefore will be an enormous potential for abuse. This abuse in turn would not only affect the owners of the respective devices or its manufacturers but also through sheer economy of scale, it would affect society at large. It is imperative to say that we need a well-defined security model that can ensure secure communications with these devices.

1.1 The question of privacy

New technology always comes with merits but also has potential concerns. In case of IoT, the concerns are with security and privacy. In the digital world, it is better known as cyber threat. The companies manufacturing the IoT devices focused on the functionality of the device and security take a back seat. The time-to-market became important to get the competitive edge. A user study was conducted with nine parent-child pairs to understand their privacy concerns about internet connected toys, like Hello Barbie and CogniToys Dino [38]. Unlike smart phones, these devices are always on, blending into the background until needed. All the parents in the research voiced their privacy concern with these toys. Another interesting study to find out if smart toys like Hello Barbie is maintaining the privacy of the children and users [25]. They concluded that Barbie's technology is not advanced enough to indicate the user whether it can keep a secret. The toy tends to share the private conversation of the child with the parents, third parties and potentially entire network of Twitter followers. A bad

actor within this ecosystem could cause enormous damage.

Preserving security and privacy of the devices is one of the biggest challenges of IoT. Due to the inherent vastness and openness, IoT is vulnerable to internet attacks. For example, there has been multiple incidents where the internet connected toys has been breached and hackers used those devices to eavesdrop on the owners. These issues can be avoided using a proper encryption technique [50]. However, implementing security solution in IoT is challenging because of its minimal capacity and lack of powerful wireless communications. As the devices lack computational resources, development of a lightweight efficient security techniques is of great interest.

Much of the data collected and communicated in IoT pertains to personal identifiable information (PII). Preserving privacy of the most sensitive data has utmost needs. Users must be provided with options to manage their own personal data and who they want to share it with. To ensure that the unauthorized access doesn't happen, identity management plays an important role. Social acceptability of IoT based applications depends on the privacy, anonymity, liability and trust. Many research has been done to portray the state-of-the-art security systems for IoT communications [50, 60].

1.2 Trust issues in heterogeneous environment

The concept of trust is used in different meanings. Trust is a complex notion about which no definitive consensus exists in the research community although its importance is widely acknowledged. A main problem with many approaches towards trust definition is that they cannot be easily quantified or measured. Moreover, the satisfaction of trust requirements is strictly related to user identity management and access control issues.

A variety of research efforts focus on assessing the trust level assessment of IoT systems [8, 14]. These researchers are conducted with the assumption that they are

Table 1.1: Contribution of ongoing European projects on IoT security [60]

	Butler	EBBITS	Hydra	uTRUSTit	iCore	HACMS	NSF	FIRE	EUJapan
Authentication	x			x	x	x	x	x	
Confidentiality	x	x	x		x	x	x	x	x
Access control	x	x		x	x	x	x	x	
Privacy	x			x		x	x	x	
Trust				x	x		x		
Enforcement									
Middleware		x	x		x				
Mobile	x						x		

used by general population without much awareness to information security and thus they are vulnerable to malicious attacks. The world of IoT is heterogeneous due to the numerous players in the market. As regards to worldwide projects shown in Table 1.2, several attempts are made to address IoT requirements in terms of security, privacy and trust in order to develop an unified framework or middleware. In IoT security, open issues faced by each project are summarized. Currently, the efforts are aimed at specific application contexts and the impact of these proposals on a mass-scale market still needs to be checked.

A better approach would be to not lay a trust on anyone but to validate every interaction. In that way, every device or person in the heterogeneous ecosystem of IoT can be sure that they are talking to a genuine other party. By eliminating the trust factor, we can better focus on privacy and security of the environment.

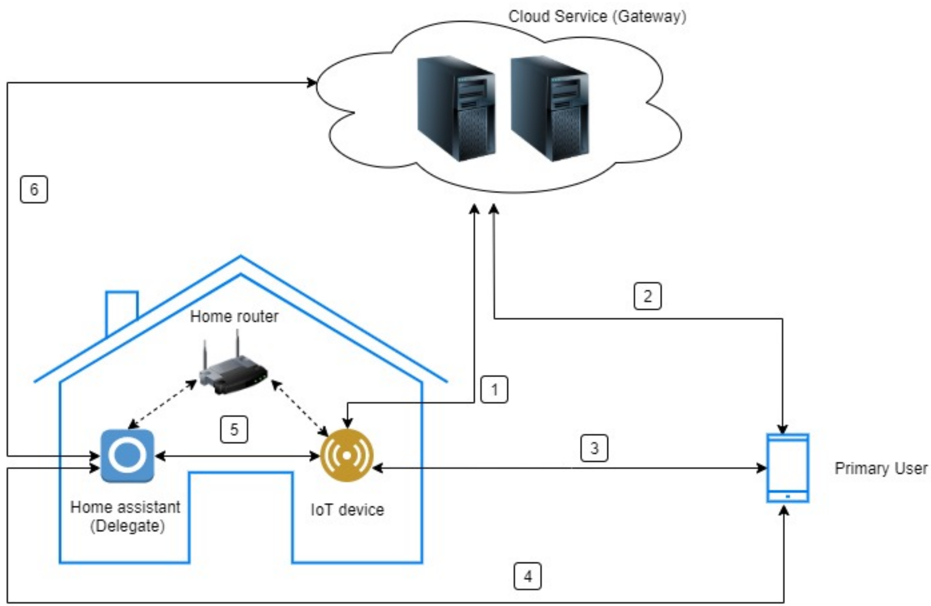


Figure 1.1: IoT architecture

1.3 The world without trust

As we see above, trust is a complex issue and most of the time is difficult to quantify. The problem we are trying to solve in this thesis is to eliminate *trust*, so that we can focus deeply on the remaining two aspects, namely, privacy and security. The authors [74] presented a eZTrust, a network-independent parameterization solution for microservices, where they shifted parameterization targets from network endpoints to fine-grained, context-rich microservice identities. If we closely look at the IoT devices, and talk about the functions that they play, we will notice that each internet connected IoT device is a small computer made for a specific purpose. They are communicating with a cloud-based endpoint (gateway) to pass some form of data to the device or person requesting it.

Figure 1.1 shows a generalized view of the IoT architecture along with its components. All of the actors communicate with each other seamlessly over the internet. If we categorize the communications individually, we notice the following:

1. IoT device talk with the cloud-based gateway
2. User device talk with the cloud-based gateway
3. User device talks to the IoT device
4. User device talks to the delegate device
5. Delegate device talks to the IoT device
6. Delegate device talks to the cloud-based gateway

The above clearly demonstrate that each actor communicates with the other for a specific purpose. Each of them may be manufactured by a different organization, having different build and configurations. Putting trust in another party in this diverse ecosystem would not be wise. Taking an analogy of the real world, we don't hand over the key to our house to anyone without knowing who they are. The situation is similar here with IoT devices and all the other components. Neither one of the components should accept or give data to another without validating their identity. The data must also be protected as it is being transmitted over the wire. Data should be protected in store as well as in transit and should be made available to only those who should have access to it. This points back to the CIA trident of information security, *i.e.* confidentiality, integrity and availability.

We propose a **zero-trust framework** for protecting the communication over the untrusted internet. The zero-trust model [27] was introduced by the analyst firm Forrester Research in 2010 to confront new attack methods in information security. Behind this model there is the simple principle of "never trust, always verify" Thus, all data traffic generated must be untrusted, no matter if it has been generated from the internal or external network. It is a strategic initiative that helps prevent successful data breaches by eliminating the concept of trust from an organization's network architecture.

Zero-Trust is a new way of thinking about information security. By adopting the concept of zero-trust architectural framework, companies can become more secure by

enforcing the compliance requirement at every step of the way. A zero-trust management model for loT would help to guarantee that the authenticity of every resource of the infrastructure are validated every time they try to communicate. Thus, evading the system is prevented. Also, a zero-trust management model might help to guarantee that every message between resources is compliant to the policies of the company. Finally, it would help to ensure that the nature of every transaction is verified before executing it [55].

The methodology we are going to use to implement zero trust is by provide a unique key for each pair of device and user that communicate with each other. The device and gateway verify each other using their own signature that is generated by their individual private keys. While in transit, we are planning to use SSL/TLS which encrypts the data and protects it from replay and other network-based attacks. Our goal here is to show that we can use the existing cryptographic infrastructure to provide an end-to-end encryption that can be easily integrated into the current network infrastructure. The framework provides the guidelines to enforce the principle of “never trust, always verifyâĂŹâĂŹ and can also be easily integrated in the current network architecture. We provide guidelines to move heavy data driven operations to the gateway and only use the device resources for its specific purpose. The gateway also provides logging for audit management.

1.4 Validating zero-trust

With the growth of number of internet-connected devices, the amount of data collected and stored is growing at scale. With the proposed security architecture, we anticipate protecting this data both in storage as well as in transit. This data needs to be encrypted at all places and should be only available to the intended recipient. Taking an example of an internet enabled thermostat, the owner can use a mobile app to get the current

temperature of the house or ask the device to change the temperature based on the userâĂŹs need. He can also ask a home assistant (like Alexa or Google Home) to do so on his behalf.

If we consider the example of the owner (primary user) asking Alexa (delegate) to increase the temperate in the thermostat, a lot of communications are happening over the wire. The user is using a Bluetooth technology to talk to Alexa. Then Alexa is interpreting the command and talk to the cloud provider of the thermostat to pass the instruction to the thermostat to do the work. Our goal with this thesis is to ensure that all this communication is secure, and each party specifically knows who they are communicating with. No one in this ecosystem must blindly trust anyone else. In the proposed model the cloud providers work as relay of information to the device. They should not be able to interpret the instruction that was given to the thermostat, nor should the providers or Alexa be able to change the instruction that was given by the primary user. On the other hand, if the primary user wants to know the current temperature of the room, no one other than the user should be able to read the data.

To demonstrate the above scenario, we want to setup AWS API endpoints to work as the gateway for us, and use a mobile app as a primary user. For the device, we want to build a temperature and humidity sensor using DHT22 Digital Temperature and Humidity Sensor connected to NodeMcu V3 board that can work as our IoT device. The internet connectivity of the device would be provided by WIFI Module in the ESP8266 microcontroller. The device will also be able to communicate with the Bluetooth module HC-05 present in it. The Bluetooth module will be used for setting up communications with the user and delegates. The device (with the help of the gateway) will be able to interact the user as well as the delegate and send notifications as and when necessary. Figure 1.2 demonstrate the proposed internal structure of the setup.

Looking at the taxonomy of IoT architectures [62], we realize the resource limitations of the IoT devices. Like mentioned above, the IoT devices are built with a specific

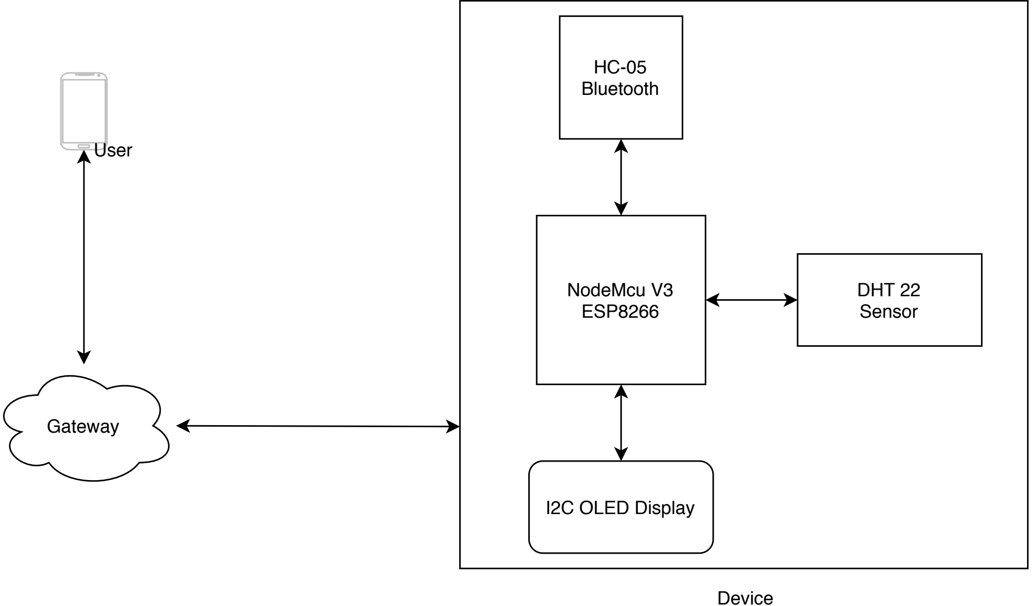


Figure 1.2: IoT proposed internal architecture for temperature and humidity sensor

functionality in mind. They have very limited memory and CPU resource available to them. With the cryptographic techniques that we are going to use to in the process to ensure privacy and security, should not use too much resource, that the device becomes useless. We are keeping a threshold of 80% of memory utilization to would measure the actual usage in the device.

The other validations that we want to perform are security, time of execution and cost. Providing adequate security to all communications is the key to this thesis. We would validate that there is no information leakage anywhere in the model. We would also verify the resistance of the device against a massive load of TCP requests and see how it protects the availability of the device. Time of execution is critical for user interactions. For web applications three seconds round trip time (RTT) is considered a good benchmark. We would keep this benchmark to verify how much time it takes for a command to get executed and the result to be returned to the user. We would investigate the cost of the device as well. As we noted above, we are assuming that this technology would be used in devices that are used by general users and we want

to keep the cost to an optimal so that its affordable to everyone. In short, our analysis and validation of the model would be in respect to security, memory, time and cost.

1.5 Thesis organization

The rest of the thesis is organized as follows:

Chapter 2 investigates the security concerns of IoT ecosystem. We explore the cause and effect for the vulnerabilities of IoT devices and talk about the unprecedented growth of the number of IoT devices in the market. We also look into the different attacks that was conducted by making the IoT devices as botnets and see how malware like Mirai exploited the security loopholes to cause a 1.1 Tbps DDoS attack with 148,000 vulnerable IoT devices.

Chapter 3 discusses the current architectures and security frameworks that are proposed in related works. We look at the current protocols like Zigbee light link and other network protocols proposed by the research community. We look in the trust issues and introduce zero-trust framework. We discuss how using zero-trust framework we can implement the simple principle of *"never trust, always verify"*.

Chapter 4 walks us through the lab setup and provide details of how we implemented zero-trust in the heterogeneous environment. We discuss code samples and look at data structures that are implemented to achieve the goal. We also discuss the different encryption techniques we use to optimize the cryptographic setup, maintaining the resource limitations.

Chapter 5 validates our claim of zero-trust security with limited resource usage for IoT devices. We will use a variety of advance tools like Wireshark to validate the network logs. No one other than the intended user should be able to read the data. We validate the logs in the gateway and the device to ensure that there is no data leak. We also discuss the resource utilization of the microcontroller board to verify our claim of no-more-than 80% resource usage. We analyze in terms of security provided, memory utilized on the device, time of operation and cost of building the device.

Chapter 6 concludes with our learning and outcomes. We go over the security aspects that are not covered by the zero-trust framework and provide some insight as to how we can mitigate them in future projects. We briefly discuss the future of IoT and how it can shape the world. We also point out how law and policies can effect the usage of IoT devices.

Chapter 2

Security of IoT Devices

With the sudden growth of the market of internet connected smart devices, researchers are both exited and concerned about the security aspect of internet enabled devices. Multiple proposals have proposed to support the security infrastructure of these devices. However, we have not come up with a standard that is followed by all manufacturers. Although there are multiple RFS proposed by Internet Engineering Task Force (IETF). Exhaustive survey about IoT security has been done to understand the current vulnerabilities of these devices [43]. As mentioned before, privacy and security are the number one concern for both the researchers and general users.

However, before discussing the proposed security frameworks and concerns with IoT devices, we would investigate the reasons for the unprecedented growth in the IoT market and how it has been abused to perform massive attacks.

2.1 The unprecedented growth

The internet of things market is highly competitive owing to the presence of many large and small players in the market operating at a global level. It was valued at USD 193.60 billion in 2019 and is expected to reach USD 657.31 billion by 2025, at a

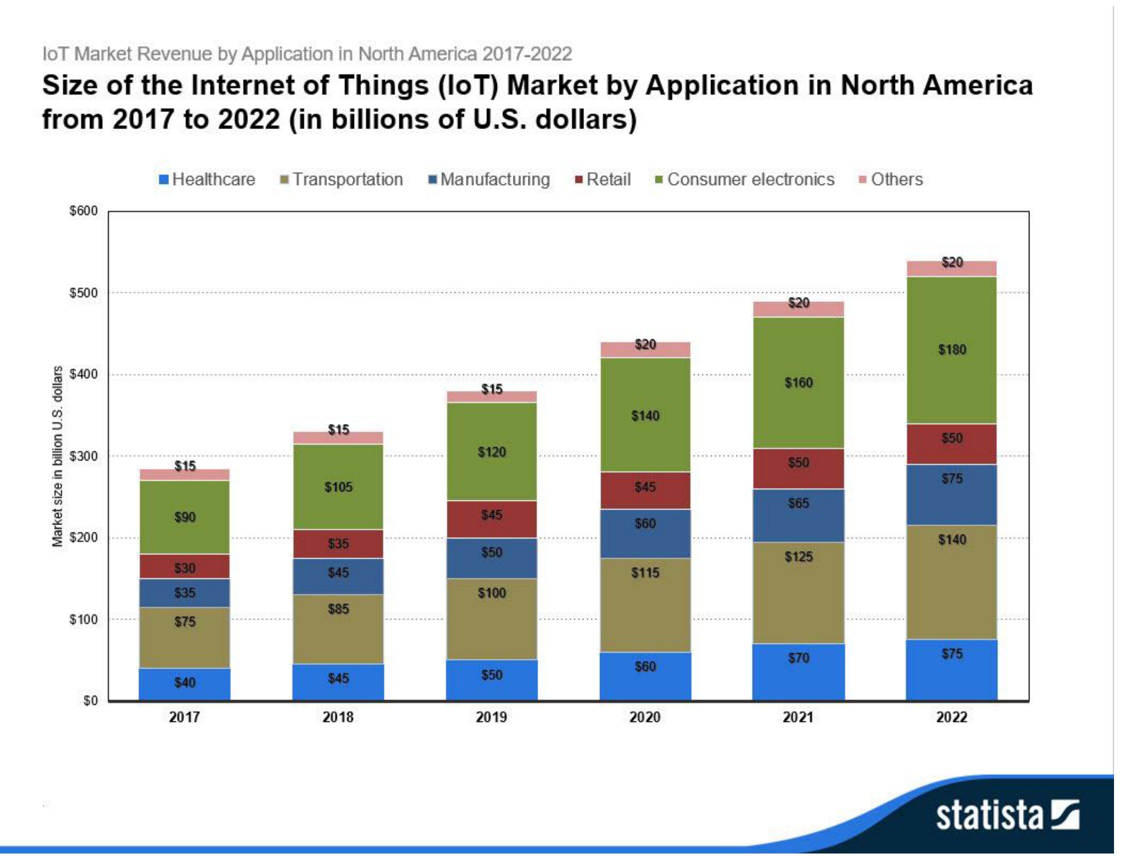


Figure 2.1: Growth of IoT by industry 2017 – 2022 [16]

CAGR of 21% over the forecast period 2020 – 2025. With the development of wireless networking technologies, the emergence of advanced data analytic, a reduction in the cost of connected devices, and an increase in cloud platform adoption, the market is expected to grow at a positive rate. According to the report from Gartner [16], there will be 5.8 billion internet connected IoT endpoints in use by the end of 2020. Utilities will be the highest user of IoT endpoints, totaling at 1.17 billion in 2019, and increasing by 17% in 2020 to reach 1.37 billion endpoints as seen in Figure 2.1.

Forbes [11] did a round up of the market estimate for IoT in 2018 and here are their findings:

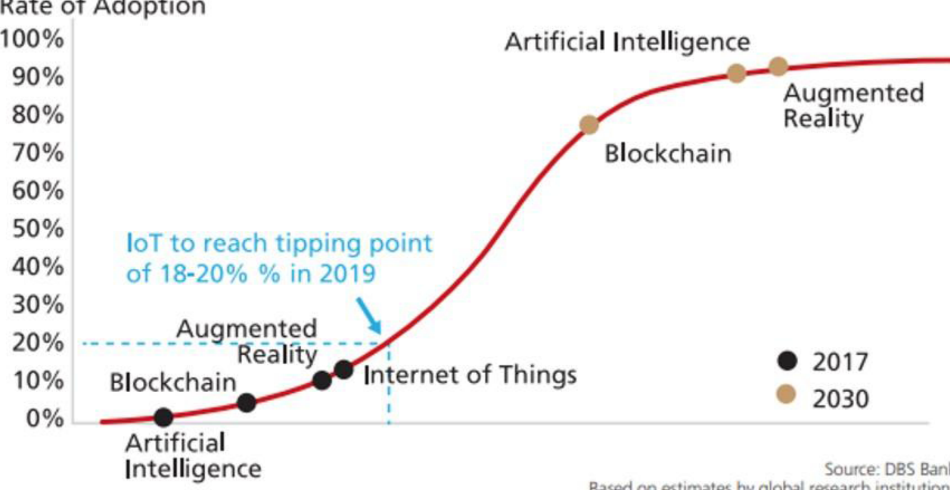
- The number of cellular IoT connections is expected to reach 3.5 billion in 2023, increasing at a CARG of 30%

- IoT installed base will grow from 6.3 million units in 2016 to 1.25 billion in 2030
- Worldwide technology spending on Internet of Things to reach USD 1.2 trillion in 2022
- The market of industry 4.0 products and services is expected to grow to USD 310 billion by 2023
- 90% of senior execs in technology, media and telecommunication says IoT in some or all lines if their business

The combination of new technologies like artificial intelligence, machine learning and real-time data streams and mission specific devices, the business case for using IoT devices in the organizations is compelling. IoT is becoming the cornerstone of many organizations' remodeling, enabling them to optimize existing processes and excel at new business models. Since the IoT devices are very focused in doing one operation, businesses are finding extreme value in installing these devices to meet their needs.

Similarly, general home users are also finding lots of value in using these devices. The iRobot Roomba has changed the landscape with home vacuum systems. These IoT devices are doing an excellent job of automation of everyday work. With the marketing done for the connected light system, it is also a source of amusement for everyone. The connected light bulbs like Philips Hue are giving options to users to control their home lights with home assistants like Google Home and Alexa. The demand for these devices is growing fast and so the manufacturers are trying their best to put up with it. In the process, a thorough penetration and security test is being sidelined. As per the Figure 2.2, IoT adoption rate will rise to 176% by 2030.

With the sudden growth of connected devices, security issues in the implementation is obvious. Many malware are released in the market, some even open sourced to increase the target rate. With Mirai being made public, the number of infected devices has doubled. In the following section we will discuss the different attacks conducted



IoT adoption gaining momentum

	2016	2017	2018	2030
IoT units installed base - total (m)	6,382	8,381	11,197	125,000
Consumer devices (m)	3,963	5,244	7,036	75,000
Consumer devices as a % of total devices	62%	63%	63%	60%
Connected devices per person	5	5	5	5
World population (m)	7,400	7,600	7,700	8,500
IoT adoption rate	**11%**	**14%**	**18%**	**176%**

Source: DBS Bank based on estimates by Gartner, United Nations, World Bank

Figure 2.2: Adoption of IoT by 2030 [16]

by using the weakness of IoT devices and misusing the technology to cause massive DDoS attacks.

2.2 Attack on IoT devices

Looking at the numbers above, it is obvious that Internet of Things is a sweet spot for the security researchers and hackers alike. With this huge competition, the manufacturers have enormous pressure to bring these smart devices to the market to get the

competitive edge. This resulted in lack of security testing of the devices before sending it out to the stores. Many small manufacturers didn't even provide option to patch the devices and so if a vulnerability is found, there is no way to mitigate the problem. Many papers have been written to demonstrate the weakness of the connected lighting systems.

Research was conducted to demonstrated how vulnerabilities in the Zigbee Light Link (ZLL) protocol can be used to exploit the connected lighting system [41]. With their penetration framework, researchers were able to gain full control over three popular connected lighting system, namely, Philips Hue, Osram Lightify and GE Link. They were able to gain access to the devices from beyond 30 meters making ZLL-based systems susceptible to war driving. In another experiment the researchers took it further to demonstrate that they were able to read leaked data from a distance of over 100 meters using cheap and readily available equipment [52]. They tested both high end smart lighting systems (Philips Hue) as well as low-end systems (LimitlessLED). A common conclusion from both the above-mentioned attack frameworks was that IoT designers need to focus on security issues during design, implementation and integration of IoT devices.

2.2.1 Malware

Malware (or "malicious software") is a code or program that propagates through the network to infect victims. It steals data, performs reconnaissance and other operations that an attacker wants to do. Based on their functionality, malware can be divided into different categories [7, 53, 63, 72]. Usually, they work to achieve following objectives:

- Provides command and control (C&C) to the attacker
- Sends other malware from the infected system to other targeted systems.

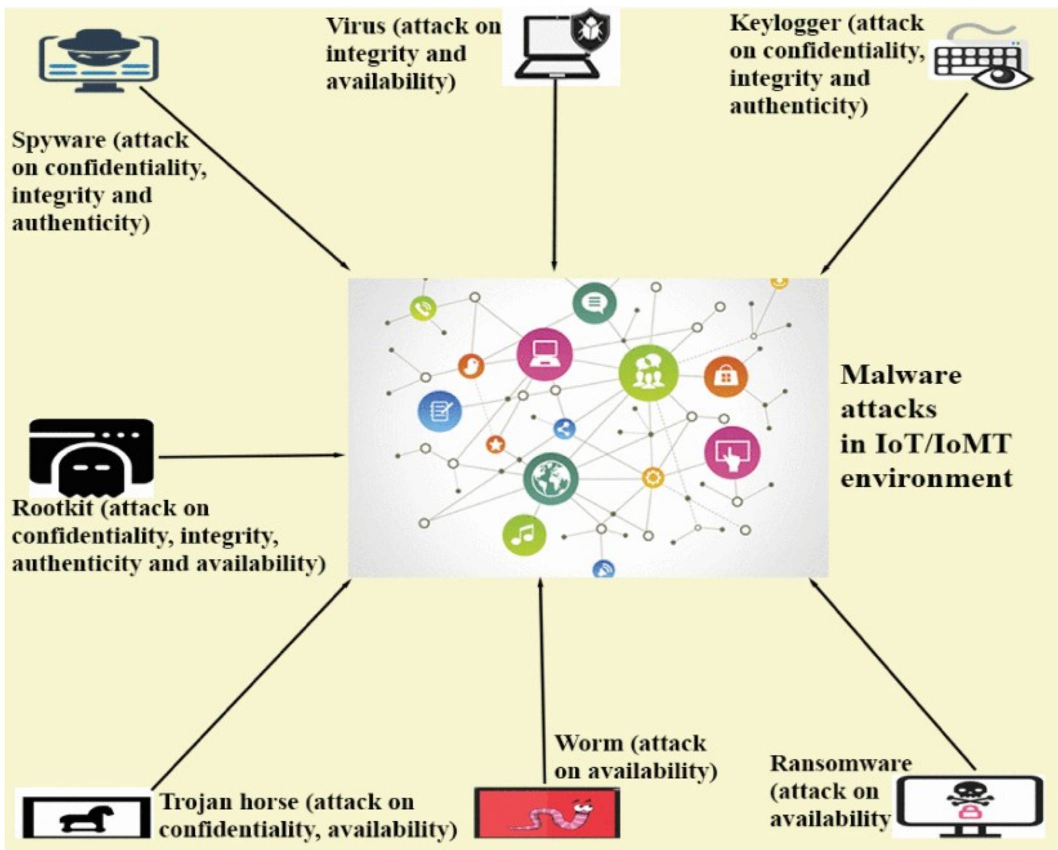
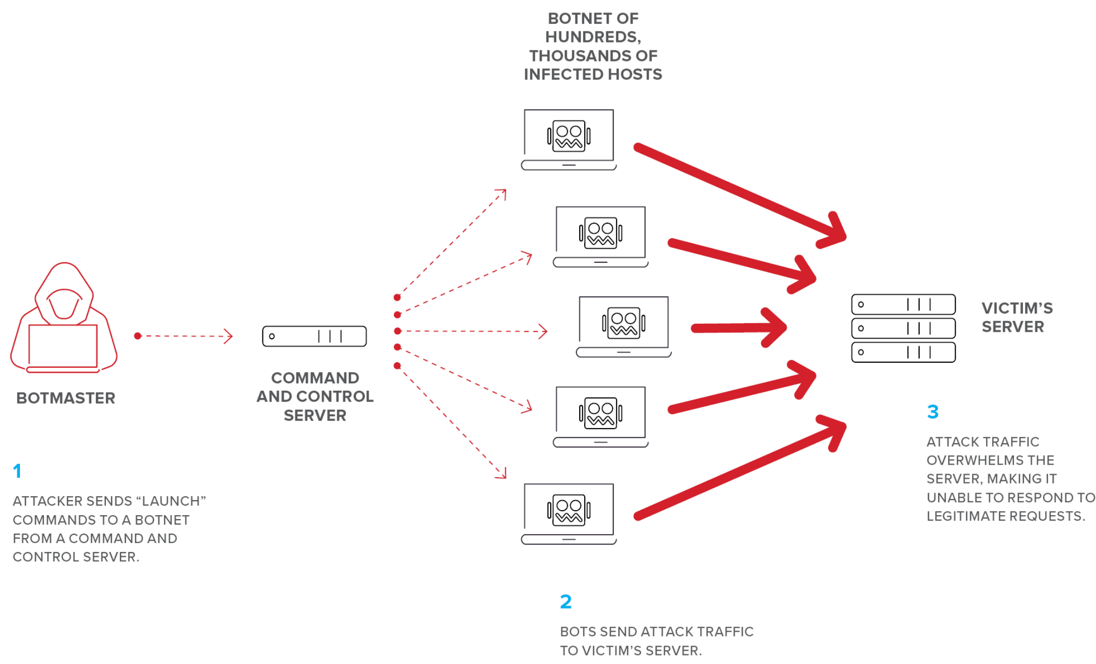


Figure 2.3: Different types of malware attacks in IoT [70]

- Investigates the local network of the infected users' system to launch further malware attacks.
- It is used to steal the sensitive data (*i.e.*, credit card information) from an infected

A details of various types of malware attacks on IoT devices is provided in Figure 2.3. Each one of them effect one or more of the CIA triad (confidentiality, integrity and availability). For example, a ransomware attacks availability of the system whereas a virus targets both availability and integrity of the victim. Rootkit targets every aspect of security, confidentiality, integrity, authenticity and availability of the effected system [70].



Source: f5.com/labs/articles/education/what-is-a-distributed-denial-of-service-attack-

Figure 2.4: Distributed Denial of Service (DDoS)

2.2.2 DDoS and IoT Botnets

Distributed Denial of Service (DDoS) attack is one of the most interesting and widely seen cyber-attack in the modern times [68]. In DDoS, a hacker temporarily enslaves a number of internet enabled devices into an arrangement known as *botnet* as shown in Figure 2.4, and overwhelms the victim with simultaneous requests so that it ignore legitimate requests from end users. It can exhaust the bandwidth (communication medium) and resources of the device. These attacks are carried out by organized criminals for financial gains, revenge, extortion or activism.

Another important aspect about DDoS attack is that the attacker deploys a layered attack with multiple attack vectors to camouflage the real attack [4]. Therefore, the real intention of a DDoS attack can be difficult to comprehend. Apart from disrupting the daily operations, DDoS can be used to probe the defense of a victim or to just distract the target during the actual attack using a different technique. Verisign in

its report [40] observed that 52% of DDoS attacks that were investigates in Q2 2018 employed multiple attack types. There was a 35% increase in the number of attacks, with a 49% decrease in the average of attack peak sizes, when compared to Q1 2018; however, the average of attack peak sizes has increased by 111%, year over year.

Looking at the recent incidents where the vulnerabilities of IoT devices are exploited, it is mainly utilized by botnets to launch wide range of DDoS attacks. Researches have shown that most DDoS attacks in recent time originate from three types of devices of which almost 96% were IoT devices, approximately four percent were home routers and less than one percent were compromised Linux servers. The IoT botnets not only affect the owners of the device but also anyone on the internet.

The threat of these IoT devices are important concern because they are hard to fix. IoT devices are low hanging fruit for the attackers. The existence of these botnets is known since 2008. Key characteristics of IoT malware used to orchestrate DDoS attack are as follows:

- IoT malware are Linux based.
- Majority of the malware has limited or no side effects on performance of the host. They are activated by the command and control head.
- The malware are stored in the device memory (RAM).
- The IoT malware are difficult to re-mediate since it doesn't use any traditional technique of DoS/DDoS attack like amplification.
- The IoT based botnets generate high volume of traffic ranging from 100 Gbps to 1.6 Tbps.
- The botnet consist of devices distributed all over the world.
- Other than the conventional TCP, UDP traffic, some IoT botnets generates unconventional payload like GRE traffic and use uncommon "DNS water torture" technique during DDoS attacks.

2.2.3 Recent IoT based attacks

As mentioned in Section 2.2.1, one of the main objective of the attacker is to remote control the infected machine. In most of the attacks that took place on IoT device, the attack vector targeted to setup a command and control (CC) with the infected devices. This was done, so that the attacker can later use these devices to perform massive attack on some other victim

Before looking into the different IoT malware, we would like to highlight the attack on Dyn Inc. DNS servers [21]. On October 21, 2016 a series of DDoS attacks were targeted on the systems of the Domain Name Service (DNS) provider Dyn. The attack caused major internet platform and services to be unavailable to a large group of users in Europe and North America as shown in Figure 2.5. Major companies like Amazon, Paypal, Visa and others were effected. The DDoS was accomplished through numerous DNS lookup requests from tens of millions of IP addresses. The activities are believed to have been executed through a botnet consisting of many internet-connected devices, such as printers, IP cameras, residential gateways and baby monitors that have been infected with the Mirai malware. The common issue in all these devices was that it used default credentials in the device.

Some of the active botnets which can launch various malware attacks in IoT environment are discussed below [26, 30, 32].

- **Mirai** is one the most popular DDoS IoT botnet in as of today. At its peak, Mirai infected 4,000 IoT devices per hour and currently it is estimated to have little more than half a million infected active IoT devices. It is famous for being used in a 1.1 Tbps attack with 148,000 IoT devices. Mirai targets mainly CCTV cameras, DVRs and home routers. With the release of the Mirai source code, the number of IoT infected device has doubled. Based on the IP addresses, we can identify that the devices are distributed in over 164 countries with majority in Vietnam, Brazil, US, China and Mexico. With its massive attacks it came into light and along with

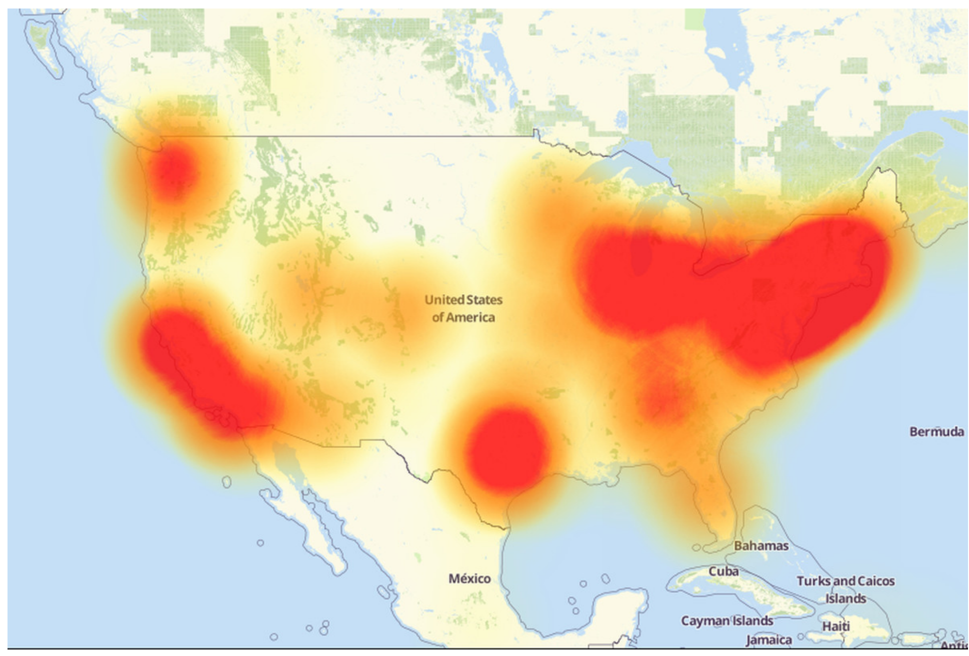


Figure 2.5: Parts of US effected by Dyn DNS attack [21]

it brought the awareness of DDoS attacks to the media.

- **Reaper** also known as IoTroop. In the fall of 2017, information security researchers discovered a different strain of botnet with improved functionality. It was capable of compromising IoT device much more quickly as compared to the Mirai botnet. Mirai infects the IoT devices which use default usernames and passwords. However, reaper is more aggressive since it targets nine different vulnerabilities in the devices of different makers, such as D-Link, Netgear and Linksys. Using this botnet, the attacker could also morph the code to avoid detection from signature based IDS . As per the information provided by 'Recorded Future', it was also used to attack on some EU banks (for example, ABN Amro).

- **Echobot** was discovered in the beginning of 2019. It is a variation of Mirai which uses 26 malicious scripts to spread itself. Like other botnets, it takes the advantage of unpatched IoT devices and then uses these vulnerabilities to harm other enterprise applications (for example, weblogic of oracle). It was discovered by Palo

Alto Networks, and designed to create a larger botnet to perform DDoS attacks.

Recent studies have shown commercial availability of DDoS service. Once can anonymously order five to six Gbps DDoS attack lasting ten minutes or more for as low as USD 15.00. DDoS-as-a-Service (DDoSaaS) providers sell the attack capabilities for knocking websites offline or perform stress test on different network infrastructure. Recent studies [56] have found out over 435 booter and stresser websites are available in open internet. However, there is much more offer on the Darknet. Popular once include ExoStress, BetaBooter, ZStress, Titanium Stresser, *etc*.

