University of Zurich[UZH]

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

Enforcing Privacy in a Smart Home Environment via Pi-hole Integration

Elliott Wallace
Zurich, Switzerland
Student ID: 11-915-956

Supervisor: Dr. Bruno Rodrigues, Katharina O. E. Müller, Prof. Dr. Burkhard Stiller
Date of Submission: August 9th, 2023

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

ifi

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

Kurzfassung

Das Internet der Dinge (Internet of Things - IoT) hat die Entwicklung von Smart-Home-Technologien ermöglicht und dadurch den Markt revolutioniert. Trotz zahlreicher Vorteile, welche moderne Smart Homes bieten, gibt es Bedenken hinsichtlich ihrer Sicherheit und bezüglich des Datenschutzes. Die zunehmende Komplexität von Smart Homes und die geringe Hardware-Kapazität vieler IoT-Geräte machen es oft erforderlich, dass die von ihnen gesammlten Daten in Cloud-Umgebungen verarbeitet und gespeichert werden. Dies birgt die Gefahr eines möglichen Missbrauchs oder der Offenlegung sensibler Informationen über die Nutzer und Nutzerinnen. Soweit dem Autor bekannt ist, bietet keine existierende Technologie zur Verbesserung der Privatsphäre (Privacy Enhancing Technology - PET) einen schlanken Ansatz zur Sicherung der Privatsphäre in Smart-Home-Umgebungen durch die Zusammenführung vorhandener Tools in einem System. Ziel dieser Arbeit ist es, einen ersten Lösungsansatz in Richtung eines erweiterbaren Open-Source-Softwaresystems zu konzipieren, das leicht in bestehende Smart Homes integriert werden kann. Damit soll die Kommunikation von Smart-Home-Geräten überwacht und ihr Kommunikationsverhalten durch benutzerdefinierte Regeln gesteuert werden. Zu diesem Zweck wird ein Prototyp entwickelt, der die Domain-Name-System-Anfragen (DNS requests) von Smart-Home-Geräten überwacht und gesetzte Regeln über einen DNS-Sinkhole-Mechanismus durchsetzt. Der Prototyp wird auf einer Ein-Chip-System-Plattform installiert und in einer aktiven Smart-Home-Umgebung getestet, um die Funktionalitäten des Protoyps zu prüfen. Auf diese Weise sollen die Leistung, Effektivität und Grenzen des Prototyps untersucht werden, um den allgemeinen Ansatz zu validieren. Die Ergebnisse der durchgeführten Versuche haben ergeben, dass der Protoyp die zu Beginn der Arbeit gesetzten Ziele erfüllt. Der Prototyp überwacht zuverlässig die Netzwerkaktivitäten von Smart-Home-Geräten. Die dabei gesammelten Daten werden aufbereitet, um damit mehr Transparenz für den Nutzer oder die Nutzerin zu schaffen. Darüber hinaus ermöglicht der Prototyp die Defintion von einfachen Regeln, um bestimmte Domänen für ein Smart-Home-Gerät entweder zuzulassen oder zu blockieren. Diese Regeln können anschliessend erfolgreich vom System umgesetzt werden.

Abstract

The Internet of Things (IoT) platform is one of the key drivers of the smart home market, having revolutionized the advancement of smart home technology. Besides the many benefits for convenience and efficiency, there are also concerns about security and privacy in such environments. The increasing complexity of smart homes and hardware limitations of individual devices necessitate the storage and processing of data in remote cloud environments. This raises privacy issues due to potential misuse or disclosure of sensitive information about residents. To the author's knowledge, no existing Privacy Enhancing Technology (PET) offers a lightweight approach to enforce privacy in smart home environments by combining existing tools into a unifying framework. The goal of this thesis is to take a first step towards an extensible open source software system that integrates into the smart home environment with the purpose of monitoring smart home device communications and controlling their communication behavior through user-defined policies. To this end, a prototype application is developed, which monitors smart home devices' Domain Name System (DNS) requests and enforces policies via a DNS sinkhole mechanism. The prototype system is deployed to a system-on-chip platform and evaluated in a live smart home environment to gain insight into the viability of the prototype. The aim is to examine the performance, effectiveness, and limitations of the prototype with the intention of validating the general approach. The results of these experiments indicate that the prototype successfully achieves the goals outlined in this thesis. The application prototype is capable of monitoring the network activity of smart home devices. The collected data are processed to gain insights and make this information transparent to the users. Furthermore, the prototype allows users to define simple allow/block policies which are subsequently enforced by the system.

Acknowledgments

I would like to express my sincere gratitude towards my supervisors Dr. Bruno Rodrigues, Katharina O. E. Müller and Prof. Dr. Burkhard Stiller at the Communication Systems Group of the University of Zurich for granting me the opportunity to write this thesis. Especially, I wish to thank Dr. Bruno Rodrigues for his support, advice, and inspiration in guiding me through the process of this thesis.

To my family and friends, I am thankful for their encouragement, understanding, and unwavering belief in me. Their support has been a constant source of motivation.

Contents

Kurzfassung i

Abstract ii

Acknowledgments iii

1 Introduction 1

1.1 Motivation . . . 2

1.2 Thesis Goals . . . 2

1.3 Methodology . . . 3

1.4 Thesis Outline . . . 4

2 Fundamentals 6

2.1 Internet of Things . . . 6

2.2 Smart Home . . . 7

2.2.1 Smart Home Components . . . 7

2.2.2 Smart Home Security . . . 8

2.2.3 Smart Home Privacy . . . 10

2.3 Domain Name System . . . 11

2.3.1 DNS Sinkholes . . . 12

2.4 Related Work . . . 12

2.4.1 Privacy Enhancing Technologies . . . 12

2.4.2 Tools . . . 15

3 Design **17**

3.1 Application Scenario 17

3.2 Requirements 18

3.2.1 Functional Requirements 19

3.2.2 Non-functional Requirements 19

3.3 Architecture 20

3.3.1 Components 20

4 Implementation **25**

4.1 Core Features 25

4.1.1 Pi-hole Integration 26

4.1.2 Monitoring 27

4.1.3 Notification Service 30

4.2 Development and Build Pipeline 34

5 Evaluation **35**

5.1 Configuration 35

5.1.1 Hardware 35

5.1.2 Software 36

5.1.3 Networking 37

5.2 Scenarios 38

5.2.1 Monitoring 38

5.2.2 Policy Enforcement 38

5.2.3 Weekly Notification 39

5.2.4 Smart Device Operability 40

5.2.5 Performance Evaluation 40

5.3 Results 41

5.3.1 Monitoring 41

5.3.2 Policy Enforcement 43

5.3.3 Weekly Notification 45
5.3.4 Smart Device Operability 47
5.3.5 Performance Evaluation 49
5.4 Discussion 51

6 Final Considerations **54**
6.1 Summary 54
6.2 Conclusions 56
6.3 Contributions 57
6.4 Future Work 58

Bibliography **58**

Abbreviations **64**

List of Figures **65**

List of Tables **66**

List of Listings **67**

A Contents of the Repository **69**

B Code **70**

Chapter 1

Introduction

The global smart home market is expected to experience a significant surge in the coming years, reaching a value of more than USD 330 billion by 2030. This growth is due to the growing number of Internet users, the higher disposable income of people in developing countries, and the increasing demand for energy-efficient and low-carbon solutions [33]. The Internet of Things (IoT) platform is one of the main drivers of the market, having revolutionized the advancement of smart home technology. It enables us to introduce smartness into our homes by fully interconnecting devices, objects, and appliances with the local network and the Internet [25].

Smart homes offer a multitude of benefits, including automation and control through seamless integration of services. They promote energy efficiency by enabling real-time monitoring and intelligent scheduling, which can result in lower utility bills and environmental sustainability. Smart home environments promise an improved quality of life through increased comfort and convenience [42]. Applications of smart home technologies range from ambient lighting and intelligent heating systems to smart security infrastructures, and even smart cities.

In spite of the much-described benefits of smart home systems, there are also concerns about security and privacy in such environments. The growing number of devices that use a multitude of communication protocols and technologies presents a larger surface for potential attackers [29]. There are significant differences in hardware and software quality from different manufacturers. Low production values, often caused by increased market pressure, lead to negligence in security issues [18].