2.3 Security concerns

There have been other minor DDoS attacks caused by the IoT botnet networks, like on November 07, 2016 the email publication server of Wikileaks services were made unavailable by a DDoS attack that continued for 24 hours. Next lets evaluate a few of the common vulnerabilities that are present in IoT devices that make it an easy target for the attackers [43]:

- **Insufficient physical security:** The majority of IoT devices operate autonomously and not supervised [37]. With minimal effort, an adversary might obtain unauthorized physical access to such device can cause damage to them. This can lead to unveiling employed cryptographic scheme, replication of the firmware using malicious nodes, or simply corrupting the device itself.
- **Limited resources:** IoT devices characteristically have limited memory and storage and do not necessarily possess the technology to recycle them [66]. An attacker might drain the memory by generating flood of spammed messages, rendering the device unavailable for valid processes and users.
- **Inadequate authentication:** Limited resource in the devices challenge the implementation of complex authentication mechanism [19]. An attacker might exploit ineffective authentication to install spoofed malicious nodes, or violate data integrity, thus intruding into the corporate network. Many platforms use default credentials for authentication, which doesn't provide adequate authentication and can be easily replicated.

- **Improper encryption:** Data need to be protected both in transit as well as in storage especially those operated in critical systems (*i.e.* power utility, manufacturing plants, building automation, *etc.*). Encryption is an effective tool to store and transmit data in a way that only authorized users can read it. As the strength of the cryptographic system depends on the randomness and size of the key, with limited resources it becomes challenging to store huge keys. To this end, the attacker might be able to circumvent the encryption technique to compromise the device [9].

- **Lack of access control:** Strong credential management is necessary to protect IoT devices and its data from unauthorized access. In the state-of-the-art platforms many organizations do not enforce the requirement of a strong credential [49]. Moreover, after installation, manufacturers do not require a change of the default credentials. To make matters more difficult, these users operate with the highest privilege.

- **Backdoor ports:** Many a times developers and technicians keep ports open in these devices to perform troubleshooting steps. However, these ports might have vulnerabilities that can be exploited by an adversary. They might use these backdoor channels to deploy malware and use the device as a bot.

- **Lack of patch management:** Every code has some or the other bug in it. A proper patch management system is essential to fix these bugs in the upcoming releases of the firmware or software. In multiple occasions we notice that the manufacturer either doesn't provide a patch or there are no capability available in the device for patching [64].

- **Missing audit management capabilities:** An audit trail helps to diagnose the issues in a device or system. It is an important tool for the support team to perform a root cause analysis (RCA). In numerous situations it is seen that the manufacturer doesn't have a audit mechanism for the device and thus any issues or attacks on it goes unnoticed and rectified.

- **Weak programming practice:** It is always recommended to follow a proper development guidelines and implement an effective testing mechanism by which known errors and issues can be identified, many researchers [29] have reported that multiple firmware are released with known vulnerabilities such as backdoors, root access, and lack of Secure Socket Layer (SSL) usage. Hence an attacker might easily exploit these weaknesses to perform attacks like buffer overflow, information modification, or gain unauthorized access to the device.

In a study with connected lighting systems [41], it was realized that the first critical point is the trust in the safekeeping of a pre-shared master key that is shared among multiple manufacturers. These keys play an important role in securing the network

communications with these devices. In a ZLL-based connected lighting system, manufacturers rely on the NDA-protected shared key. The ZLL link key, used for classic communication hasn't been leaked yet, but of course it can happen anytime even by a mistake. Once leaked, almost every device using the ZLL protocol are prone to be attacked and used for other malicious purpose like being used in a command and control system by a attacker.

2.4 Proposed frameworks to secure the gaps

There are a lot of research done to define a security architecture for IoT. The biggest hurdle in creating an IoT framework is its heterogeneous nature. Multiple devices from different manufacturer have already flooded the market. Establishing an objects identity in IoT is critical to the privacy and security of both the device as well as the user. A lack of a unified model has led to the manufacturers coming out with their own proprietary model and researchers aiming at formalizing it. An effective authentication mechanism is crucial to establish trust in the IoT ecosystem. In the following section we will evaluate the proposed frameworks in terms of architecture, connection with the cloud and processing commands.

We started our literature review with a broad overview of the security challenges and reference architectures of IoT [61]. IoT devices are used in every sector including healthcare, retail, airlines, and others. It can also perform micro-transactions on itself. The major market for e-commerce in IoT is smart homes and wearable devices. Figure 2.6 shows an IoT architecture in e-commerce consists of warehouse management center, cloud data centers (PaaS, IaaS and SaaS), and backend databases consisting of catalog, customer, online purchase, *etc.* The warehouse is connected through RFID sensors to catalog and shipping product database, and cloud data centers. Similarly, delivery is connected through RFID sensors to shopping carts, shipped products, and

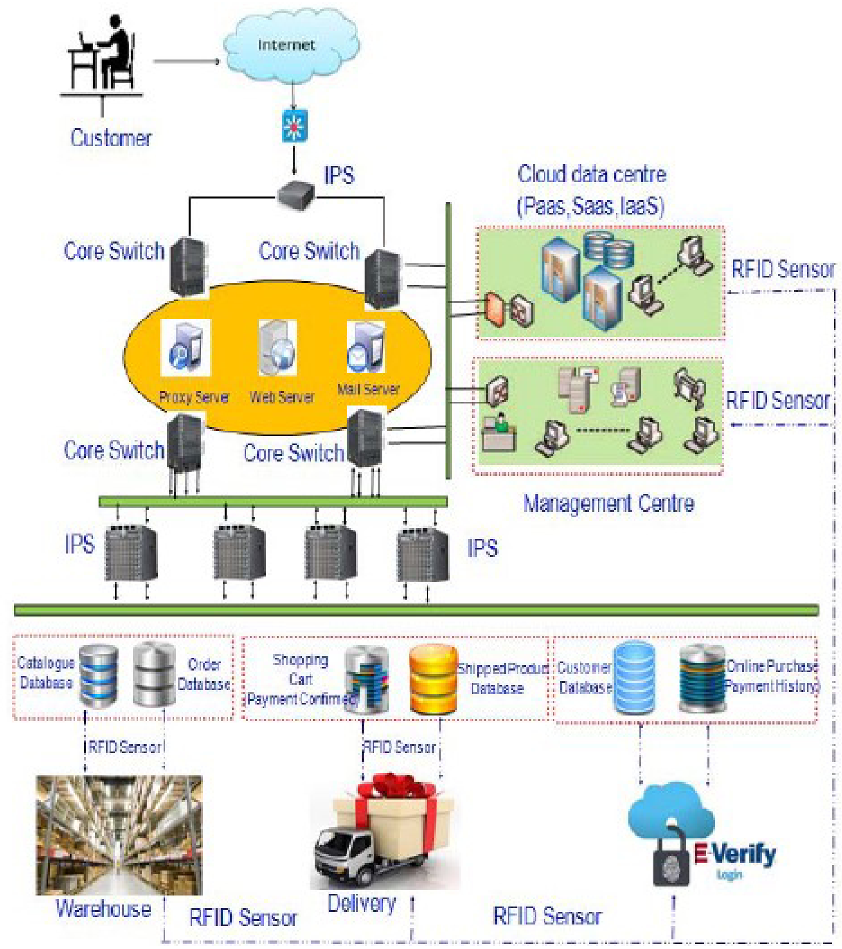


Figure 2.6: Reference IoT Architecture for E-commerce [61]

management centers. With the advent of cloud architectures, the devices can easily connect to the gateways and exchange data on a regular basis.

2.4.1 IoT security

Many low power radio technologies suitable for IoT already exists in the market, including Bluetooth Low Energy (Bluetooth LE), IEEE 802.15.4, Z-Wave, ANT, Dash7, Wave2M and low power variant of IEEE 802.11 [18]. Bluetooth LE has the highest potential for IoT use which is still missing IP capabilities. Nieminen et. al. [44] presented a holistic solution for integrating Bluetooth LE devices with the IoT. The central piece of the solution is the IPv6 over Bluetooth LE specification that is currently produced by the

IETF. Data transmission is protected by Bluetooth LE link layer security which supports encryption and authentication by using the Cipher Block Chaining Message Authentication Code (CCM) algorithm and a 128-bit AES block cipher [20]. Using this solution to build an end-to-end security framework would require a change in the current protocol structure. It requires a change in the Bluetooth LE protocol and would need substantial discussion and verification before implementation. However, the pairing of devices using the Bluetooth LE framework is powerful and secure.

ConnectOpen was proposed to automate the integration of IoT devices [47]. In this framework a flexible communication agent is deployed at the gateway and can adopt to multiple communication protocols. The agent is automatically deployed on the gateway in order to connect the device to a central platform where data is consolidated and exposed via REST API to third party services. It showed some practical implementation of the framework to implement in a large scale. The framework has the security service embedded in the integration services. It is an interesting framework and remove the IoT tangle. However, it does not address the security aspects specifically. There is no clear explanation as to how the author wants to address them.

A DTLS based end-to-end security architecture was proposed for the low power hardware platform suitable for IoT [31]. The proposed security scheme uses public key cryptography (RSA), and works on top of the standard low power communication stack. One strong notion that we can get from this article is that we can re-use the established standards, existing implementations and security infrastructure to enable an easy solution. Another study showed that RSA, the most commonly used public key algorithm in the internet, can be used in sensor networks with the assistance of Trusted Platform Module (TPM) [22]. A TPM is an embedded chip that provides tamper proof generation and storage of RSA keys as well as hardware support for RSA algorithm. RSA provides strong encryption. However, the key length of RSA for efficient security is 2,048 bits. This requires a lot of storage in the device to store the key and certificate

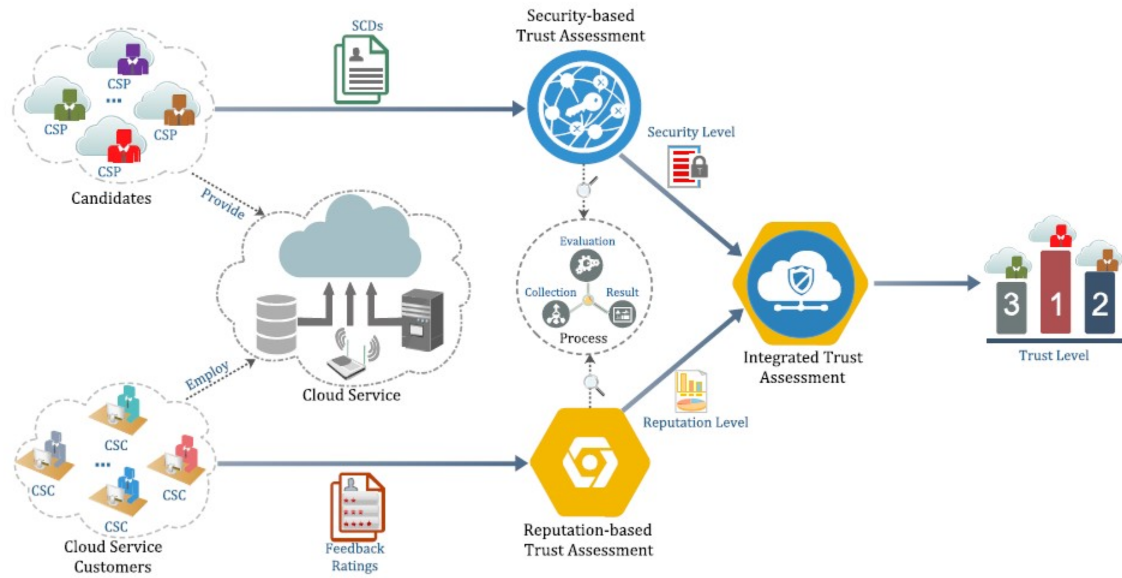


Figure 2.7: Trust assessment framework for cloud services [34]

for RSA.

A dynamic defense architecture was also proposed to handle security threats to an extent in the past period of time [35]. The proposal explains six dynamic and circular defense segments. The result from all the segments are used to create a defense strategy. It causes the defense measure against security threat active and the whole defense procedure is dynamic. Another cloud based IoT security model was proposed by using trustworthy cloud services [34]. It detailed a trust assessment framework for security and reputation of cloud services. The framework enables trust evaluation of cloud services in order to ensure security of the cloud based IoT by integrating security and reputation based trust assessment model as shown in Figure 2.7. As we mentioned before measuring and quantifying trust is a difficult task.

2.4.2 IoT authentication

Establishing the identity of a user or device comes before verifying trust by access control mechanisms. Cryptographic mechanisms incur high computation and communica-

tion load. The resource constrained IoT devices does not easily support cryptographic solutions. The gathering and use of data from smart devices that people interact with on a daily basis has several implications related to privacy [3]. In this section we evaluate the different techniques proposed to authenticate actors in the IoT ecosystem.

A one time password (OTP) scheme for IoT was proposed based on elliptic curve [59]. The private key generator generates the OTP and at the same time, it assumes the role of validating at the IoT platform. OTP generation occurs in a scheme that runs in 4 phases, namely, setup, extraction, generation and validation. This method depends on the Lamport algorithm for its security by generating the OTP. It has been proven with experimentally that experienced attackers can perform replay and modification attack on OTP based authentication systems and access the system. Moreover, it becomes cumbersome to generate and use a OPT for every communication to and from a device.

A certificate based authentication technique was presented to redress the issue of password based authentication [6]. This technique requires that a certification authority issues certificates to users. The certificate is issued and certified by a remote authority who seals the link between them and public cryptographic key. The certificate-based authentication works under very strict principles, providing that those who issue certificate and those who award access be differentiated. The interconnection proposed, consist of smart objects, gateways and services.

An identity management (IDM) [54] system was proposed to focus on providing access authorization as well as authentication for IoT users. The presented authentication and key based authentication method provide single sign-on to IoT devices. This technique integrates four components, namely, the entity or the user, their identity, identity provider (IdP) and service provider. The benefits of IdP is that it makes non-interactive login possible. This method provides better identity check via private keys. They are more secure than passwords, as malicious users must obtain the private key and their corresponding passphrase to use the system.

The work published in [24] focused on authentication of devices in smart home environment using the physical properties of IoT device and communication. The smart home environment uses a variety of communication protocol. The proposed methods rely on using both the physical unclonable function (PUF) and physical key generation (PKG). The authors promote that the combination of two methods results in an immediate enhancement of security. They also suggest reusing existing hardware to reduce overall system cost. The security mechanism used in this technique is a random set of challenges and symmetric key encryption.

Another novel idea was proposed [23] to use blockchain system with credit based consensus mechanism for Industrial IoT (IIoT). The proposal consists of a credit-based proof of work (POW) mechanism for IoT devices, which can guarantee system security and transaction efficiency simultaneously as shown in Figure 2.8. In order to protect sensitive data confidentiality, they designed a data authority management method to regulate the access to sensor data. In addition, the system is based on directed acyclic graph structured blockchains, which is more efficient than the Satoshi-style blockchain in performance. The structure described is not only suitable for smart factories but also is able to adopt to various IIoT scenarios.

2.5 Summary

There are several shortfalls in terms of secure communications of IoT systems. This attribute to the lack of proper encryption being employed to cipher the data being transmitted. This results in security breaches costing companies huge penalties. The data also requires hiding when in transit. Otherwise it might be a target of sniffing and that can lead to critical and personal data being leaked to the unwanted recipient. As noted in the Section 2.3, some solutions use a pre-shared master key that is protected by NDA. A strong key management system is important to evaluate to protect

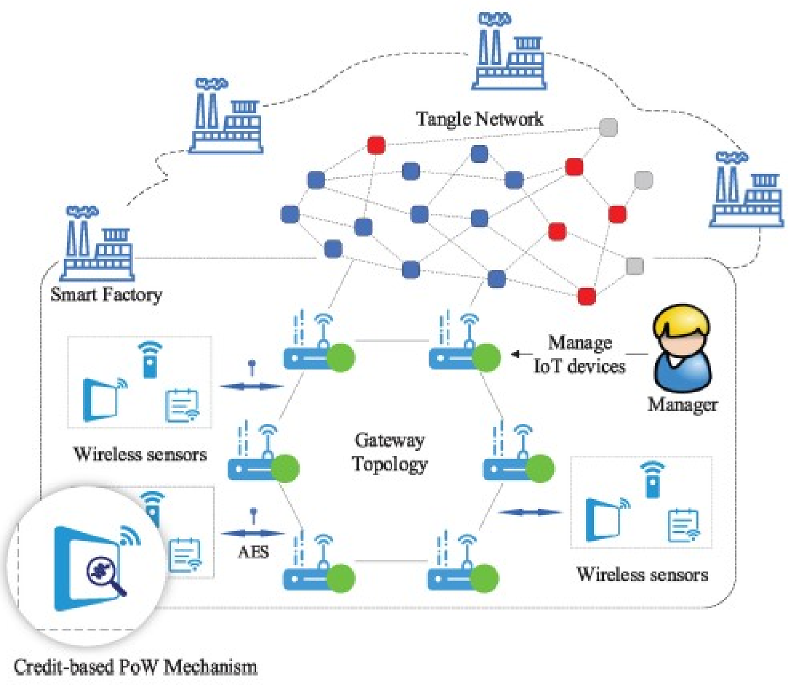


Figure 2.8: Architecture of blockchain-based IIoT system for smart factory [23]

the devices along with the data it deals with.

The methods discussed for implementing security and authentication for IoT devices have undermined the limitation of the one password scheme. If an attacker were able to intercept messages transmitted at the login stage, they can replay it and access the system. In a password-based system, once an attacker can get into the system as a legitimate user, they have broken the kill chain. It will be very difficult to distinguish an attacker from a legitimate user in this scenario. In practical IoT devices we have seen that the manufacturers use a default credential to allow the user to plug-and-play. This caused such devices to be used as bots using malware like Mirai. Many authentication techniques mentioned above are prone to impersonation attack. Most of the devices along with their cloud backend, failed to enforce strong password policy.

It has been mentioned by almost all the research that for IoT the biggest challenge is the constraint in resources. IoT devices have a constraint on resources including energy, memory, computational speed, and communication bandwidth. Scares re-

sources necessitates only lightweight operations on the smart objects, particularly if a distributed authentication is implemented to achieve scalability.

The IoT ecosystem is very heterogeneous. Users and devices are interacting with each other over an unreliable and untrusted medium. It is unavoidable to verify every access from the user as well as the device. This is because neither of them knows from where the request or response is coming. One thing is obvious, we need to use some cryptographic solution to overcome the privacy and security problem. Keeping the resource limitation in mind, the IoT industry needs a solution that can be easily and effectively used without putting trust on the medium.

We propose a zero-trust framework that can work with the above limitation and provide an end-to-end authentication system for every communication. In this solution we rely on the existing cryptographic infrastructure that can be easily integrated into the existing network topology.

Chapter 3

Zero-Trust Framework

The internet was build with a *trust* that it would help people communicate easily and knowledge can be shared. People will have easy access to information and would be able to learn if they are willing to. From the early days of internet, websites have made a concerted effort to reassure their customers that their personal data and transactions are safe. In his article "trust online, trust offline" Eric [67] raised the question, *"are people generally trusting?"* He describes that there are two types of trust, namely, strategic trust and moralistic trust. Strategic trusts reflect our experiences with particular people doing things. Moralistic trust is an optimistic view we learn at an early age: the world is a good place and it is not so great a risk to agree that most people can be trusted.

The internet can be both trusting and mistrusting at the same time. People who have a higher percentage of moralistic trust, are open to try new things. They are of the opinion that a few bad incidents cannot define that the world is a bad place. It is generally good to be open to try new things, but we need to be careful as well. Experiments fuel innovation and provides encouragement to people who are constantly thriving towards making the world a better place.

3.1 Understanding zero-trust

Looking at all the attacks described in Section 2.2, a person with strategic trust would get scared and avoid using IoT devices, and a person with moralistic trust would also become skeptical. We are connected by devices all around us in homes, offices, cafes, *etc.* In today's connected world, trust is only achieved by proof. If we go for a driver's license, we need to prove our identity before they allow us to go ahead with the process. Similarly, in the digital world, every entity needs to prove who they are. The burden of proof lies on the entity asking the service as well as those from whom the service is asked for.

Zero-trust is a strategic initiative that helps prevent successful data breaches by eliminating the concept of trust. Rooted in the principle, "never trust, always verify" zero-trust is designed to protect modern digital environment. It leverages network segmentation, prevent lateral movement, provide threat prevention and simplifies granular user access control. The zero-trust model recognizes trust as a vulnerability. Once in the network, users can move freely and access or exfiltrate whatever data or resource they are have access to. A malicious user in a network would want to escalate his privilege. We must remember from the Lockheed-Martin kill chain [51] that the point of infiltration of an attack is not the target location. Thus, these devices can be used for reconnaissance and network monitoring as well.

The concept of zero-trust is particularly important in the heterogeneous ecosystem of smart devices. With the huge growth in the number of connected endpoints, it is difficult to have a trust on a request or response that is coming from an unknown source over an untrusted medium. The solution is to verify *every* incoming packet. One of the biggest concerns in IoT devices, as seen in the previous section, is that the resources are minimal. So, to implement a strong authentication mechanism, we need to build an efficient framework where every request can be validated, and access can be determined before any other operation. The concept is the same as that for a web-

based authentication. For a protected route, we ensure that the web token is valid, and the user has the required role to perform an action.

In the next section we will look at the actors that are involved in the IoT ecosystem and understand the nomenclature that would be used throughout the rest of the book.

3.2 IoT architecture

We will start by analyzing a general scenario of using a home automation system like iRobot Roomba or a Philips Hue smart bulb. With the cloud connectivity, the device talks to some server to keep its information and then that information is transmitted into our mobile phone via an app. We issue commands to the device to perform some task. For an iRobot Roomba, we schedule its cleaning cycle or ask it to clean a certain portion of the house. Similarly, we instruct the Philips Hue to change color. With the advent of natural language processing (NPL) and devices like Alexa and Google Home, we send these instructions to the devices using voice commands. Using such home assistant, we issue commands in English language like, "*Hey Google, turn off the light in the bedroom!*".

In the above situation, we have a Philips Hue smart light in the bedroom, and the owner has downloaded the Philips mobile app to connect and configure the smart bulb. Then, the same light is added into the Google Home app as a bedroom light. Now let's look at the network communications that are happening as shown in Figure 3.1. On listening to "Hey Google" the Google Home device gets activated, and interprets the remaining part of the statement. It catches the phrases "turn-off" "light" and "bedroom." It contacts the Google server to find out what bulb is connected in the bedroom. It gets the information and then contacts the bulb manufacturer's server (in our case Philips) and requests it to contact the bulb and turn it off. Now, the manufacturerâĂŹs server looks through its database and finds the IP address of the bulb and sends out a com-

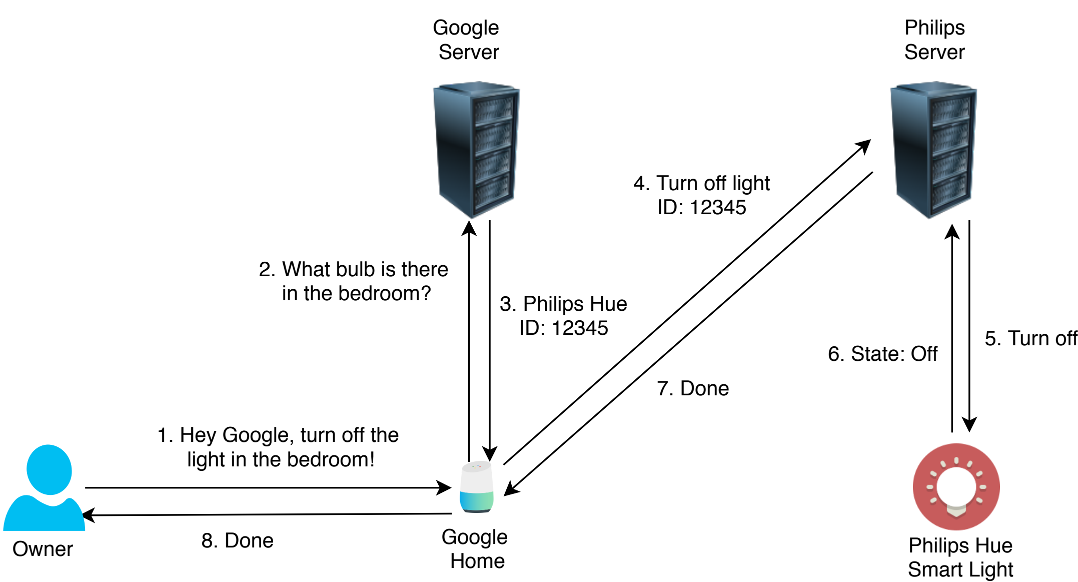


Figure 3.1: Generalized network communications for a IoT request

mand to turn off. The bulb performs the operation and informs the manufacturer that the work is done. The manufacturer saves the state and informs the user and Google Home that the operation is completed.

In the above scenario, both the Google Home and Philips Hue can be considered as IoT devices and they communicated with each other using the cloud infrastructure. Below is the taxonomy for each of these actors:

- **Device:** From now on we will be referring the IoT device as *device*. These are physical devices that have limited resources and are built to serve a specific purpose (*e.g.* Philips Hue smart light). They are used for home automation like lighting, heating, air conditioning, media and security. They can also be used in medical and healthcare, better known as internet of medical things (IoMT), as well as transportation to control inter and intra-vehicular communications.

- **User:** The user is the owner of the device, how purchased it from the market. The user downloads the manufacturer's app in their phone and help pair the device to his/her account. In the proposed architecture, there can be at most one primary

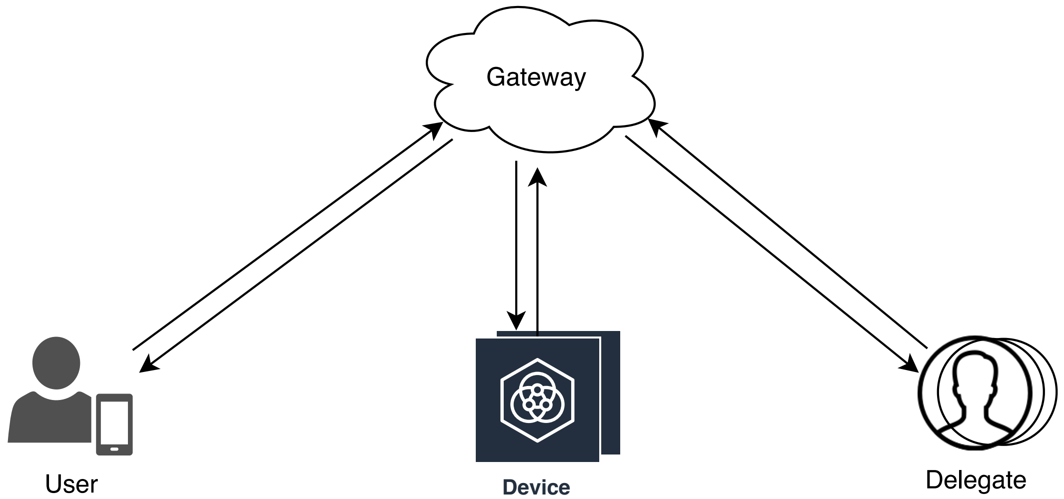


Figure 3.2: Proposed architecture for zero-trust framework

user to the device who registered it with the gateway server. Any other user who would want to connect with the device would need approval from the user.

- **Delegate:** They are all the other users and devices who would want to connect to the device. Google Home, in the earlier discussion, would be called a delegate in our architecture. They are responsible to communicate with the device on behalf of the user. Any interaction between the device and delegate should be logged and communicated to the user.

- **Gateway:** The gateway is the manufacturer's cloud interface that the device and user connect with to store and retrieve information. In the example above, both the Google server and Philips server are gateways, each to interact with its own manufactured device. The user can interact with the device through the gateway. Here, the gateway works as a relay to pass information between the user and the device.

A generalized proposal is shown in Figure 3.2. We are proposing that all communication to the device happens *only* through the gateway. This will avoid the device to respond to any request that comes to it. Any request whose origin and referrer

headers are not the gateway, will be responded with a error message. This will prevent the device from having to respond to every incoming request. The crux of this is that network chatter will increase and there will be two hops for the user to communicate with the device, but this latency is acceptable for implementing a strong security architecture. A lot of spoofing attacks can be avoided by implementing this solution. In our experiment we will measure the delay and validate if the latency is acceptable.

We are also proposing only one primary user for a device. Any other user will be considered a delegate. The delegate will also be able to communicate in the same way as the user, the only difference is the pairing. For the user, during the pairing process, the device is provided the Wi-Fi credentials and a shared key is established. For the delegates, they just create a shared key. However, before the shared key is activated, the user needs to authorize the pairing. Only after the approval, can the delegate communicate with the device.

3.3 Data structure

Following the basic principle of zero trust of "never trust, always verify" we are to build a strong security framework that can verify every request to any of the component in the ecosystem. In the proposed security framework, every component maintains a database of all the devices it is connecting to. This ensures that any request coming from a non-registered device is rejected and logged. In our architecture, the gateway acts as a relay for any communication that happens between the user (or delegate) and device. So, we will start by looking at the data structures that are maintained by the gateway.

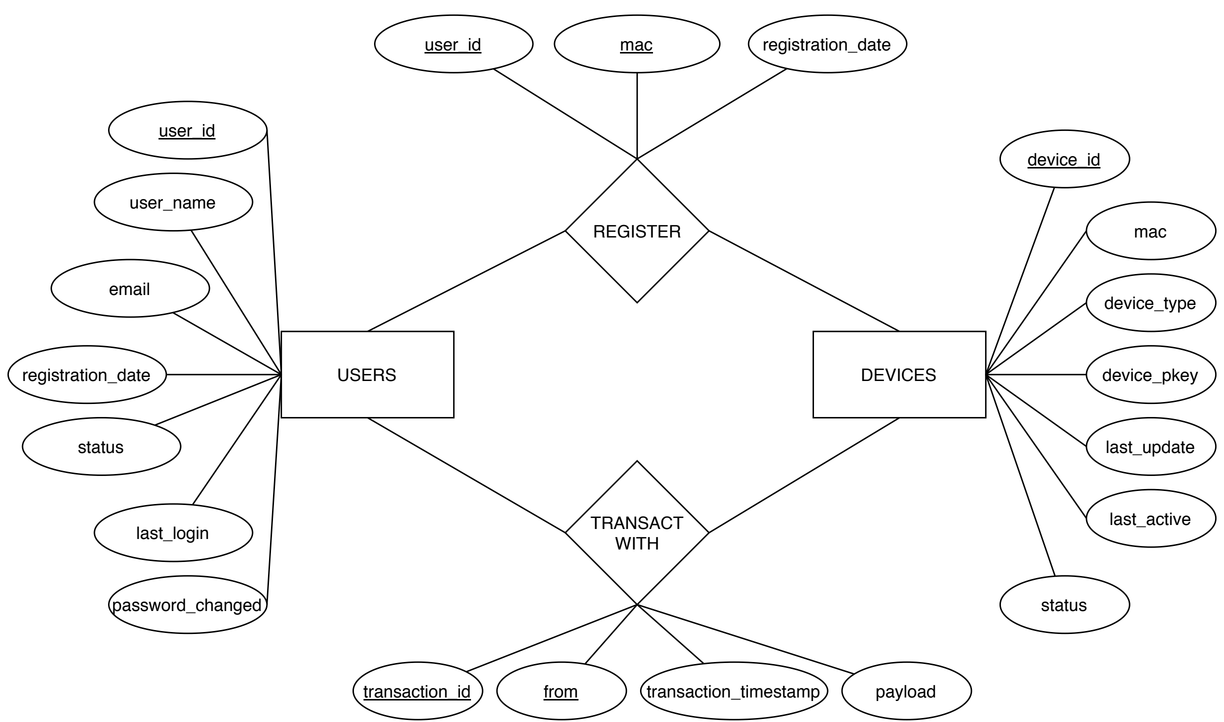


Figure 3.3: ER diagram of the gateway data structure

3.3.1 Data stored in gateway

The gateway is a cloud infrastructure that is hosted by the manufacturer. Since they create and market these devices, they will maintain a database of the same. In our design, we are proposing the following relations and attributes for a security framework.

DEVICES (device_id, device_mac, device_type, device_pkey, last_update, last_active, is_online, status)

USERS (user_id, user_name, email, status, last_login, password_changed)

REGISTRATION (user_id, device_mac, registration_date, is_primary, status)

TRANSACTIONS (transaction_id, from, to, activity_date, payload)

Figure 3.3 shows the Entity-Relationship (ER) diagram of the data structure in the gateway.

The gateway maintains a record of all the devices that is produced. During the production, each device is embedded with the *device_id* and an ECC private key of the device and an ECC public key of the gateway. The corresponding public key *device_pkey* of the device is stored in the database. The device is also provided with the server fingerprint as well as the public key of the server. This helps the device to communicate with the gateway in a secured manner. The initial status of each device is inactive "I" until the device is registered with a user. We will look closely at the registration process in Section 3.4.

Similarly, every user must register with the gateway as well. The user relation keeps a track of all the entities that interact with the devices. If, there is a device to device interaction, the device needs to be added as a user in this model. As described earlier in Section 3.2, every device has only one primary user, every other user/device that interact with it is a delegate and the user must approve the request for a delegate to interact with the device. The user approves the request for a delegate to interact with the device.

We have eluded to the fact earlier, that for security reasons we are enabling all communications via the internet with the device should go through the gateway. The gateway can keep a log of all transactions that happen through it. If there are multiple steps that are happening in a transaction, then all the steps are recorded individually by the *from* and *to* field, and the entire transaction gets the same *transaction_id*. We will see a clear example of this in command execution.

3.3.2 Data stored in devices

As mentioned earlier, the device maintains its own private key and the public key of the gateway along with it's *device_id*. For each of the user and delegates that it is paired with, it maintains a symmetric key. Since, the resources of the device are limited, it is recommended that we restrict the number of delegates to three per device. This

number can vary based on the storage provided with the device. For each user/delegate, it stores the relevant information as shown here:

```
struct UserInfo {
  char* userId;
  byte key[32];
  byte iv[16];
};
```

We will be using 32 bytes (256 bit) keys for each of the pairing along with 16 bytes (128 bit) initialization vector. Overall to save a record for one pairing, we would need 84 bytes (672 bits) including the UUIDv4 format for user ID. With four such pairing we will consume around 336 bytes (2688 bits). Adding the device's private key and gateway's public key and the device identifier we can restrict the storage to two kilobytes. If the device can support more storage, then more pairing is possible. The device also stores the Wi-Fi credentials that are shared by the user during pairing. The credentials are stored in a structure as follows:

```
struct WiFiCred {
  char* ssid;
  char* password;
};
```

It is difficult to assess the size of the credentials accurately since this can be any string set by the user.

3.3.3 Data stored in users and delegates

Users and delegates can have a similar data structure. Like, we mentioned above, both need to register with the gateway as a user. For this thesis, we will not consider users who are also devices. We will keep this for further research. Each of these entities are to maintain a record of the devices that they are paired with. If they are the primary

users, they can also sync a list of delegates that are connected to the device. Most of this information will be synced with the cloud, except for the pairing keys.

A symmetric key will be shared between the user/delegate and device. This key will be known only to the sharing parties and no one else. This key will be used to pass information between the user/delegate and device. The gateway will not have the authority to send commands, only approved users and delegates can do so. The key must be refreshed after every regular interval.

One thing we need to understand in implementing a security infrastructure is that, the complexity of the system should not be too high for a general user to use. Generating a key during paring will remove the requirement of having a static key embedded in the system or a default password. Those can be easily cracked, but an auto generated key which is refreshed at a regular interval poses some challenge to the attacker. Another advantage of this strategy is that the user doesn't need to remember any complex password.

A simple data structure to store the above information would be as follows:

DEVICES (device_mac, device_key, iv, key_refresh, last_update)

The *key_refresh* holds the information as to when the last time the key was refreshed. The manufacturer's app can keep a rule, that the key need to be refreshed after *X* days. From a general recommendation, we would say to follow the change password rule followed in most web/mobile applications, where the key is refreshed every 60 days. For the initiation, it will be the day of pairing. The *last_update* attribute will hold the timestamp as to when the user last interacted with the device. Most of the other information that the user would need, like the list of other devices can be got from the gateway and thus we donâĂŹt want to duplicate that information in the user. The user should have the option to add/remove/update a delegate.

One thing to note from the above data structure is that the device is identified primarily by the *device_id* between the device and gateway, whereas by *device_mac*

between the user and device. This is done to avoid using the same identifier in both ends. During paring, the device lets the user know it's MAC address and it is a unique value that can be used by the user to register itself with the device at the gateway. The gateway can link the MAC address with the right device_id and map the keys correctly during communication with the device.

3.4 P3 connection model

In the proposed model we separated the connection initiation from the communication over internet. For connection initiation, we propose using Bluetooth 4.0 or Bluetooth LE [12,17,44]. Bluetooth LE has been designed for ultra-low-power application, yet keeping similarities with classic Bluetooth. All modern mobile phones and smart devices are enabled with Bluetooth LE. Another reason to use Bluetooth in setting up secret keys is the area of access. Since the Bluetooth connection can be established only in proximity of the device, the attack vector becomes smaller. The initiation steps are similar between the user and delegates, except that the delegates need to get additional approval from the user to access the device. And the user need to provide the Wi-Fi credentials to the device so that the device can connect to the internet and reach the gateway. In the world of computers, we often hear the term plug-and-play, it refers to the concept where you can connect the device and start using it. For IoT we prefer to use the internet as a medium of communications so that we can access these devices from anywhere in the world. So, to establish a connection we use a *plug-pair-play* method termed as *P3 connection model*. In this method, the device is plugged to a power source and then it pairs with a user to establish the security credentials and only then the user can play with the device (or start using the device). The *pair* sets up the keys for the device and user to communicate over the internet.

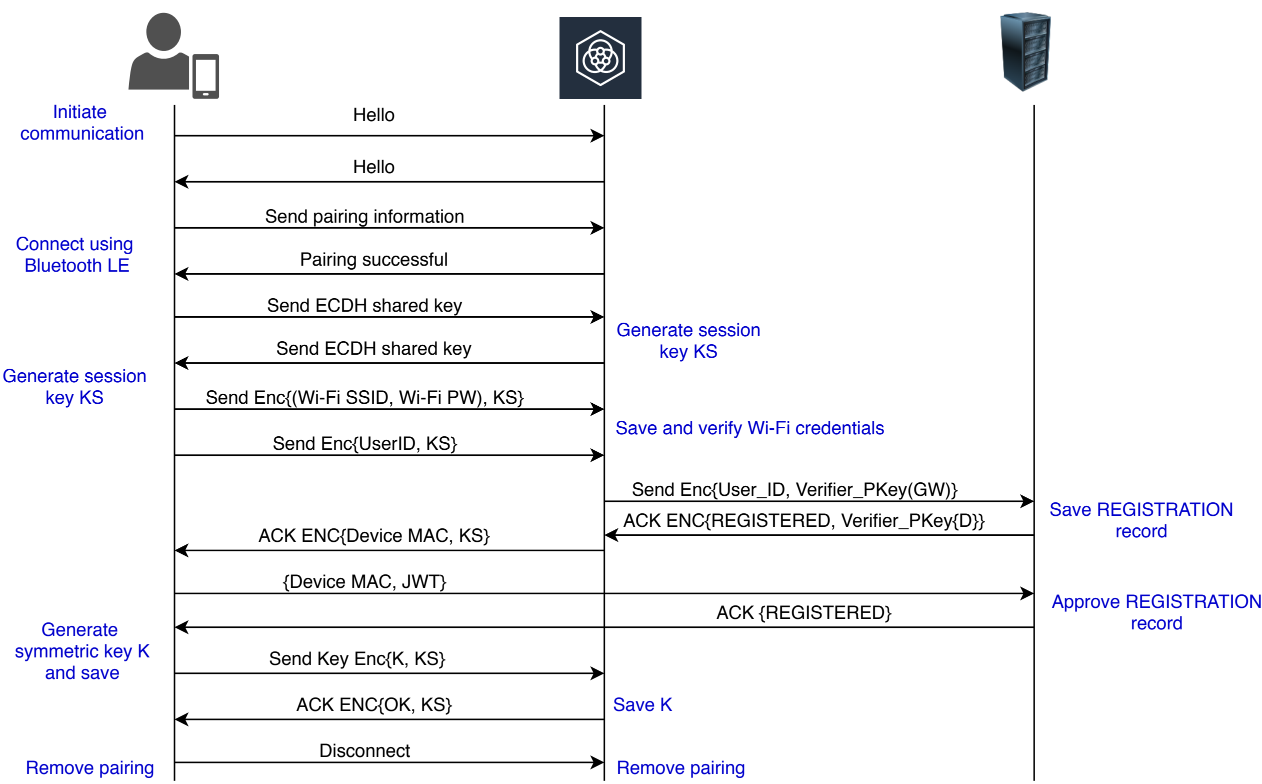


Figure 3.4: P3 connection between user and device

3.4.1 Connecting user and device

For the user, the connection process is initiated when the device is bought brand new and is connected for the first time. In this step the user connects the device to the internet and enables it to talk to the gateway. Before this process, we are assuming that the user has registered with the manufacturer and a record is added to the *USERS* relation. The process is shown in Figure 3.4.

- **Pairing:** The first step of the initiation is the pairing between the user and the device. The user from his mobile app searches to find available devices. In Bluetooth the connection happens between a master and a slave. In this case, the user's phone acts as a master and the device acts as a slave. Once the user can find the device, it pairs with it using the default pairing key embedded in the app and initiates a connection.

- **Generate session key:** In cryptography, Curve25519 is an elliptic curve offering 128 bit of security and designed for use with the elliptic curve Diffie-Hellman (ECDH) key arrangement scheme. Here, both the device and user, generate a shared key using the public keys available. On exchanging each other's public key, the session key is created. This session key K_s is used to secure the remaining transactions.

- **Connect to Wi-Fi:** Once the session key is established, the next step is for the device to connect to the internet. For this, the user sends out the Wi-Fi SSID and password encrypted with the session key $Enc\ \{<$ *Wi-Fi SSID, Wi-Fi password* $>, K_s\}$. On receiving this information, it tries to connect to the Wi-Fi and ensures a successful connection. Once connected, it saves the information into its memory to maintain a constant connection to the internet. It returns a "success" to the user.

- **User verification:** After connecting to the internet, the device need to verify the identity of the user. The user sends his *user_id* to the device encrypted $Enc\ \{user_id, K_s\}$. The devices send this information to the gateway along with the device's digital signature for verification over TLS. On receiving this information, the gateway validates if the given *device_id* is valid and is in an inactive state with no prior user. It also verifies, if the given *user_id* is valid and registered. If any of the validation fails, it returns a failure with status code 401. If the validations are successful, it creates a partial verified record in *REGISTRATIONS* table.

- **Device verification:** On receiving a successful verification of the user from the device, the device returns a *device_mac* to the user encrypted with the prior session key $Enc\ \{device_mac, K_s\}$. The user forwards this session key over a TLS session to the gateway API. The authorization used in this transaction is a JWT token that was created when the user logged into the manufacturer's app. The gateway

verified the JWT token for authorization and then checks the device_mac to verify it against the partial verified record in the *REGISTRATIONS* relation. Once verified, the gateway completes the transaction and informs the user in the app that a new device is added to its record.

- **Generate and share the symmetric key:** Once the user receives the confirmation from the gateway that everything is good, the user generates 256 bits symmetric key and shares it with the device $Enc\ \{K, K_s\}$. The device saves the same and acknowledges the user that the key is saved securely.
- **Disconnect:** The Bluetooth interface is only used to help connect and verify the user and device. Once this connection is established, there is no need to hold on to the connection. The user initiates a disconnect request and the device comply.

Now, the user can send commands using the shared key K to the device relaying through the gateway.

3.4.2 Connecting delegate and device

The delegate connects the same way as the user except the Wi-Fi credentials doesn't need to be provided any more. The user had already provided the details and the device is connected to the gateway via internet. A new step for the delegate is to get an approval from the user. This step ensures that the user is aware of all interactions to the device. This avoids unauthorized access and provides an additional layer of security for the device. The Figure 3.5 details the steps.

- **Pairing:** The pairing step is the same as in the user and device. The delegate acts as a master and the device as slave. The pairing happens with a default pairing key that is embedded in the app.

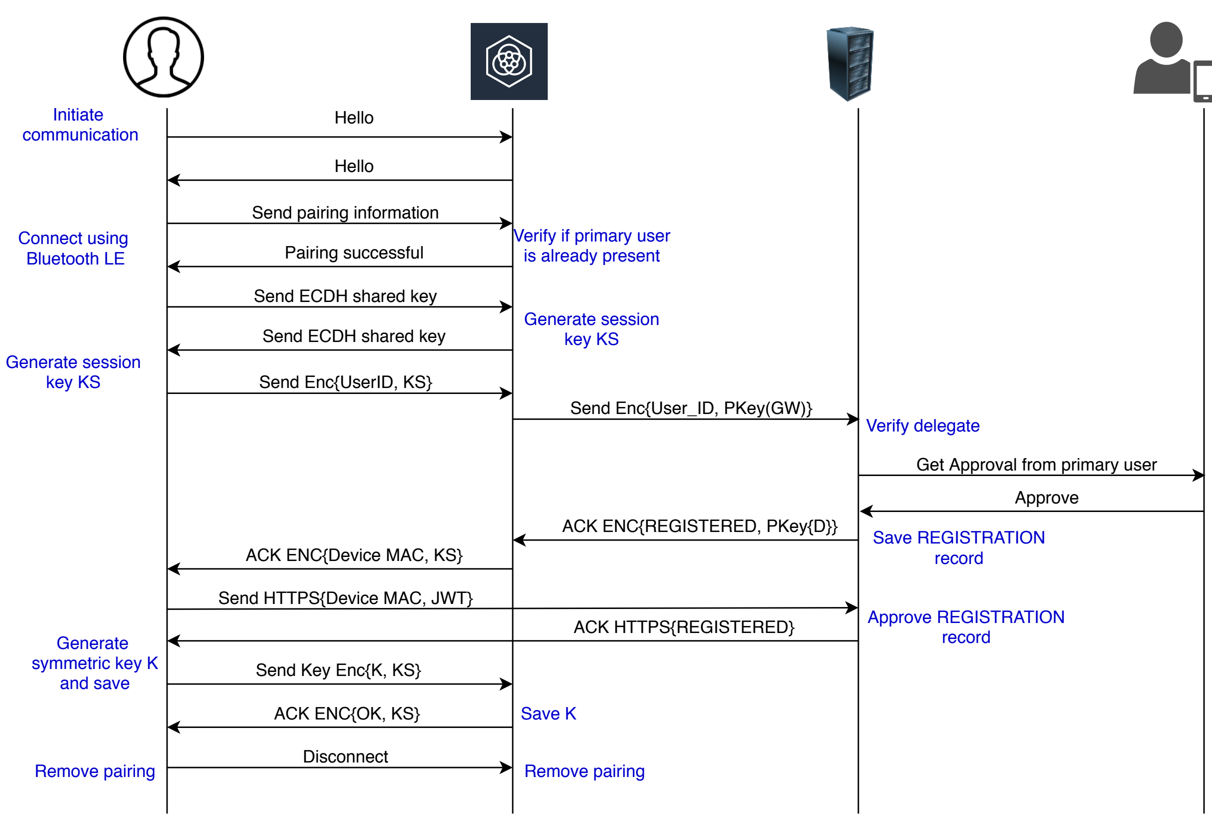


Figure 3.5: P3 connection between delegate and device

- **Generate session key:** Curve25519 is used as a cryptographic technique to generate a ECDH session key. This session key K_s is used to for all the remaining transactions.

- **User verification:** After connecting to the internet, the device verifies the identity of the user. The user sends his *user_id* to the device encrypted $Enc\ \{user_id, K_s\}$. The devices send this information to the gateway encrypted with the gateway's public key $Enc\ \{< user_id, device_id >, PKey_{GW}\}$. On receiving this information, the gateway validates if the given *device_id* is valid and finds the primary user. It also verifies, if the given *user_id* is valid and registered. If any of the validation fails, it returns a failure with status code 401.

- **User approval:** This is an additional step that takes place for the delegate. The user needs to approve the request to add another user to the device from the app. The gateway sends the request on behalf of the device to the user using notifications or email. The user looks at the request and approves the same from the app. Once approved, the gateway creates a partial verified record in *REGISTRATIONS* and sends the information to the device. If the user rejects the transaction, then the same is communicated back to the device and the pairing is canceled.

- **Device verification:** This step is same as in the user and device. The device sends the device_mac $Enc\ \{device_mac, K_s\}$ which in turn is send to the gateway for verification and completing the registration process. The gateway verified the JWT token for authorization and then checks the device_mac to verify it against the partial verified record in the *REGISTRATIONS* relation. Once verified, the gateway completes the transaction and informs the delegate and user with email that a new device is added to its record.

- **Generate and share the symmetric key:** A 256 bit symmetric key is generated by the delegate and shared with the device $Enc\ \{K, K_s\}$. The device saves the same

and acknowledges the user that the key is saved securely.

- **Disconnect:** The Bluetooth communication is terminated and the delegate can interact with the device now.

One of the advantages of the P3 model is that there is no pre-shared key or password embedded in the device for communicating with it. This has been a criticism of the existing IoT implementations as described in Section 2.3. This model helps generate the key and can also be used to re-generate them at a regular interval. The user doesn't have to type in or remember a complicated password. The system generates and saves a secured key that can be used in all communications.

This model also follows the principle of zero-trust where both the parties verify the identity of each other before creating a shared key. If any of the step fails validation, the communication is terminated, and the user is informed of the invalid connection request. It prevents the attackers from gaining access on the device without the knowledge of the user. The user also has the option to terminate a delegate as and when he wishes. If by some means, the attacker becomes the user, the gateway is informing the legitimate user that the device already has a primary user and the user can report back the issue to the manufacturer. The manufacturer can reset the device and trace back the attacker from its transaction history.

This security model of plug-pair-play will be simple to use for an end user and can also maintain a strong security protection for both the user and device.

3.5 Communication over untrusted medium

P3 connection model creates a strong authentication mechanism for the user and device to interact with each other safely. The generated keys help with authentication as well as confidentiality. The next important issue we wanted to tackle is the communication with the devices. The internet is an untrusted medium. There can be people listening

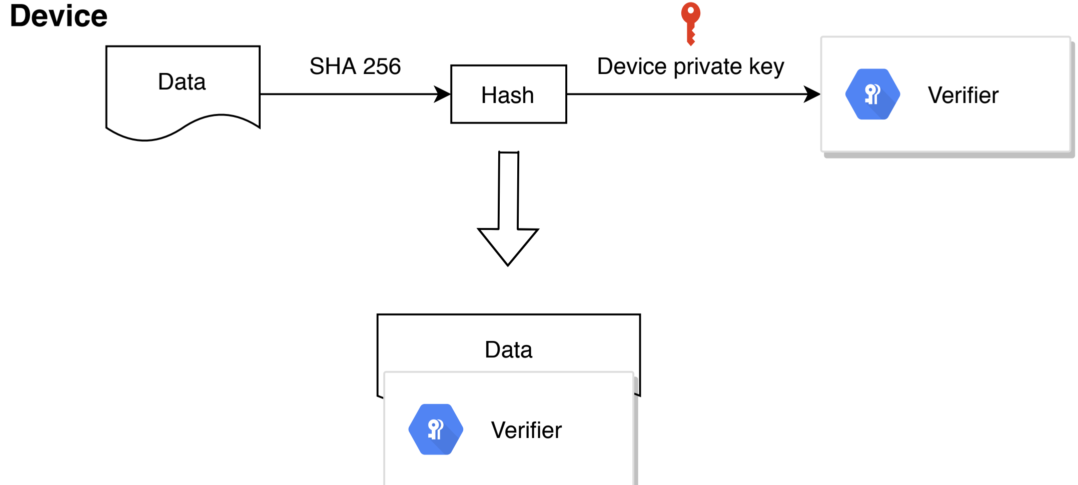
to the conversations by wiretapping or man-in-the-middle (MITM). Sniffing is beyond the control of the IoT ecosystem because the data is traveling over many routers and it is beyond the control of anyone to protect all of them. The next best thing we can do is secure the conversations and make the information unreadable in transit. Even though someone is listening to it, they will not be able to make out anything from it. We will be using the keys and data structures described above to implement a strong end-to-end communication where all transactions are verified and logged.

3.5.1 Heartbeat communication

The heartbeat protocol has been implemented in many frameworks and network level applications. This is used to indicate the health of a device. In our architecture, we are going to use the heartbeat to tell the gateway that the device is active and functioning well. The same can also be used to send data to the server for analysis. For example, we can use the heartbeat to send CPU and memory utilization of the device for it to perform an analysis of the device usage. For a sensor device, it can send the sensor data through the heartbeat to the gateway to perform machine learning and predictions. In short, we can reuse the heartbeat communication to cater to the need of the device and user.

In our model, the gateway provides an API endpoint to which the device can send the heartbeat communication. This URL is protected by SSL/TLS with a server certificate. The device verifies the certificate before sending the data to the gateway using the server's fingerprint. From the perspective of the gateway, it also needs to verify the identity of the device. As mentioned above, the device has its own private key. The corresponding public key is saved in the *DEVICES* relation of the gateway. The detailed process of the device verification is shown in Figure 3.6.

The device creates a SHA 256 hash of the heartbeat data and encrypts it with its own private key to create a digital signature. Then it sends hash along with the current



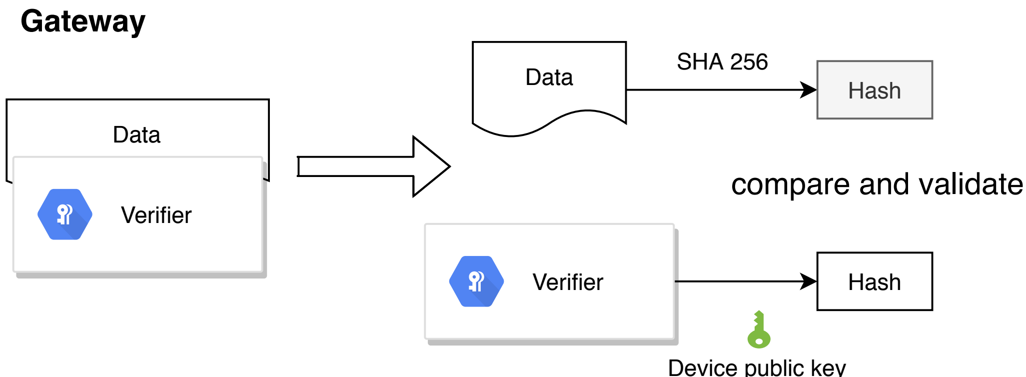


Figure 3.6: Heartbeat from device to gateway

timestamp, device_id and heartbeat data.

$$< device_id, current_timestamp, heartbeat_data > \rightarrow data$$

$$< data, Enc\ \{data, PrivKey_{device}\} > \rightarrow package$$

The gateway receives the *package* and extracts out the data from it. It takes the device_id and finds the public key for the device from the *DEVICES* relation. If the *device_id* doesn't return any record from the database, then the message is logged as error and discarded. Once the record is found, it extracts the last *heartbeat_timestamp*

record and compares the timestamp. The given timestamp must be later than the one saved in the record otherwise this is a replay attack. Once both the verification succeeds, the gateway computes a hash of the same. Next it extracts the provided hash using the device public key and compares it against the computed hash. If there is a match, the device is verified, and the heartbeat operation proceeds. The pseudo code for the operation is described below:

```
heartbeat(data, encryptedHash):
  assert data.deviceId, ''Missing deviceId''
  assert data.curTimestamp, ''Missing curTimestamp''
  assert data.data, ''Missing data''
  deviceinfo = getDeviceInfo(data.deviceId)
  if not deviceInfo:
    raise Exception(''Invalid deviceId'')
  if data.curTimestamp <= deviceInfo.hbTimestamp:
    raise Exception(''Invalid timestamp'')
  computedHash = computeHash(SHA256, data)
  providedHash = decrypt(encryptedHash, deviceInfo.pubKey)
  if computedHash == providedHash:
    deviceInfo.hbTimestamp = data.curTimestamp
    update(deviceInfo)
    processData(data.data)
  else:
    raise Exception(''verification failed'')
```

The gateway saves the given timestamp in the *heartbeat_timestamp* field. The gateway also runs a cron job that verifies the *heartbeat_timestamp* for all the active device. If the device has not received a heartbeat in last five minutes, the user is notified. There can be multiple cause of the device not sending the heartbeat. The device may be offline, the resources of the device is consumed, or the device is under a DoS/DDoS attack. The user gets notified in real time that the device is non-responsive and can take actions to prevent any further damage.

3.5.2 Sending command to device

The user and delegates can request information form the device by sending it specific commands. In this model, we have generalized the command formatting. The list of commands that can be sent to the device can vary based on the usage and need. IoT devices can be diverse and the set of commands that it can respond to depends on the type of operation that it performs. For example, a smart thermostat can have commands like, raise the temperature of the room, change from heating to cooling, and other. For a smart light bulb, the set of commands can be totally different, switch on the bulb, or change the color of the bulb. So, the commands that can be interpreted by the device can be decided by the manufacturer. However, the commands can only be issued by the user or delegate who are registered with the device.

Like we mentioned before, all communications to the device can only happen through the gateway. Any communication that is not from the gateway is discarded by the device. The operation is similar for both the user and delegate. Here we will work through the security framework for the user. The same can be applied for the delegate as well. We have established earlier in Section 3.4 that both user and delegate set up a shared key with the device for communication. We can use that shared key to send instructions and queries to the device.

For communicating between the user and the device, the query can be encrypted with the shared key from the user and send to the gateway. The *package_to_gateway* is sent to the gateway using an API which is protected by SSL/TLS. The user also sends the authorization header containing the JWT token.

$$Enc\ \{< current_timestamp, query >, K\} \rightarrow package_to_gateway$$

The gateway receives the package along with the headers. From the authorization token, it verifies the identity of the user. After verification it creates a package for the

device including the *package_to_gateway*, *user_id* and the current timestamp. Then it creates a hash of the whole and encrypts it with the private key of the server.

$$\{package_to_gateway, user_id, current_timestamp\} \rightarrow package$$

$$\{package, Enc\ \{Hash(package, SHA256), PrivKey_{gateway}\}\} \rightarrow package_to_device$$

The *package_to_device* is sent to the device over HTTP. On receiving the package, the device first verifies the identity of the request. For that it checks the origin header to find out where the request is from. If it is someone other than the gateway, it rejects the request. The serverâĂŹs identity is verified by provided hash with a computed hash from the package. The timestamp is checked next to verify that the given timestamp is not older than one minute, otherwise the request is rejected. After the verification, the *user_id* is taken from the package and the respective shared key K is extracted. The key K is used to decrypt the *package*. This verifies the identity of both the gateway and the user successfully. The next step is to process the query and send the response back to the user via gateway. The algorithm is shown below:

```
requestReceived(package, headers, encryptedHash):
  # verify gateway
  assert headers.origin == GATEWAY_URL, ''Invalid origin''
  computedHash = computeHash(SHA256, package)
  providedHash = decrypt(encryptedHash, GATEWAY_PUBKEY)
  assert computedHash == providedHash, ''Invalid verifier''
  curTimestamp = getCurrentTimestamp()
  timeDiff = getTimeDiff(MIN, curTimestamp, package.timestamp)
  assert timediff <= 1, ''Invalid package''

  # verify user
  key = getSharedKey(package.userId)
  assert key != None, ''Invalid user''
```

```
14 queryData = decrypt(package.package, key)
15 assert queryData.userId == package.userId, ''Invalid user''
16 timeDiff = getTimeDiff(MIN, curTimestamp, queryData.timestamp)
17 assert timediff <= 2, ''Invalid query''
18
19 # provide response
20 provideResponse(key, queryData.query)
```

One thing to note here is that both the user and gateway is sending the current timestamp when they are creating the request. The device verifies both the timestamp. The reason for that is to prevent the gateway from using an old package from the user. The gateway will log every transaction, and that can include the package as well. With the encryption in place, the server will not be able to see the query but can replay the same at a later point of time. The double timestamp verification prevents that from happening. This process falls back to the principle of "never trust, always verify". Here since the package is coming via the gateway, it is important to verify both the user and gateway before sending a response.

The next step is for the device to generate a response and send it in a secured way. The device reads the query, performs the required operation and gathers the response *data*. First it creates the package for the user. It encrypts the *data* along with the *device_mac* and current timestamp with the shared key K.

$$Enc\ \{< data, device_mac, current_timestamp >, K\} \rightarrow package_for_user$$

Similarly, the *package* is prepared for the gateway to accept and verify. For that the *package_for_gateway* is created with *package* along with the *device_id* and current timestamp.

$$\{package_for_user, device_id, current_timestamp\} \rightarrow package$$

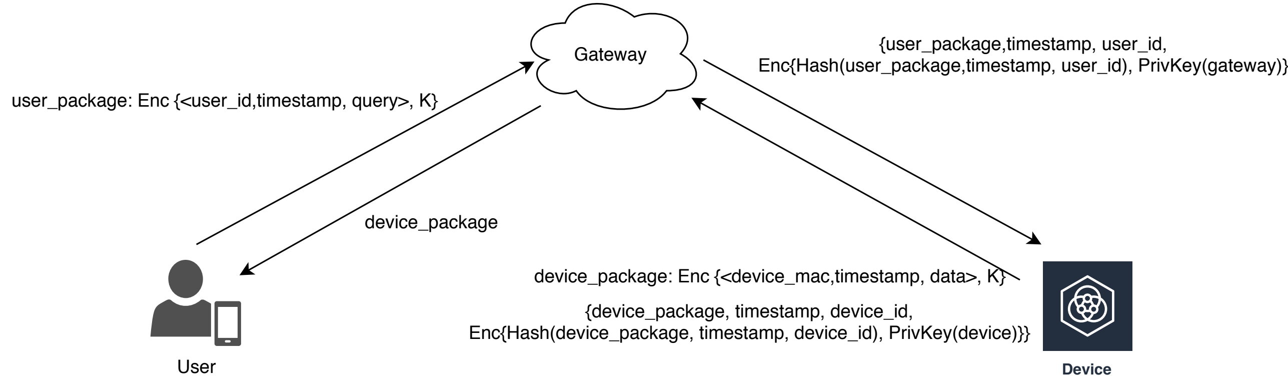


Figure 3.7: Communication between user and device via gateway

The *package* is hashed and the hash is encrypted with the private key of the device to create the digital signature.

$$\{package, Enc\ \{Hash(package), PrivKey_{device}\}\} \rightarrow package_for_gateway$$

This *package_for_gateway* is send to the gateway.

On receiving the package, it extracts out the device_id and finds the corresponding public key. Like the heartbeat communication, the gateway verifies the hash and timestamp and extracts the package for the user. The gateway responds back to the earlier request of the user with the package.

The user on receiving the response from the gateway, performs a similar verification of the *package*. The verification process like the one performed by the device. The user verifies that the response is for the earlier request and is from the gateway. Next it decrypts the package with the shared key to get the device_mac and timestamp. It verifies that the device_mac matches with the one requested for and the timestamp is no older than two minutes. This concludes the communication between the user and the device via gateway. Figure 3.7 shows the encryption process for the exchange. The communication between the delegate and the device will be exactly similar.

In the entire communication, every party verifies the identity of each other. The

gateway can log the package that is being send to the device, but it cannot read any information out of it because the same is encrypted by a key that is known only to the user and device. Similarly, the data being transmitted to the user from the device cannot be interpreted by the gateway. The gateway acts as a relay to pass the information and for the device to receive request from only one medium.

In the following section we will look at other security issues that can be solved by this framework.

3.6 Patch management

Firmware update or patch management is one of the security issues we wanted to tackle with this framework. This is a common problem with most of the IoT devices out there in the market today. There is no clean way of patching the devices to rectify issues and correct programming bugs. Thus, a bug that is found is not mitigated and such issues can be taken advantage of by adversaries.

A lot of work has been done to come up with a security framework for updating firmware for IoT devices [28]. In the article [75] the researchers surveyed the different techniques for device management. The Lightweight Machine-to-Machine (LwM2M) protocol is the most prominent. It uses CoAP, which can be secured with DTLS. It provides a simple data model and RESTful interfaces for remote management of IoT devices. Other researches proposed solution based on block chaining [13,73]. The firmware versions and hash are compared against the information stored in the block chain server before an IoT device is added to the network. The proposed solution allows identification of a suitable type of firmware update and ensures integrity of the firmware.

The software on the IoT device must be prepared to support the firmware update mechanism [75]. The typical firmware update procedure is simple. The manufacturer's development team fixes bugs, tests the changes, re-compiles the code and generates

a entirely new firmware image, which is then patched on to the devices. The flash memory of the IoT device is split into memory region containing the bootloader and firmware images. The new image is stored in one of the slots and the IoT device is then reset with the bootloader and the new firmware. This process is used in MCUboot and ESPer.

For implementing a security framework around the patch update, we followed the architecture defined by the OMA LwM2M device management standard v2.0 [39]. After going through the standard, we realized we can use the communication mechanism designed in Section 3.5.2 and enhance it to securely initiate the firmware update process.

As we have noted many times before, the user is the owner of the device and has total control over the device. The patch management is no exception. The initiation of the process to inform the device starts with the user. The gateway notifies the user that a new update is available for the device and the user requests the gateway to send the update URL over a TLS protected API. Once it receives the update URL, it creates a package like the one for device queries and sends it to the device via the gateway. The device on receiving the package, verifies the identity of the gateway and user and calls the URL provided in the package. The URL returns the updated firmware along with a hash over TLS protected API. The device verifies the hash and initiates the update process. Once updated, the device restarts and runs the new firmware over the bootloader. The details of the process is shown in Figure 3.8

One thing to note here is that the framework dose not describe how the update process is done in the device. It ensures that the update communication is received by the device in a secured way so that it is not tampered in transit. The heartbeat communication can be used to send the notification to the gateway about the current version of the device. That can help the gateway notify the user if the device is not updated after the new firmware is released. The gateway can give alert level to the notifications to inform the user about the criticality of the update. From the user's

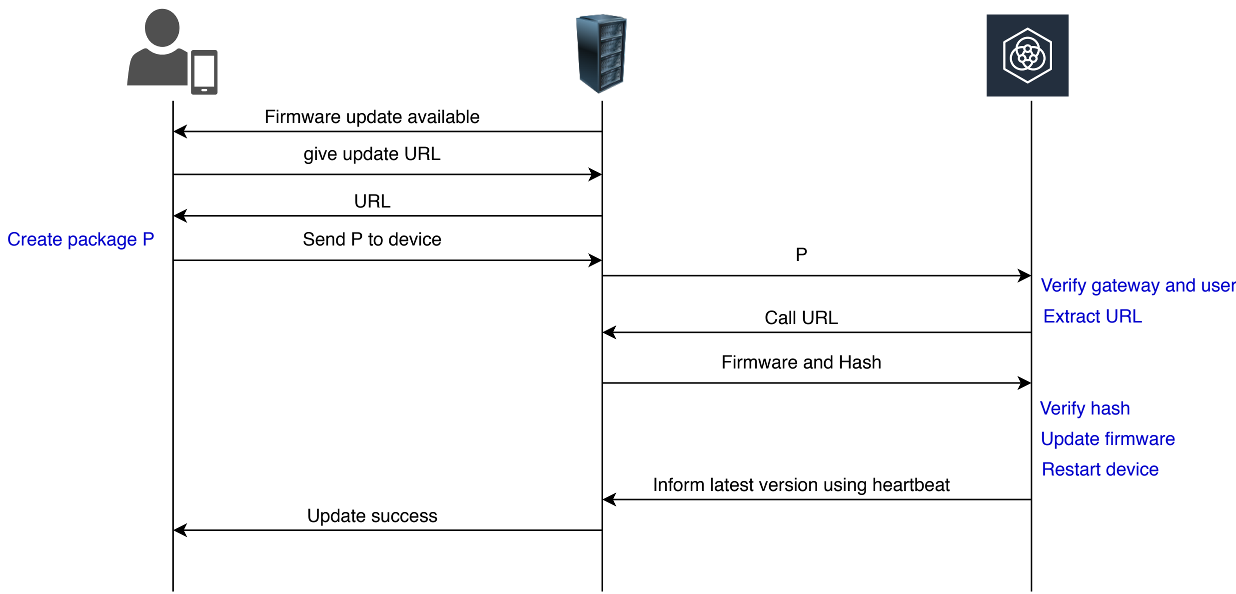


Figure 3.8: Updating device firmware

perspective, updating the firmware of the device is just a click of a button.

The goal of this framework is to ensure that security implementation is not a hindrance to operation. This framework takes into consideration this fact and ensures that the user doesn't have to go an extra step to implement security in the system. Security is baked in as a backbone.

3.7 Addressing other security concerns

There are a few security issues that cannot be solved by this framework. This includes physical security and weak programming practice. For physical security, a software solution cannot solve it completely. However, we can take a reactive approach and let the user know if the device is not responding. As seen in Section 3.5.1, the heartbeat communication protocol informs the gateway that it is alive and well every minutes. The gateway also runs a cron job regularly to check the last *heartbeat_timestamp*. If the job notices that an active device is not communicating more than fifteen minutes (skipped around five pulse) then a notification will be sent to the user is notified via

email/in-app notification. This will inform the user that the device is offline and needs attention. We will not be able to solve the physical security issue but at least provide enough information in real time for the user to take some action. The reason for waiting for few failures is to avoid the false positive. There can be two misses due to some other reason like a network glitch but more consecutive failure than that is a serious problem and needs attention.

For weak programming the framework takes a reactive response, similar to physical security. The manufacturer gets to know if there is a breach of the device in any way. With the advent of internet, it is not difficult to get information. Once the manufacturer rectifies the issue and updates the source code, then he can use the patch management technique described in Section 3.6 to update the device.

Another issue we noted in the security concerns for IoT is audit management. Having every communication flow through the gateway gives us a central location to perform audit and logging. In data structures described in Section 3.3, we have added a relation called *TRANSACTIONS*. This relation keeps track of all communications flowing through the gateway. Lets take an example of the communication happening between the user and device as described in Section 3.5.2. There we see that the user sends a request to the gateway, which gets forwarded to the device. Then the device communicates back to the gateway and it finally reaches the user. There are four steps in this communication, but it is one transaction. An example if the data stored in the *TRANSACTIONS* Table 3.1.

The above is a dummy dataset but it proves some valuable insights about the audit management. One thing to note in Table 3.1 is that all the four records have the same *transaction_id*. This way we can trace back and chain the transactions that happened for one communication. Also, the *timestamp* attribute records the time when the transaction was recorded. With this we can also measure the performance of one communication. This will measure the effectiveness of the framework in terms of time

Table 3.1: Transactions of the communication between user and device

Transaction_Id	From	To	Timestamp	Payload
515e9cf8-522c-4e2c-bad7-9bc41a2fcd4b	User: df234545-b36b-4971-b0e3-7daf18f088f3	Gateway	2020-02-28T17:57:24	76jDv+yOBC7 //vzcuxQkKnE5jfSD nuBsRBVuCmn-weS8=
515e9cf8-522c-4e2c-bad7-9bc41a2fcd4b	Gateway	Device: 89dda487-586d-4fad-931f-2735323102f7	2020-02-28T17:58:19	jB2rvzDufZlcdkL LWVvrsHgTNNXK5p 1aV5OVtWD5OeY=
515e9cf8-522c-4e2c-bad7-9bc41a2fcd4b	Device: 89dda487-586d-4fad-931f-2735323102f7	Gateway	2020-02-28T17:58:37	JkMjG2eVRwn J9Zo8V17nNuLS j6UeD5NzP Ur-jWDe4gk=
515e9cf8-522c-4e2c-bad7-9bc41a2fcd4b	Gateway	User: df234545-b36b-4971-b0e3-7daf18f088f3	2020-02-28T17:59:02	4oMSspNupP 9P0uo50mBRF HRW8d8err1b p7Wk7jQt2sQ=

and we can optimize operations to better the performance. The *from* and *to* fields gives a clear map as to how the data flowed and the *payload* field, though unreadable, can provide feedback on the load of data transmitted over the wire.

3.8 Summary

Revisiting the security issues mentioned in Section 2.3, we realized that the concerns for IoT security are the following:

- Insufficient physical security
- Limited resources
- Inadequate authentication
- Improper encryption
- Lack of access control
- Backdoor ports
- Lack of patch management
- Missing audit management capabilities
- Weak programming practice

With the proposed zero-trust model we have covered most of the above-mentioned issues. With the P3 connection model we setup the encryption technique that we can use for proper authentication and authorization. It also helps to uniquely identify each entity in the system and investigate the access rights for each of them. The communication model described ensures that data is protected in transit as well as from other parties who are not supposed to listen to the details. Everyone is provided information to the extent that they need to perform their role in the ecosystem. Another thing to note about this model is that all communications happen over the HTTPS application

protocol and thus can be easily integrated into the current network architecture. We are not using any port other than 443 in the system.

The patch management provides a clear guideline for the user to control the update process but provides flexibility for the manufacturer to inform the user of an important patch that can make the system vulnerable. A strong update process as mentioned helps recover from zero-day bugs quickly and efficiently. The audit logging and notification services keeps the user updated about the well-being of the device and the device through the heartbeat communication is in constant touch with the gateway. Any missed pulse is logged and reported to the user to take actions.

The framework is designed to provide the user of the device full control on the device without having to know complex passwords. The system eradicates the necessity for default keys or passwords in the systems. It provides the manufacturers the ability to monitor the flow of operations without interfering with the activities of the user. This framework proves that we can follow the principle of "never trust, always verify" within the boundaries of the resources available in an IoT device.

Chapter 4

Implementing Zero-Trust for IoT Security

This chapter discusses the implementation of the zero-trust framework we discussed in Chapter 3. We will start by building the individual elements of the framework, *i.e.* device, user and gateway. Next we will be building the API endpoints in the gateway to perform the different operations like registration, audit logging, relay and others. From there we will move on to discuss the source code to build the P3 connection model as described in Section 3.4 and then we will look into the implementation of the communication protocol as detailed in Section 3.5.

For building the framework different technologies and programming languages is being used including C++, Python and JavaScript. The reason for using different programming language is that each element in the framework works best with a specific language to perform its operation. For example, for the device works directly with the hardware and microcontroller and thus for hardware level efficiency it requires the source code to be in C++ or Python. We will detail out the reason for each of our choice in implementing the solution as we move along.

4.1 Building the environment

As mentioned earlier, the environment consists of four key members, *i.e.* device, user, delegate and gateway. Each one plays a unique role in the ecosystem. We build up each of the component from scratch to have better control on the functionality. The development was done on a 64 bit Windows 10 machine on an Intel(R) Core(TM) i3-7100 U CPU @ 2.04GHz and 8 GB RAM.

4.1.1 Setting up the device

For this thesis we wanted to build a temperature and humidity sensor which can be reached via the internet. The user and delegate will be able to interact with the device and send specific commands. They can also request for the current reading of the sensor as well as change the display of the device from Celsius to Fahrenheit and *vice versa*. A broad overview of the device internals was shown in Figure 1.2 in Section 1.4. Here Figure 4.1 shows the physical device after implementation.

For the microcontroller we looked at Arduino which can be used to build simple IoT devices. We decided to use NodeMcu V3 ESP8266 as the microcontroller. The advantage of a NodeMcu over Arduino Uno or other similar boards is that it comes with an in-build Wi-Fi chip that can be used to connect to the internet. It can be used as both a web server and a HTTP client using the *ESP8266WiFi.h* library.

NodeMCU v3 is a development board which runs on ESP8266 with the Espressif Non-OS SDK, and hardware is based on the ESP-12 module. The device features 4MB of flash memory, 80MHz of system clock, around 50k of usable RAM and an on chip Wi-Fi Transceiver. The specification of the NodeMcu board is detailed in Table 4.1.