Generally, depending on the complexity of the tasks performed by a smart home environment, more data needs to be collected and more resources are required to process the data. Due to hardware limitations, this data is typically processed and stored remotely in a cloud environment. This leads to users having to surrender control over their personal data and trust the data practices of providers [73]. Hence, privacy becomes a crucial topic in smart homes, as aggregated data can reveal sensitive information about their occupants [41].

1.1 Motivation

Previous research has shown that users are often unaware of the privacy implications of using smart home devices [15, 41]. Users have also suggested that they want more control over their privacy in their smart home environments, but feel that the mechanisms provided by manufacturers for this purpose generally do not meet their expectations [41, 74].

While the emergence of the Internet of Things has undeniably ushered in a new era of the smart home with many advantages, the important issue of privacy protection must be carefully reviewed. Smart home environments are constantly sensing and collecting data. It is essential to ensure that people are aware of and consent to the collection, processing, and sharing of their data to maintain a balance between technological development and the individual's right to privacy. For example, in the cases of Amazon owned Ring doorbell and Echo/Alexa, users of these smart home devices fell victims to privacy violations and data compromises. In 2019, Amazon employees gained access to the cameras of private houses by misusing the Ring video doorbell functionality [69]. Similarly, in the incident involving Amazon Alexa, customers reported that employees and contractors were able to listen to recordings of sensitive and private information [20].

Legally, data collection should be in accordance with the principles of data minimization established by the General Data Protection Regulation (GDPR). The principle states that the processing of personal data must be: "adequate, relevant and limited to what is necessary in relation to the purposes for which they are processed", as stated in Article 5 (1c) of the GDPR [57]. In practice, compliance with these rules is not always easy to verify. The ability to adopt and enforce the GDPR data minimization principle will pave the way toward empowering smart home residents with more control over their personal data and fostering an environment of trust and privacy in the realm of smart home technology.

To the best of the author's knowledge, no existing Privacy Enhancing Technology (PET) provides a lightweight solution to enforce privacy in smart home environments through data minimization by integrating existing tools. Therefore, this thesis addresses this gap by proposing an extensible architecture that builds on existing knowledge and incorporates established systems. These tools are intended to facilitate the monitoring and control of network interactions of smart home devices. Furthermore, existing technologies mainly focus either on privacy preservation or on observability, but not both in conjunction. For this reason, a system that promotes transparency while offering users options for privacy protection is warranted.

1.2 Thesis Goals

Given the challenges described above, this thesis aims to raise awareness of data collection in smart homes and regain control over the process. To this end, the aim is to make the communication behavior of devices in smart home environments more transparent. Equipped with this knowledge, it is envisioned that smart home residents will be able to

make informed decisions about their privacy. As a way to ensure that these preferences can be enforced, the goal is to create simple privacy control mechanisms for users in smart home environments.

To achieve the overarching objective, the following goals are defined for this thesis:

- **Design an architecture** for a system that manages smart home devices to monitor and restrict their traffic with external networks, such as cloud services. For this purpose, the system shall monitor Domain Name System (DNS) requests of the devices, make them transparent to users, and enable them to define simple policies that are enforced via a DNS sinkole. Achieving this goal involves the elicitation of requirements, specifying the system architecture, and designing the planned software components.
- **Implement a prototype** that meets the specified requirements. The implementation should leverage existing tools and build on established frameworks. It is intended that the built application is suitable for execution on a System-on-Chip (SoC) platform.
- **Evaluate the prototype** using a real-world deployment in a live smart home environment. This includes the definition of scenarios to test the functionality of various features of the prototype. Conducting experiments in these use cases will provide qualitative and quantitative results that validate the implementation and provide insight into the effectiveness and limitations of the chosen solution. In particular, the advantages and disadvantages of the application prototype are examined in terms of the balance between privacy and possible limitations on the functionality of smart home devices.

The vision is to take a first step towards an extensible open-source framework that integrates into the smart home environment with the purpose of monitoring devices and controlling their communication behavior through user-defined policies. Therefore, the system should be extensible, allowing it to support different types of policies, additional data sources for device behavior, and more mechanisms to enforce policies.

1.3 Methodology

The methodology applied in this thesis is designed to comprehensively achieve the goals listed in Section 1.2. The approach taken can be divided into two main stages: A literature review and applied research.

Literature review: The initial stage involves conducting a literature review to gather essential knowledge about fundamental concepts and related work in the context of smart home security and privacy. From this research, an overview of related work on privacy enhancing technologies is derived, which is summarized in Table 2.1 in Section 2.4.1. Furthermore, existing solutions and tools are studied to understand their functionalities, strengths, and limitations. This literature review will serve as the basis for the subsequent

phases and ensures that the proposed system builds upon existing knowledge and best practices.

Applied research: The second stage entails the creation and assessment of a prototype to validate the approach outlined in the thesis goals in Section 1.2. The details of the procedure in this phase are described below.

- Design: Based on the previously defined thesis goals and the insights gained from research, this phase features the elicitation and documentation of functional and non-functional requirements (see Section 3.2). A comprehensive system architecture will be designed encompassing the components and key interactions necessary to fulfill the requirements. The deliverables of this phase include the documentation of this design in the form of descriptions and software design artifacts, presented in Section 3.3.
- Implementation: The design specified in the previous phase is implemented, resulting in a prototype of the software system. During this process, implementation details for key features are reported to highlight the approach and the reasoning behind it (see Section 4.1).
- Evaluation: In a final phase, a set of scenarios for the evaluation of the prototype is defined. A description of these use cases can be found in Section 5.2. Following this, the prototype system is deployed in a live smart home environment to test its properties in a real-world setting. A series of experiments are conducted on this platform as outlined in Section 5.2, and the results are presented in Section 5.3.

1.4 Thesis Outline

The contents of this thesis are structured into several chapters. **Chapter 2** lays the foundation for the research by exploring fundamental concepts related to the smart home domain in general, particularly focusing on security and privacy aspects. Moreover, the chapter covers Domain Name Systems (DNS) and DNS sinkholes. Lastly, related work, including privacy enhancing technologies and selected tools, is discussed.

Chapter 3 provides a description of the design of the proposed system. Here, the context is explained by presenting a possible application scenario for the solution. Furthermore, the requirements for the system are specified and divided into the categories functional and non-functional. In addition, the architecture and design of the associated software components and their interaction are introduced.

To provide insight into the technical realization of the prototype, selected implementation details of its core features are presented and discussed in **Chapter 4**. This includes specifics for the integration with Pi-hole, monitoring DNS requests, and sending email notifications. The elaborations in this chapter also give an overview of the technology stack that was utilized to realize the implementation. The chapter further covers the development and build pipeline used in the project.

Chapter 5 addresses the evaluation of the application prototype in a smart home environment. It describes the configuration of hardware, software, and networks used in the process and defines several scenarios to test the performance and functionality of the system. The results of these experiments are presented with the aim of providing insights into the viability of the prototype's core features, effectiveness, and overall system performance. The chapter concludes with a discussion of the results and their implications.

Finally, **Chapter 6** summarizes the results of this thesis, lists the contributions made, and draws conclusions based on the results obtained. It also highlights potential areas for future work, suggesting opportunities for further development and improvement of the proposed system.

Chapter 2

Fundamentals

The following chapter covers the fundamental subjects that provide the background for this thesis. First, a brief introduction to the concept of the Internet of Things is given in Section 2.1. Subsequently, in Section 2.2, smart homes are discussed in detail, including their basic characteristics and components. Furthermore, security and privacy in the context of smart home environments will be explored more closely. In addition, the topic of Domain Name System (DNS) and DNS sinkholes is introduced in Section 2.3. The chapter concludes with an analysis of related work in Section 2.4. It focuses in particular on existing literature on privacy enhancing technologies. In addition, the tools used to realize this work are introduced.

2.1 Internet of Things

The Internet of Things (IoT) paradigm refers to everyday objects equipped with technology and connected via pervasive communication networks to serve as sensors and actuators. The goal is to embed intelligence into our environment by establishing smart connectivity with existing networks and context-aware computation [26].