To build the temperature-humidity sensor we needed other components as shown in Table 4.2. The *HC-05 Wireless Bluetooth RF Transceiver* is used as a Bluetooth receiver for setting up the initial negotiation as described in the P3 connection model. The

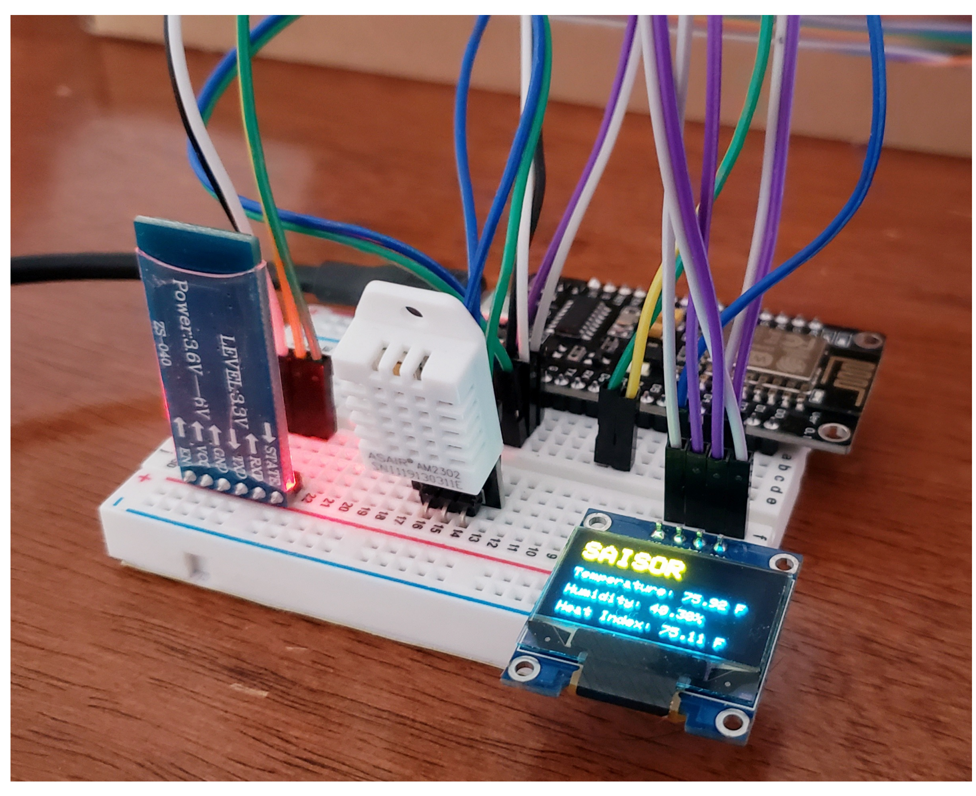


Figure 4.1: The physical device

Table 4.1: NodeMcu V3 specification [76]

Input voltage	7–12V
Operating voltage	3.3V
CPU (Microcontroller)	32-bit RISC Tensilica Xtensa LX106 running at 80 MHz
External QSPI flash	512 KB to 4 MB* (up to 16 MB is supported)
EEPROM	512 KB
Wi-Fi	IEEE 802.11 b/g/n, Integrated TR switch, balun, LNA, power amplifier and matching network, WEP or WPA/WPA2 authentication, or open networks
Digital I/O Pins (DIO)	16
Analog Input Pins (ADC)	1
UARTs	1 on dedicated pins, plus a transmit-only UART can be enabled on GPIO2
SPIs	1
I2Cs	1
Flash Memory	4 MB
SRAM	64 KB
Clock Speed	80 Mhz

UCTRONICS 0.96 Inch OLED Module is used to display the temperature and humidity of the room. The display was added to verify that the temperature that is taken by the sensor is the same that is being transmitted to the user. *HiLetgo DHT22* is the sensor which is getting the actual temperature and humidity of the environment. There were multiple options for the sensor like DHT11, DHT22, AM2302 and others. The advantage of using the DHT22 over others is that it can measure a temperature of -40 to 125 degree centigrade with an accuracy of +/- 0.5 degrees. It also has a better humidity measuring range from 0 to 100

The device is coded using the Arduino IDE (`https://www.arduino.cc/en/main/software`). For allowing the Arduino to code for an ESP8266 board, we had to perform the following steps:

Add ESP8266 board manager. To add additional board manager open the Arduino IDE-File-Preferences. In the *Additional Board Manager URLs* add `https://arduino.esp8266.com/stable/package_esp8266com_index.json`. See Figure 4.2.

Install the ESP8266 board. To add the ESP8266 board, go to Tools-Boards-Board Manager. Search for *esp8266* and click install. See Figure 4.3

Select the generic board and the com port. Once the installation is complete, the next step is to select the right board, so that we can build the sketch for ESP8266 using the Arduino IDE. Select the board by going to Tools - Board - Generic ESP8266 Module. Next we need to select the COM port that the device is connected to by selecting Tools - Ports - The available port. We can also verify the correct port from the system's device manager. *Note* that after connecting the device to the development system, we need to install the device driver. In Windows 10, this automatically happens when the device is connected for the first time. See Figure

Once the setup is complete, the IDE can be used to code the NodeMcu board. A lot

Table 4.2: Components of the IoT device

Component	Description	Quantity
	ESP8266 microcontroller NodeMCU Lua V3 WiFi with CH340G	1
	HiLetgo HC-05 Wireless Bluetooth RF Transceiver Master Slave Integrated Bluetooth Module 6 Pin Wireless Serial Port Communication BT Module for Arduino	1
	UCTRONICS 0.96 Inch OLED Module 12864 128x64 Yellow Blue SSD1306 Driver I2C Serial Self-Luminous Display Board for Arduino Raspberry PI	1
	HiLetgo DHT22/AM2302 Digital Temperature and Humidity Sensor Module Temperature Humidity Monitor Sensor Replace SHT11 SHT15 for Arduino Electronic Practice DIY	1
	Breadboard Solderless with Jumper CablesâĂŞ ALLUS BB-018 3Pc 400 Pin Prototype PCB Board and 3Pc Dupont Jumper Wires (Male-Female, Female-Female, Male-Male) for Raspberry Pi and Arduino	1 set

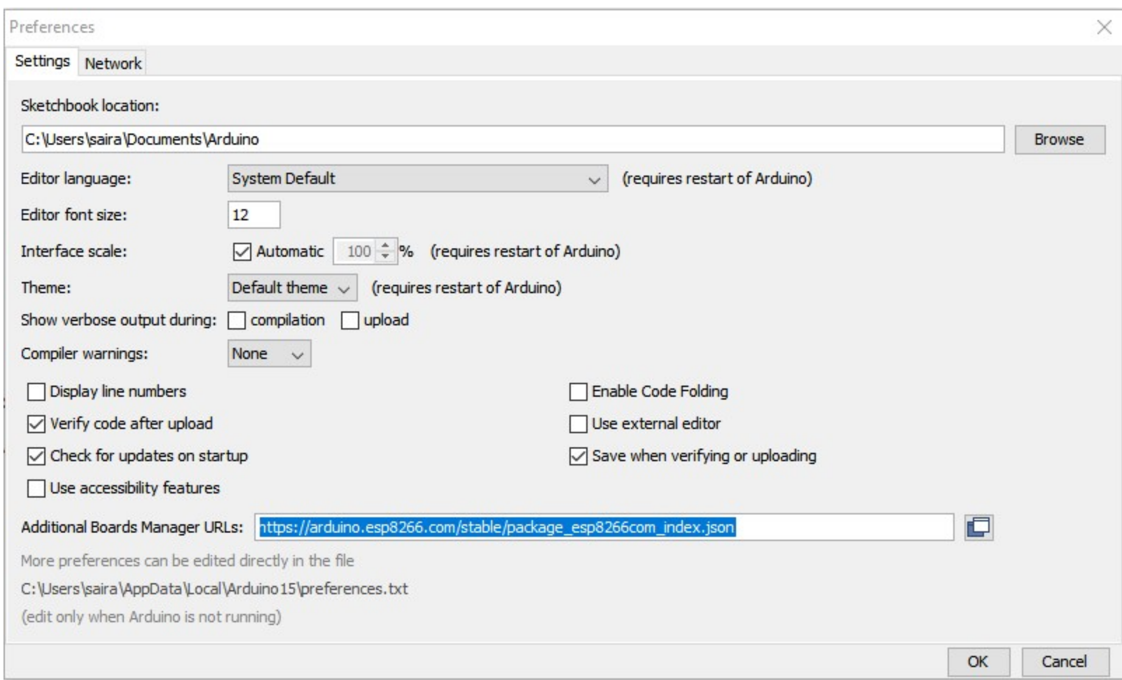


Figure 4.2: Adding ESP8266 board manager to the IDE

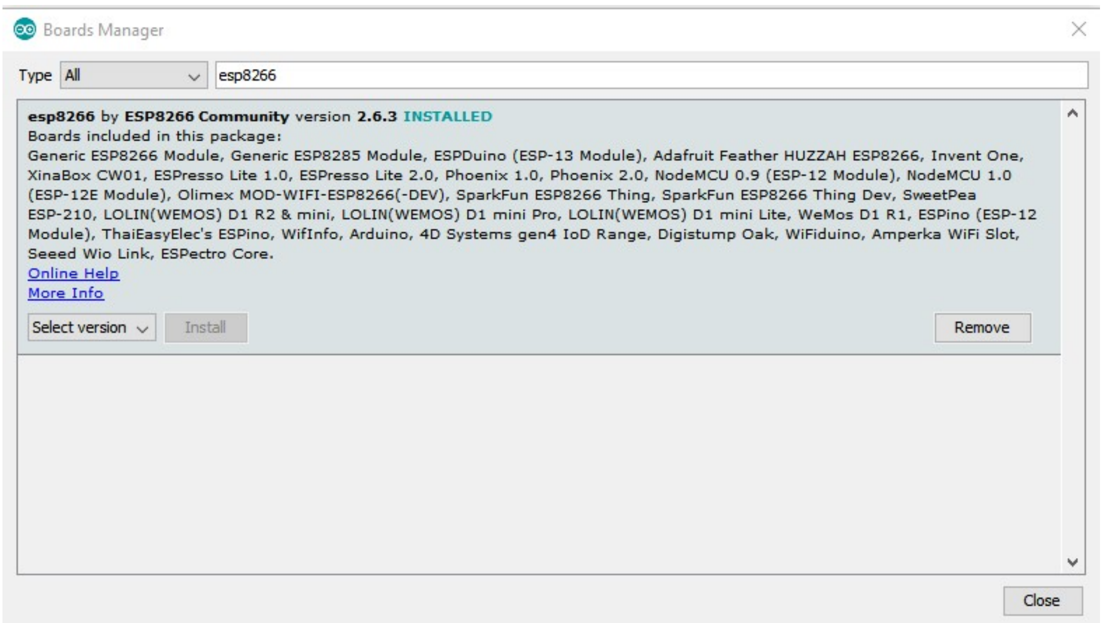


Figure 4.3: Installing ESP8266 board

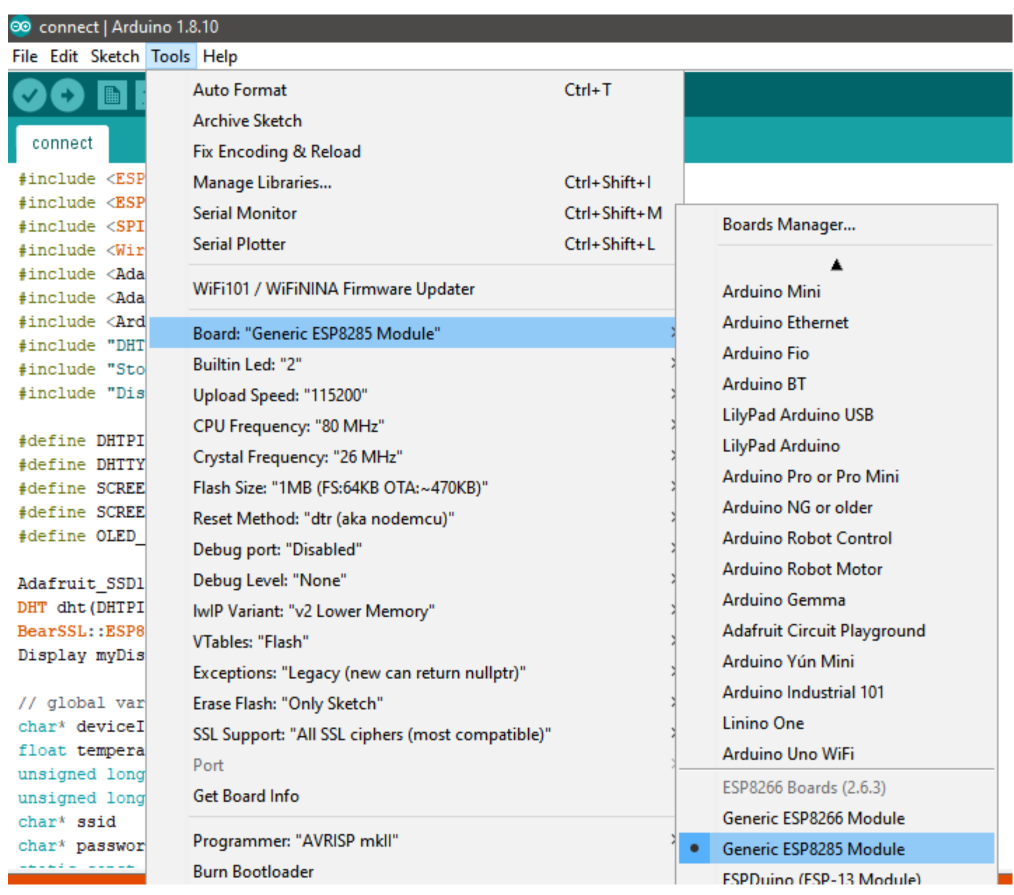


Figure 4.4: Selecting the required setting for Arduino IDE

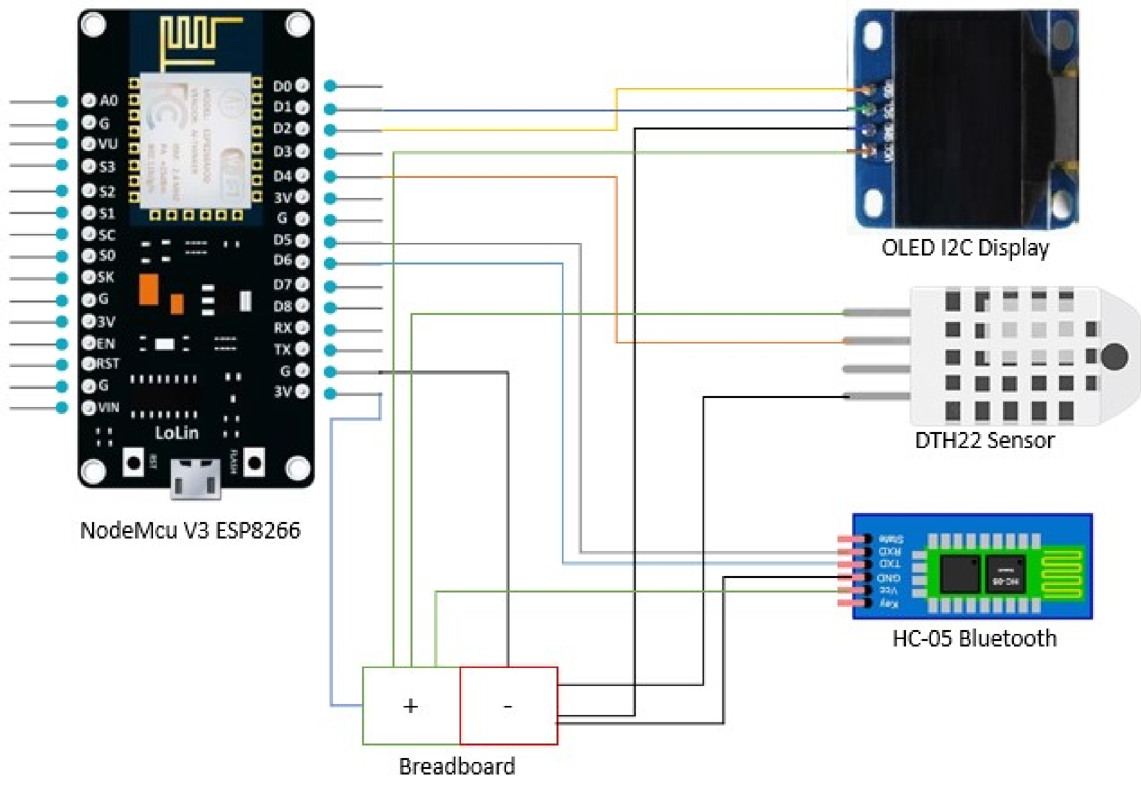


Figure 4.5: Pinout diagram for the device

of examples come along with the board to get us started. To test the connection, we used the one in File - Examples - ESP8266 - Blink. The program is attached in Appendix A. We Verified and uploaded the sketch to the device using the IDE. Once uploaded, the blue LED light in the NodeMcu module started blinking and we successfully were able to code the ESP8266.

We connected all the components described in Table 4.2 with the NodeMcu microcontroller. The pin diagram is as shown in Figure 4.5 and described in Table 4.3. For the power and ground, we are connecting the 3V and G pins of the microcontroller to the breadboard and all the other components are getting their power from it. This is done because there are limited power and source outlets from the microcontroller and so we used the parallel outputs of the breadboard to get the power from one outlet into all the other components.

Table 4.3: Pinout connection of the components with the microcontroller

Microcontroller pin	Component	Pin
V3 (via breadboard)	HC-05 Bluetooth	VCC
G (via breadboard)	HC-05 Bluetooth	GND
D5	HC-05 Bluetooth	RXD
D6	HC-05 Bluetooth	TXD
V3 (via breadboard)	DHT22 sensor	VCC
G (via breadboard)	DHT22 sensor	GND
D4	DHT22 sensor	DATA
V3 (via breadboard)	OLED I2C display	VCC
G (via breadboard)	OLED I2C display	GND
D1	OLED I2C display	SCL
D2	OLED I2C display	SDA

We named the device **"Saisor"**.

4.1.2 Mobile app for the user and delegate

We are using the Android platform to build our app. We used a Samsung Galaxy S9 with Android 10 and kernel version 4.9.186-17655189. The mobile phone was updated with the latest security patch as of February 01, 2020. For testing purpose, we are also using an emulator of Google Pixel 3XL with API 29 from the Android Virtual Device (AVD) manager.

To profile and debug our app we had to enable the developer options on our Samsung S9. Developer options lets us configure system behavior and enables debugging of the apps. To enable this option we had to go to Settings – About phone – Software information and then tap the *Build number* seven times. We can enable and disable the option from Settings – Developer options. We enabled the following features for debugging:

- **USB debugging:** This option enables debug mode when USB is connected. This helped us to deploy the debugging version of the source into the phone. In the debugging mode. React Native reflects the changes into the device app as we

update.

- **Stay awake:** This keeps the screen awake when the USB is connected. This was more of a convenience where we didn't have to unlock the phone as we were developing and debugging the screens.

To develop the app, we used *React Native*. React Native combines the best part of native development with React, a JavaScript library for building user interfaces. It provides the tools to build apps for both Android and iOS platform with the same code base. React components wrap existing native APIs via React's declarative UI paradigm and JavaScript. It is developed by Facebook and has a huge community base behind it. For this app, we are only concentrating on the core native aspects of Android and not iOS.

The *Getting Started* page [1] is a very helpful place to start building apps using React Native. For developing the React Native app for Android using a Windows machine, the following setup is required:

Installing dependencies. We need to install Node, Python2 and Java SE Development Kit (JDK) of the latest version for React Native to run. We used, Chocolatey, a popular package manager for Windows to install the dependencies. For React Native to work, we need the minimum version for Node to be 8.3 and JDK to be 8.

```
choco install -y nodejs.install python2 jdk8
```

Android development environment. We need to download and install the Android studio from `https://developer.android.com/studio/index.html`. During the installation we must make sure that the following are checked:

- Android SDK

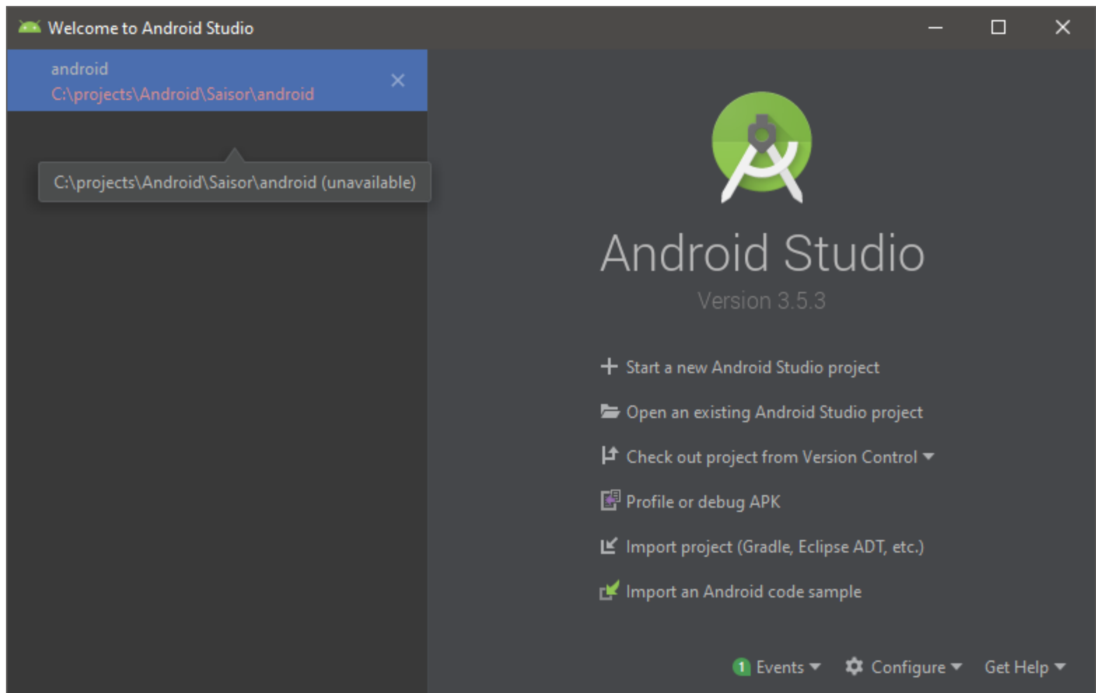


Figure 4.6: Android Studio 3.5.3

- Android SDK platform
- Performance (Intel HAXM)
- Android Virtual Devices

Android installs the latest SDK (currently 10) by default. For building apps with React Native we require the Android 9 (Pie) SDK. This can be installed using the SDK manager in Android Studio. The SDK manager can be found under Configuration – SDK Manager.

Configuring the development machine. React Native require some environment variables to be set up in order to build apps with native code. To set the *ANDROID_HOME* environment variable in Windows 10 go to Windows – Settings – About – System Info – Advance System Settings. This opens the system properties. Go to Advance – Environment Variables. Add a new environment variable as shown in Figure 4.7. The default path for the SDK will be in the user folder.

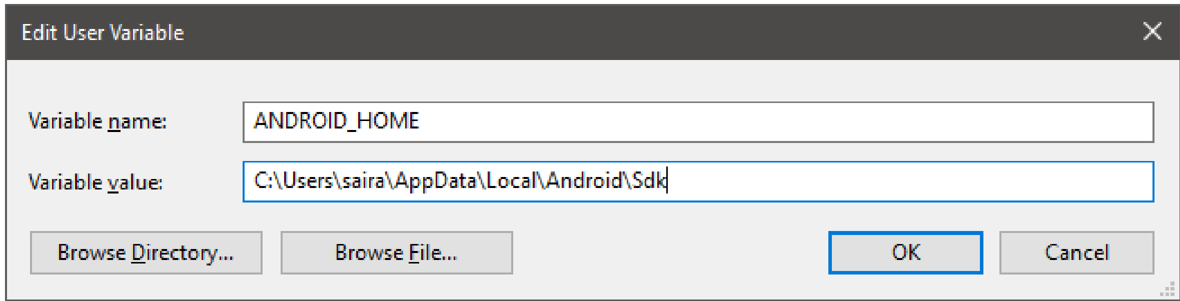


Figure 4.7: Setting environment variable for React Native to work with Android Studio

```
c:\Users\YOUR_USERNAME\AppData\Local\Android\Sdk
```

We also need to add the platform tools in the environment variables for React Native to build and deploy the app. In the *Environment Variables* under *System Variables* click on Path and click Edit. Click on New and add the path to the platform tools as shown in Figure 4.8. For default installation, it will be under the user folder.

Using React Native command line interface. React Native provides a command line interface (CLI) to create, deploy and manage the app. To install the CLI, use Node to install it using NPM to install it globally.

```
npm i react-native -g
```

To verify the installation, open command prompt and check the version as shown in Figure 4.9.

Creating a React Native test app. To test the setup, we wanted to create a test app using the CLI. For that we ran the following command to create a basic app. Figure 4.10 shows the *TestApp* open in VSCode.

```
react-native init TestApp
```

Running the test app. Before deploying the app, we need to be sure that the emulator and the real device is connected and accessible. For the emulators we can go into the

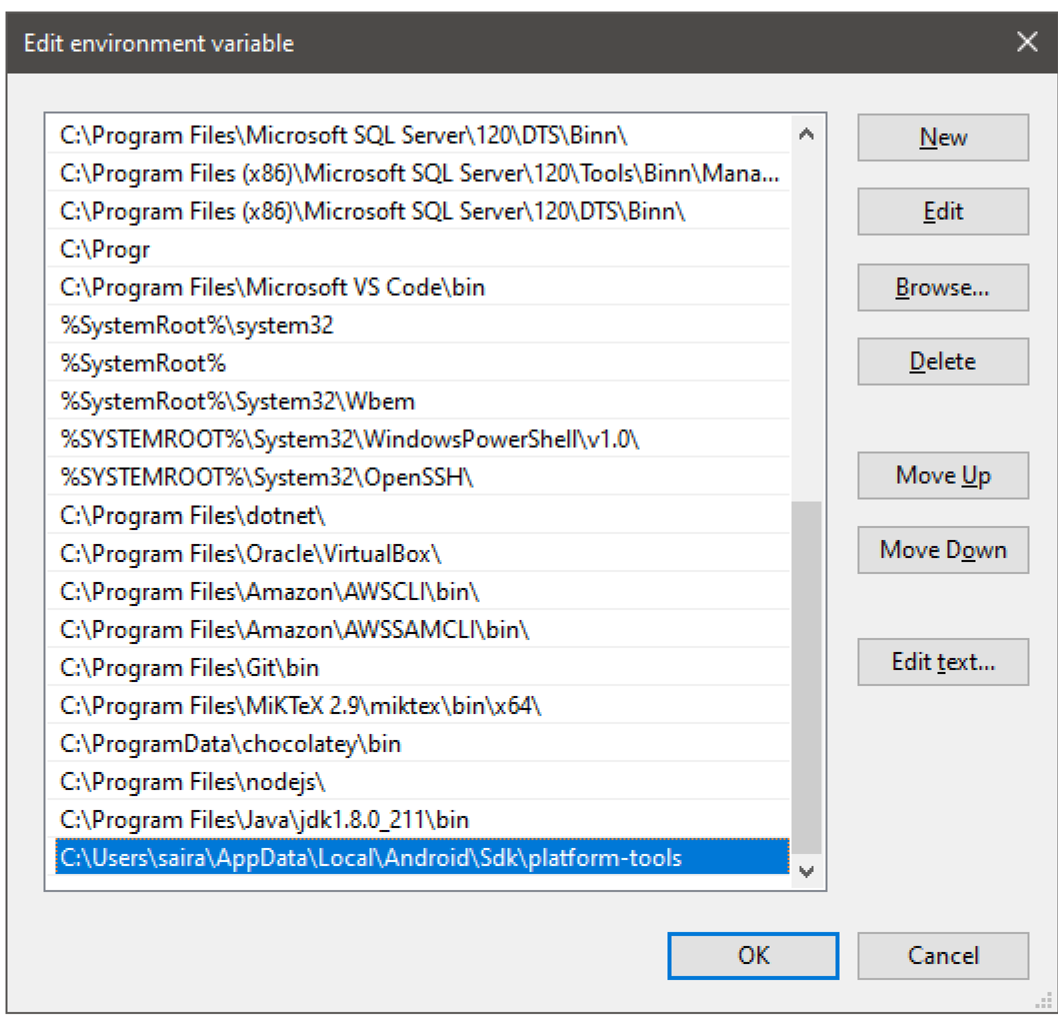


Figure 4.8: Setting path variable for platform tools

```
C:\Users\saira>react-native -v
react-native-cli: 2.0.1
react-native: 0.61.5

C:\Users\saira>
```

Figure 4.9: Verify React Native CLI installation

```
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
  package="com.testapp">

    <uses-permission android:name="android.permission.INTERNET" />

    <application
      android:name=".MainApplication"
      android:label="@string/app_name"
      android:icon="@mipmap/ic_launcher"
      android:roundIcon="@mipmap/ic_launcher_round"
      android:allowBackup="false"
      android:theme="@style/AppTheme">
      <activity
        android:name=".MainActivity"
        android:label="@string/app_name"
        android:configChanges="keyboard|keyboardHidden|orientation|screenS
        android:windowSoftInputMode="adjustResize">
        <intent-filter>
            <action android:name="android.intent.action.MAIN" />
            <category android:name="android.intent.category.LAUNCHER" />
        </intent-filter>
      </activity>
      <activity android:name="com.facebook.react.devsupport.DevSettingsAct
    </application>

</manifest>
```

Figure 4.10: Test app opened in Visual Studio Code

```
C:\projects\tests\TestApp>adb devices
List of devices attached
4930563041553398        device
emulator-5554   device
```

Figure 4.11: Mobile devices connected to the development machine

AVD manager and check the devices that are connected. For knowing all the active devices, we can run the command *adb devices*. Figure 4.11 shows a list of all the connected devices to the development machine. The real devices are always represented with numbers. The command must return at least one device for the deployment to work. If no devices are shown, we need to make sure that the above steps are properly done.

After verifying the connected devices, we run the following command to deploy the test app into the device. Figure 4.12 shows the *TestApp* deployed in the emulator.

```
react-native run-android
```

Once we are able to see the screen with the *App* screen. To verify that the live

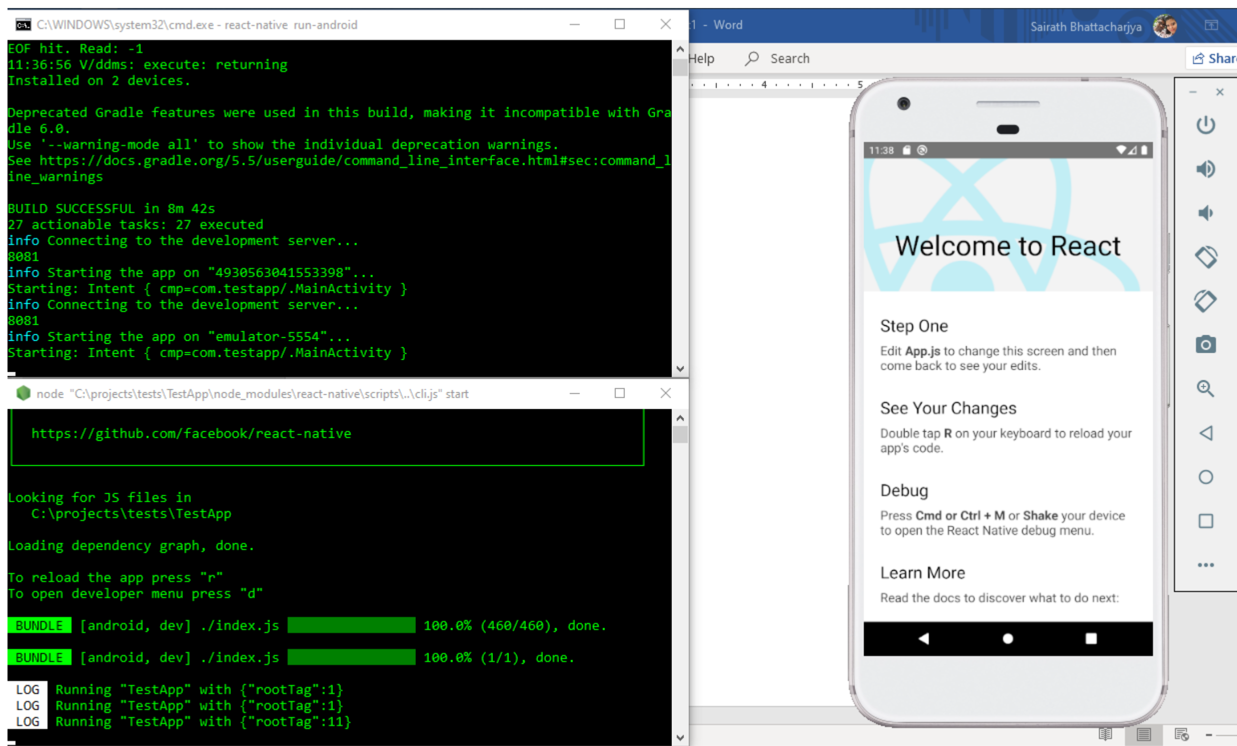


Figure 4.12: Deployment of the app in emulator

debugging is working fine, changes the text “Welcome to React” with “Hello World!” and verified that the screen is accordingly updated. This enables us to use the environment to develop the app for *Saisor*. The next step is to setup the gateway.

4.1.3 Configuring the gateway

For the gateway, we used Amazon Web Services (AWS). AWS is a subsidiary of Amazon that provides on-demand cloud computing platforms and APIs to individuals, companies, and governments, on a metered pay-as-you-go basis. In aggregate, these cloud computing web services provide a set of primitive abstract technical infrastructure and distributed computing building blocks and tools.

Setting up an account with AWS is easy. They provide a free one-year subscription with the sign-up. We created a free account by entering the required details and credit card information. Immediately we got an email confirmation and we used the provided

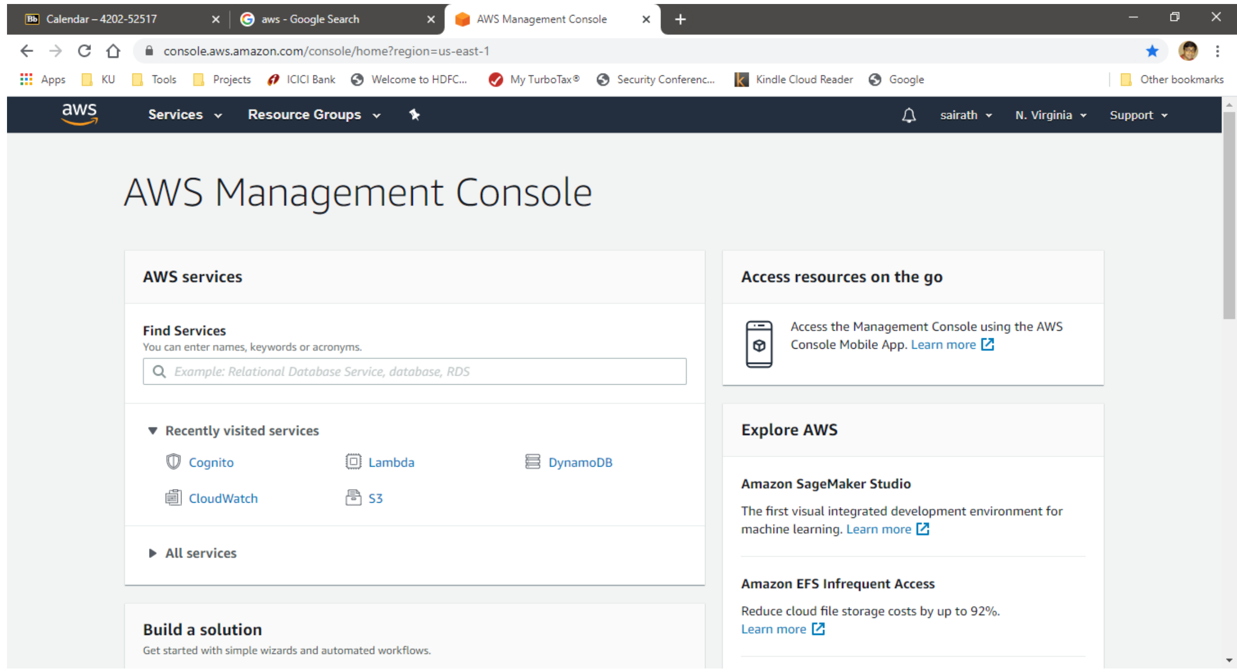


Figure 4.13: AWS console dashboard

link to login into the console as shown in Figure 4.13.

We used several services offered by AWS in this thesis, including:

- **Amazon Simple Storage Service (Amazon S3)** is an object storage service that offers industry-leading scalability, data availability, security, and performance. We can use it to store and protect any amount of data for a range of use cases, such as websites, mobile applications, backup and restore, archive, enterprise applications, IoT devices, and big data analytics. Amazon S3 provides easy-to-use management features so we can organize our data and configure finely tuned access controls to meet your specific business, organizational, and compliance requirements.

- **Amazon DynamoDB** is a key-value and document database that delivers single-digit millisecond performance at any scale. It's a fully managed, multiregion, multimaster, durable database with built-in security, backup and restore, and in-memory caching for internet-scale applications. DynamoDB can handle more than

10 trillion requests per day and can support peaks of more than 20 million requests per second.

- **Amazon Cognito** lets you add user sign-up, sign-in, and access control to your web and mobile apps quickly and easily. Amazon Cognito scales to millions of users and supports sign-in with social identity providers, such as Facebook, Google, and Amazon, and enterprise identity providers via SAML 2.0.

- **Amazon API Gateway** is a fully managed service that makes it easy for developers to create, publish, maintain, monitor, and secure APIs at any scale. APIs act as the "front door" for applications to access data, business logic, or functionality from your backend services. Using API Gateway, you can create RESTful APIs and WebSocket APIs that enable real-time two-way communication applications. API Gateway supports containerized and serverless workloads, as well as web applications. API Gateway handles all the tasks involved in accepting and processing up to hundreds of thousands of concurrent API calls, including traffic management, CORS support, authorization and access control, throttling, monitoring, and API version management.

- **AWS Lambda** lets us run code without provisioning or managing servers. With Lambda, we can run code for virtually any type of application or backend service – all with zero administration. Lambda takes care of everything required to run and scale our code with high availability. We can set up the code to automatically trigger from other AWS services or call it directly from any web or mobile app.