The result is a system of physical objects that are connected to the Internet and can communicate with each other. These items can be used to observe and interact with their environment through Wireless Sensor Networks (WSN). WSNs use a network of small, low-power sensors to collect environmental data. They can be used to monitor and control physical states, such as temperature, humidity, and pressure. Furthermore, they enable location-based services that can be used for navigation and asset tracking. Wireless sensor networks are a cornerstone of modern IoT applications, which also include smart homes [27].

The Internet of Things has revolutionized the development of smart home technology. By interconnecting everyday objects, it has enabled the automation of numerous tasks in the home. This has led to a variety of smart home technologies, such as voice-controlled assistants, automatic lighting and smart security systems. In addition, the IoT has enabled the development of more efficient energy management systems that allow residents to

monitor and control their energy consumption. Consequently, the emergence of the IoT has had a significant impact on the development of smart home technology [71].

2.2 Smart Home

A smart home is a residence equipped with smart objects or devices interconnected by a communication network. It enables remote access, monitoring, and control for a system that provides smart home services to its residents. The smartness of the home arises through its fully interconnected nature [25]. Remote home control, home automation, and ambient intelligence are services a smart home system provides to its residents. These services can be context-sensitive, automated, or assistive[2]. In [6], the authors identify four key aspects that characterize a smart home: A communications network enabling smart devices to exchange messages, intelligent controls for system management, sensors that collect information, and smart features responding to user input or sensor data.

The primary goal of smart home systems is to improve the quality of life of users in their homes. It is supposed to increase convenience and comfort, provide security and entertainment, and optimize energy use [42].

Smart home systems automate tasks that would otherwise have to be performed manually, such as turning on the lights when a user enters a room or adjusting the temperature in the home to the time of day. They can also improve home security by alerting users to potential hazards. Furthermore, they provide entertainment, for example, by streaming music or movies. By monitoring energy consumption, smart homes can also help improve energy efficiency. Based on monitoring data, lights in unused rooms can be turned off or the temperature of the house can be adjusted according to the weather forecast. In addition, the energy consumption of individual appliances can be monitored, allowing users to identify which appliances consume the most energy and take action to reduce their power usage.

2.2.1 Smart Home Components

In [70], a conceptual framework for IoT-based smart home systems is introduced. The framework encompasses the following levels: smart home, hub, cloud, third party, user interface, and utility.

- Smart home: The smart home level includes a Wireless Sensor Network (WSN), comprising smart devices with modest processing power and wireless communication capabilities.
- Hub: Another element of a smart home is a central hub, a device that stores and processes data locally and connects the WSN to other networks. It is crucial for the interoperability between smart devices as it translates their differing communication protocols. Data generated through sensing by smart devices are forwarded to a hub. After local pre-processing, the hub pushes the data to the cloud.

- Cloud: In this model, the cloud assumes a central role and is responsible for massive data storage and processing.
- Third party: Third parties develop software applications offering services to the end user based on the data provided by the cloud.
- User interface: End users interact with these smart home and third party solutions on the User Interface (UI) level. UIs are responsible for intuitively displaying the accumulated data, providing smart home controls, as well as notifying and giving recommendations to the user.
- Utility: The utility level pertains to the integration of smart home technology into smart power grids. Smart grids, however, will not be discussed in the scope of this thesis.

2.2.2 Smart Home Security

Security has become a growing concern for the IoT field in general, specifically also for smart home environments. Due to the fact that smart home infrastructures rely heavily on the devices communicating with each other as well as with the service providers' cloud infrastructure via the Internet, they are at risk of being compromised. As the number of different protocols, technologies and application scenarios for smart home devices continues to grow, so does the potential attack surface for smart homes [29]. The heterogeneous nature of IoT, smart home devices, and technologies further weakens the security of these systems. Other factors increasing security risks in smart homes include the short time to market for IoT devices and cost reductions, which are valued higher by manufacturers than protection of their hardware and software [18]. Further, privacy is one of the most important aspects to consider in the home environment. In the following paragraphs, we define security and privacy in the context of smart homes. We discuss vulnerabilities and threats to smart home security and some of the attack vectors they enable.

A well-known model in the field of information security is the CIA triad of confidentiality, integrity, and availability [18, 39]. It describes the goals for the security of a system and serves as a guideline for developing and implementing information security strategies.

- **Confidentiality** refers to the principle that data can only be accessed by authorized entities. Encryption and access control mechanisms are possible ways to achieve data confidentiality.
- **Integrity** ensures that the data is not modified without authorization to maintain an accurate and consistent state of the data. This attribute is especially relevant in view of possible interference during data transit.
- **Availability** guarantees that authorized entities can access any information resources at any given time, even if the network is under attack.

In addition to the CIA triad, authenticity, authorization, and non-repudiation have been identified as important security objectives for smart home environments [39].

- **Authenticity** refers to proving identity and validating this information by the respective communication partner. Mutual authentication prevents the spoofing of false roles.

- **Authorization** ensures that access is controllable for any entity that is part of the system and that those rights can be managed accordingly.

- **Non-repudiation** ensures that a claim is provable and prevents any entity from denying that it has performed a specific action.

Based on the CIA categories, the authors in [30] define security requirements for smart home services: For confidentiality, they list encryption mechanisms as well as secure key and identity management. Further, responsible password practices are suggested. Mutual authentication between devices and reliable communication are important to maintain integrity. Unauthorized user or device access and forging encryption keys should be prohibited. To ensure the availability of the resources, it should be possible to set up device-specific security policies and to receive regular security updates for software components. Additionally, they suggest monitoring for abnormal device behavior, blocking unnecessary remote access and providing external attack detection capabilities.

Most data transmissions in IoT use cases, such as smart homes, are carried out by WSNs. Hence, they are naturally vulnerable to many attacks targeting wireless networks [70]. In [47], IoT vulnerabilities are classified by their attack vectors targeting different layers and analyzed in terms of the security goals they threaten. Inadequate authentication and improper encryption at the network layer are identified as threats to the confidentiality and integrity of IoT resources. However, limitations in computational capabilities and energy supply of devices, on the other hand, are determined to hinder the implementation of sufficiently sophisticated authentication and encryption algorithms. This enables attackers to reveal sensitive information or take control of the system. Common problems highlighted as a representation of these vulnerabilities are the unsecured sharing of symmetric keys for authentication and the unencrypted transmission of WiFi passwords when setting up devices. IoT devices with unnecessary open ports are another network-based vulnerability listed to threaten availability, providing an attack surface for denial-of-service (DoS) attacks.

The authors in [47] further describe software-based vulnerabilities such as insufficient access control and weak programming practices that threaten both confidentiality and integrity. In conjunction with the fact that most users have elevated permissions, inadequate access control also refers to the problem of weak credential management, such as using default passwords. This may lead to unauthorized third parties gaining access. Furthermore, deficiencies in the firmware of IoT devices are commonly found due to programming that includes known vulnerabilities.

These vulnerabilities expose an IoT-based smart home to a variety of threats. In [19], a threat model examines the possible risks of active or passive adversaries directed at the smart home's infrastructure or the data stored in the associated cloud services. The identified threats are described below:

- **Eavesdropping:** An adversary may capture network traffic between the different components of the smart home infrastructure. The information acquired during this process can extract knowledge about the user's behavior passively or learn details about the communication mechanisms that can later be used in an active attack.
- **Impersonation:** To access smart home devices, an adversary can try to impersonate a legitimate user, which enables them to control the devices or extract confidential information from them. The adversary may use information like unique identifiers of devices and Internet Protocol (IP) addresses from previous eavesdropping attacks to successfully impersonate an authorized user.
- **DoS:** Denial of Service (DoS) or Distributed Denial of Service (DDoS) attacks can target either the hubs in a smart home or individual devices. As most IoT devices are based on hardware with low processing capabilities, they are especially vulnerable to DoS.
- **Software exploitation:** Smart home IoT devices can be infected with malware to gain access to information and influence operations. One possible scenario is the creation of a botnet by infecting many IoT devices with the goal of launching large-scale DDoS attacks. Other approaches may exploit vulnerabilities in applications on an authorized user's device that are used to interact with the smart home system.