- **Amazon CloudWatch** is a monitoring and observability service built for DevOps engineers, developers, site reliability engineers (SREs), and IT managers. CloudWatch provides us with data and actionable insights to monitor our applications, respond to system-wide performance changes, optimize resource utilization, and get a unified view of operational health. CloudWatch collects monitoring and

```
C:\Users\saira>aws --version
aws-cli/1.16.278 Python/3.6.0 Windows/10 botocore/1.13.14

C:\Users\saira>
```

Figure 4.14: AWS CLI installed in the development machine

operational data in the form of logs, metrics, and events, providing us with a unified view of AWS resources, applications, and services that run on AWS and on-premises servers. We can use CloudWatch to detect anomalous behavior in our environments, set alarms, visualize logs and metrics side by side, take automated actions, troubleshoot issues, and discover insights to keep our applications running smoothly.

- **AWS CloudFormation** provides a common language to model and provision AWS and third party application resources in the cloud environment. AWS CloudFormation allows us to use programming languages or a simple text file to model and provision, in an automated and secure manner, all the resources needed for our applications across all regions and accounts.

To use the amazon services we need to install and configure the *AWS command line interface (CLI)* on our development machine. The AWS CLI is an open source tool that enables you to interact with AWS services using commands in your command-line shell. We can download the CLI from `https://s3.amazonaws.com/aws-cli/AWSCLI64PY3.msi`. To confirm the installation is successful we need to run the command *aws -version* as shown in Figure 4.14.

The next step is to configure AWS CLI to use our AWS account. For this we went into the AWS console and went into the identity and access management (IAM). We went into users and selected our user “sairath.” Under the security credentials tab, we created clicked on “create access key.” This generated a new *access key ID* and *secret access key*. We need to add them in our development environment so that the machine can directly

```
C:\Users\saira>aws configure list
      Name                    Value             Type    Location
      ----                    -----             ----    --------
   profile                <not set>             None    None
access_key     ****************FFC5 shared-credentials-file
secret_key     ****************jBJS shared-credentials-file
    region                us-east-1      config-file    ~/.aws/config

C:\Users\saira>
```

Figure 4.15: Configuring AWS account for development machine

```
C:\Users\saira>aws s3 ls
2020-03-04 13:56:57 amplify-saisor-prod-135648-deployment
2020-03-04 12:22:04 saisor-20200304122051-hostingbucket-prod
2020-03-04 14:04:22 saisor-20200304140242-hostingbucket-prod
2020-02-15 10:45:22 saisor-deployment

C:\Users\saira>
```

Figure 4.16: Buckets present in AWS S3

access the AWS resources. We need to make sure that the user which we are using for development has admin privileges over the different services we mentioned above. This will be required to create and maintain them from the development machine. In our case, user "sairath" is the admin and has full privileges.

In the development machine, we opened a command prompt and typed the command *aws configure*. It will ask us to enter the access ID and secret key which we got from the above step in the console. Once we enter the correct credentials, the development environment is ready to access the AWS resources and services. This creates a file called *credentials* under the *.aws* folder to store the credentials as the default profile. The folder is found under the *users* folder for the logged in user in Windows. To check if the values are correctly entered, type the command *aws configure list* in the command prompt and it should show the details as shown in Figure 4.15.

To verify, we tried to look at the S3 buckets that are currently available in the account and so we typed the command *aws s3 ls* as shown in Figure **??**. This returned the correct set of buckets and confirmed that our setup was successfully done and we are ready to use AWS as our gateway.

4.1.4 Creating the AWS resources

To create the databases tables in *DynamoDB* we used a script that we developed using Python. Appendix B shows the sample script for the *Users* relation. AWS provides a Python library *boto3*, which allows us to interact with the AWS resources. The library in turn uses the AWS configuration that we created above. The setting exports appSettings JSON that holds the region where we want to create the table.

```
appSetting = {
    "region": "us-east-1"
}
```

First, we create a object for the *DynamoDB* service from boto3. Then we call the function *create_table* on it. This returns a *table* object. Since, DynamoDB is a document database, we don't need to provide all the attributes during the table creation. It only requires the key attributes. The primary table is going to have the partition key as *"user_id"*. We are also creating a secondary index on the table so that we can access the record using the userâĂŹs email. In the billing mode, we specified the table to have resources on demand. We wait for the table to get created and then print the status of the table.

For the other resources, we either create it manually in the console or via *CloudFormation* scripts in yaml format. The resources that are a onetime creation, we used the manual method and directly created them from the console. This includes the IAM role that all the APIs are going to use. We went in IAM and selected *roles*. There we created multiple roles to meet the needs of the services. AWS prefers to keep the roles tight, so that the service has only enough permission to perform the task it is assigned to. We can also check the roles form the CLI using *"aws iam list-roles"* as shown in Figure 4.17.

The APIs and lambda functions for deployed using the *CloudFormation* template. For the ease of deployment, we used AWS Serverless Application Model (SAM). SAM is

```
C:\Users\saira>aws iam list-roles
{
    "Roles": [
        {
            "Path": "/",
            "RoleName": "amplify-saisor-prod-135648-authRole",
            "RoleId": "AROA3FGV4Q375ORYU7XQC",
            "Arn": "arn:aws:iam::767100225279:role/amplify-saisor-prod-135648-authRole",
            "CreateDate": "2020-03-04T19:56:56Z",
            "AssumeRolePolicyDocument": {
                "Version": "2012-10-17",
                "Statement": [
                    {
                        "Effect": "Allow",
                        "Principal": {
                            "Federated": "cognito-identity.amazonaws.com"
                        },
                        "Action": "sts:AssumeRoleWithWebIdentity",
                        "Condition": {
                            "StringEquals": {
                                "cognito-identity.amazonaws.com:aud": "us-east-1:9550360a-8ff4-472a-8c3d-42d09ce7da31"
                            },
                            "ForAnyValue:StringLike": {
                                "cognito-identity.amazonaws.com:amr": "authenticated"
                            }
                        }
                    }
                ]
            },
            "Description": "",
            "MaxSessionDuration": 3600
        },
        {
            "Path": "/",
            "RoleName": "amplify-saisor-prod-135648-authRole-idp",
            "RoleId": "AROA3FGV4Q37R2PPS26MD",
            "Arn": "arn:aws:iam::767100225279:role/amplify-saisor-prod-135648-authRole-idp",
            "CreateDate": "2020-03-04T20:04:15Z",
            "AssumeRolePolicyDocument": {
                "Version": "2012-10-17",
                "Statement": [
                    {
                        "Effect": "Allow",
```


Figure 4.17: Roles present in IAM

an open-source framework for building serverless applications. It provides shorthand syntax to express functions, APIs, databases, and event source mappings. With just a few lines per resource, we can define the application we want and the model using YAML. During deployment, SAM transforms and expands the SAM syntax into AWS CloudFormation syntax, enabling us to build serverless applications faster. A sample YAML file is attached in Appendix D. By default, we will be allowing CORS for all APIs and allow the methods GET, POST, DELETE and PUT. Authentication would be added by default to all the API endpoints unless specified for a API. The authorization check is provided by another lambda function that is invoked before the API lambda. This ensures that all the APIs are protected.

4.2 Implementing the P3 connection model

The P3 connection model is described in detail in Section 3.4. This model allows the IoT device to connect securely with a user (mobile app). It uses Bluetooth LE to establish a connection with the user to setup a session key. This key is used for all further exchange between the user and the device. Once both the user and device verify each other, a secret key is established and saved in both places. This key will be used for all further communication between the pair. In this section we are going to describe in detail the steps to implement the model.

In the implementation we had some limitations for lack of compatible security frameworks for each framework that we used. So, we had to made small arrangements to bypass the shortfall. We will explain the differences between the implementations and the model wherever relevant. For the device, we used the Arduino cryptographic library (`https://rweather.github.io/arduinolibs/`). This library provides the implementation for AES256, SHA256, ED25519 and Curve25519, along with other cryptographic algorithms. For the user app, we used the JavaScript NPM library react-native-crypto-js (`https://www.npmjs.com/package/react-native-crypto-js`). Most of the crypto libraries are build into Python, for ED25519, we had to use a extra library PyNaCl (`https://pynacl.readthedocs.io/en/stable/signing/#ed25519`). Next we will look at the implementation of the steps of the P3 connection model.

Pairing and generating the session key. As of now there is no compatible library to implement elliptic curve cryptography. All the options we found were hacks to make it work, but there was no clear option available. We switched to passing the key and initialization vector in a base 64 encoded format from the app to the device.

In the app, the user gets a screen that shows the devices that are currently registered with the user as shown in Figure 4.18. There is a button that allows the user to add a new device. On clicking on the button, the app scans and finds all the devices that

matches the signature of "Saisor" as shown in Figure reffig:app-add-device. In our case, we matched the name *HC-05*.

On selecting the device, the following JavaScript code runs on the user app to pass the hello message.

```
1  async connectToDevice() {
2    const {device} = this.state;
3    const crypto = new CryptoService();
4    const key = crypto.generateRandom(256);
5    const iv = crypto.generateRandom(128);
6    console.log('connecting');
7    let isConnected = await BluetoothSerial.isConnected();
8    if (!isConnected) {
9      await BluetoothSerial.connect(device.id);
10   }
11   console.log('connected');
12   const data = {
13     message: 'HELLO',
14     key: key,
15     iv: iv,
16   };
17   await BluetoothSerial.write(encode(JSON.stringify(data)));
18   this.setState({key, iv});
19   // wait for the response from device
20   console.log('waiting response');
21   await sleep(5000);
22   let response = await BluetoothSerial.readFromDevice();
23   let responseInfo = crypto.decrypt(response, key, iv, 5);
24   // confirm that the response message is a HELLO message
25   if (responseInfo !== 'HELLO') {
26     await BluetoothSerial.write('CLOSE');
27     throw new Error('No response found');
28   }
```

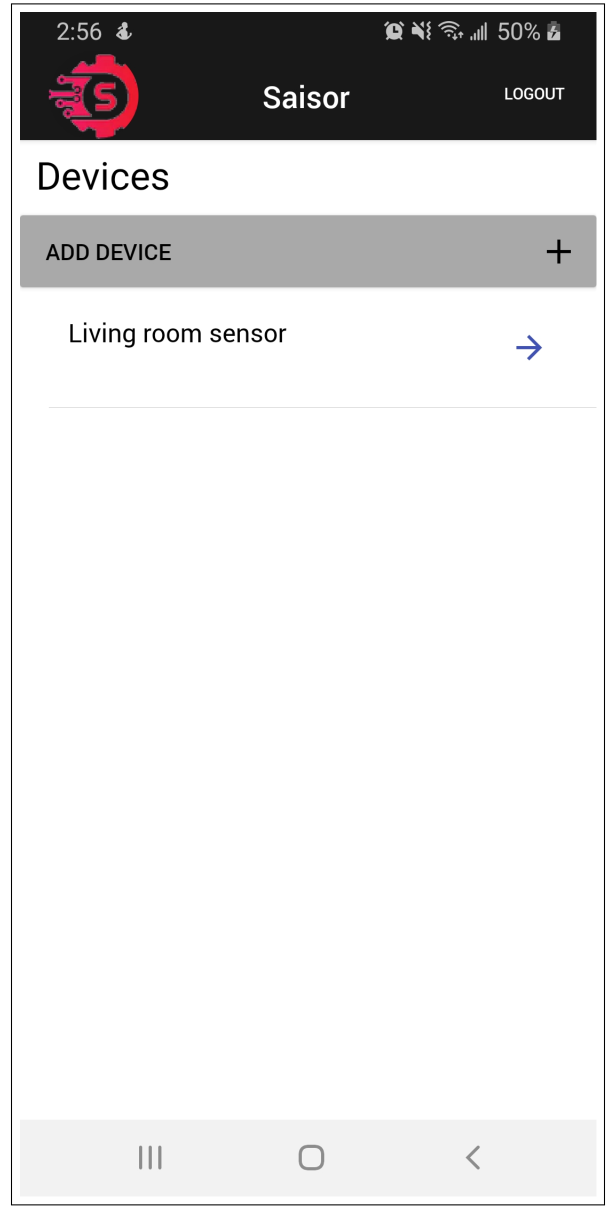


Figure 4.18: App screen to show the current devices added

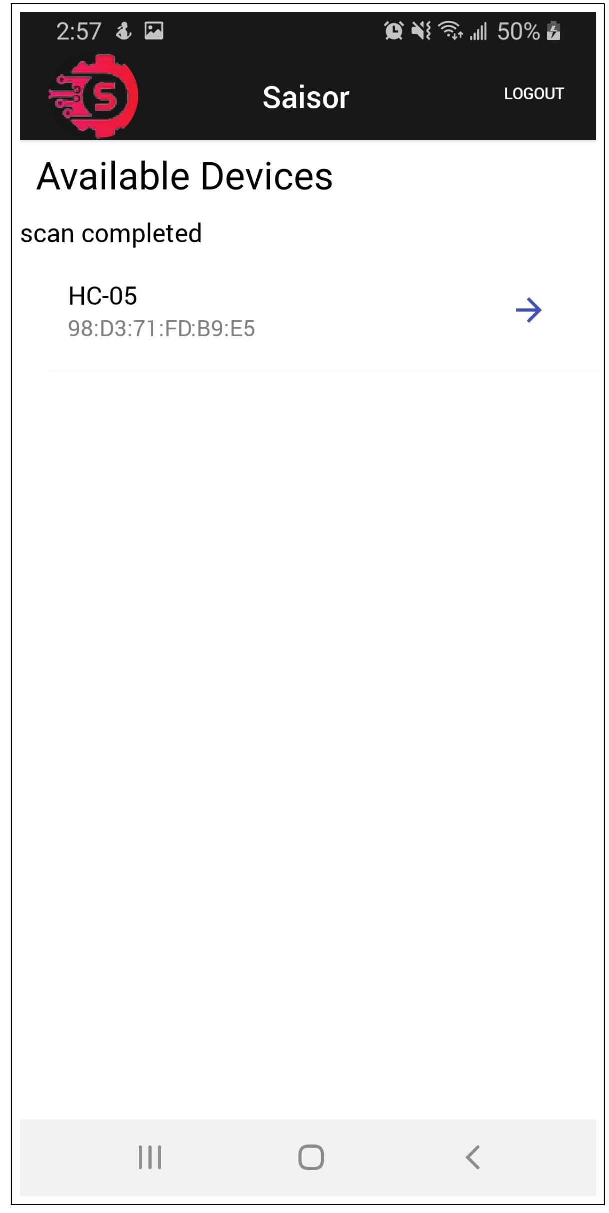


Figure 4.19: App screen to add a new device

```
  console.log(responseInfo);
}
```

A better option would have been to do a Dephi-Hellman key exchange using Curve25519 as described in the model. This step would be encrypted using the session key generated from the DH key exchange. In the current implementation, a random 256 bits key and 128 bits IV is generated and send along with a hello message in an encoded format after creating a connection. Then we wait for a response from the device for five seconds. The device is expected to return a hello message encrypted with the key and IV send to it. If the message is properly decrypted and we get a hello message, we are sure the next steps will be secured for the exchange of the Wi-Fi credentials. If the message is anything other than the expected hello message is received, the app terminates the connection with a cancel request to the device. The below Arduino function runs in the device to check if a user is requesting a Bluetooth connection.

```
void getConnected() {
  // wait till someone try to connect to the device using Bluetooth
  if (btSerial.available() > 0) {
    Serial.println("user connecting");
    availableMemory();
    // get the initial encoded connection request
    String input =  btSerial.readString();
    const int inputLen = input.length();
    char dInput [inputLen];
    device.decode(input, dInput);
    StaticJsonDocument<200> inputJson;
    DeserializationError error = deserializeJson(inputJson, dInput);
    if (error) {
      Serial.print(F("failed to deserialize input: "));
      Serial.println(error.c_str());
      return;
    }
```

```
    // check what is the connection message type
    if(strcmp(inputJson["message"],"HELLO") == 0){
      // get the key and IV
      hexCharacterStringToBytes(key, inputJson["key"]);
      hexCharacterStringToBytes(iv, inputJson["iv"]);
      // initiate adding a user
      addUser();
    } else {
      Serial.println("Invalid message");
    }
    // cleanup
    inputJson.clear();
  }
}
```

The device waits to see if there is a connection request. Once it receives the string, the device decodes it and extract the values for *message*, *key* and *iv*. The data is passed from the app as a JSON serialized string and so the device deserializes it after decoding. One thing to note here is that both the key and IV are passed as a Hex string. So, we are using the function *hexCharacterStringToBytes* to convert the key and IV into bytes. Then the device performs the next steps by calling the function *addUser*. In this function, the device sends a hello message back in a encrypted format. The function *sendUserBT* is called from the *addUser* function to send the response. We are using AES256 which has a block size of 128 bits. So, the first step to return the message was to pad the hello message to make it 16 bytes. Here we padded with zero. Then we encrypted the message with the given key and IV and encoded it to base 64 format to pass it back to the user app.

```
void sendUserBT(String message) {
  // get required variables
  int msgLen = message.length();
  int cipherLen = msgLen - (msgLen % 16) +
```

```
5     (msgLen % 16 > 0 ? 16 : 0);
6     //form message with padding
7     byte bMsg[cipherLen];
8     for(int i = 0; i < msgLen; i++) {
9       bMsg[i] = message[i];
10    }
11    // add padding
12    for(int i = 0; i < (cipherLen - msgLen); i++) {
13      bMsg[msgLen + i] = 0;
14    }
15    //encrypt
16    byte cipher[cipherLen];
17    byte plain[cipherLen];
18    crypto.clear();
19    crypto.setKey(key, 32);
20    crypto.setIV(iv, 16);
21    crypto.encrypt(cipher, bMsg, cipherLen);
22    //encode
23    int encLen = base64.encodedLength(cipherLen);
24    char response[encLen];
25    base64.encode(response, (char*)cipher, cipherLen);
26    Serial.println(response);
27    btSerial.write(response);
28  }
```

Connect to Wi-Fi. All the steps from here on are part of the *addUser* function and in each step, we will explain the specific part. he below code gets the Wi-Fi credentials from the user app.

```
1  /* STEP2: get user id of the user and WiFi credentials if primary user */
2  StaticJsonDocument<200> inputJson;
3  char plain[128];
4  rcvUserBT(plain);
```

```
Serial.println(plain);
DeserializationError error = deserializeJson(inputJson, plain);
if (error) {
  Serial.print(F("failed to deserialize input: "));
  Serial.println(error.c_str());
  return;
}
Serial.println("Message serialized");
const char* ssid = inputJson["ssid"];
const char* password = inputJson["password"];
const char* userId = inputJson["user_id"];
availableMemory();
```

The function *rcvUserBT* waits till it gets a response from the user back. Once it gets the response, it decodes the same and decrypts it using the key and IV that was passed before.

```
void rcvUserBT(char* plain) {
  // wait to get data from user
  while(btSerial.available() <= 0) {}
  String message =  btSerial.readString();
  Serial.println(message);
  int msgLen = message.length();
  char * msg = const_cast<char*>(message.c_str());
  int encLen = base64.decodedLength(msg, msgLen);
  byte bMsg[encLen];
  // decode
  base64.decode((char*)bMsg, msg, msgLen);
  Serial.println("Message decoded");
  // decrypt
  crypto.clear();
  crypto.setKey(key, 32);
  crypto.setIV(iv, 16);
  crypto.decrypt((byte*)plain, bMsg, encLen);
```

```
  Serial.println("Message decrypted");
}
```

As in the previous steps, the data is passed as a JSON serialized string. The device extracts the *ssid*, *password* and *user_id* from it. Once it is deserialize, the function tries to connect to the Wi-Fi of the user with the given credentials. The device waits for ten tries to connect. If it fails it returns a failed message to the user and wait for another new connection request. Once connected the device performs the next step to verify the user.

```
/* STEP3: Connect to Wifi with the given credentials */
Serial.print("Connecting to SSID: ");
Serial.println(ssid);
WiFi.begin(ssid, password);
myDisplay.message("Connecting to", const_cast<char*>(ssid));
// check if able to connect to WiFi
int counter = 0;
while (WiFi.status() != WL_CONNECTED) {
  delay(500);
  counter++;
  Serial.print(".");
  if(counter >= 10) {
    Serial.println("Failed to connect to WiFi");
    return;
  }
}
Serial.println();
Serial.println("Connected");
Serial.println(WiFi.localIP());
myDisplay.message("ACTION: ", "verifying user");
availableMemory();
```

From the user app, the code is straight forward. The user provides the inputs which

is converted into a serialized JSON and then encrypted. The encrypted text is forwarded to the device, which is processed as described above. The screenshot of the app is given in Figure 4.20. The following code does the encryption in the user app. The app waits for a period of one minute for the device to respond back after verification from the gateway. If it doesn't respond, the app rejects the connection request and lets the user know of the failed attempt and asks the user to try again.

```
const {ssid, password, key, iv} = this.state;
const user = await Auth.currentAuthenticatedUser();
console.log(key);
console.log(iv);
const data = {
  user_id: user.attributes.sub,
  ssid,
  password,
};
const crypto = new CryptoService();
const encData = crypto.encrypt(data, key, iv);
await BluetoothSerial.write(encData);
this.setState({
  message: 'WiFi credentials send, waiting response',
  state: 'WAITING_RESPONSE',
});
```

User verification. Once the device is able to connect to the Wi-Fi, it sends the information to the gateway to verify the identity of the user. From the *addUser* function the *verifyUser* function is called. The *user_id* that is given by the app is passed as an argument. The function creates a connection with the gateway by calling the *connect* on the *WiFiClientSecure* object. The client is passed the fingerprint of the gateway to verify the server certificate. Once the connection is established, the device generates a signature and attaches it in the body of the request. Then the request is send to

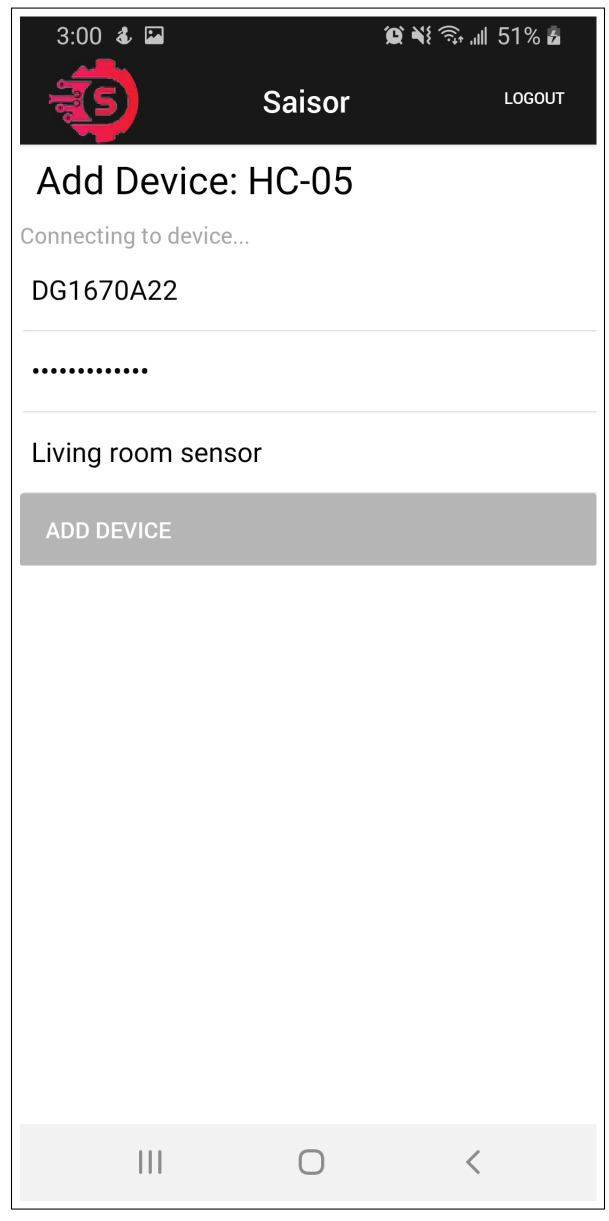


Figure 4.20: App screen to pass Wi-Fi credentials to the device after successful connection

the gateway over TLS (HTTPS) and the device waits for a response. If the response is success, the function returns true, otherwise false. The user verification is a important step to ensure that the user is who claims he is.

```
bool verifyUser(const char* userId) {
  WiFiClientSecure client;
  Serial.print("Verifying user ");
  Serial.println(userId);
  // verify that we are able to connect to the gateway
  client.setFingerprint(fingerprint);
  if (!client.connect(host, httpsPort)) {
    Serial.println("connection failed");
    return false;
  }
  String url = "/Prod/registration";
  uint8_t signature[64];
  Ed25519::sign(signature, devicePrivKey, devicePubKey,
     deviceId, strlen(deviceId));
  int encLen = base64.encodedLength(64);
  Serial.println("Generated signature");
  String data = String("{") +
        "\"device_id\":\"" + deviceId + "\"," +
        "\"user_id\":\"" + userId + "\"," +
        "\"millis\":\"" + millis() + "\"," +
        "\"verifier\":\"" + verifier + "\"" +
        "}";
  String payload = String("PUT ") + url + " HTTP/1.1\r\n" +
           "Host: " + host + "\r\n" +
           "Cache-Control: no-cache \r\n" +
           "Content-Type: application/json \r\n" +
           "Content-Length: " + data.length() + "\r\n" +
           "Connection: close \r\n\r\n" +
           data;
```

```
30   client.print(payload);
31   // clear the header
32   while (client.connected()) {
33     String line = client.readStringUntil('\n');
34     if (line == "\r") {
35       break;
36     }
37   }
38   hash.clear();
39   String line = client.readStringUntil('\n');
40   client.stop();
41   if (line.equals("{\"message\": \"SUCCESS\"}")) {
42     Serial.println("Success response from gateway");
43     return true;
44   } else {
45     Serial.println(line);
46   }
47   return false;
48 }
```

As mentioned before, the gateway consists of APIs which are hosted in the AWS cloud. For the user verification, a lambda with the name *saisor-api-AddRegistrationsFunction-10UVAJZSI0KLE* is created. This is the backend for the API endpoint PUT:/Prod/registration that is called by the device. On receiving the input, the gateway verifies that the required values are passed, including the *device_id*, *user_id* and *verifier*. As we have seen before, the gateway already has the public key of the device and so it takes the user_id and generates a signature of the same and matches it against the verifier that is provided. Once verified, the gateway is sure that the request has come from the device itself. Next the gateway verifies the state of the device along with the users registered to it. If this is not the primary user, then the gateway notifies the primary user of the request for approval. This additional step happens when a delegate tries to connect to

the device. Otherwise, it tags the current user as the primary user and creates a request in the “Registrations” table. A success response is sent to the device. The below Python lambda function does the user verification and device registration.

```
def lambda_handler(event, context):
  deviceInfo = None
  controller = None
  body = json.loads(event['body'])
  print(body)
  # authenticate
  try:
  # verify input
  assert 'device_id' in body, 'Missing device_id'
  assert 'user_id' in body, 'Missing user_id'
  assert 'millis' in body, 'Missing millis'
  assert 'verifier' in body, 'Missing verifier'
  # validate user
  user = User(body['user_id'])
  assert user.exists(), 'Invalid user'
  # validate device
  deviceInfo = validateAuth.validateDevice(
    body['device_id'], body['user_id'], body['verifier'])
  print('device_mac: ', deviceInfo['device_mac'])
  except Exception as ex:
    print(ex)
    return responses.AuthErrorResponse('Unauthorized access. {}'
                                       .format(ex.args[0]))
  # add new record to database
  try:
    notificationId = None
    # verify device
    controller = Registration(body['user_id'])
    primaryUser = controller.getPrimaryUser(deviceInfo['device_mac'])
```

```
30      print(primaryUser)
31      # First user converts the status to True for the device
32      assert primaryUser == None
33      and deviceInfo['device_status'] == False, \
34          'Invalid device state, contact customer support'
35      # notify the primary user
36      if primaryUser != None:
37          notification = Notification(body['user_id'])
38          message = 'Need approval for device ' + \
39              primaryUser['device_nickname']
40          notificationId = notification.add(
41              message, 'approve', {
42                  'device_mac': deviceInfo['device_mac'],
43                  'user_id': body['user_id']
44              })
45          print('user notified')
46      isPrimary = primaryUser == None
47      controller.add(deviceInfo['device_mac'], isPrimary)
48      print('record added successfully')
49      # if there is no previous user, send success
50      # else send approval send notice
51      if notificationId == None:
52          return responses.successResponse({
53        'message': 'SUCCESS'
54      })
55      else:
56          return responses.successResponse({
57        'message': 'NEED_APPROVAL'
58      })
59    except Exception as ex:
60      print(ex)
61      return responses.errorResponse('Failed to add. {}'
62                                      .format(ex.args[0]))
```

Once the device receives a success from the gateway, it sends a MAC address, encrypted with the key and IV passed in the first step, to the user. Otherwise it responds with a failed message.

Device verification. Like we mentioned earlier, the user app waits for a response from the device. Once it receives a response, it checks if itâĂŹs a success or failed response. If failed, the user is notified. Otherwise, the program extracts the MAC address and sends the same to the gateway for verification. The user calls the endpoint POST: /prod/registration for this operation. This API is protected behind a authorizer which extracts the authentication header and validates the JWT token from it. If the validation fails, the gateway responds with a "Unauthorized access" message.

In the previous step the gateway created the registration record. However, there is a field *user_approved* which is set to false. Here the gateway extracts the registration record created in the previous step and validates that all the required field are as they are supposed to be. Any discrepancy results in a error response with status code 400. After validation, the *user_approved* is updated to true and enables the device and user to communicate with each other. The gateway returns a success response to the user. The below Python code does the registration approval.

```
1 def lambda_handler(event, context):
2     username = None
3     # authenticate
4     try:
5         username = validateAuth.validate(event['requestContext'])
6         print('username: ', username)
7     except Exception as ex:
8         print(ex)
9         return responses.AuthErrorResponse('Unauthorized access. {}'
10                   .format(ex.args[0]))
11     # add new record to database
```

```
12    try:
13        body = json.loads(event['body'])
14        print(body)
15        assert 'device_mac' in body, 'Missing device_mac'
16        # if no nickname is provided, set the MAC as the nickname
17        if not 'device_nickname' in body:
18            body['device_nickname'] = body['device_mac']
19        controller = Registration(username)
20        # validate the registration record
21        registration = controller.get(body['device_mac'])
22        assert registration != None, 'Missing registration record'
23        assert registration['user_approved'] == False,
24      'operation already performed'
25        assert registration['reg_status'] == False,
26      'invalid device state'
27        assert registration['device_approved'] == True,
28      'record not approved by device'
29        # set required attributes
30        # if this is the primary user, no other approval required
31        if registration['is_primary'] == True:
32            controller.primaryUserApproved = True
33            controller.status = True
34        controller.userApproved = True
35        controller.deviceMac = body['device_mac']
36        controller.deviceNickName = body['device_nickname']
37        controller.update()
38        # return success
39        return responses.successResponse({'message': 'SUCCESS'})
40    except Exception as ex:
41        print(ex)
42        return responses.errorResponse('Failed response. ERROR: {}'
43          .format(ex.args[0]))
```

Generate and share the symmetric key. On getting a success response from the gateway, the user app generates a 32-byte key and 16 byte initial vector and sends it to the device similar to step 1 but encrypted with the previously generated key and vector. The device saves the same in EEPROM along with the user_id. It also saves the Wi-Fi credentials. This way, if the device is disconnected and then connected back, it can use the saved credentials to connect to the internet. Then the user disconnects from the Bluetooth session and opens it for another user to connect.

The entire source code of the device can be found in Appendix C. The above implementation effectively replicates the P3 connection model. With this model, the device and user can securely establish a connection and verify each other. The connection establishment is also simple from the user's point of view. As a user, we had to only select the device and enter the Wi-Fi credentials. Everything else was done behind the scene. Also, one of the weakness in the current state-of-the-art platforms is that there are default keys set for communicating between the user and device. This model eliminates the defaults and sets up a key on the fly. This enhances the security of the model and the whole IoT ecosystem. It also opens the possibility of refreshing the key and IV on a regular basis. For a user, that will only be a click of a button.

4.3 Communicating with the device

As described in Section 3.5 the communication can be broadly classified as heartbeat communication and command execution. With the heartbeat, the device informs the gateway that it is functioning properly. The command execution defines the standard that the user must follow in order to talk to the device.

4.3.1 Implementing the heartbeat

In the heartbeat communication, the device sends a pulse every minute. We have set the timing for the effectiveness of analysis but should be done based on the requirement of the user or manufacturer. The function performing the heartbeat operation starts by gathering the current temperature, humidity and heat index from the DHT22 sensor. The rest of the process is like the one we have seen in user verification in the P3 connection model. The device signs the request (we have chosen the device ID) and attaches it as a “verifier” along with the rest of the data. The data is sent over a secure HTTPS connection to the gateway. The below code performs the operation for the heartbeat in the device.

```
void sendHeartBeat() {
  unsigned long timeDiff = millis() - heartbeatCounter;
  // send heartbeat every 10 min
  if(timeDiff < 60000) {
    return;
  }
  unsigned long startTime = millis();
  heartbeatCounter = millis();
  Serial.println("Initiating heartbeat");
  availableMemory();
  WiFiClientSecure client;
  // verify that we are able to connect to the gateway
  client.setFingerprint(fingerprint);
  if (!client.connect(host, httpsPort)) {
    Serial.println("connection failed");
    return;
  }
  String url = "/Prod/sensor/data";
  uint8_t signature[64];
  Ed25519::sign(signature, devicePrivKey,
```

```
21      devicePubKey, deviceId, strlen(deviceId));
22    int encLen = base64.encodedLength(64);
23    char verifier[encLen];
24    base64.encode(verifier, (char*)signature, 64);
25    Serial.println("sending payload...");
26    String data = String("{") +
27          "\"device_id\":\"" + deviceId + "\"," +
28          "\"temperature\":" + temperature + "," +
29          "\"humidity\":" + humidity + "," +
30          "\"heatindex\":" + heatIndex + "," +
31          "\"millis\":" + millis() + "," +
32          "\"verifier\":\"" + verifier + "\"" +
33          "}";
34    String payload = String("PUT ") + url + " HTTP/1.1\r\n" +
35             "Host: " + host + "\r\n" +
36             "Cache-Control: no-cache \r\n" +
37             "Content-Type: application/json \r\n" +
38             "Content-Length: " + data.length() + "\r\n" +
39             "Connection: close \r\n\r\n" +
40             data;
41    client.print(payload);
42    // clear the header
43    while (client.connected()) {
44      String line = client.readStringUntil('\n');
45      if (line == "\r") {
46        break;
47      }
48    }
49    String line = client.readStringUntil('\n');
50    Serial.println("...response received");
51    client.stop();
52    if (line.equals("{\"message\": \"SUCCESS\"}")) {
53      Serial.println("Success response from gateway");
```

```
  } else {
    Serial.println(line);
  }
  availableMemory();
  Serial.print("[Heartbeat] Total time: ");
  Serial.println(millis() - startTime);
}
```

On receiving the request, the gateway verifies that all the required values are present. Then it verifies the signature passed in the “verifier” field with the public key stored in the *Devices* database. Then the gateway updates the *activity_timestamp* field to indicate the last time the heartbeat was received. The below code performs the operation from the gateway to record the heartbeat.

```

import json
import validateAuth
import responses
from devicedata.controllers.DataController import DeviceData
from transaction.controllers.TransactionController import Transaction

log = Transaction()


def lambda_handler(event, context):
    deviceInfo = None
    controller = None
    body = json.loads(event['body'])
    print(body)
    # authenticate
    try:
        # verify input
        assert 'device_id' in body, 'Missing device_id'
```

```
        assert 'temperature' in body, 'Missing temperature data'
        assert 'humidity' in body, 'Missing temperature data'
        assert 'heatindex' in body, 'Missing temperature data'
        assert 'millis' in body, 'Missing millis'
        assert 'verifier' in body, 'Missing verifier'
        # validate device
        deviceInfo = validateAuth.validateDevice(
            body['device_id'], body['device_id'], body['verifier'])
        print('device_mac: ', deviceInfo['device_mac'])
    except Exception as ex:
        print(ex)
        return responses.AuthErrorResponse('Unauthorized access. {}'
                                           .format(ex.args[0]))
    # add new record to database
    try:
        payload = {
            'temperature': body['temperature'],
            'humidity': body['humidity'],
            'heatindex': body['heatindex']
        }
        print()
        controller = DeviceData(body['device_id'])
        controller.add(payload)
        # auditlog transaction
        log.add('device:' + body['device_id'], 'gateway', payload)
        return responses.successResponse({'message': 'SUCCESS'})
    except Exception as ex:
        print(ex)
        return responses.errorResponse(
  'Failed to save device data. ERROR: {}'.format(ex.args[0]))
```

One thing to note in the above code is the introduction of “log.” The class *Transaction* creates the interface for every operation to log the activity. This helps in audit

operation. Logs are an essential part of any software solution It helps in identifying flaws and errors in code execution.