2.2.3 Smart Home Privacy

Smart homes offer various services to increase their inhabitants' convenience. Providing such functionalities often requires collecting various types and amounts of information. Smart home equipment can amass extensive data concerning its users, encompassing details regarding their activities and surroundings. An instance of this would be a smart temperature regulator that can gather data about when users are present, how they manipulate the temperature settings, and when they do so. Similarly, a smart door lock can accumulate data about the identities of individuals entering or leaving the property, at what times, and for what duration. This data may serve diverse objectives such as enhancing energy efficiency or fortifying home protection. Due to hardware limitations, this data is typically processed and stored remotely. In most cases, this happens in a cloud environment run by the device manufacturer, where user data may be outside of their control. Hence, they have to rely on the data practices of device manufacturers and third parties instead [73]. Consumers are forced to weigh their privacy needs against the utility and comfort of smart device services and accept the trade-offs [74]. Privacy poses a complex topic in IoT use cases in general. In the specific case of smart home environments, the gathered information concerns the domestic intimacy of users' personal lives. The resulting privacy implications become important as the aggregated data can reveal sensitive insights about smart home residents [41].

Consumers are often unaware of the privacy implications and risks caused by their installation and use of smart home technology. Comprehension thereof is not effectively promoted by displaying lengthy privacy policies that are difficult to understand. Furthermore, buyers cannot monitor the data practices of smart device producers and service

providers [15, 41]. Hence, the perspective users have to trust the intentions of the companies whose products they are buying, concerning their data. To warrant this trust, it should be a goal to establish well-communicable transparency about the types of data collected, data retention time, who the information is shared with, and how securely it is stored [41].

Studies [41, 74] suggest that even though smart home devices, like smart speakers with integrated voice assistants, come with some privacy indication and control features, they are not aligned with the needs or capabilities of their users. However, they still wish for enhanced transparency and control over their data collection and sharing.

An important requirement to preserve privacy in an IoT environment like a smart home is the residents' consent regarding which data is gathered and stored and how it is processed and shared with third parties. In a fully integrated smart home, sensing and collecting data is a ubiquitous process that spans many areas and operates at various levels. For people living in such environments, expressing their consent in a differentiated manner so the data capturing fits their wishes for privacy without proper awareness and control mechanisms may not be a trivial task. Consequently, the definition of access rules to protect residents' privacy is difficult to achieve [1]. This suggests the necessity of new approaches to raise awareness about data gathering and dissemination connected to using smart devices. Furthermore, implementing easy-to-use privacy control mechanisms to leverage this awareness is proposed.

Collecting and using personal data raises a range of ethical questions that are important to consider concerning smart homes. For example, is it ethical to use data from smart homes to target ads to inhabitants, or to sell the data to third parties [46]? Do users have the right to opt out of data collection in general, or should they be required to share their data as a condition of using smart home devices?

2.3 Domain Name System

The Domain Name System (DNS) is a crucial component of the Internet infrastructure. It serves as a name service that translates human-readable domain names into IP addresses, which are used for accessing online resources. The domain name space is structured as a variable-depth tree, with nodes representing domains, each having a label. The node's domain name is constructed by concatenating its label with its parent domain name, separated by a dot [45].

Essentially, the Internet's DNS acts as a distributed database. The database nodes are hosted on DNS name servers that store zone files containing records related to a specific domain or DNS zone. When a client requests a domain name, the request is forwarded to the appropriate DNS server by the DNS resolver. The DNS server then searches its zone files for the requested domain's information [45].

The DNS's structure follows the hierarchy given by the domain name space: Root name servers are at the tree's top and top-level domain (TLD) servers contain information about the different TLDs. Authoritative DNS servers contain data about specific domain names.

When a user enters a domain name into their browser, it sends a DNS query to the local DNS resolver. If the resolver has the IP address of the domain name in its cache, it returns it. Otherwise, it forwards the query iteratively or recursively to other DNS servers, beginning with the root servers, until it finds the IP address on the authoritative name server associated with the domain name. This entire process is known as a DNS lookup [45].

2.3.1 DNS Sinkholes

Sinkholing is a technique to intercept outgoing DNS queries requesting undesired domains to redirect traffic to a different IP address [9, 50]. Various lists of such malicious or unwanted domains are publicly available and maintained by communities of security professionals. A DNS sinkhole essentially takes the role of a DNS server that receives DNS queries from clients and, in the first step, checks the configured instance of such a blacklist against the requested domain. If the domain is not listed on the blacklist, the DNS request is forwarded to an upstream DNS server to resolve the legitimate query. Otherwise, if the requested domain is blacklisted, the DNS sinkhole does not forward the request and instead may respond with a different privately controlled IP address [7].

2.4 Related Work

2.4.1 Privacy Enhancing Technologies

The authors in [76] survey different privacy-enhancing technology (PET) proposals for hub-based smart homes. Proposed systems are categorized on the basis of their approach to implementing privacy preservation mechanisms. The following categories have been identified:

- Network traffic obfuscation
- Local data processing
- Device activity control and patching
- Cross-cloud data flows control
- App activity control
- Data minimization
- Data obfuscation

In addition to categorizing them according to their privacy preservation mechanisms, the paper also differentiates between different system models that the PETs are based on, i.e., hub-centric, hub-only, cloud-centric, or hybrid. Furthermore, the stakeholders of the smart home system that are considered potentially dangerous in the threat model of the respective design are identified. Lastly, the locations in the smart home system model, where the different security mechanisms of the systems operate, are determined [76]. Utilizing this analytical framework, the solution proposed in this thesis can be described as a hub-centric system that fits in the data minimization category. The stakeholders considered by our threat model are device and application providers. Regarding the generic system model described in [76], the solution proposed in this thesis implements security and privacy mechanisms at the home gateway security checkpoint.

Based on these observations, we take a closer look at those research proposals similar to the work presented in this paper and compare them based on their distinctive attributes.

Privacy-enhancing technologies that deal with app activity control try to minimize the privacy and security threats from third-party applications that run within a smart home or other IoT platforms [76]. HomePad [75] is a smart home hub system that contains a platform for third-party apps. It requires developers to build isolated modules and explicitly define the data flows between them. Users can specify expressive privacy preferences that the runtime of HomePad checks against the apps' specifications. Only functions that comply with the user's policy are executed.

Network traffic obfuscation techniques aim at reducing the possibility of an adversary inferring privacy sensitive information from observed network traffic, such as the daily routines of a smart home inhabitant [76]. As such, inference attacks can succeed even with encrypted communication through pattern detection [5]. Using stochastic traffic padding to reshape the traffic generated by devices, this is no longer easily possible.

Other approaches use data obfuscation to anonymize user data before propagating outside the home network. Systems that use such a process need access to the raw sensor data generated by devices to alter data that can individually identify a user [76]. The authors of [38] describe a privacy agent that uses a neural network to learn a user's privacy policy and enforce it directly on the respective devices. In contrast to the other technologies described here, this approach is not implemented at the gateway or hub, but is directly deployed onto the device. The agent runs with the embedded software in a hypervisor emulating the device's hardware interfaces. The agent can either let data pass through or modify it and drop or inject traffic to achieve anonymity.

Data minimization techniques aim to reduce the risk of leakage of privacy-sensitive data by only disclosing as little data as possible to service providers, while still maintaining all relevant functionalities of the system [76]. Peekaboo [34] is a proposal for a privacy-sensitive smart home architecture. The authors argue that existing smart home apps collect more data than necessary. Therefore, peekaboo offers a framework for application developers to transparently collect user data on a need-to-know basis by explicitly specifying the collection behaviors in a manifest. The peekaboo runtime uses the manifest to build a preprocessing pipeline, manipulating the data so that it only discloses the agreed upon information. PFirewall [8] is another data minimization technology that focuses on home automation. It uses a virtualization approach mediating the communication between IoT

Name	Category	Objectives	Method	Environment	Platform
PFirewall [8]	Data minimization	Minimize data sent to platform. Enable users to define custom policies.	Mediator between devices and platform. Filters raw IoT data, forwarding only policy compliant data.	Cloud or hub-based smart home.	Local hub
HomePad [75]	App activity control	Users control how apps access and process data collected by smart devices.	Apps modeled in a prescribed way. Only execute if compliant with users' settings.	Dedicated hub-centric smart home	Local hub
Klingensmith et al. [38]	Data obfuscation	Learning user's global privacy policy and enforce it on all devices.	AI agent disables or modifies data streams directly at device level.	General IoT	Device hardware
IoTrim [44]	Data minimization	Only sending data that is required by primary device functionalities.	Automatically classify destinations as required or not and create firewall rules for the latter.	Cloud or hub-based smart home	IP router
Apthorpe et al. [5]	Network traffc obfuscation	Making it harder to infer genuine user interactions by analyzing traffic patterns.	Stochastic traffic padding. Reshapes traffic during user activity, injects dummy traffic during down time.	Cloud-centric IoT/Smart home system	Local hub (WiFi access point)
IoT Inspector [32]	Traffic analysis	Gather network traffic data from smart home devices. Making traffic observable in real-time.	Intercept traffic between devices and router via ARP spoofing. Process data and visualize in UI.	Cloud or hub-based smart home	Local Unix host. Central server.
Klement et al. [36]	Traffic analysis	Gathers network traffic of IoT devices and associated mobile applications.	ARP Spoofing. MITM-proxy used to capture encrypted mobile app traffic.	Cloud or hub-based smart home	Local hub
Peekaboo [34]	Data minimization	Transparent data collection. Centralized privacy features across apps and devices.	Developers specify data collection behaviors in manifest. Local preprocessing of data.	Dedicated hub-centric smart home	Local hub
Kazlouski et al. [35]	Data minimization, App activity control	Block unnecessary third party connections of Fitbit associated apps.	Block unwanted domains through publicly available blocklists.	Fitbit ecosystem	Smart phone

Table 2.1: Related work

devices and the smart home platform, acting as a hub. PFirewall builds its own internal model of the automation rules in a smart home. It uses this model to check if the data received from devices should trigger a configured automation. The request is forwarded only if the condition is met. The last privacy-enhancing technology we discuss in this category is IoTrim [44]. The prototype consists of two components: IoTrigger and IoTrimmer. IoTrigger uses experimentation to classify network traffic destinations into required and non-required destinations for the tested smart device functionality. This results in a block list that is then used by the IoTrimmer component to install firewall rules on the router. Besides the other hub-based technologies, IoTrim runs natively on the router, making it not a trivial installation for the average user. The approaches described above effectively enforce the data minimization principle stated in the General Data Protection Regulation (GDPR) [57].

Not exactly fitting into the definition of PETs are [32, 36], as the software systems described in the papers does not actively intervene in smart home communications. The objective of these approaches is to allow its users to observe the network traffic generated in their smart homes, which, in turn, can help them make informed decisions regarding their privacy. Both presented systems use Address Resolution Protocol (ARP) spoofing to intercept traffic between devices and the home router. A notable difference between the two approaches is that IoT Inspector [32] uses a central cloud server for data aggregation and storage. In contrast, the proposal in [36] offers a local-only option.

The system proposed in this thesis stands out as a lightweight PET, specifically designed to run on system-on-chip (SoC) environments, such as the Raspberry Pi. This system is based on the integration of existing and proven tools to achieve data minimization. In contrast to related technologies that require complex virtualization or custom runtimes, the prototype developed here uses a simple DNS-based monitoring and policy enforcement approach. By operating directly at the network level, this approach avoids the need for direct modifications to devices or the smart home platform.

2.4.2 Tools

For the creation and testing of a prototype in the context of this thesis, various tools are used. Some of the main examples are listed below:

- **Home Assistant** [31] is an open source home automation platform that allows users to control smart home devices and create custom automations. It integrates with various services and devices, such as Amazon Alexa, Google Assistant, Philips Hue, and Sonos. Home Assistant provides a web interface and a mobile app for users to monitor and manage their smart home devices. Users can also create automation and scripts to customize their smart home scenarios.

- **Pi-hole** [62] is a network-level ad blocker that runs on a Raspberry Pi. It acts as a DNS sinkhole that blocks unwanted domains for ads, trackers, malware, and phishing sites. Pi-hole also provides a web interface and an Application Programming Interface (API) for users to configure and query their Pi-hole settings and statistics.

- **Raspberry Pi** is a series of small, low-cost computers that can be used for various purposes. They are based on a single-board design that contains a processor, memory, input/output ports, and other components. As their operating system, they run Raspberry Pi OS, an official operating system based on Debian Linux, optimized for Raspberry Pi hardware. The Pi supports various programming languages, such as Python and C [12].

- **Docker** technology allows developers to create, deploy, and run applications using containers. Containers are isolated environments that contain all the dependencies and configurations necessary for an application to run. This makes it easier to ensure consistency and portability across different platforms and environments. Docker also provides tools to manage and orchestrate containers, such as Docker Compose and Docker Hub [10].

Chapter 3

Design

This thesis focuses on designing and implementing a prototype to enhance user privacy by offering insights into the interactions of smart home devices with the Internet and providing users with simple tools to give them more control over their data. The intention is to address some of the privacy limitations in smart homes described in Section 2.2.3. To this end, the capabilities of the existing tools listed in Section 2.4.2 should be leveraged by using or integrating them into the prototype design. The prototype must capture the network behavior of smart home devices by monitoring DNS requests and enforce simple policies through a DNS sinkhole mechanism. The interaction with the system will be facilitated by a user-friendly interface. It provides observability through reporting and data visualization and enables the user to manage devices and configure policies.

The purpose of this chapter is to provide a comprehensive description of the design of the proposed system. Section 3.1 presents a possible application scenario, providing context for the work done in this thesis. Furthermore, the functional and non-functional requirements for the prototype system are defined in Section 3.2.1. Lastly, the system architecture and details of the software design are presented in Section 3.3.

3.1 Application Scenario

The increasing number of smart devices in a home environment prompts demands for improved device security and data privacy. Smart devices often rely on cloud-based services to provide various automation and remote services at the expense of relying on collected personal data. To provide some context and exemplify the environment in which the proposed system will be embedded, an application scenario for the prototype is described below. The suggested scenario is illustrated in Figure 3.1, organized into five abstraction layers. The first abstraction layer concerns several smart devices, including but not limited to smart light bulbs, TVs, IP cameras, thermostats, and fans/air filters. At the bridge layer, devices may connect directly to a router in a cabled or wireless connection, or via a bridge that connects to the router. The automation layer often relies on integration APIs offered by bridges to automate two or more devices in pre-configured scenarios by

the user, such as “entering or leaving home”, or based on predefined time schedules. Major home automation services, such as Apple HomeKit [4], Amazon Alexa [3], Google Home [24], and Home Assistant (open source) [31] offer interfaces for managing smart devices and often rely on a cloud service. The exception is Home Assistant [31], a community-driven project that is deployed locally and treats remote control through the cloud as optional. The proposed system is intended to operate as a network layer service that offers visibility into device network usage and the exchange of personal data. The service layer features management interfaces that can be accessed remotely over the Internet or locally via APIs.