Along with the heartbeat pulse that is sent from the device, the gateway runs another program called the **heartbeat checker.** The job of the checker is to make sure that all active devices are sending the heartbeat regularly. If there is no heartbeat within the last few minutes (we have specified it to five minutes for quick analysis), the checker sends a alert email notification to the user informing that the device is not responding. The below code performs the operation of the checker.

```
import boto3
from devicedata.controllers.DataController import DeviceData
from device.controllers.DeviceController import Device
from registration.controllers.RegistrationController import Registration
from user.controllers.UserController import User
from datetime import datetime
from settings import appSetting


def lambda_handler(event, context):
  print('Getting all devices')
  deviceController = Device()
  devices = deviceController.getAllActive()
  assert len(devices) > 0, 'No active device found'
  for device in devices:
    try:
    dataController = DeviceData(device['device_id'])
    deviceData = dataController.get()
    assert deviceData != None, 'No device data present for {}'
     .format(dataController.activityDate)
    hbTime = datetime.fromisoformat(
      deviceData['activity_timestamp'])
    curTime = datetime.now()
```

```
24    timeDiff = curTime - hbTime
25    deviceController.deviceId = device['device_id']
26    if timeDiff.total_seconds() <= 900:
27      print('device {} is healthy'.format(device['device_id']))
28      deviceController.updateStatus(True)
29    else:
30      print('device {} is not responding'.format(
31        device['device_id']))
32      print(timeDiff.total_seconds())
33      deviceController.updateStatus(False)
34      # notify user via email
35      regController = Registration()
36      pUser = regController.getPrimaryUser(device['device_mac'])
37      assert pUser != None, 'Failed to find primary user'
38      userController = User(pUser['user_id'])
39      email = userController.getEmail()
40      assert email != None, 'Failed to get user email'
41      sendEmail(email, pUser['device_nickname'],
42              timeDiff.total_seconds())
43      print('User notified')
44    except Exception as ex:
45    print('error in processing {}'.format(device['device_id']))
46    print(ex.args[0])
```

The checker is an important part of the framework to ensure the physical safety of the device. If the device is offline by an attack or a power outage, the user will be informed immediately. The same architecture can also be used for critical situation like a fire. In the temperature and humidity sensor, we can verify that the temperate is within an acceptable range. If not, the device can inform the gateway and a similar checker can be implemented to alert the user. For this thesis we are using email as an alert mechanism because that is the only contact information we are collecting from the user. A system can easily collect the phone number of the user and send a SMS

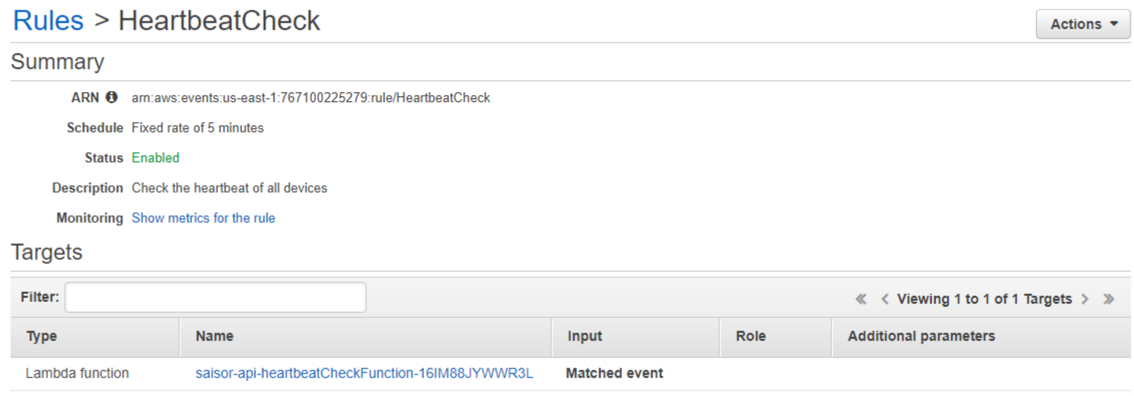


Figure 4.21: Setting up heartbeat checker in AWS

instead. Another option would be to implement a push notification to the users app.

To implement the checker, we added an event rule in AWS cloud watch as shown in Figure 4.21. AWS provides the interface for setting up cron jobs that we can schedule to run on a fixed interval or based on a occurrence of another event. For our case, we have scheduled the job on a regular interval of 5 minutes. The timing can be customized based on the need of the manufacturer or the user.

4.3.2 Executing command on the device

The command execution described in Section 3.5.2 model ties all the framework together. Majority of the time, a user asks an IoT device to perform some operation or action. In our case, we had designed the temperature-humidity (HT) sensor with two major tasks. The user can:

- Ask the current readings of the sensor
- Ask the device to switch the display from Celsius to Fahrenheit and vise-versa

Both the operations are carried out in the same way. The user encrypts the command using the shared key created in the P3 connection model. This command is sent to the gateway along with the device MAC and JWT token for user authentication header. On receiving the request, the gateway verifies the user and the user's registration with

the device. After successful verification, the gateway creates a signature with his own private key and the given command. The signature along with the encrypted command is sent to the device.

The device on receiving the package, verifies the signature to ensure that the request is indeed coming from the gateway. Like we mentioned before, all request that comes to the device must be from the gateway. This minimizes the load on the device to verify the identity of every sender. Also, it ensures that no request comes to the device from any other source. After verification, the device decrypts the command from the user and performs the necessary action. While returning a response, the device again encrypts the device data with the shared key and adds its own signature for the gateway to verify. The gateway on receiving the response and successful verification, sends the encrypted data back to the user.

One thing to note here is the upkeep of the zero-trust principle. The command that is sent from the user to the device cannot be read by the gateway. This ensures that the communication, though passing through the gateway, is secured between the user and device. No other party can decipher that conversation. Next we will explain the above implementation in detail.

We will start by explaining the request sent from the user to the gateway. The below code runs in the user's mobile. As you see below, the app extracts the user's key and IV. This key and IV was stored in the *AsyncStorage* by the P3 connection model when the user connected with the device. Next the app uses that information to encrypt the command passed as a function parameter and then executes the API call to pass the device MAC and the encrypted command. In the P3 connection model implementation we had seen that the user knows only the MAC address of the device, which is passed to the gateway for registration confirmation.

```
1 const {device} = this.state;
2 let service = new Command();
```

```
const crypto = new CryptoService();
const key = await AsyncStorage.getItem(device.device_mac + '_key');
const iv = await AsyncStorage.getItem(device.device_mac + '_iv');
console.log(key);
console.log(iv);
let cipherCmd = crypto.encrypt(command, key, iv);
let response = await service.execute(device.device_mac, cipherCmd);
```

On receiving the request from the user, the gateway verifies the JWT token using the *authorizer* header. If the authorization fails, an error response is sent with a status code of 401. If successfully verifies, the gateway next checks that the required information is available in the request and goes on to verify that the user is registered to the device. The next step is to generate the signature using the *signingKey* which is the private key of the gateway and then executes a GET request to the device. One thing to note here is that the information is being send as query parameters to the GET request. This is because GET method doesn't accept body field.

```
def lambda_handler(event, context):
    username = None
    # authenticate user
    try:
        username = validateAuth.validate(event['requestContext'])
        print('username: ', username)
    except Exception as ex:
        print(ex)
        return responses.AuthErrorResponse('Unauthorized access. {}'.format(ex.
    args[0]))
    try:
        body = json.loads(event['body'])
        print(body)
        assert 'device_mac' in body, 'Missing device_mac'
        assert 'command' in body, 'Missing command'
```

```python
        # validate registration
        print('Validating user registration for device')
        controller = Registration(username)
        registration = controller.get(body['device_mac'])
        assert registration != None, 'No registration record found'
        assert registration['reg_status'] == True, 'Registration not active'
        # get device info
        controller = Device()
        device = controller.getByMac(body['device_mac'])
        assert device != None, 'No device found'
        print('device IP ', device['ip_address'])

        # execute command on device
        print('getting data from device')
        signingKey = nacl.signing.SigningKey(appSetting['signingKey'],
                                             encoder=nacl.encoding.HexEncoder)
        signed = signingKey.sign(str.encode(body['command']))
        verifier = base64.b64encode(signed.signature).decode('utf-8')
        uri = '{}?data={}&verifier={}'.format(
            device['ip_address'], body['command'], verifier)
        print(uri)
        response = requests.get(uri)
```

On receiving the packet from the gateway, the device runs the function *handleCommand()*. To start the device does the sanity check of the passed query parameters and the request method.

```c
Serial.println("received command...");
availableMemory();
// verify GET method
if(server.method() != HTTP_GET){
  Serial.println("Invalid HTTP method");
  server.send(400, "application/json", "Invalid request");
  return;
```

```
8 }
9 // verify parameters
10 String cipherCmd = server.arg("data");
11 String verifier = server.arg("verifier");
12 verifier.replace(' ', '+');
13 cipherCmd.replace(' ', '+');
14 if(cipherCmd == "" || verifier == "") {
15   Serial.println("...Missing parameters");
16   server.send(401, "text/plain", "Invalid parameters");
17   return;
18 }
19 Serial.println(cipherCmd);
20 Serial.println(verifier);
```

Then it verifies the signature and ensures that it is indeed from the gateway. It uses the public key of the gateway for verifying the identity. If the verification fails, a 401 response is sent back.

```
1 // verify the signature - must be from gateway
2 int sLen = verifier.length();
3 char* encSignature = const_cast<char*>(verifier.c_str());
4 int encLen = base64.decodedLength(encSignature, sLen);
5 byte signature[encLen];
6 base64.decode((char*)signature, encSignature, sLen);
7 bool isVerified = Ed25519::verify(signature, serverPubKey, cipherCmd.c_str(),
    cipherCmd.length());
8 if(!isVerified) {
9   Serial.println("...Signature verification failed");
10   server.send(401, "text/plain", "Invalid verifier");
11   return;
12 }
13 Serial.println("Signature verified successfully");
14 availableMemory();
```

The command is decrypted using the user's key and IV and based on the command, the *inFahrenheit* flag is set. This flag helps to determine the display between Celsius and Fahrenheit.

```
// decode command
Serial.println("Decrypting command");
UserInfo user = storage.getUser();
int cipherLen = cipherCmd.length();
char* cipher = const_cast<char*>(cipherCmd.c_str());
encLen = base64.decodedLength(cipher, cipherLen);
byte bCmd[encLen];
char command[6];
base64.decode((char*)bCmd, cipher, cipherLen);
Serial.println("Command decoded");
crypto.clear();
crypto.setKey(user.key, 32);
crypto.setIV(user.iv, 16);
crypto.decrypt((byte*)command, bCmd, encLen);
command[6] = '\0';
if(strcmp(command, "TYPE_C") == 0) {
  inFahrenheit = false;
} else if (strcmp(command, "TYPE_F") == 0) {
  inFahrenheit = true;
} else {
  Serial.println("...Invalid command");
  server.send(400, "text/plain", "Invalid command");
  return;
}
bool isRefreshed = refreshData(true);
if(!isRefreshed) {
  Serial.println("...Invalid command");
  server.send(400, "text/plain", "Failed to refresh data");
  return;
```

```
30 }
```

The device then encrypts the sensor data using the user's key which it gets form the storage using *storage.getUser()*. In this case, we have only implemented one user, if there were more, then the gateway would have to send the user ID for the device to determine who is talking to the device and get the appropriate key and IV. Also, notice how we used padding here to make sure that the data is a multiple of 128 bits. This is required because we are using CBC mode on AES for encrypting the data. The cipher response is then encoded in Base64 format for transferring it over the wire.

```
1  Serial.print("Temperature: ");
2  Serial.println(temperature);
3  String data = String("{") +
4        "\"temperature\":" + temperature + "," +
5        "\"humidity\":" + humidity + "," +
6        "\"heatindex\":" + heatIndex + "," +
7        "\"counter\":" + millis() +
8    "}";
9  availableMemory();
10
11 // encrypt response data
12 int dataLen = data.length();
13 cipherLen = dataLen - (dataLen % 16) + (dataLen % 16 > 0 ? 16 : 0);
14 //form message with padding
15 byte bData[cipherLen];
16 for(int i = 0; i < dataLen; i++) {
17   bData[i] = data[i];
18 }
19 // add padding
20 for(int i = 0; i < (cipherLen - dataLen); i++) {
21   bData[dataLen + i] = 0;
22 }
23 //encrypt
```

```
24 byte cipherData[cipherLen];
25 crypto.clear();
26 crypto.setKey(user.key, 32);
27 crypto.setIV(user.iv, 16);
28 crypto.encrypt(cipherData, bData, cipherLen);
29 //encode
30 encLen = base64.encodedLength(cipherLen);
31 char resData[encLen];
32 base64.encode(resData, (char*)cipherData, cipherLen);
33 resData[encLen] = '\0';
34 Serial.println("Encrypted response");
35 availableMemory();
```

The last step in the device is to add the signature for the gateway to verify. Then a 200 response is sent to the gateway with the *data* and *verifier*.

```
1 // add signature for verification at gateway
2 uint8_t resSign[64];
3 Ed25519::sign(resSign, devicePrivKey, devicePubKey, resData, strlen(resData));
4 encLen = base64.encodedLength(64);
5 char resVerifier[encLen];
6 base64.encode(resVerifier, (char*)resSign, 64);
7 // send response to gateway
8 String response = String("{") +
9       "\"data\":\"" + resData + "\"," +
10      "\"verifier\":\"" + resVerifier + "\""
11   "}";
12 server.send(200, "application/json", response);
13 Serial.println("...response send");
14 availableMemory();
```

The gateway on receiving the response verifies that the signature of the device is valid and passes the *data* component to the user with a *successResponse*.

```
1 if(response.status_code != 200):
```

```
        raise Exception('Failed to get data from device')
# verify device signature
data = json.loads(response.content.decode('utf-8'))
print('verifying device signature')
assert 'data' in data, 'Missing data element in device response'
assert 'verifier' in data, 'Missing verifier in device response'
print(data['data'])
print(data['verifier'])
validateAuth.validateDevice(
    device['device_id'], data['data'], data['verifier'])
# return device data to user
response = {
    'data': data['data']
}
        return responses.successResponse(response)
```

The user on receiving the response from the gateway, decrypts the message and converts the response to a JSON. This ensures that the data is in the correct format. One thing to note here is that the device sends the *millis()* value along with the response. This helps the user to verify that this is not a replayed packet and it is coming from the device itself. To check for the replay attack, the user verifies that the given *counter* value is greater than the one previously sent. The user saves this information in the *AsyncStorage*. This is mainly to maintain a consistent state in the app. The next time the user comes back to this screen, he is shown the last values extracted from the device. He can also refresh the data or change the display value in the device.

```
console.log(response);
let decData = crypto.decrypt(response, key, iv);
let index = decData.indexOf('}');
decData = decData.substring(0, index + 1);
console.log(decData);
let data = JSON.parse(decData);
```

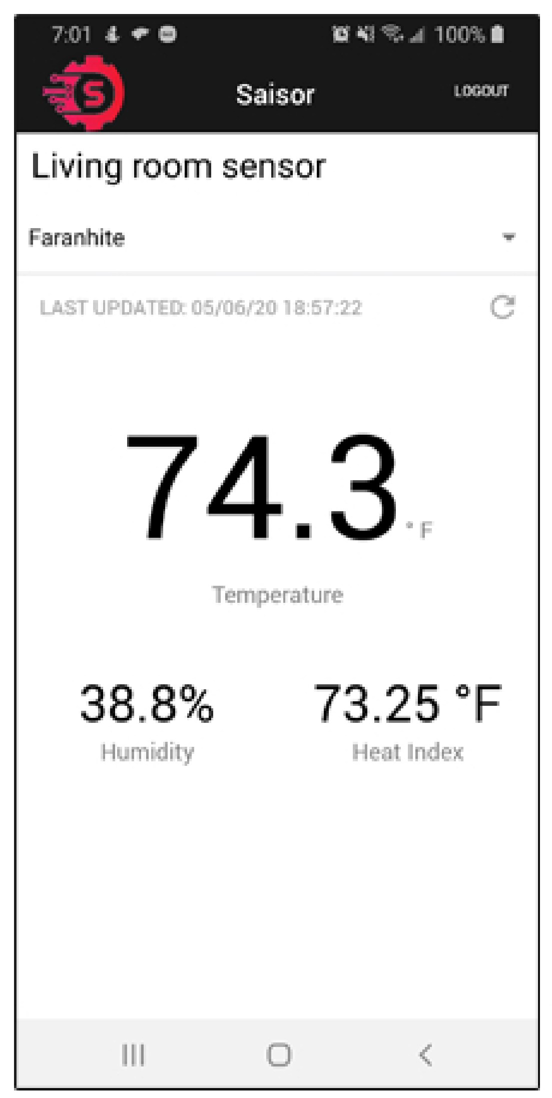
```
// make sure there is no replay attack
if (data.counter <= this.state.data.counter) {
  throw new Error('Invalid response from device');
}
let now = new Date();
let lastUpdated = now.toLocaleDateString() + ' ' + now.toLocaleTimeString();
await AsyncStorage.setItem('last_updated', lastUpdated);
await AsyncStorage.setItem(device.device_mac, JSON.stringify(data));
await AsyncStorage.setItem('type', command);
this.setState({
  lastUpdated,
  data,
});
```

The UI for the user app is simple. From the devices screen, the user clicks on the arrow right to the name of the device and is taken to the detailed screen. There the user is presented with the last data recorded from the device. The user can change the data type and the same is reflected in the device as well as in the screen as shown in Figure 4.22. The user can also refresh and get the latest sensor values.

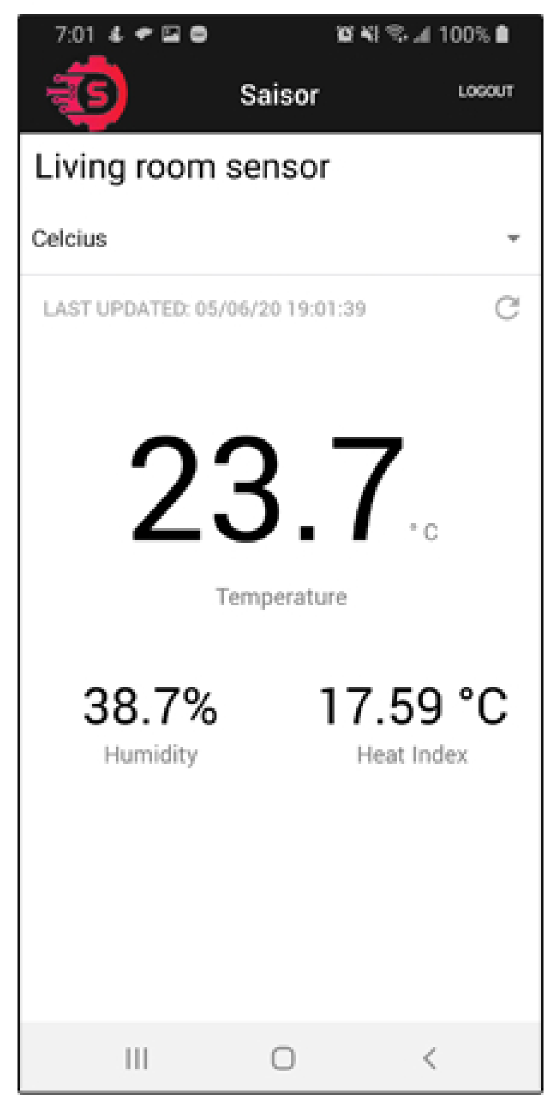
The command execution uses all the security principles described in all the other models and protocols in the framework. Like we mentioned before, for security reasons, we ensured that all the communications to the device flows through the gateway. However, we didn't want the gateway to have the command that the user wished to execute on the device. The blanket layer of protection provided by the shared key ensured that the communication between the device and user only within the two of them.

4.4 Summary

To summarize the security implementation, we used an ED25519 key pair for the gateway, another Ed25519 key-pair for the device. These key-pairs provide authentication,



(a) Device data in Fahrenheit



(b) Device data in Celsius

Figure 4.22: Data captured from the device on the user's app

integrity and non-repudiation between the device and gateway. Each provide a signature to the other using their private key and the same is verified by the other party. Since, all communication to the device comes only through the gateway, it needs to verify only one sender. Any failed verification is responded with an unauthorized 401 status. Elliptic curve cryptography (ECC) is particularly effective for devices with limited resources. The key size we used for the signing is 32 bytes (256 bits). This is way smaller in comparison to the 2048 bits key from RSA. The operation was fast and could be easily performed within the resource limitation of our device.

The heartbeat communication is a good example of the above. All the API in the gateway are protected by SSL/TLS and thus protected in transit. This helps maintain the confidentiality of data in transit. We used the device's private key to sign the payload that was send to the gateway. This added authentication and integrity check to the data. The heartbeat checker played a role to ensure physical security of the device as well. It maintained a threshold of five minutes to check if an active device had sent a pulse. If not, the user is notified that the device is not functional. Here we took a more reactive approach for detection and notification.

For the communication between the device and the user, we used AES 256 with CBC mode. The P3 connection model helped setup this key. In the model we described using ECDH key exchange to create a session key between the user and device to protect the remaining communications. However, due to limitation in available technology, we were not able to implement the ECDH exchange. We shared a key and IV in a base64 encoded format and then used that as a session key. All other Bluetooth communication in the P3 connection model were encrypted using that. The CBC mode provides an effective diffusion of the cipher data and is widely used. Since, the key is auto generated we used a symmetric encryption to secure the communication between the user and device. The command execution model brought all the above together. Here the user encrypted the command using the shared key generated in the P3 connection and send

it to the device via gateway. This prevents the gateway from knowing the command itself. However, the gateway adds its signature to the request to let the device know that the request is coming from a genuine source. Similarly, in response, the device encrypts the device data using the shared key and adds its own verifier for the gateway to verify and the user to read the sensor data.

The zero-trust framework effectively secures the communications between all the three parties, namely, device, user and gateway. Each part of the framework plays an important role in providing the overall security. The next question to ask is how efficient the model is. In the next chapter we will analyze the performance of the model in four aspect: security, operational time, memory utilization of device and cost of implementation.

Chapter 5

Analyzing the Framework

In this chapter we analyze the effectiveness of the zero-trust framework that we described and implemented so far. The framework should effectively secure the communications between the different actors. So, we will look at network logs to check if there is any information leakage when the information is flowing over the insecure medium. We will also check the memory usage of the device in each step. One of the key issues as we have seen before with IoT devices is that the resource available for them is very small. We will analyze if we are able to provide adequate encryption with the limited resource of the device. The next thing we want to measure is the time of operation. In the security framework that we have defined, there are lot of different models and transactions. Some transactions take multiple hops to get the response to the requestor. We need to ensure that the responses are returned in a acceptable time-frame. Another angle we wanted to explore was the cost to build and run a IoT device we implemented. Cost is a important factor and we kept this in mind during our implementation. We restricted the cost of the device which can be affordable to everyone.

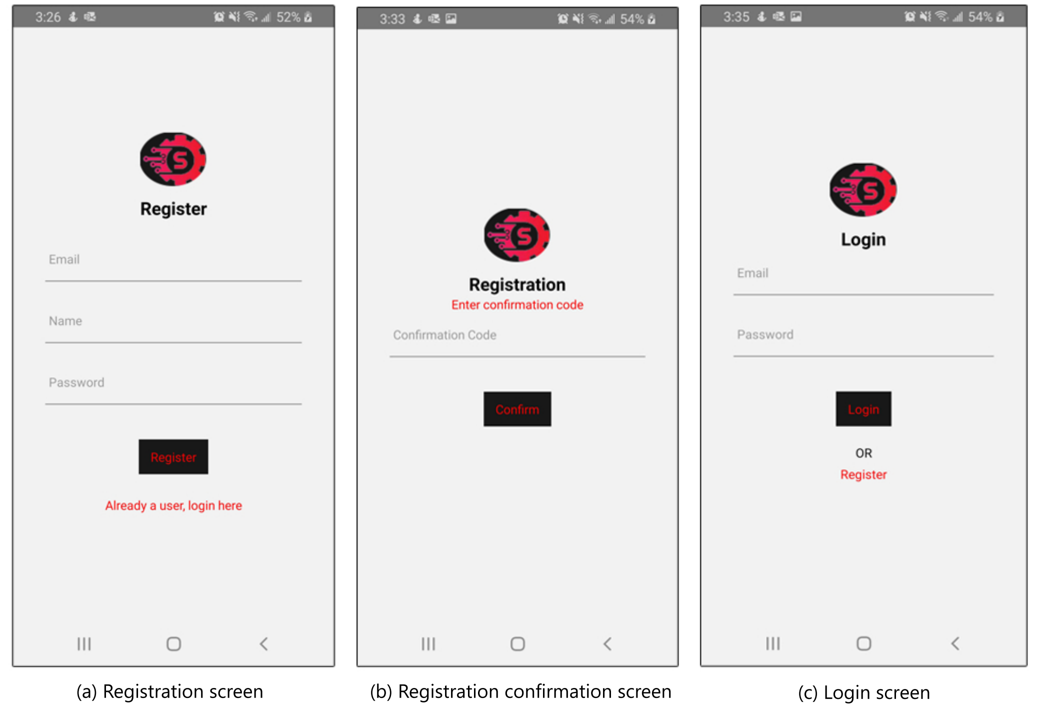
So, to summarize, in the following sections we will analyze the level of security provided by the framework, the memory utilization, the time to respond and cost of

building the device. In each of the sections we will delve into the different parts of the framework, namely, P3 connection model, heartbeat communication and command execution. We will start by looking at the effectiveness of the security framework.

5.1 Security in transit

Any user who wants to use the "Saisor" devices, need to register themselves with the gateway. For this thesis, we kept the user details to very few required fields, namely, email, name and password. Once the user enters the details and clicks on register, a confirmation code is sent to the provided email. The user must enter the code in the confirmation screen and only then they can use any device from Saisor. After a successful confirmation of the user, the system redirects the user to the login screen where they need to provide their email and password and enter the system. Figure 5.1 shows the registration, confirmation screen and the login screen.

Figure 5.2 shows the confirmation email sent to the user. This is the first step for authenticating the user because in this process, the user is assigned a user identification in the form of a UUIDv4. This unique identifier is used to locate the user record and used everywhere else in the application. The user authentication service is maintained using AWS Cognito. It provides a secure user directory that can scale to millions of users. It supports identity federation using SAML, Facebook and other social media platforms. Cognito has support for multi-factor authentication and encryption of data both at rest and in transit. It provides compliance with standards like HIPPA, SOC, PCI DSS and others. With a Cognito backend, we can ensure that the user authentication and authorization is securely handled. Any post login request sent to the gateway, we use the JWT generated by the app and pass it to the backend API. All user centric API are protected by an auth-guard. The auth-guard verifies the token and ensures the identity of the user. Before passing it to the relevant AWS Lambda, the authorizer adds



(a) Registration screen (b) Registration confirmation screen (c) Login screen

Figure 5.1: App screen to show registration, confirmation and user login

Figure 5.2: Email send to user for verification

the identity of the user in the event request. The lambda can read and verify the user information which is available in the request before processing it. Any failure in the authentication step results in a 401 unauthorized access error.

The security of the device is also important. During the manufacturing of the device, a public and private key pair is generated and embedded in the device along with its device identification (a UUIDv4 value). Whenever communicating with the gateway, the device adds a verifier to the request. The verifier is a signature generated and signed with this key pair. The private key is only available to the device, but the pubic key is

ip.addr == 192.168.137.182

No.	Time	Source	Destination	Protocol	Length	Info
6278	255.222317	192.168.137.1	192.168.137.182	DHCP	344	DHCP Offer - Transaction ID 0x5604d8f8
6280	255.256814	192.168.137.1	192.168.137.182	DHCP	344	DHCP ACK - Transaction ID 0x5604d8f8
6284	255.651547	192.168.137.182	192.168.137.1	DNS	106	Standard query 0x0001 A pbiuhtxv1l.execute-api.us-east-1.amazonaws.com
6286	255.700903	192.168.137.1	192.168.137.182	DNS	170	Standard query response 0x0001 A pbiuhtxv1l.execute-api.us-east-1.amaz
6287	255.707238	192.168.137.182	13.226.182.84	TCP	62	63846 → 443 [SYN] Seq=0 Win=2144 Len=0 MSS=536 SACK_PERM=1
6288	255.737778	13.226.182.84	192.168.137.182	TCP	62	443 → 63846 [SYN, ACK] Seq=0 Ack=1 Win=29200 Len=0 MSS=1460 SACK_PERM=
6289	255.742235	192.168.137.182	13.226.182.84	TCP	54	63846 → 443 [ACK] Seq=1 Ack=1 Win=2144 Len=0
6290	255.782187	192.168.137.182	13.226.182.84	TLSv1.2	305	Client Hello
6291	255.820582	13.226.182.84	192.168.137.182	TCP	54	443 → 63846 [ACK] Seq=1 Ack=252 Win=30016 Len=0
6292	255.822902	13.226.182.84	192.168.137.182	TLSv1.2	590	Server Hello
6293	255.822975	13.226.182.84	192.168.137.182	TCP	590	443 → 63846 [ACK] Seq=537 Ack=252 Win=30016 Len=536 [TCP segment of a
6294	255.823072	13.226.182.84	192.168.137.182	TCP	590	443 → 63846 [ACK] Seq=1073 Ack=252 Win=30016 Len=536 [TCP segment of a
6295	255.823134	13.226.182.84	192.168.137.182	TCP	590	[TCP Window Full] 443 → 63846 [PSH, ACK] Seq=1609 Ack=252 Win=30016 Le
6296	255.831509	192.168.137.182	13.226.182.84	TCP	54	63846 → 443 [ACK] Seq=252 Ack=1073 Win=1072 Len=0
6297	255.832542	192.168.137.182	13.226.182.84	TCP	54	[TCP ZeroWindow] 63846 → 443 [ACK] Seq=252 Ack=2145 Win=0 Len=0
6298	255.879309	192.168.137.182	13.226.182.84	TCP	54	[TCP Window Update] 63846 → 443 [ACK] Seq=252 Ack=2145 Win=536 Len=0
6299	255.886040	192.168.137.182	13.226.182.84	TCP	54	[TCP Window Update] 63846 → 443 [ACK] Seq=252 Ack=2145 Win=1072 Len=0

Figure 5.3: Wireshark logs showing use of TLS

shared between the device and gateway. On receiving a request from the device, the gateway uses the given device ID to locate the public key of the device. Then it uses the given signature and stored public key to verify the identity of the device. Like the user, a failed verification results in a 401 unauthorized access. This technique is used in both the P3 connection model and heartbeat communication to authenticate the device to the gateway.

The gateway is a set of API endpoints that are hosted in AWS API Gateway. All the APIs are by default enabled with HTTPS. Thus, all communications that are happening to the gateway are encrypted in transit using SSL/TLS as you can see in Wireshark logs in Figure 5.3. To verify the identity of the gateway to the device, the fingerprint of the server's digital certificate is provided to the device. The device can use that to verify the identity of the server in TLS handshaking. The server's public key is also provided to the device for any request coming from the server to the device. The device can utilize this public key to verify the identity of the server. This is used in sending command from the user app to the device via gateway. According to the model, all request to the device should only come from the gateway and the gateway's public key is used to verify the request. Any unauthorized attempt is responded with a 401 unauthorized access.

Till now we have established the protection around all communication to and from

the gateway. To recap, the user passes the JWT token in authorization header to validate itself. For the device, it sends it device identifier along with a signature that is signed with its own private key. This signature can be verified by the gateway. The next step is to establish security for communication between the device and user. The P3 connection model plays a major role in setting up the keys for them to talk. As we have described in the Section 3.4, the device and user communicate with each other to setup a shared key. In the process both communicates with the gateway to verify the identity of the other. This way, both the device and the user can be sure that they are talking to a genuine other party. This key that is generated is stored locally in the user's app and the device. This is not passed onto the gateway. This way we can be sure that even when the gateway is compromised, it cannot communicate to a device pretending to be the user. The principle of "trust noone, verify everyone" prevails with this model.

The P3 connection model sets the stage for the implementation of zero-trust. In this process, the user and device not only set up the shared key but also validates the identity of each other. The gateway is the repository of the identity of both the user and device. It helps validate the claim of both the device and user to each other. In the cryptographic techniques we have used both symmetric as well as asymmetric encryption. The asymmetric encryption is necessary between the gateway and the device because the keys are prepopulated. Without an asymmetric encryption, compromising one can also compromise the other. Among the available public key encryption technology RSA is the most popular for digital signature. However, implementing RSA would have been expensive with the limited resources available in the device. We used ED25519 (Elliptic Curve Cryptography Digital Signature) instead of RSA. The reason for choosing ECC is that it provides similar security with minimal resource utilization and energy. Table 5.1 and Table 5.2 shows the signing and verification time between RSA and ECC. As we see that RSA outperforms ECC in verification, but the signing time becomes better for ECC in comparison to RSA as the key size grows [65]. In our case we are using a

Table 5.1: Comparison of signing time between RSA and ECC [65]

Key Length (bits)		Time (secs)	
RSA	ECC	RSA	ECC
1024	163	0.01	0.15
2240	233	0.15	0.34
3072	283	0.21	0.59
7680	409	1.53	0.18
15360	571	9.20	3.07

Table 5.2: Comparison of verification time between RSA and ECC [65]

Key Length (bits)		Time (sec)	
RSA	ECC	RSA	ECC
1024	163	0.01	0.23
2240	233	0.01	0.51
3072	283	0.01	0.86
7680	409	0.01	1.80
15360	571	0.03	4.53

32-byte key length and thus we see that the operational time for signing is around 400 milliseconds in the device logs.

The heartbeat communication happens on a regular basis and in there the device must sign the request which is verified by the gateway. With the limited resources available for the device, ECC becomes more effective when it comes to public key cryptography. It also has a small key size which makes it easier to store in the device. As we have seen in the user verification of the P3 connection model, a similar approach for security is taken in the heartbeat communication as well. The device signs the message with its private key and sends it to the gateway. The gateway stores the public key for the device in its database. It uses the public key to verify the signature and confirm the identity of the device. As we have seen earlier, failure of verification results in a 401 unauthorized access response.

Table 5.3: Comparison of different symmetric encryption technique [2]

Algori thm	File Name	File Size (MB)	Enc. Time (Sec.)	Dec. Time (Sec.)	Total (Sec)
AES	1	0.30	0.047	0.031	0.078
BF	1	0.30	0.020	0.020	0.040
DES	1	0.30	0.062	0.032	0.094
3DES	1	0.30	0.078	0.140	0.218
AES	2	1.38	0.062	0.047	0.109
BF	2	1.38	0.060	0.050	0.110
DES	2	1.38	0.109	0.094	0.203
3DES	2	1.38	0.266	0.480	0.746
AES	3	3.95	0.141	0.140	0.281
BF	3	3.95	0.130	0.110	0.240
DES	3	3.95	0.280	0.250	0.530
3DES	3	3.95	0.671	1.295	1.966
AES	4	7.42	0.218	0.219	0.437
BF	4	7.42	0.220	0.210	0.430
DES	4	7.42	0.483	0.468	0.951
3DES	4	7.42	1.217	2.402	3.619

The symmetric encryption is used between the device and the user. For the symmetric encryption we choose Advance Encryption Standard (AES). The study [2] shows that AES outperforms most of the other symmetric encryption standard. The Table 5.3 shows the comparison between different algorithms for different file size. Using the P3 connection model we are setting up the 32 bytes key between the user and device along with a 16 bytes initialization vector (IV). For the mode of operation, we are using cipher block chaining (CBC). It is the most commonly used mode of operation and can perform better diffusion than other modes like electronic codebook (ECB).

The command execution works particularly on the zero-trust principle. As discussed before, all communications to the device via internet goes through the gateway. However, we didn't want the gateway to know the command that is being executed. This is because the command execution is between the user and the device. The gateway has no part in it except for working as a relay of information. Figure 5.4 shows that

Tree
Item {5}
activity_date String : 2020-05-08 20:14:21
payload Map {2}
command String : CLEkCsTzSPcKDmz8iofIMQ==
device_mac String : CC:50:E3:55:C5:A4
transaction_from String : user:c5114815-434b-4984-8c90-67fe3902c0a1
transaction_id String : 53934472-32b0-4aa8-8635-9413466e49a8
transaction_to String : gateway

Figure 5.4: Encrypted command send from the user to the device via gateway

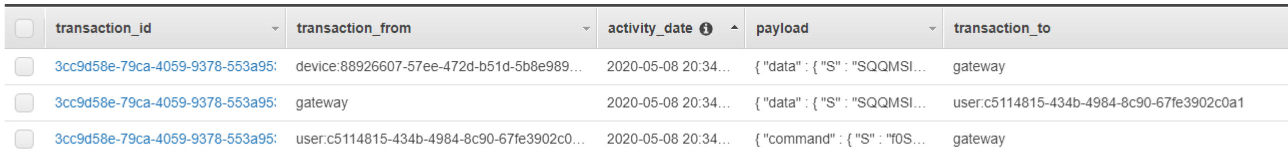
Tree
Item {5}
activity_date String : 2020-05-08 20:14:21
payload Map {2}
data String : /Yz1aRtZj3Az6XtnVPQHg9XST04BWwc+sHv/6bD+BYqpxHpyQQq9vAxzFz1/7BCIC2xeSZI6ArFt2kycl/UZHtuhDESjEFmOJqhpJEZNvM4=
verifier String : jFhA2l1NTzjd7WS6nKhR4CCRz/0AJo/oA/ysht9i2Sw+lQMFWsXsYq8fGIrRikFgB1odMzdLXiz5qZ5waLIBBQ==
transaction_from String : device:88926607-57ee-472d-b51d-5b8e9898a5f6
transaction_id String : 53934472-32b0-4aa8-8635-9413466e49a8
transaction_to String : gateway

Figure 5.5: Encrypted data sent from device to the user via gateway

the data received in the payload is totally encrypted. The gateway only gets the information that it needs, i.e., the device MAC address. From that information the gateway can extract the correct device URL and send the information to the device properly.

Similarly, when the data is returned from the device to the gateway, we see that the payload consist of *data* and the *verifier* as shown in Figure 5.5. We see the Base64 encoded message in both the field. The gateway can use the verifier to verify the identity of the returned message. However, it will not be able to decipher the returned data. This is because the message is again encrypted with the shared key that is known only to the user and device.

Another important aspect of security is auditing. In our model, we maintained a rule, that all information flows through the gateway to the device. This helps provide a single location to audit and log all transactions. The *Transactions* relation plays a



transaction_id	transaction_from	activity_date	payload	transaction_to
3cc9d58e-79ca-4059-9378-553a95:	device:88926607-57ee-472d-b51d-5b8e989...	2020-05-08 20:34...	{ "data" : { "S" : "SQQMSI...	gateway
3cc9d58e-79ca-4059-9378-553a95:	gateway	2020-05-08 20:34...	{ "data" : { "S" : "SQQMSI...	user:c5114815-434b-4984-8c90-67fe3902c0a1
3cc9d58e-79ca-4059-9378-553a95:	user:c5114815-434b-4984-8c90-67fe3902c0...	2020-05-08 20:34...	{ "command" : { "S" : "f0S...	gateway

Figure 5.6: Transaction audit at the gateway

key role in audit management. Figure 5.6 show a snapshot of the table. It records the information about the sender as well as the receiver. It also records the transaction time. Each operation can be uniquely identified using the *transaction_id* and *transaction_from*. Any group of transactions that are related has the same *transaction_id*. This way we can trace and easily perform an audit trail of any operation.

Overall, the framework provides end-to-end protection of data in transit as well as in storage. The P3 connection model helps setup the shared key in a secured manner between the user and device. This key is used to protect the data even when it is going through the gateway. The gateway and device use their own private key to communicate with each other. Since, these keys are preset it provides more flexibility if each have their own set of keys and not have to share with anyone else. The signature ensures integrity and authentication. The confidentiality of data is maintained by enabling TLS/SSL in the gateway APIs. This makes sure that the data is protected in transit.

5.1.1 Physical security

Another aspect of the security triad is availability. To provide a complete security model, we must address all the three aspects of the security triad, i.e., confidentiality, integrity and availability. We have discussed confidentiality and integrity in the previous section. The availability aspect touches briefly in the physical security of the device. In the paper [57], the authors discuss each area of the security triad in terms of IoT. One of the biggest concerns in availability is the Dos/DDoS attack. An IoT device can easily be brought down using a DDoS attack due to the limited resources available

in it.

To test the aspect of a DDoS attack we setup a lab with two Linux machine and attacked the device using a TCP SYN flood. As anticipated, the device was reset by the number of incoming TCP requests within two minutes and 10,000 hits approximately. We continued the attack for 15 minutes to see how the device behaves. For each TCP SYN request the device saved some storage in memory and when the memory was overwhelmed it ended up in a stack overflow error and dumped the memory in the console logs as shown in Figure 5.7. After that the device got reset.

We traced the available memory of the device and Figure 5.8 shows the memory going down during the attack before it got reset.

In the zero-trust framework we took a reactive approach rather than proactive approach for physical security. Using the heartbeat protocol, we can determine the last time a active device has sent the heartbeat pulse. In our experiment, we set the pulse interval to one minute. At the gateway, we ran a **heartbeatChecker**. The responsibility of the checker is to ensure that there is at least one pulse sent from the active devices in every five minutes. If not, it sends an email informing the user that the device is not responding and offline for X minutes as shown in Figure 5.9.

Our solution doesn't provide adequate protection in terms on physical security but provides the information in real time on the health of the device. More research is required to harvest the energy of the devices and ensuring full-time availability [5, 71].

Ensuring device security is of utmost importance. These devices are becoming an integral part of our everyday lives. They are storing and transacting on our personal data. Securing these devices will ensure privacy and protection for our personal information as well. This model effectively provides a secure channel for the devices to communicate with the gateway and the users. It also ensures the proper health of these devices using the heartbeat.

```
17:27:28.897 -> Memory available: 43504
17:27:33.917 -> Memory available: 43504
17:27:38.898 -> Memory available: 32256
17:27:43.914 -> Memory available: 24952
17:27:48.906 -> Memory available: 16080
17:27:53.920 -> Memory available: 16136
17:27:58.926 -> Memory available: 16960
17:28:03.904 -> Memory available: 12184
17:28:08.922 -> Memory available: 12320
17:28:13.936 -> Memory available: 12232
```

```
17:30:13.957 -> Memory available: 6040
17:30:13.992 ->
17:30:13.992 -> Exception (29):
17:30:13.992 -> epc1=0x40212fdf epc2=0x00000000 epc3=0x0000000
17:30:14.094 ->
17:30:14.094 -> >>>stack>>>
17:30:14.094 ->
17:30:14.094 -> ctx: cont
17:30:14.128 -> sp: 3ffffc60 end: 3fffffc0 offset: 01a0
17:30:14.163 -> 3ffffe00:  3ffffe20 00000000 3ffffe20 40207502
```

```
17:30:15.400 -> 3fffff80:  00000000 00000000 00000001 3ffef2
17:30:15.435 -> 3fffff90:  3fffdad0 00000000 3ffef2b0 40204f
17:30:15.503 -> 3fffffa0:  3fffdad0 00000000 3ffef2b0 402144
17:30:15.538 -> 3fffffb0:  feefeffe feefeffe 3ffe8530 401016
17:30:15.571 -> <<<stack<<<
17:30:15.571 ->
17:30:15.571 -> last failed alloc call: 40207AAE(16709)
17:30:15.638 -> ʕʕʕʕ□1ʕ@HʕConnecting to SSID: DG1670A22
17:30:16.319 -> ..
```

```
17:48:45.470 -> Memory available: 14120
17:48:50.491 -> Memory available: 14232
17:48:53.185 -> Initiating heartbeat
17:48:53.185 -> Memory available: 14648
17:48:53.288 -> connection failed
17:48:55.457 -> Memory available: 14400
17:49:00.487 -> Memory available: 14728
17:49:05.475 -> Memory available: 15056
17:49:10.460 -> Memory available: 16376
17:49:15.485 -> Memory available: 20960
```

Figure 5.7: Memory error caused by DDoS attack on device

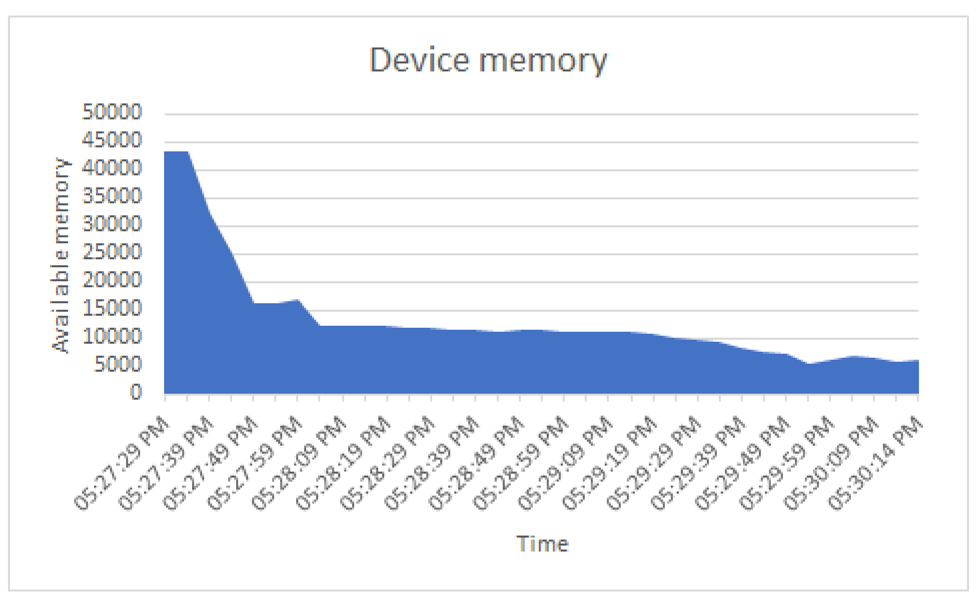
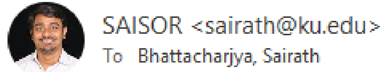


Figure 5.8: Available memory of the device during the attack

[ALERT] Living room sensor not responding



Living room sensor not responding. Please check the device. It is unavailable for 52 min.

Figure 5.9: Alert email send to the user when device is offline

5.2 Memory utilization of device

The gateway consists of lambda functions that are deployed in AWS and have one gigabytes of memory each. Similarly, the mobile phone we are using for building and testing the app is having four gigabytes of RAM. The weakest link in our ecosystem in terms of memory is the device. It has the least memory available for processing. The NodeMcu V3 ESP28266 model we used for building out device has four megabytes of flash memory, 64 KB of instruction RAM and 96 KB of RAM for data. This model represents a real world IoT device with limited resources. So, we had to be very careful in utilizing the resources of the device. We had to account for every byte that we were using of the available space.

For convenience, we printed out the available heap memory on the serial console of Arduino. For that we created a small function called *availableMemory* as shown below:

```
// Function to print the current memory usage
void availableMemory(){
  Serial.print("Memory available: ");
  Serial.println(ESP.getFreeHeap());
}
```

This simple function is called everywhere in the sketch of the device to print the available heap space. We start by analyzing memory usage in the P3 connection model. Like we mentioned above, there is 64 KB of RAM available for instruction. The current source code for the sketch uses 37.18 MB of the available space. With all the global variables defined in the code along with the included headers, at start we had 43.88 KB of available data RAM out of 96 KB. Figure 5.10 shows the memory usage during P3 connection between the user and device.

As we notice, in the first two steps of getting the hello message and sending an encrypted hello reply, the device uses only an extra 200 bytes. In the next section we see a drop of five kilobytes to decrypt the message send by the user containing the

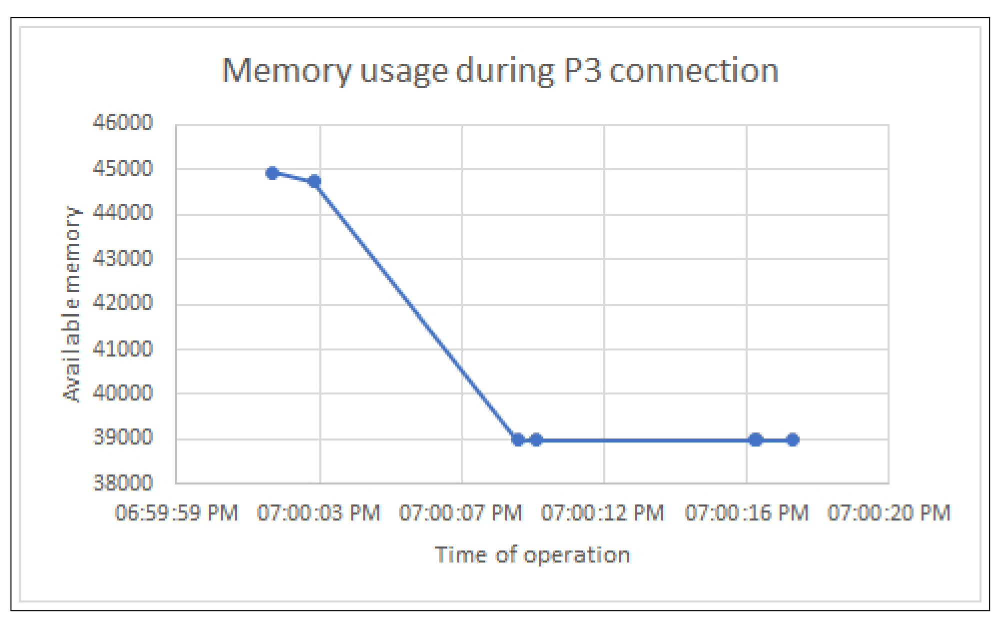


Figure 5.10: Memory usage during P3 connection model

Wi-Fi credentials. Post that we don't see any further change in the memory usage. This process is efficient considering we have 40% of memory still left for other processes.

The heartbeat call was important for the memory utilization. This call happens regularly, and we had to make sure that the memory used by the function gets released properly for the next call. We ran the heartbeat call on a interval of one minute and Figure 5.11 shows the available memory before and after the process. As you see around 43 KB of memory was available at the start of the function and at the end, we see around 37 KB left. The function uses around six kilo bytes of memory for the entire operation. We ran the test for 30 minutes and every time the available memory remained constant. This is a good sign to indicate that there is no memory leakage happening in the function.

We noticed a similar pattern for memory utilization in the command execution mode. We must Note here that the heartbeat is running parallel as the command is getting executed. We noticed that the available memory at the start of each operation is around 42 MB. Each execution cycle took around 2,080 bytes. Figure 5.12 shows the available memory per command execution on the device. We repeated the experiment

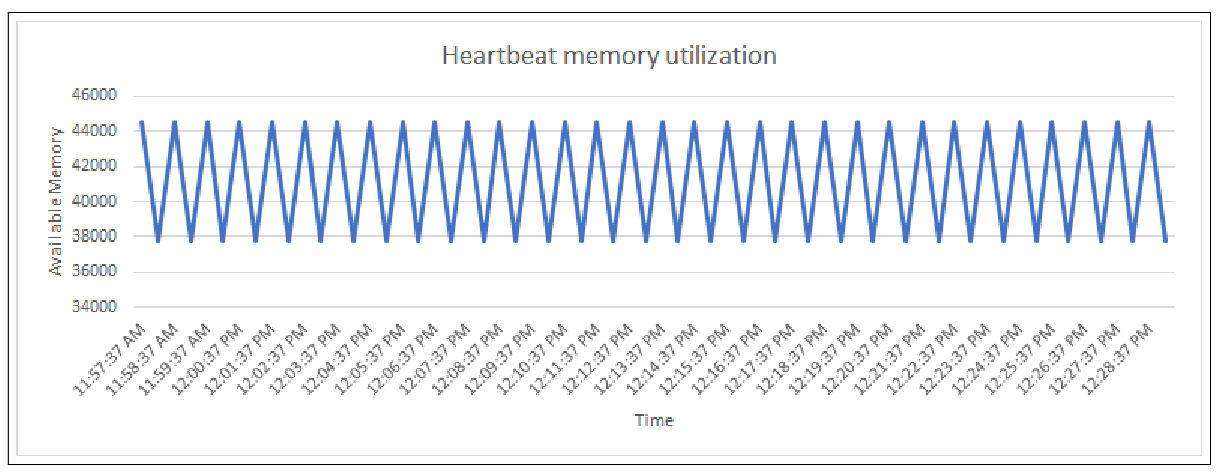


Figure 5.11: Memory utilization of device for heartbeat communication

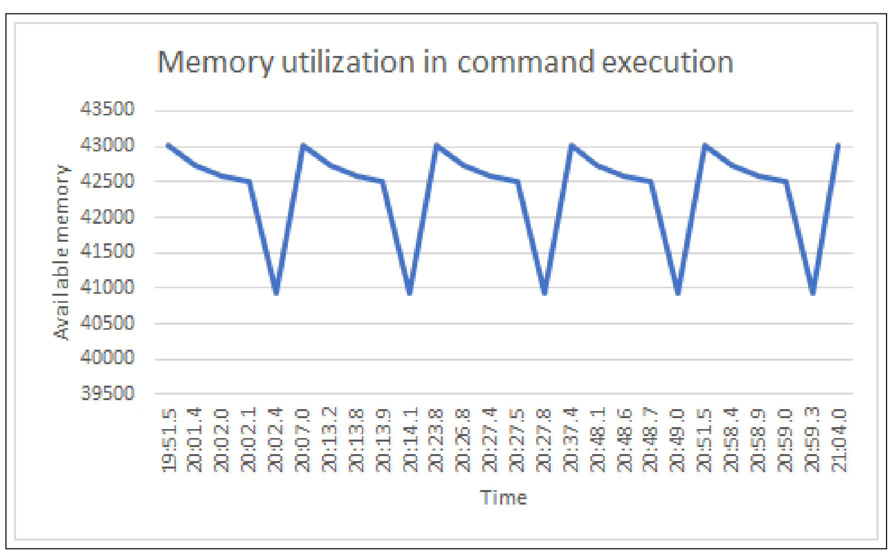


Figure 5.12: Memory utilization of device for command execution

five times and we got the same results.

We noticed a similar pattern for memory utilization in all the three models of the framework. Each of the process at its own cycle utilizes around 5–10% of memory. We kept a threshold of 80% of memory utilization for any operation to determine it as a red flag. However, at the end of each cycle, the available memory is close to 60%. For our tests, we have run one command at a time and seen the above-mentioned results.

Table 5.4: Operational time for each step in P3 connection model

Operation	**Time (msec)**
Pairing and generating the session key	1,210
Connect to Wi-Fi	3,249
User verification	6,527
Device verification	2,032
Generate and share the symmetric key	1,140
Total	**14,158**

5.3 Time to response

We considered time for an operation as an important factor for our analysis. In the framework, for maintaining the security of the device, we decided to pass all request to the device only from the gateway. So, any communication between user and device, post the P3 connection model, would happen via the gateway. This can potentially make the process slow and difficult for the user to use. We decided to run a full test of the time-to-respond for each of the operations that are happening in the framework.

We start by analyzing the time for each of the step of the P3 connection model. One thing to note here is that there are user actions involved in the process and the system is designed to accommodate those. For example, in the P3 connection model, the user app waits for five seconds to get a return hello message from the device. Here we will calculating the actual time the device is taking to encrypt the hello message and sending it back to the user. Table 5.4 shows the time taken for each operations in the P3 connection model.

As we see in the table, it takes around 14 seconds to complete the whole operation. User verification takes the maximum time of six seconds to get the result to the device. We will have to consider the cold start of the lambda functions. Cold start happens when the lambda is called for the first time, when no other instances exist. This is when the lambda is brought in the server memory for processing for the first time. On checking the execution time for the lambda functions, we can see that the lambda for

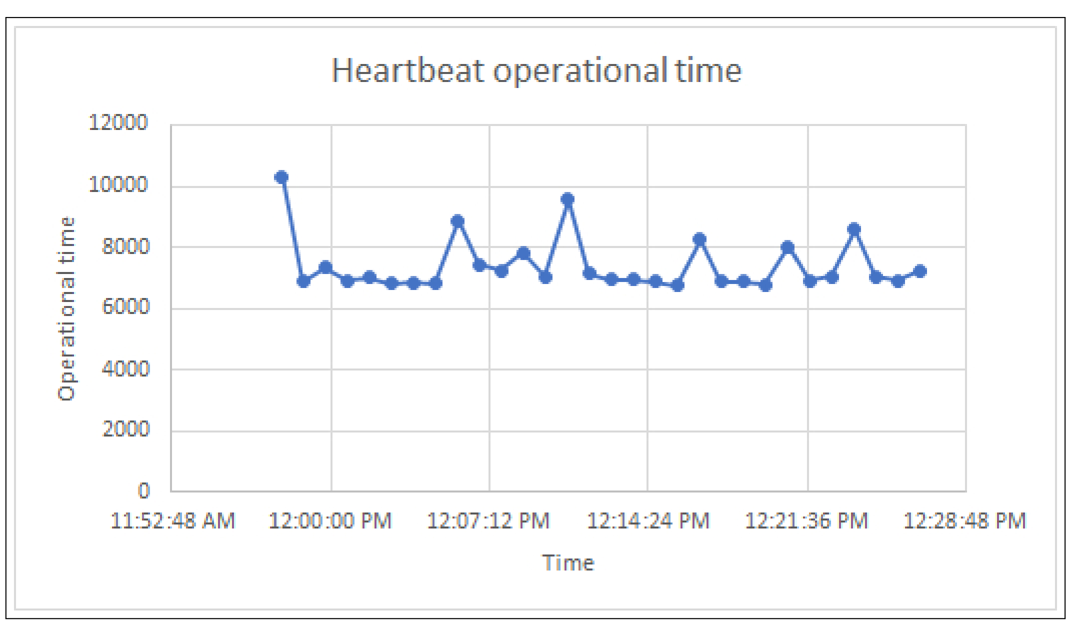


Figure 5.13: Time utilization of device for heartbeat communication

adding a registration record (user verification) took around 170 milliseconds and the one for updating the registration (device verification) took around 193 milliseconds.

The applications allow more time for the responses. For setting up a connection, the app allowed the device to respond in five seconds. As we see in the table, it took around 1.2 seconds to respond. Similarly, for user verification, the user app waits for a period of 60 seconds (one minute) for the complete operation and we see that it completes in less than 10 seconds. The timing is not the most optimized, but within the acceptable range considering the number of operations that are happening and the resource availability of the device.

The cold start is clearly visible in the heartbeat calls as well. In the start we see it takes an operational time of around ten seconds. But then we notice that the time for a heartbeat call takes roughly seven seconds. This line up with the time taken to perform a user verification in the P3 connection model. From the logs itâĂŹs clear that the round trip from the AWS gateway to the device is around 5.5 seconds. We ran the test for 30 minutes and the results can be seen in Figure 5.13.

For executing commands on the device, we saw a drastic decrease in the execution

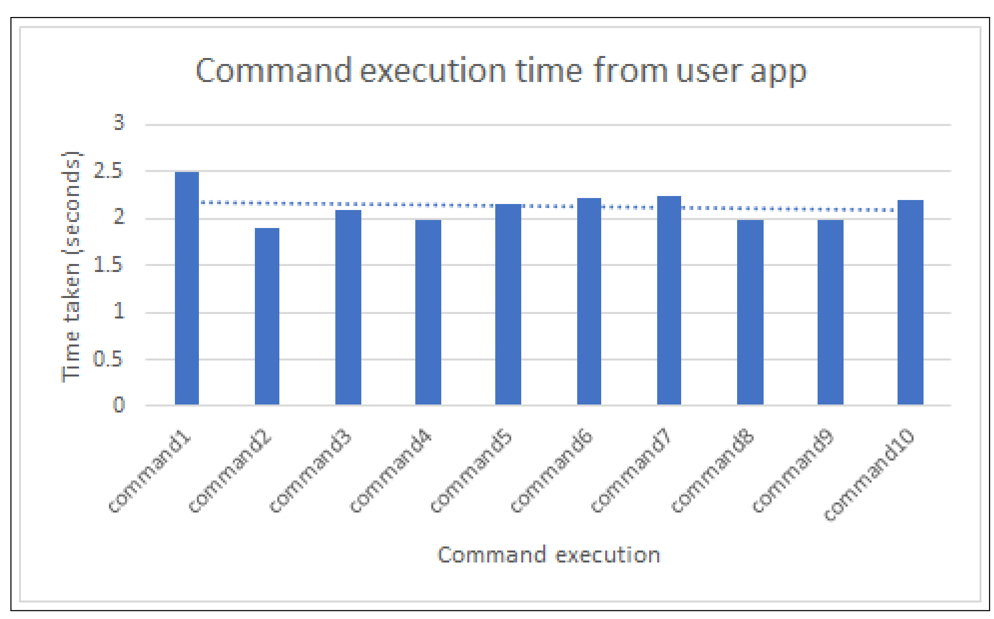


Figure 5.14: Command execution time

time. We used a mix of both changing the temperature type as well as refreshing the data on the app. The device took on an average around 938.8 milliseconds to complete the entire operation. We repeated our experiment for ten iterations and only once we noticed that the time taken for operation on the device went above one second, i.e., 1037 milliseconds. The Figure 5.14 gives the execution time from the user app. We noticed a round trip time (RTT) from the app to the device via gateway was around 2.1271 seconds. In all the ten executions we noticed the same pattern.

In comparison to the heartbeat protocol, we changed from HTTPS to HTTP for passing information from the gateway to the device. Since both our data and verifiers are encrypted using separate keys, we found using SSL/TLS as redundant when calling the service. By avoiding the TLS handshaking, we dramatically improved on the network latency. This brought down the execution time from 7400 milliseconds in heartbeat to 938.8 milliseconds in command execution. We also added a replay attack protection by passing the *millis()* value along with the output data. Millis is the amount of time the device has been up and running. This value constantly increases with every microcontroller clock cycle. In the device we are storing the information of the last counter

Table 5.5: Cost of building the device

Component	**Cost (USD)**
NodeMcu V3 ESP8266	4.75
DHT22 Sensor	3.40
HC-05 bluetooth module	7.99
OLED I2C serial display	6.99
Breadboard and wires	3.50
Total	**26.63**

value. On receiving the new response, we compare the given value to the last stored value to ensure that it is greater. If not, we reject the given result and inform the user that the data is not current.

We decided on a three seconds threshold for the command execution. This is an acceptable value for a web API call. We noticed that even though the RTT went through the gateway and not directly to the device, we were able to maintain the same level of performance. The entire operation didn't take much extra time for completing the RTT. The gateway operated quickly to relay the information to the device and then after verification, sending the device data back to the user app. Overall, in terms of execution time, all the operations of the framework proves that we are able to do the required tasks in a acceptable timeframe, even after maintaining a high level of security.

5.4 Cost of the device

The cost of the device is important when we talk about IoT devices. The cost of the IoT devices vary greatly depending on the utility. A Philips hue light bulb costs around USD 15.00 whereas a iRobot Roomba costs USD 500.00. For our temperature and humidity sensor, we shopped from Amazon and took the components which were recommended by Amazon and had good user reviews. Table 5.5 details the cost of the components used to build the device.

As you see above, the cost of our device is less than USD 30.00. However, we have

not considered the cost of cloud hosting and development efforts. For our thesis we used the free subscription provided by AWS for one year. And since we developed the solution ourselves, it is difficult to quantify the cost associated with it. Considering all possibilities, we can safely say that a device like this with total security implementation should not cost more than USD 50.00. This is just a rough guess on what we think the cost should be.

5.5 Summary

In this chapter we wanted to analyze the effectiveness of framework in terms of security, memory utilization, time of operation and cost to build the device. We verified each of these aspects for the different parts of the zero-trust framework, namely, P3 connection model, heartbeat communication and the command execution model. We also defined thresholds in each sector of our analysis based on the standards provided by the industry. We measured the performance of our framework against those established standards to determine the ease of implementation of this framework in real world.

In terms of security, we saw how both the device and the gateway are equipped with their own ED25519 private and public key pairs. This removed the reliance of a single shared key between them. We used the device's private key to sign the request to get the device verified in the gateway. Similarly, in the command execution, we see the gateway signing the request and sending a verifier to the device for authentication and integrity check. The P3 connection model helped establish a shared key and initialization vector, for AES256 with CBC mode symmetric encryption, between the user and the device in an automated way to secure their future communication. This shared key was utilized to relay the command via the gateway in the command execution model and to send the response back in an encrypted way without the gateway knowing the

information. We also established that the heartbeat protocol ensures physical security of the device in a reactive way. The transaction table maintained an audit trail for investigating communications between the different actors. Passing all communications via the gateway provided a central location to maintain the logs effectively.

Memory utilization yielded very good results for every operation of the framework. We initially kept a threshold of 80% and anything above that was not an acceptable outcome. In all the different operations running in parallel in the device, we noticed the maximum memory usage to go up to 60–65%. We also performed a stress test on the device by conducting a DoS attack with TCP SYN flood. We noticed the memory utilization go down and reset the device after a stack overflow. The heartbeat checker played an effective role to inform the user when the device was offline for more than five minutes.

Similarly, we had a good result when measuring the operational time. For the P3 connection execution we noticed an overall time taken was around 15 seconds. This is excluding the time for the user interacting with the platform. For the heartbeat we saw an execution time of less than seven seconds in an average to send each heartbeat pulse and get a success response form the gateway. We were exited with the outcome of the command execution. As stated above the commands were sent from the user app to the device via the gateway. We realized that it took significantly less time (roughly around two seconds) to respond back with all the cryptographic steps in each stop. We concurred that the delay in the heartbeat was due to the handshaking with TLS in every cycle, which we eliminated in the command execution using HTTP. We maintained the secrecy using the shared key from the P3 connection model and the individual signing and verification key of the device and gateway. Lastly, we compared the cost of the device with other devices in the market. We investigated the cost of each component we used in building the device and calculated the average price of building the device was around USD 30.00. With the implementation and server cost added, we estimated

that the cost of the temperate and humidity sensor should not be more than USD 50.00.

Overall, from the analysis of the zero-trust framework, we conclude that with the proper utilization of memory, we can circumvent the resource limitation of IoT devices and can effectively respond to user queries in an acceptable timeframe after maintaining the proper level of security. This work proves that it is possible to maintain high privacy and security of communication with IoT devices.

Chapter 6

Future of IoT Security

The world has seen many revolutionary changes that have changed the entire fabric of the human society. With every such change, we move to a new normal. *Internet of Things (IoT)* is the next big revolution that is slowly making its way into our society. Things are becoming smarter and more sophisticated with the change. These devices are learning to adopt to the needs and started to communicate within themselves. The lines between physical and virtual worlds are becoming more and more blur.

Credit cards and net banking have already taken over paper money. Now crypto currency and micro transactions are changing the landscape to a globally unified financial exchange system. IoT are playing a major role in performing micro transactions – offering a potential for easier living with technology. For example, a car can buy its own gas when it needs, or a washing machine can buy its own detergent when it runs out. These are not that futuristic anymore.

If the IoT devices are taking over the financial dealings of general populations, it is imperative to think that we need a strong security system to ensure the safety of those devices. In this thesis we demonstrated how the *zero-trust framework* can effectively provide the safety in terms of communication with these IoT devices. We started off by exploring the security weakness of IoT devices in Section 2.3. We claimed that our

solution would effective solve most of the problems. Here we will discuss how our framework addressed them:

- **Insufficient physical security:** The heartbeat communication from the device and the heartbeat checker at the gateway together provided reactive solution to detect malfunctioning of the device and informing the user in real time. We performed a DoS TCP SYN attack to bring down the device for 15 minutes and noticed how an email was sent to the primary user of the device.

- **Limited resources:** With effective use of memory utilization for each operation we proved that we were able to perform every operation with a maximum usage of 60%. We noticed that the memory that was used in a function was effectively getting released by the end of the job. We noticed this in both the heartbeat cycle as well as in command execution.

- **Inadequate authentication:** We used public key encryption for communication between the device and the gateway. Each provided a signature to the other using their private key. That provided authentication as well as an integrity check for the data. For the user and the gateway, we noticed how the JWT token was passed for every request to the gateway for validation and authentication. Since the user never communicated directly with the device and had to go through the gateway, there was no need to authenticating each other.

- **Improper encryption:** With the zero-trust framework, we wanted to maintain total confidentiality of data. An actor gets only the information that it needs to operate. This was effectively shown in the command execution, where the user sent the command to the device via the gateway. However, we didn't want the gateway to know what command was getting executed. The command was encrypted using the shared key generated in the P3 connection model and only known to the user-device pair.

- **Lack of access control:** The architecture clearly defined the roles and access of each actors. Using the P3 connection model, we made sure that a user and device gets to verify the identity of each other before providing access for command execution. We also established using the same model, that a delegate has to get approval from the user to access the device. The gateway, acting as a relay for commands, is not provided access to execute commands. The model defined the boundaries as to what each actor can perform.

- **Backdoor ports:** In this thesis, the only way of accessing the device via internet was using the web server that was established in the device. It only had port 80 open and all other ports were nonfunctional. Restricting it to a single port, we were able to track the kind of requests that were send to the device. Moreover, we required that certain parameters be passed in the request to validate the

authenticity of the sender. Any unauthorized access was responded with a 400 error.

- **Missing audit management capabilities:** In the framework we ensured that all communications flow through the gateway to establish a central hub to log all operations. The *Transactions* table maintained audit logs for each and every operation. For a group of related operations, they were assigned the same *transaction_id*. At any point of time, we can exactly determine what happened on the device.

Few of the key notable things about this framework is that:

- We followed the principle of “never trust, always verifyâĂŹâĂŹ in every transaction in the framework. Every operation was authenticated to determine the identity of the sender. We also included the verifier that helped maintain the message integrity.

- The P3 connection model helped a device and user pair with each other in a seamless way such that we eliminated the need of a default credential or common shared key.

- Removing the need for transport layer security in request from gateway to device during command execution ensured effective memory utilization on the device as well as increased the response time 5 times.

- Routing every request through the gateway enforced single point of verification for the device as well as allowed the gateway to do the heavy lifting of the security requirements on behalf of the device. The gateway recorded all the logs and the device didnâĂŹt require extra resources to store them.

- The framework effectively showed that resource limitations on the device could be avoided by carefully programming and using cryptographic techniques designed for such systems.

The Internet Engineering Task Force (IETF) is yet to come out with a security framework for IoT devices. Many RFC has been produced to circumvent the problems including RFC-8576 [46]. This RFC talks about the state-of-the-art platform for IoT and the challenges related to it. They talk about the same issues that we targeted in this thesis, including, resource limitation, end-to-end security, privacy protection and others. There is no formal guidelines as of now. However, there is an IETF IoT directorate which is an advisory board. The group coordinates the work on IoT and increases visibility of IETF IoT standards to industry alliance, standard development organization (SDOs) and others.

6.1 Future research

In the above we talked about the security issues and how the zero-trust framework effective solves each of those problems. However, we didn't discuss a few topics which fall beyond the scope of this thesis. These topics are presented here.

Improper patch management capabilities. This is one area that we haven't explored in the thesis. Patch management is vital for the operation of the IoT devices. As we have seen in our laptops and phones, with every patch the device is improved in its performance, security and optimizations. It also ensures that existing bugs are fixed in the new releases. There is a lot of research going on to define a patch management strategy [15, 42, 45].

Weak programming practice. This is another area which we didn't explore in our thesis. Weak programming practice causes a lot of security issues including buffer overflow, privilege escalation, backdoor access, to name a few. A proper security testing can be used to catch these issues during development. Unfortunately, due to the quick time-to-market, many organizations provide minimum attention to the security testing.

Proper patch management need to be developed to counter these issues in the deployed devices to counter such existing problems.

Deficient physical security. In our solution, we provided a more reactive approach of informing the user. However, this is not enough to protect the devices. Physical security is a huge research area. Many frameworks are proposed to counter the issue [10,58,69]. Many of these devices, including the sensor that we built, store sensitive data that can be used to jeopardize the user and leak personal information. It is very important to protect these data from falling into the wrong hands.

Blockchaining solution to secure IoT devices. Blockchain is another technique that is being used by the research community to come up with solutions for securing IoT devices. There has been multiple proposals to secure industrial IoT devices and the micro-transactions that occur through them. Similar technology has been proposed to patch IoT devices as well [13, 23, 45] .

IoT is a growing area but it has not reached its full potential yet. There are many areas of security we will learn as it starts getting more and more attention. The usage will reveal the need to a security framework around it. Overall, it is important to come up with a efficient solution to the current issues that we are aware of.

6.2 Need for policy

In many situations we require the cloud to provide services to perform data analytics. For example, in the sensor we build for our thesis, we might want to check how the temperate is changing on a day to day basis and generate a prediction for the near future based on the historic data. In this scenario, the gateway must know the data gathered by the IoT device. There can be other examples where we as users want the manufacturer to have the data to serve us better. One example of this is troubleshoot-

ing the device. Many times we see that the manufacturers want to gather the device data to see what caused a particular failure and so genuinely the crash data can help both the user as well as the manufacturer.

We need to note here that these devices are handling financial transactions as well as dealing with personally identifiable information (PII) of the users. It is important that the country's judicial system steps in to provide some guidance what the manufacturers can and cannot do with the data that they gather from the IoT devise. California is the first state to come up with an IoT law that was brought into effect on January 01, 2020. As per the IoT security law, manufactures of connected devices must equip such devices *"with a reasonable security feature"* that are all of the following:

- Appropriate to the nature and function of the device
- Appropriate to the information it may collect, contain, or transmit
- Designed to protect the device and any information contained therein from unauthorized access, destruction, use, modification, or disclosure

One thing we notice here is that the language of the bill is not very clear. Setting a default password can be considered as a "reasonable security." The law need to provide more guidance and restrictions on the manufacturers to protect the privacy and security of the citizens.

This is a good first step to bring awareness in the legislature about the need for coming up with such bills to protect the interest of the people. However, this law is only applicable to the state of California. The federal government need to implement a similar statute in the country's legislature to have a common ground throughout the country.

The market of IoT devices is growing rapidly. As per the prediction from Gartner [16] by the end of this year there will be 5.8 billion IP connected IoT devices in the market. In the days to come it will be part of our everyday life and we should not

wait for that day to build a security framework. This is the perfect moment to define a solution that is going to shape the future. With all its resource limitations and huge heterogeneous ecosystem, it is possible to build a strong security framework that will verify all requests and not lay a blind trust. This thesis lays the foundation for a "never trust, always verify" principle for the IoT devices. We must go a long way to secure every aspect but from this research we are positive its possible.

Bibliography

[1] React Native 0.61. Getting started. `https://reactnative.dev/docs/getting-started.html`, 2020. accessed March 16, 2020.

[2] N. A. Advani and A. M. Gonsai. Performance analysis of symmetric encryption algorithms for their encryption and decryption time. In *2019 6th International Conference on Computing for Sustainable Global Development (INDIACom)*, pages 359–362, 2019.

[3] A. Albalawi, A. Almrshed, A. Badhib, and S. Alshehri. A survey on authentication techniques for the internet of things. In *2019 International Conference on Computer and Information Sciences (ICCIS)*, pages 1–5, April 2019.

[4] Kishore Angrishi. Turning internet of things(IoT) into internet of vulnerabilities (IoV): IoT botnets. *CoRR*, abs/1702.03681, 2017.

[5] N. Ashraf, A. Hasan, H. K. Qureshi, and M. Lestas. Combined data rate and energy management in harvesting enabled tactile IoT sensing devices. *IEEE Transactions on Industrial Informatics*, 15(5):3006–3015, 2019.

[6] Yahya Atwady and Mohammed Hammoudeh. A survey on authentication techniques for the internet of things. In *Proceedings of the International Conference on Future Networks and Distributed Systems*, ICFNDS '17, New York, NY, USA, 2017. Association for Computing Machinery.

[7] A. Azmoodeh, A. Dehghantanha, and K. R. Choo. Robust malware detection for internet of (battlefield) things devices using deep eigenspace learning. *IEEE Transactions on Sustainable Computing*, 4(1):88–95, Jan 2019.

[8] Fenye Bao and Ing-Ray Chen. Dynamic trust management for internet of things applications. In *Proceedings of the 2012 International Workshop on Self-Aware Internet of Things*, Self-IoT '12, pages 1–6, New York, NY, USA, 2012. Association for Computing Machinery.

[9] Alex Biryukov, Daniel Dinu, and Yann Le Corre. Side-channel attacks meet secure network protocols. In *ACNS*, 2017.

[10] A. Burg, A. Chattopadhyay, and K. Lam. Wireless communication and security issues for cyber-physical systems and the internet-of-things. *Proceedings of the IEEE*, 106(1):38–60, 2018.

[11] Louis Columbus. 2018 roundup of internet of things forecasts and market estimates. `shorturl.at/bzOST`, 2019. accessed January 21, 2020.

[12] A. C. Davies. An overview of Bluetooth wireless technology/sup TM/ and some competing LAN standards. In *ICCSC'02. 1st IEEE International Conference on Circuits and Systems for Communications. Proceedings (IEEE Cat. No.02EX605)*, pages 206–211, June 2002.

[13] S. Dhakal, F. Jaafar, and P. Zavarsky. Private blockchain network for IoT device firmware integrity verification and update. In *2019 IEEE 19th International Symposium on High Assurance Systems Engineering (HASE)*, pages 164–170, Jan 2019.

[14] Fenye Bao and Ing-Ray Chen. Trust management for the internet of things and its application to service composition. In *2012 IEEE International Symposium on a World of Wireless, Mobile and Multimedia Networks (WoWMoM)*, pages 1–6, June 2012.

[15] M. Ge, J. Cho, C. A. Kamhoua, and D. S. Kim. Optimal deployments of defense mechanisms for the internet of things. In *2018 International Workshop on Secure Internet of Things (SIoT)*, pages 8–17, 2018.

[16] Laurence Goasduff. Gartner says 5.8 billion enterprise and automotive IoT endpoints will be in use in 2020. `shorturl.at/zMN29`, 2019. accessed January 21, 2020.

[17] Jonathan Goldberg. 802.15.1-2005 - ieee standard for information technology– local and metropolitan area networks– specific requirements– part 15.1a: Wireless medium access control (MAC) and physical layer (PHY) specifications for wireless personal area networks (WPAN). `https://standards.ieee.org/standard/802_15_1-2005.html`, 2005. accessed February 21, 2020.