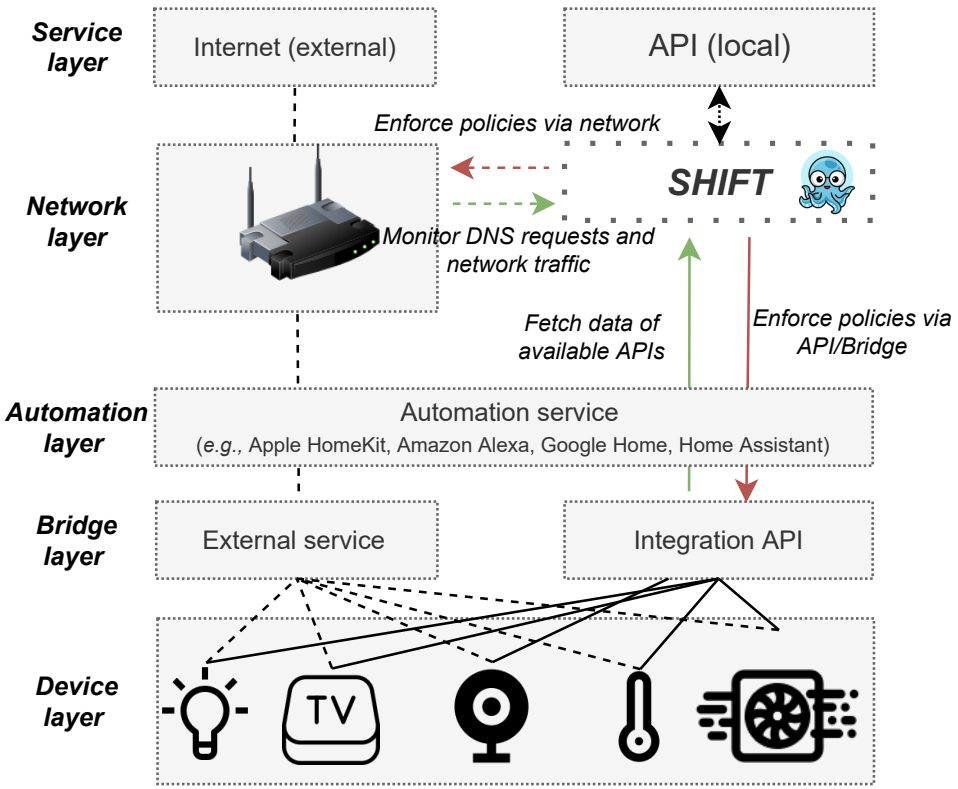


Figure 3.1: Overview of application scenario

In this scenario, the purpose is to observe which devices are sending what information and how frequently, noting that barriers hinder visibility. For instance, some devices may use encrypted end-to-end communication, preventing the observation of transmitted data. Although this secures personal information from third parties, the manufacturer may anonymously use the data. In the scope of this thesis, the focus lies on monitoring DNS requests and enforcing policies at the network layer.

3.2 Requirements

To develop an overview of the tasks that must be considered in the design stage to achieve the desired functionality of the system, as outlined by the thesis objectives in Section 1.2,

this chapter lists the requirements for the prototype. They are categorized into functional and non-functional requirements.

3.2.1 Functional Requirements

1. The user shall be able to configure which smart home devices the system manages using a web-based admin interface.
2. The user shall be able to enter their credentials and API keys for integration with peripheral systems, such as Pi-hole, through the user interface.
3. The system shall persist the user- and device-specific configurations in a relational database.
4. The system shall monitor the DNS requests of the specified devices by periodically querying the Pi-hole API.
5. The data gathered by monitoring the connections and requests shall be stored in a time series database.
6. The network behavior data of managed devices shall be presented to the user on a dashboard.
7. The system shall periodically inform the user about the network behavior of the specified devices.
8. The user interface shall allow the user to set simple privacy policies for individual devices.
9. The system shall periodically check whether the observed metrics violate any policies specified by the user.
10. A policy violation shall trigger an appropriate action in the system to enforce the policy in the future using the integration of peripheral systems.

3.2.2 Non-functional Requirements

1. The system shall be easy to deploy and integrate into the existing smart home infrastructure.
2. The system shall run on a system-on-chip environment such as the Raspberry Pi platform.
3. Monitoring and policy enforcement functions shall be designed modularly so that they can be extended to other mechanisms.
4. To protect users' privacy, the system shall store and process all data locally without sharing it with any third parties.

3.3 Architecture

The prototype design allows for a non-restrictive integration with an existing smart home environment. As its functionality is based on the local network's DNS communication, the prototype system is not directly dependent on the specific variation of the smart home platform in operation, such as Home Assistant, Apple Homekit, or any other variant. However, as Home Assistant itself can be operated locally, it is especially well suited for the purpose of data minimization. The core of the suggested system is formed by the lightweight web application prototype and a Pi-hole instance, serving as the network's primary DNS server. Both are designed to be hosted on a system-on-chip (SoC) platform (*e.g.*, Raspberry Pi) in the local network. The interaction between these two parts of the system composition enables the realization of the key functionalities of the system, such as monitoring and policy enforcement.

3.3.1 Components

The prototype design is based on the multi-component architecture illustrated in Figure 3.2. The diagram shows the prototype's environment, its various components, and their relationships. Its aim is to visualize the system design at a high level. This section details the purpose and intended operation of each module in the design.

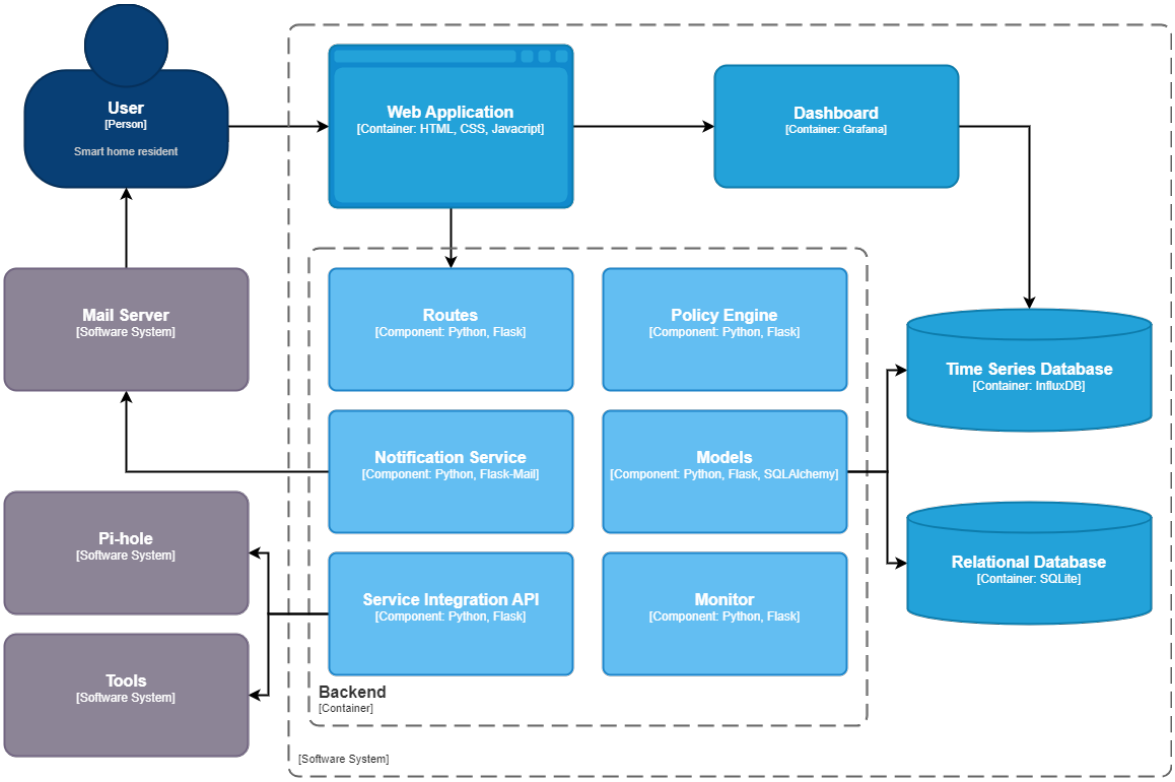


Figure 3.2: System Components

Web Application: Users interact with the system through the application's user interface in a web browser. It provides an administrative interface through which users can

log in to configure devices and policies. Moreover, it grants access to user-specific settings.

Dashboard: The user interface provides a dashboard that contains data visualization and statistics of device-network interactions. This component may be directly integrated into the web application or powered by an external tool such as Grafana [40], which relies on a data source provided by the system.

Routes: Incoming HTTP(S) requests from the frontend of the web application are handled by the routes component. This serves as an entry point for the backend. It is responsible for managing user authentication, redirecting requests, and serving the requested web pages to the web client.

Service Integration API: Interactions with different software systems are facilitated by the service integration API component, which acts as a consumer to integrate third-party services. Within the scope of the prototype, this component integrates mainly the Pi-hole HTTP API, but it can be extended to incorporate other systems or tools.

Monitor: The monitor component is responsible for executing scheduled jobs to supervise network activity, *e.g.* by periodically triggering queries directed at the Pi-hole API.

Models: The system's data models are defined in the models component, encapsulating all functionality related to interactions with the system's databases. This abstraction is facilitated by the use of an object relational mapper (ORM) library.

Policy Engine: User-defined policies are evaluated against metrics observed by the recurring monitoring activities. The policy engine checks for violations and may trigger actions to prevent future deviations.

Notification Service: The service generates reports on the behavior of smart devices and handles their delivery to users.

Relational Database: A relational database is used to store data concerning properties of users, policies, and devices. The entity relationship diagram in Figure 3.3 illustrates a simple model for a relational database, capturing user, policy, and device data.

Time Series Database: A time series database may store device networking information collected from various data sources in a unified format.

Pi-hole: Pi-hole serves as the primary DNS server of the smart home environment. Its API can be queried to monitor DNS requests made by devices. Its DNS sinkhole capabilities enable the enforcement of blocking policies.

Tools: The system design can be extended to integrate other third-party services and tools that may be used as data sources or to enforce user policies.

Mail Server: User notifications are delivered via a standard email service.

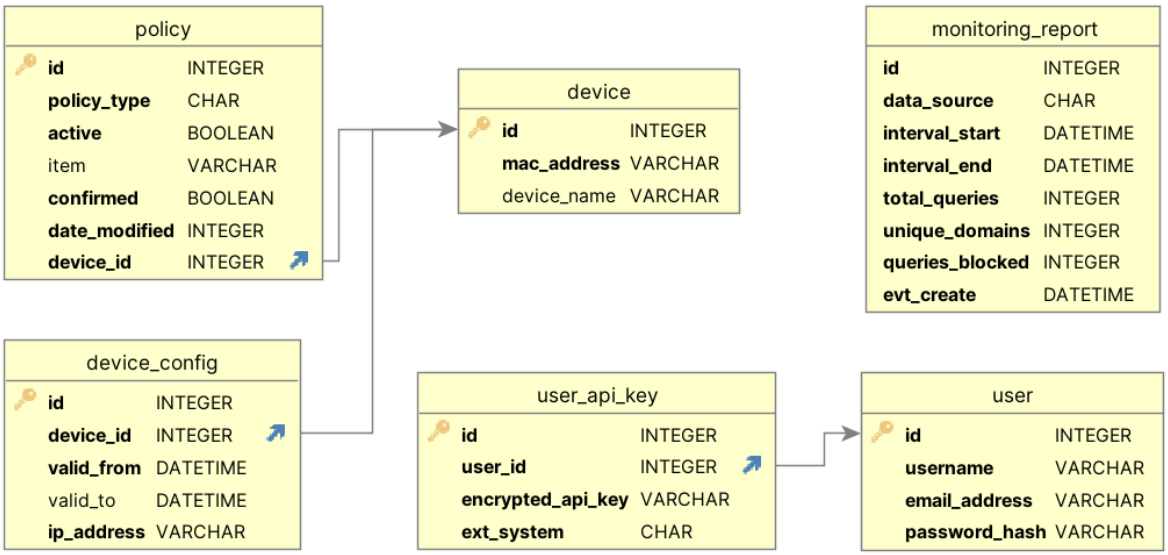


Figure 3.3: ER-Diagram

3.3.1.1 Frontend

In the prototype design, users interact with the application through a web-based frontend that consists of an administration interface and a dashboard. This user interface lets users register and log in with their credentials, allowing them to access their data and settings. After installing the application in a smart home environment, users must define smart home devices that the system monitors and manages using the administration interface. Furthermore, the interface lets users define simple device policies to allow or block connections. Users can review the configurations made in the administration interface and edit them anytime.

In addition to the administrative interface, the frontend provides a dashboard that displays statistics and data visualizations. It lets users interactively discover their devices' network behavior. The prototype design features a native dashboard integrated into the web application and uses Pi-hole's API as its main data source. In an extended version of the system that monitors additional data sources, the design can be adapted to incorporate a dashboarding platform such as Grafana [40]. This configuration requires dedicated data storage, such as a time series database that persists the transformed data from all monitoring activities and serves as the data source for the dashboard.

3.3.1.2 Backend

The backend of the system runs on a lightweight web server and contains the main business logic of the application. Its responsibilities include data aggregation, storage and retrieval, handling incoming frontend requests, integrating third-party services, performing monitoring and policy evaluation tasks, and generating user notifications.

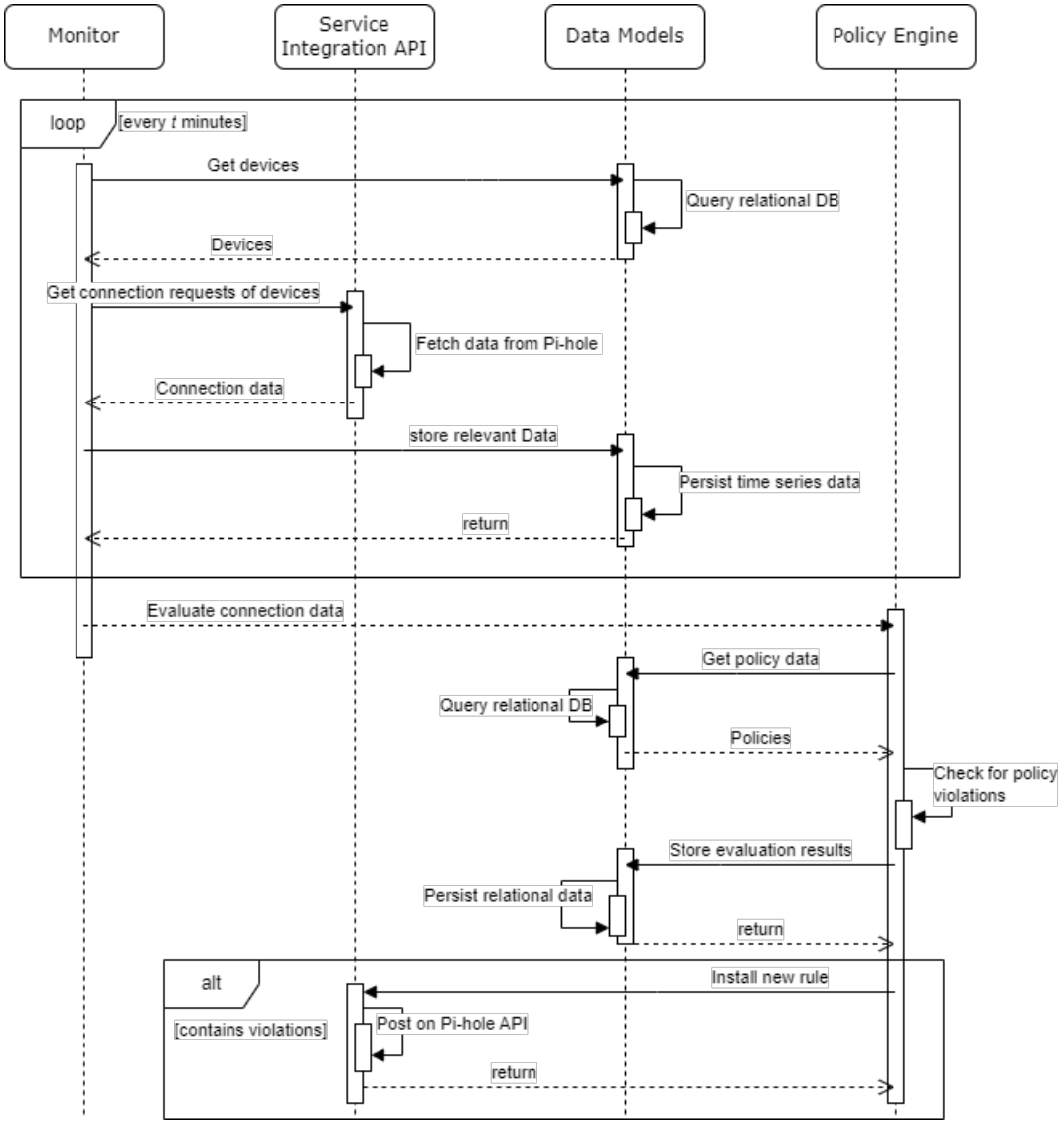


Figure 3.4: UML Sequence Diagram

Figure 3.4 illustrates two core functionalities of the prototype's backend design. The Unified Modeling Language (UML) sequence diagram features the interactions between the four components introduced above: Monitor, Service Integration API, Data Models, and Policy Engine.