[18] C. Gomez and J. Paradells. Wireless home automation networks: A survey of architectures and technologies. *IEEE Communications Magazine*, 48(6):92–101, June 2010.

[19] Ibbad Hafeez, Aaron Yi Ding, Lauri Suomalainen, Alexey Kirichenko, and Sasu Tarkoma. Securebox: Toward safer and smarter IoT networks. In *Proceedings of the 2016 ACM Workshop on Cloud-Assisted Networking*, CAN '16, pages 55–60, New York, NY, USA, 2016. Association for Computing Machinery.

[20] Haolin Wang, Minjun Xi, Jia Liu, and Canfeng Chen. Transmitting IPv6 packets over Bluetooth low energy based on BlueZ. In *2013 15th International Conference on Advanced Communications Technology (ICACT)*, pages 72–77, Jan 2013.

[21] Scott Hilton. Dyn analysis summary of Friday October 21 attack. `https://dyn.com/blog/dyn-analysis-summary-of-friday-october-21-attack/`, 2016. accessed January 28, 2020.

[22] Wen Hu, Hailun Tan, Peter Corke, Wen Chan Shih, and Sanjay Jha. Toward trusted wireless sensor networks. *ACM Trans. Sen. Netw.*, 7(1), August 2010.

[23] J. Huang, L. Kong, G. Chen, M. Wu, X. Liu, and P. Zeng. Towards secure industrial IoT: Blockchain system with credit-based consensus mechanism. *IEEE Transactions on Industrial Informatics*, 15(6):3680–3689, June 2019.

[24] C. Huth, J. Zibuschka, P. Duplys, and T. Guneysu. Securing systems on the internet of things via physical properties of devices and communications. In *2015 Annual IEEE Systems Conference (SysCon) Proceedings*, pages 8–13, April 2015.

[25] M. L. Jones and K. Meurer. Can (and should) hello barbie keep a secret? In *2016 IEEE International Symposium on Ethics in Engineering, Science and Technology (ETHICS)*, pages 1–6, May 2016.

[26] T. Kelley and E. Furey. Getting prepared for the next botnet attack : Detecting algorithmically generated domains in botnet command and control. In *2018 29th Irish Signals and Systems Conference (ISSC)*, pages 1–6, June 2018.

[27] J. Kindervag. No more chewy centers: Introducing the zero trust model of information security. `https://www.ndm.net/firewall/pdf/palo_alto/Forrester-No-More-Chewy-Centers.pdf`, 2010. accessed January 26, 2020.

[28] A. Kolehmainen. Secure firmware updates for IoT: A survey. In *2018 IEEE International Conference on Internet of Things (iThings) and IEEE Green Computing and Communications (GreenCom) and IEEE Cyber, Physical and Social Computing (CPSCom) and IEEE Smart Data (SmartData)*, pages 112–117, July 2018.

[29] C. Konstantinou and M. Maniatakos. Impact of firmware modification attacks on power systems field devices. In *2015 IEEE International Conference on Smart Grid Communications (SmartGridComm)*, pages 283–288, Nov 2015.

[30] Maria Korolov. What is a botnet? when armies of infected IoT devices attack. `https://www.csoonline.com/article/3240364/what-is-a-botnet.html`, 2019. accessed January 28, 2020.

[31] T. Kothmayr, C. Schmitt, W. Hu, M. Brunig, and G. Carle. A DTLS based end-to-end security architecture for the internet of things with two-way authentication. In *37th Annual IEEE Conference on Local Computer Networks - Workshops*, pages 956–963, Oct 2012.

[32] Ayush Kumar and Teng Joon Lim. EDIMA: early detection of IoT malware network activity using machine learning techniques. *CoRR*, abs/1906.09715, 2019.

[33] T. Lackorzynski and S. Koepsell. "hello barbie" - hacker toys in a world of linked devices. In *Broadband Coverage in Germany; 11. ITG-Symposium*, pages 1–7, March 2017.

[34] Xiang Li, Qixu Wang, Xiao Cun Lan, Xingshu Chen, Ning Zhang, and Dajiang Chen. Enhancing cloud-based IoT security through trustworthy cloud service: An integration of security and reputation approach. *IEEE Access*, 7:9368–9383, 2019.

[35] C. Liu, Y. Zhang, Z. Li, J. Zhang, H. Qin, and J. Zeng. Dynamic defense architecture for the security of the internet of things. In *2015 11th International Conference on Computational Intelligence and Security (CIS)*, pages 390–393, Dec 2015.

[36] Paolo Magrassi. Why a universal rfid infrastructure would be a good thing. `https://www.gartner.com/en/documents/356347`, 2002. accessed January 21, 2020.

[37] R. Mahmoud, T. Yousuf, F. Aloul, and I. Zualkernan. Internet of things (IoT) security: Current status, challenges and prospective measures. In *2015 10th International Conference for Internet Technology and Secured Transactions (ICITST)*, pages 336–341, Dec 2015.

[38] Emily McReynolds, Sarah Hubbard, Timothy Lau, Aditya Saraf, Maya Cakmak, and Franziska Roesner. Toys that listen: A study of parents, children, and internet-connected toys. In *Proceedings of the 2017 CHI Conference on Human Factors in Computing Systems*, CHI '17, pages 5197–5207, New York, NY, USA, 2017. Association for Computing Machinery.

[39] Open Mobile Alliance. Device management architecture. `http://openmobilealliance.org/release/DM/V2_0-20160209-A/OMA-AD-DM-V2_0-20160209-A.pdf`, 2005. accessed February 29, 2020.

[40] Blog moderator. Q2 2018 DDOS trends report: 52 percent of attacks employed multiple attack types. `shorturl.at/ezART/`, 2018. accessed January 28, 2020.

[41] Philipp Morgner, Stephan Mattejat, and Zinaida Benenson. All your bulbs are belong to us: Investigating the current state of security in connected lighting systems. *CoRR*, abs/1608.03732, 2016.

[42] I. Mugarza, A. Amurrio, E. Azketa, and E. Jacob. Dynamic software updates to enhance security and privacy in high availability energy management applications in smart cities. *IEEE Access*, 7:42269–42279, 2019.

[43] N. Neshenko, E. Bou-Harb, J. Crichigno, G. Kaddoum, and N. Ghani. Demystifying IoT security: An exhaustive survey on IoT vulnerabilities and a first empirical look on internet-scale IoT exploitations. *IEEE Communications Surveys Tutorials*, 21(3):2702–2733, thirdquarter 2019.

[44] J. Nieminen, C. Gomez, M. Isomaki, T. Savolainen, B. Patil, Z. Shelby, M. Xi, and J. Oller. Networking solutions for connecting Bluetooth low energy enabled machines to the internet of thing. *IEEE Network*, 28(6):83–90, Nov 2014.

[45] M. Novak and P. Skryja. Efficient partial firmware update for IoT devices with lua scripting interface. In *2019 29th International Conference Radioelektronika (RADIOELEKTRONIKA)*, pages 1–4, 2019.

[46] M. Sethi O. Garcia-Morchon, S. Kumar. Internet of things (IoT) security: State of the art and challenges. https://tools.ietf.org/html/rfc8576, 2019. accessed May 12, 2020.

[47] N. Pazos, M. Muller, M. Aeberli, and N. Ouerhani. Connectopen - automatic integration of IoT devices. In *2015 IEEE 2nd World Forum on Internet of Things (WF-IoT)*, pages 640–644, Dec 2015.

[48] R. S. Raji. Smart networks for control. *IEEE Spectrum*, 31(6):49–55, June 1994.

[49] Kristi Rawlinson. Hp study reveals 70 percent of Internet of Things devices vulnerable to attack. https://www8.hp.com/us/en/hp-news/press-release.html?id=1744676, 2014. accessed January 18, 2020.

[50] S. Raza, H. Shafagh, K. Hewage, R. Hummen, and T. Voigt. Lithe: Lightweight secure coap for the internet of things. *IEEE Sensors Journal*, 13(10):3711–3720, Oct 2013.

[51] Thomas Rid and Ben Buchanan. Attributing cyber attacks. *Journal of Strategic Studies*, 38(1-2):4–37, 2015.

[52] E. Ronen and A. Shamir. Extended functionality attacks on IoT devices: The case of smart lights. In *2016 IEEE European Symposium on Security and Privacy (EuroS P)*, pages 3–12, March 2016.

[53] Ethan M. Rudd, Andras Rozsa, Manuel Günther, and Terrance E. Boult. A survey of stealth malware: Attacks, mitigation measures, and steps toward autonomous open world solutions. *CoRR*, abs/1603.06028, 2016.

[54] O. Salman, S. Abdallah, I. H. Elhajj, A. Chehab, and A. Kayssi. Identity-based authentication scheme for the internet of things. In *2016 IEEE Symposium on Computers and Communication (ISCC)*, pages 1109–1111, June 2016.

[55] M. Samaniego and R. Deters. Zero-trust hierarchical management in IoT. In *2018 IEEE International Congress on Internet of Things (ICIOT)*, pages 88–95, July 2018.

[56] J. J. Santanna, R. d. O. Schmidt, D. Tuncer, J. de Vries, L. Z. Granville, and A. Pras. Booter blacklist: Unveiling DDoS-for-hire websites. In *2016 12th International Conference on Network and Service Management (CNSM)*, pages 144–152, Oct 2016.

[57] V. G. Semin, E. R. Khakimullin, A. S. Kabanov, and A. B. Los. Problems of information security technology the “Internet of Things”. In *2017 International Conference “Quality Management, Transport and Information Security, Information Technologies” (IT QM IS)*, pages 110–113, 2017.

[58] V. Sharma, I. You, K. Yim, I. Chen, and J. Cho. BRIoT: Behavior rule specification-based misbehavior detection for IoT-embedded cyber-physical systems. *IEEE Access*, 7:118556–118580, 2019.

[59] V. L. Shivraj, M. A. Rajan, M. Singh, and P. Balamuralidhar. One time password authentication scheme based on elliptic curves for internet of things (IoT). In *2015 5th National Symposium on Information Technology: Towards New Smart World (NSITNSW)*, pages 1–6, Feb 2015.

[60] S. Sicari, A. Rizzardi, L.A. Grieco, and A. Coen-Porisini. Security, privacy and trust in internet of things: The road ahead. *Computer Networks*, 76:146 – 164, 2015.

[61] S. Singh and N. Singh. Internet of things (IoT): Security challenges, business opportunities reference architecture for e-commerce. In *2015 International Conference on Green Computing and Internet of Things (ICGCIoT)*, pages 1577–1581, Oct 2015.

[62] A. Taivalsaari and T. Mikkonen. A taxonomy of IoT client architectures. *IEEE Software*, 35(3):83–88, May 2018.

[63] Hayate Takase, Ryotaro Kobayashi, Masahiko Kato, and Ren Ohmura. A prototype implementation and evaluation of the malware detection mechanism for IoT devices using the processor information. *International Journal of Information Security*, 19(1):71–81, Feb 2020.

[64] A. Tekeoglu and A. Åđ. Tosun. A testbed for security and privacy analysis of IoT devices. In *2016 IEEE 13th International Conference on Mobile Ad Hoc and Sensor Systems (MASS)*, pages 343–348, Oct 2016.

[65] D. Toradmalle, R. Singh, H. Shastri, N. Naik, and V. Panchidi. Prominence of ECDSA over RSA digital signature algorithm. In *2018 2nd International Conference on I-SMAC (IoT in Social, Mobile, Analytics and Cloud) (I-SMAC)I-SMAC (IoT in Social, Mobile, Analytics and Cloud) (I-SMAC), 2018 2nd International Conference on*, pages 253–257, 2018.

[66] W. Trappe, R. Howard, and R. S. Moore. Low-energy security: Limits and opportunities in the internet of things. *IEEE Security Privacy*, 13(1):14–21, Jan 2015.

[67] Eric M. Uslaner. Trust online, trust offline. *Communications of the ACM*, 47(4):28–29, April 2004.

[68] An Wang, Wentao Chang, Songqing Chen, and Aziz Mohaisen. Delving into internet DDoS attacks by botnets: Characterization and analysis. *IEEE/ACM Transactions on Networking*, 26(6):2843–2855, December 2018.

[69] D. Wang, B. Bai, K. Lei, W. Zhao, Y. Yang, and Z. Han. Enhancing information security via physical layer approaches in heterogeneous IoT with multiple access mobile edge computing in smart city. *IEEE Access*, 7:54508–54521, 2019.

[70] M. Wazid, A. K. Das, J. J. P. C. Rodrigues, S. Shetty, and Y. Park. IoMT malware detection approaches: Analysis and research challenges. *IEEE Access*, 7:182459–182476, 2019.

[71] J. Xiong, J. Ren, L. Chen, Z. Yao, M. Lin, D. Wu, and B. Niu. Enhancing privacy and availability for data clustering in intelligent electrical service of IoT. *IEEE Internet of Things Journal*, 6(2):1530–1540, 2019.

[72] Ping Yan and Zheng Yan. A survey on dynamic mobile malware detection. *Software Quality Journal*, 26(3):891–919, Sep 2018.

[73] A. Yohan and N. Lo. An over-the-blockchain firmware update framework for IoT devices. In *2018 IEEE Conference on Dependable and Secure Computing (DSC)*, pages 1–8, Dec 2018.

[74] Zirak Zaheer, Hyunseok Chang, Sarit Mukherjee, and Jacobus Van der Merwe. Eztrust: Network-independent zero-trust perimeterization for microservices. In *Proceedings of the 2019 ACM Symposium on SDN Research*, SOSR '19, pages 49–61, New York, NY, USA, 2019. Association for Computing Machinery.

[75] K. Zandberg, K. Schleiser, F. Acosta, H. Tschofenig, and E. Baccelli. Secure firmware updates for constrained IoT devices using open standards: A reality check. *IEEE Access*, 7:71907–71920, 2019.

[76] Zerynth. NodeMCU v3. `https://docs.zerynth.com/latest/official/board.zerynth.nodemcu3/docs/index.html`, 2020. accessed March 9, 2020.

Appendix A

Arduino program for blink

```
/*
  ESP8266 Blink by Simon Peter
  Blink the blue LED on the ESP-01 module
  This example code is in the public domain

  The blue LED on the ESP-01 module is connected to GPIO1
  (which is also the TXD pin; so we cannot use
  Serial.print() at the same time)

  Note that this sketch uses LED_BUILTIN to find
  the pin with the internal LED
*/
void setup() {
  // Initialize the LED_BUILTIN pin as an output
  pinMode(LED_BUILTIN, OUTPUT);
}

// the loop function runs over and over again forever
void loop() {
  // Turn the LED on (Note that LOW is the voltage level
  digitalWrite(LED_BUILTIN, LOW);
  // but actually the LED is on; this is because
  // it is active low on the ESP-01)

  // Wait for a second
  delay(1000);
  // Turn the LED off by making the voltage HIGH
  digitalWrite(LED_BUILTIN, HIGH);
  // Wait for two seconds (to demonstrate the active low LED)
  delay(2000);
}
```

Appendix B

Create table script

```
import boto3
from settings import appSetting

dynamodb = boto3.resource('dynamodb', region_name=appSetting['region'])

table = dynamodb.create_table(
    TableName='Registrations',
    KeySchema=[
        {
            'AttributeName': 'user_id',
            'KeyType': 'HASH'  # Partition key
        },
        {
            'AttributeName': 'device_mac',
            'KeyType': 'RANGE'  # Sort key
        }
    ],
    AttributeDefinitions=[
        {
            'AttributeName': 'user_id',
            'AttributeType': 'S'
        },
        {
            'AttributeName': 'device_mac',
            'AttributeType': 'S'
        }

    ],
  GlobalSecondaryIndexes=[
        {
            'IndexName': 'registationsByMac',
            'KeySchema': [
                {
                    'AttributeName': 'device_mac',
                    'KeyType': 'HASH'
                }
            ],
            'Projection': {
```

```
                'ProjectionType': 'ALL',
            }
        },
    ],
    BillingMode="PAY_PER_REQUEST"
)

table.meta.client.get_waiter('table_exists').wait(TableName='Registrations')

print("Table status:", table.table_status)
```

Appendix C

Source code of the device

```
#include <ESP8266WiFi.h>
#include <SoftwareSerial.h>
#include <SPI.h>
#include <Wire.h>
#include <Adafruit_GFX.h>
#include <Adafruit_SSD1306.h>
#include <AES.h>
#include <CBC.h>
#include <ArduinoJson.h>
#include <WiFiClientSecure.h>
#include <ESP8266WebServer.h>
#include <ESP8266mDNS.h>
#include <ED25519.h>
#include "DHT.h"
#include "Storage.h"
#include "Display.h"
#include "Device.h"

// global objects used across many functions
SoftwareSerial btSerial(14, 12);
Adafruit_SSD1306 display(128, 64, &Wire, LED_BUILTIN);
DHT dht(2, DHT22);
ESP8266WebServer server(80);
float temperature, humidity, heatIndex;
Display myDisplay(display);
Storage storage;
Device device;
bool isConnected;
byte key[32], iv[16];
Base64Class base64;
CBC<AES256> crypto;
unsigned long refreshCounter = 0, heartbeatCounter = 0;
bool inFahrenheit = true;

// Function to print the current memory usage
void availableMemory(){
  Serial.print("Memory available: ");
  Serial.println(ESP.getFreeHeap());
```

```
39 }
40
41 void connectToWiFi() {
42   isConnected = storage.isConnected();
43   if(isConnected) {
44     WiFiCred cred = storage.getWiFiCredentials();
45     Serial.print("Connecting to SSID: ");
46     Serial.println(cred.ssid);
47     WiFi.begin(cred.ssid, cred.password);
48     myDisplay.message("Connecting to", cred.ssid);
49     // check if able to connect to WiFi
50     int counter = 0;
51     while (WiFi.status() != WL_CONNECTED) {
52       delay(500);
53       counter++;
54       Serial.print(".");
55       if(counter >= 10) {
56         myDisplay.error("Failed to connect to WiFi");
57         isConnected = false;
58         return;
59       }
60     }
61     Serial.println();
62     Serial.println("Connected");
63     Serial.println(WiFi.localIP());
64     refreshData(true);
65     sendHeartBeat();
66   } else {
67     myDisplay.error("No connection info available");
68   }
69 }
70
71 void sendHeartBeat() {
72   unsigned long timeDiff = millis() - heartbeatCounter;
73   // send heartbeat every 10 min
74   if(timeDiff < 60000) {
75     return;
76   }
77   unsigned long startTime = millis();
78   heartbeatCounter = millis();
79   Serial.println("Initiating heartbeat");
80   availableMemory();
81   WiFiClientSecure client;
82   // verify that we are able to connect to the gateway
83   client.setFingerprint(fingerprint);
84   if (!client.connect(host, httpsPort)) {
85     Serial.println("connection failed");
86     return;
87   }
88   String url = "/Prod/sensor/data";
89   uint8_t signature[64];
90   Ed25519::sign(signature, devicePrivKey, devicePubKey, deviceId, strlen(deviceId)
     );
91   int encLen = base64.encodedLength(64);
```

```
92    char verifier[encLen];
93    base64.encode(verifier, (char*)signature, 64);
94    Serial.println("sending payload...");
95    String data = String("{") +
96          "\"device_id\":\"" + deviceId + "\"," +
97          "\"temperature\":" + temperature + "," +
98          "\"humidity\":" + humidity + "," +
99          "\"heatindex\":" + heatIndex + "," +
100         "\"millis\":" + millis() + "," +
101         "\"verifier\":\"" + verifier + "\"" +
102         "}";
103   String payload = String("PUT ") + url + " HTTP/1.1\r\n" +
104              "Host: " + host + "\r\n" +
105              "Cache-Control: no-cache \r\n" +
106              "Content-Type: application/json \r\n" +
107              "Content-Length: " + data.length() + "\r\n" +
108              "Connection: close \r\n\r\n" +
109              data;
110   client.print(payload);
111   // clear the header
112   while (client.connected()) {
113     String line = client.readStringUntil('\n');
114     if (line == "\r") {
115       break;
116     }
117   }
118   String line = client.readStringUntil('\n');
119   Serial.println("...response received");
120   client.stop();
121   if (line.equals("{\"message\": \"SUCCESS\"}")) {
122     Serial.println("Success response from gateway");
123   } else {
124     Serial.println(line);
125   }
126   availableMemory();
127   Serial.print("[Heartbeat] Total time: ");
128   Serial.println(millis() - startTime);
129 }
130
131 void sendUserBT(String message) {
132   // get required variables
133   int msgLen = message.length();
134   int cipherLen = msgLen - (msgLen % 16) + (msgLen % 16 > 0 ? 16 : 0);
135   //form message with padding
136   byte bMsg[cipherLen];
137   for(int i = 0; i < msgLen; i++) {
138     bMsg[i] = message[i];
139   }
140   // add padding
141   for(int i = 0; i < (cipherLen - msgLen); i++) {
142     bMsg[msgLen + i] = 0;
143   }
144   //encrypt
145   byte cipher[cipherLen];
```

```
146   byte plain[cipherLen];
147   crypto.clear();
148   crypto.setKey(key, 32);
149   crypto.setIV(iv, 16);
150   crypto.encrypt(cipher, bMsg, cipherLen);
151   //encode
152   int encLen = base64.encodedLength(cipherLen);
153   char response[encLen];
154   base64.encode(response, (char*)cipher, cipherLen);
155   Serial.println(response);
156   btSerial.write(response);
157 }
158
159 void rcvUserBT(char* plain) {
160   // wait to get data from user
161   while(btSerial.available() <= 0) {}
162   unsigned long startTime = millis();
163   String message =  btSerial.readString();
164   Serial.println(message);
165   int msgLen = message.length();
166   char * msg = const_cast<char*>(message.c_str());
167   int encLen = base64.decodedLength(msg, msgLen);
168   byte bMsg[encLen];
169   // decode
170   base64.decode((char*)bMsg, msg, msgLen);
171   Serial.println("Message decoded");
172   // decrypt
173   crypto.clear();
174   crypto.setKey(key, 32);
175   crypto.setIV(iv, 16);
176   crypto.decrypt((byte*)plain, bMsg, encLen);
177   Serial.println("Message decrypted");
178   Serial.print("[Step 2] Total time: ");
179   Serial.println(millis() - startTime);
180 }
181
182 bool verifyUser(const char* userId) {
183   WiFiClientSecure client;
184   Serial.print("Verifying user ");
185   Serial.println(userId);
186   // verify that we are able to connect to the gateway
187   client.setFingerprint(fingerprint);
188   if (!client.connect(host, httpsPort)) {
189     Serial.println("connection failed");
190     return false;
191   }
192   String url = "/Prod/registration";
193   uint8_t signature[64];
194   Ed25519::sign(signature, devicePrivKey, devicePubKey, deviceId, strlen(deviceId)
    );
195   int encLen = base64.encodedLength(64);
196   char verifier[encLen];
197   base64.encode(verifier, (char*)signature, 64);
198   Serial.println("Generated signature");
```

```
199   String data = String("{") +
200         "\"device_id\":\"" + deviceId + "\"," +
201         "\"user_id\":\"" + userId + "\"," +
202         "\"millis\":\"" + millis() + "\"," +
203         "\"verifier\":\"" + verifier + "\"" +
204         "}";
205   String payload = String("PUT ") + url + " HTTP/1.1\r\n" +
206            "Host: " + host + "\r\n" +
207            "Cache-Control: no-cache \r\n" +
208            "Content-Type: application/json \r\n" +
209            "Content-Length: " + data.length() + "\r\n" +
210            "Connection: close \r\n\r\n" +
211            data;
212   client.print(payload);
213   // clear the header
214   while (client.connected()) {
215     String line = client.readStringUntil('\n');
216     if (line == "\r") {
217       break;
218     }
219   }
220   String line = client.readStringUntil('\n');
221   client.stop();
222   if (line.equals("{\"message\": \"SUCCESS\"}")) {
223     Serial.println("Success response from gateway");
224     return true;
225   } else {
226     Serial.println(line);
227   }
228   return false;
229 }
230 
231 void addUser() {
232   unsigned long startTime = 0;
233   /*STEP1: send encrypted hello message back to the user */
234   startTime = millis();
235   sendUserBT("HELLO");
236   Serial.println("Send confirmation to user. waiting response");
237   availableMemory();
238   Serial.print("[Step 1.2] Total time: ");
239   Serial.println(millis() - startTime);
240 
241   /* STEP2: get user id of the user and WiFi credentials if primary user */
242   startTime = millis();
243   StaticJsonDocument<200> inputJson;
244   char plain[128];
245   rcvUserBT(plain);
246   Serial.println(plain);
247   DeserializationError error = deserializeJson(inputJson, plain);
248   if (error) {
249     Serial.print(F("failed to deserialize input: "));
250     Serial.println(error.c_str());
251     return;
252   }
```

```
253   Serial.println("Message serialized");
254   const char* ssid = inputJson["ssid"];
255   const char* password = inputJson["password"];
256   const char* userId = inputJson["user_id"];
257   availableMemory();
258   Serial.print("[Step 2 FULL]Total time: ");
259   Serial.println(millis() - startTime);
260 
261   /* STEP3: Connect to Wifi with the given credentials */
262   startTime = millis();
263   Serial.print("Connecting to SSID: ");
264   Serial.println(ssid);
265   WiFi.begin(ssid, password);
266   myDisplay.message("Connecting to", const_cast<char*>(ssid));
267   // check if able to connect to WiFi
268   int counter = 0;
269   while (WiFi.status() != WL_CONNECTED) {
270     delay(500);
271     counter++;
272     Serial.print(".");
273     if(counter >= 10) {
274       Serial.println("Failed to connect to WiFi");
275       return;
276     }
277   }
278   Serial.println();
279   Serial.println("Connected");
280   Serial.println(WiFi.localIP());
281   myDisplay.message("ACTION: ", "verifying user");
282   availableMemory();
283   Serial.print("[Step 3] Total time: ");
284   Serial.println(millis() - startTime);
285 
286   /* STEP4: verify user from gateway */
287   startTime = millis();
288   bool isVerified = verifyUser(userId);
289   if(!isVerified) {
290     availableMemory();
291     myDisplay.error("Failed to verify user");
292     btSerial.write("FAILED");
293     return;
294   }
295   availableMemory();
296   Serial.print("[Step 4] Total time: ");
297   Serial.println(millis() - startTime);
298 
299   /* STEP5: Store credentials in EEPROM */
300   startTime = millis();
301   myDisplay.message("ACTION: ", "completing setup");
302   storage.wifiSSID = const_cast<char*>(ssid);
303   storage.wifiPassword = const_cast<char*>(password);
304   storage.userId = const_cast<char*>(userId);
305   storage.userKey = key;
306   storage.userIV = iv;
```

```
307   isConnected = storage.addPrimaryUser();
308   availableMemory();
309   Serial.print("[Step 5] Total time: ");
310   Serial.println(millis() - startTime);
311 
312   /*STEP6: confirm success to user*/
313   startTime = millis();
314   sendUserBT(WiFi.macAddress());
315   myDisplay.message("ACTION: ", "setup complete");
316   delay(1000);
317   availableMemory();
318   Serial.print("[Step 6] Total time: ");
319   Serial.println(millis() - startTime);
320   // cleanup
321   inputJson.clear();
322 }
323 
324 // function to cater to new user connect request
325 void getConnected() {
326   unsigned long startTime = 0;
327   // wait till someone try to connect to the device using bluetooth
328   if (btSerial.available() > 0) {
329     startTime = millis();
330     Serial.println("user connecting");
331     availableMemory();
332     // get the initial encoded connection request
333     String input =  btSerial.readString();
334     const int inputLen = input.length();
335     char dInput [inputLen];
336     device.decode(input, dInput);
337     StaticJsonDocument<200> inputJson;
338     DeserializationError error = deserializeJson(inputJson, dInput);
339     if (error) {
340       Serial.print(F("failed to deserialize input: "));
341       Serial.println(error.c_str());
342       return;
343     }
344     // check what is the connection message type
345     if(strcmp(inputJson["message"],"HELLO") == 0){
346       // get the key and IV
347       hexCharacterStringToBytes(key, inputJson["key"]);
348       hexCharacterStringToBytes(iv, inputJson["iv"]);
349       // initiate adding a user
350       Serial.print("[Step 1.1]Total time: ");
351       Serial.println(millis() - startTime);
352       addUser();
353     } else {
354       Serial.println("Invalid message");
355     }
356     // cleanup
357     inputJson.clear();
358   }
359 }
360 
```

```
361 bool refreshData(bool forCommand) {
362   unsigned long timeDiff = millis() - refreshCounter;
363   if(!forCommand && timeDiff < 5000) {
364     return false;
365   }
366   refreshCounter = millis();
367   temperature = dht.readTemperature(inFahrenheit);
368   humidity = dht.readHumidity();
369   if (isnan(humidity) || isnan(temperature)) {
370     temperature = 0;
371     humidity = 0;
372     heatIndex = 0;
373     myDisplay.error("Failed to read sensor");
374     return false;
375   }
376   heatIndex = dht.computeHeatIndex(temperature, humidity);
377   myDisplay.data(temperature, humidity, heatIndex, inFahrenheit);
378   // availableMemory();
379   // Serial.println("Data refreshed");
380   return true;
381 }
382
383 void handleRoot() {
384   Serial.println("received root...");
385   server.send(200, "text/plain", "Hello Saisor!");
386   Serial.println("...response send");
387 }
388
389 void handleCommand() {
390   unsigned long startTime = millis();
391   Serial.println("received command...");
392   availableMemory();
393   // verify GET method
394   if(server.method() != HTTP_GET){
395     Serial.println("Invalid HTTP method");
396     server.send(400, "application/json", "Invalid request");
397     return;
398   }
399   // verify parameters
400   String cipherCmd = server.arg("data");
401   String verifier = server.arg("verifier");
402   verifier.replace(' ', '+');
403   cipherCmd.replace(' ', '+');
404   if(cipherCmd == "" || verifier == "") {
405     Serial.println("...Missing parameters");
406     server.send(401, "text/plain", "Invalid parameters");
407     return;
408   }
409   Serial.println(cipherCmd);
410   Serial.println(verifier);
411
412   // verify the signature - must be from gateway
413   int sLen = verifier.length();
414   char* encSignature = const_cast<char*>(verifier.c_str());
```

```
  int encLen = base64.decodedLength(encSignature, sLen);
  byte signature[encLen];
  base64.decode((char*)signature, encSignature, sLen);
  bool isVerified = Ed25519::verify(signature, serverPubKey, cipherCmd.c_str(),
    cipherCmd.length());
  if(!isVerified) {
    Serial.println("...Signature varification failed");
    server.send(401, "text/plain", "Invalid verifier");
    return;
  }
  Serial.println("Signature verified successfully");
  availableMemory();

  // decode command
  Serial.println("Decrypting command");
  UserInfo user = storage.getUser();
  int cipherLen = cipherCmd.length();
  char* cipher = const_cast<char*>(cipherCmd.c_str());
  encLen = base64.decodedLength(cipher, cipherLen);
  byte bCmd[encLen];
  char command[6];
  base64.decode((char*)bCmd, cipher, cipherLen);
  Serial.println("Command decoded");
  crypto.clear();
  crypto.setKey(user.key, 32);
  crypto.setIV(user.iv, 16);
  crypto.decrypt((byte*)command, bCmd, encLen);
  command[6] = '\0';
  if(strcmp(command, "TYPE_C") == 0) {
    inFahrenheit = false;
  } else if (strcmp(command, "TYPE_F") == 0) {
    inFahrenheit = true;
  } else {
    Serial.println("...Invalid command");
    server.send(400, "text/plain", "Invalid command");
    return;
  }
  bool isRefreshed = refreshData(true);
  if(!isRefreshed) {
    Serial.println("...Invalid command");
    server.send(400, "text/plain", "Failed to refresh data");
    return;
  }
  Serial.print("Temperature: ");
  Serial.println(temperature);
  String data = String("{") +
        "\"temperature\":" + temperature + "," +
        "\"humidity\":" + humidity + "," +
        "\"heatindex\":" + heatIndex + "," +
        "\"counter\":" + millis() +
    "}";
  availableMemory();

  // encrypt response data
```

```
468   int dataLen = data.length();
469   cipherLen = dataLen - (dataLen % 16) + (dataLen % 16 > 0 ? 16 : 0);
470   //form message with padding
471   byte bData[cipherLen];
472   for(int i = 0; i < dataLen; i++) {
473     bData[i] = data[i];
474   }
475   // add padding
476   for(int i = 0; i < (cipherLen - dataLen); i++) {
477     bData[dataLen + i] = 0;
478   }
479   //encrypt
480   byte cipherData[cipherLen];
481   crypto.clear();
482   crypto.setKey(user.key, 32);
483   crypto.setIV(user.iv, 16);
484   crypto.encrypt(cipherData, bData, cipherLen);
485   //encode
486   encLen = base64.encodedLength(cipherLen);
487   char resData[encLen];
488   base64.encode(resData, (char*)cipherData, cipherLen);
489   resData[encLen] = '\0';
490   Serial.println("Encrypted response");
491   availableMemory();
492 
493   // add signature for verification at gateway
494   uint8_t resSign[64];
495   Ed25519::sign(resSign, devicePrivKey, devicePubKey, resData, strlen(resData));
496   encLen = base64.encodedLength(64);
497   char resVerifier[encLen];
498   base64.encode(resVerifier, (char*)resSign, 64);
499   // send response to gateway
500   String response =  String("{") +
501         "\"data\":\"" + resData + "\"," +
502         "\"verifier\":\"" + resVerifier + "\""
503     "}";
504   server.send(200, "application/json", response);
505   Serial.println("...response send");
506   availableMemory();
507   Serial.print("[command]Total time: ");
508   Serial.println(millis() - startTime);
509 }
510 
511 void handleNotFound() {
512   Serial.println("invalid request received...");
513   String message = "File Not Found\n\n";
514   message += "URI: ";
515   message += server.uri();
516   message += "\nMethod: ";
517   message += (server.method() == HTTP_GET) ? "GET" : "POST";
518   message += "\nArguments: ";
519   message += server.args();
520   message += "\n";
521   for (uint8_t i = 0; i < server.args(); i++) {
```

```
522     message += " " + server.argName(i) + ": " + server.arg(i) + "\n";
523   }
524   server.send(404, "text/plain", message);
525   Serial.println("...response send");
526 }
527 
528 void setup() {
529   Serial.begin(9600);
530   btSerial.begin(9600);
531   dht.begin();
532   myDisplay.setup();
533   dht.begin();
534   connectToWiFi();
535   server.on("/", handleRoot);
536   server.on("/command", handleCommand);
537   server.onNotFound(handleNotFound);
538 
539   server.begin();
540   Serial.println("HTTP server started");
541 }
542 
543 void loop() {
544   if(!isConnected) {
545     getConnected();
546     return;
547   }
548   refreshData(false);
549   sendHeartBeat();
550   server.handleClient();
551   MDNS.update();
552 }
```

Appendix D

Template to deploy resources using SAM

```yaml
AWSTemplateFormatVersion: '2010-09-09'
Transform: AWS::Serverless-2016-10-31
Description: Saisor - Categories

Globals:
  Function:
    MemorySize: 1024
    Timeout: 180
    Runtime: python3.7
    CodeUri: .
  Api:
    Cors:
      AllowMethods: "'OPTIONS,GET,POST,DELETE,PUT'"
      AllowHeaders: "'Authorization,Content-Type,Access-Control-Allow-Origin'"
      AllowOrigin: "'*'"
    Auth:
      Authorizers:
        BasicAuthorizer:
          FunctionArn: arn:aws:lambda:us-east-1:767100225279:function:saisor-authorizer-AuthorizerFunction-GXBWTH890W5W
      DefaultAuthorizer: BasicAuthorizer
      AddDefaultAuthorizerToCorsPreflight: False


Resources:

  GetRegistrationsFunction:
    Type: AWS::Serverless::Function
    Properties:
      Handler: registration/get.lambda_handler
      Role: arn:aws:iam::767100225279:role/dynamodb-service-role
      Events:
        GetRegistrationsApi:
          Type: Api
          Properties:
            Path: /registration
            Method: GET

```

```
38  AddRegistrationsFunction:
39    Type: AWS::Serverless::Function
40    Properties:
41      Handler: registration/put.lambda_handler
42      Role: arn:aws:iam::767100225279:role/dynamodb-service-role
43      Events:
44        AddRegistrationsApi:
45          Type: Api
46          Properties:
47            Path: /registration
48            Method: PUT
49            Auth:
50              Authorizer: NONE
51
52  UpdateRegistrationsFunction:
53    Type: AWS::Serverless::Function
54    Properties:
55      Handler: registration/post.lambda_handler
56      Role: arn:aws:iam::767100225279:role/dynamodb-service-role
57      Events:
58        UpdateRegistrationsApi:
59          Type: Api
60          Properties:
61            Path: /registration
62            Method: POST
63
64  addDeviceDataFunction:
65    Type: AWS::Serverless::Function
66    Properties:
67      Handler: devicedata/put.lambda_handler
68      Role: arn:aws:iam::767100225279:role/dynamodb-service-role
69      Events:
70        addDeviceDataApi:
71          Type: Api
72          Properties:
73            Path: /sensor/data
74            Method: PUT
75            Auth:
76              Authorizer: NONE
77
78  sendCommandFunction:
79    Type: AWS::Serverless::Function
80    Properties:
81      Handler: command/post.lambda_handler
82      Role: arn:aws:iam::767100225279:role/dynamodb-service-role
83      Events:
84        sendCommandApi:
85          Type: Api
86          Properties:
87            Path: /command
88            Method: POST
89
90  addDeviceFunction:
91    Type: AWS::Serverless::Function
```

```
92        Properties:
93          Handler: device/create.lambda_handler
94          Role: arn:aws:iam::767100225279:role/dynamodb-service-role
95
96    addTransactionFunction:
97      Type: AWS::Serverless::Function
98      Properties:
99        Handler: transaction/put.lambda_handler
100       Role: arn:aws:iam::767100225279:role/dynamodb-service-role
101
102   heartbeatCheckFunction:
103     Type: AWS::Serverless::Function
104     Properties:
105       Handler: devicedata/heartbeatcheck.lambda_handler
106       Role: arn:aws:iam::767100225279:role/dynamodb-service-role
```