The upper section of the graphic represents a recurring data collection job. It monitors devices' behavior based on DNS query information retrieved from Pi-hole. The Monitor initiates the process by obtaining a list of devices from the relational database through the Data Models component. It then fetches DNS query data through the Service Integration API module, which serves as a consumer for Pi-hole's API. This process returns the query data for the time interval between job executions. The next step is to clean and transform the obtained data and store the results in a time series database. This step is only required if the data is collected from multiple data sources to make the data available in a unified format for analysis and visualization.

The lower part of the diagram illustrates the evaluation of the collected data orchestrated by the Policy Engine. First, a list of user-defined and device-specific policies is retrieved from the relational database. The Policy Engine then checks for violations by comparing the data provided through the data collection job with the retrieved policies. The results of this evaluation are then persisted through the Data Models component for use by other system features, such as reporting and user notification. The Policy Engine can take necessary steps to ensure future policy compliance if the evaluation brought forth any policy violations. In the prototype, this entails the installation of additional rules for Pi-hole's DNS sinkhole mechanism by utilizing the integration with their API.

Chapter 4

Implementation

Based on the design presented in Chapter 3, a prototype of the proposed system is developed to evaluate this approach to improving awareness and control over communications in a smart home environment. As indicated in the architecture description in Section 3.3, the prototype is realized as a lightweight web application. It is designed to run on a system-on-chip (SoC) platform located in the network of the smart home ecosystem.

To build a prototype, Flask has been chosen as a framework [55] to implement the web application. The framework enables the creation of web applications using Python [16] and adheres to the Web Server Gateway Interface (WSGI) specification described in the Python Enhancement Proposal (PEP) 3333 [11]. WSGI serves as an intermediary between web servers and Python web applications, translating HTTP requests and responses. Flask is often referred to as a microframework due to its minimal but extensible core [53]. This means that it focuses on central functionalities to create web applications, such as URL routing and request parsing. It is left up to the developer to install additional features by including extensions and libraries. Hence, the application environment can remain lightweight. This attribute is the main reason why Flask was chosen to develop the prototype.

This chapter presents selected implementation details of key features to illustrate the approach to realizing the design elaborated in Chapter 3. For this purpose, the implementation of the Pi-hole integration, the monitoring functionality, and finally the notification service are closely examined in Section 4.1. Moreover, Section 4.2 contains a description of the development and build pipeline that was applied to manage this software project.

4.1 Core Features

This section examines selected features of the prototype. This includes listing the requirements they help to meet, describing the context in which the respective feature is embedded, and explaining the chosen solution. The goal is to highlight some of the key implementation details of the application to better understand the system's operations.

4.1.1 Pi-hole Integration

In the system's architecture, the Service Integration API component serves the purpose of facilitating communication with third party services to collect data and to enforce user-defined policies. The component is intended as an abstraction of the interfaces provided by these services, mapping selected functionalities offered by them and making them accessible for other modules of the application. Although the architecture allows the implementation of multiple third-party services, the prototype focuses on Pi-hole to prove the concept. Hence, the integration with Pi-hole plays a crucial role in the prototype's design, as it enables the monitoring of device-related DNS query data and the installment of new policies.

Pi-hole offers an HTTP-based API, which was chosen as this project's primary interaction point between the two systems. At the time of writing this thesis, the developers of the Pi-hole community are working on an updated RESTful API, offering a richer set of functionalities that will eventually replace the API currently in use. However, as it is in a pre-release state with development still ongoing, and considering the stability of the integration, the decision was made to utilize the current HTTP API. There is no official documentation on the HTTP API available. Therefore, its integration relied on the analysis of Pi-hole's source code on Github [61].

```
1  class PiholeConsumer:
2
3      def __init__(self, pihole_domain, auth_token: str):
4          self._pihole_domain = pihole_domain
5          self._auth_token = auth_token
6
7      def get_all_queries(self, from_timestamp: int, until_timestamp: int) -> dict:
8          builder = QueryBuilder(self._pihole_domain)
9          builder.add_auth_token(self._auth_token)
10         builder.type_all_queries()
11         builder.add_from(from_timestamp)
12         builder.add_until(until_timestamp)
13         response = builder.query.send_request()
14         return response
```

Listing 1: Snippet taken from the PiholeConsumer class

Listing 22 of Appendix B depicts a subset of the Pi-hole HTTP API represented using the OpenAPI standard [52] to illustrate the structure of the API. The API essentially consists only of a single endpoint located at `http://pi.hole/admin/api.php`, where `pi.hole` is the standard domain for Pi-hole in the local network. This endpoint accepts various types of queries for information such as a daily summary, the top x clients or statistics on query types. Requests are formed by appending query parameters to the URL of the API endpoint and transmitted as `HTTP GET`-requests. As an example, the parameters in Listing 22 can be combined to create a request: `http://pi.hole/admin/api.php?auth=API_KEY&getAllQueries=1&from=TIME_X&until=TIME_Y`. This results in requests for all DNS queries between times `TIME_X` and `TIME_Y` in UNIX timestamp format. To authenticate

the request, Pi-hole expects an `API_KEY`, which is also passed to the API as a parameter in the query string. Using just HTTP without any encryption, this may not be the most secure way of authentication. It is therefore recommended to only allow requests from the local network.

Implementing the Pi-hole module of our Service Integration API component takes an object-oriented approach and employs the builder pattern to construct requests as described above. The module contains three classes: `PiholeConsumer`, `QueryBuilder`, and `Query`. `PiholeConsumer` is an abstraction that encapsulates the integration for clients in other components by providing a simple interface to work with the API.

```
1 def send_request(self) -> dict:
2     response = requests.get(self._base_url, params=self._query_params)
3     return response.json()
```

Listing 2: The send_request method of the Query class

Listing 1 shows a snippet of the `PiholeConsumer` class, including the `get_all_queries` method that can be used to create and execute requests for all queries within a specific time frame, as in the example above. To apply this feature, a client instantiates the `PiholeConsumer`, initializes it with the domain name for Pi-hole (pi.hole by default) and its API key, and then invokes the `get_all_queries` method. The method constructs a new instance of the `QueryBuilder` class and calls its methods to direct the creation of the query URL by telling the builder which parameters to add. Finally, the builder is asked to return the finished query and transmit it by invoking the `send_request` method, depicted in Listing 2. The method shows that the `Query` instance, which contains a base URL for the endpoint and the added query parameters, sends "itself" using the *requests* HTTP library and returns a response in `JSON` format.

4.1.2 Monitoring

The Monitor component of the architecture proposed in Section 3.3.1 plays a central role in the system's overall design. It is responsible for executing recurring or on-demand tasks that involve collecting and processing data concerning the behavior of smart home devices. To obtain the data from third-party services, it depends on the Service Implementation API component. For the implementation of the prototype, this covers the functional requirement number 4 described in Section 3.2.1.

The goal is to collect and process the data to make the relevant information available to other components, either directly or by storing it in a time-series database for later access. By preparing the data for these clients, the Monitor facilitates their functionalities and helps them fulfill the functional requirements they cover. The Policy Engine depends on the data provided by the Monitor to evaluate policy compliance, the Notification Service needs it to compile reports, and the Dashboard generates data visualizations based on it. Hence, the Monitor component lays the foundation to meet the functional requirements

5, 6, 7, and 9 from the list in Section 3.2.1. This further underscores the importance of this component for the application.

To implement the prototype, the Monitor component consists of one module, the Pi-hole monitor, which gathers data from the Pi-hole instance utilizing the integration described in the previous Section 4.1.1. The module interacts with the Service Integration API component through the `PiholeConsumer` class. Listing 3 shows how the consumer is initialized at the start of the `fetch_dns_query_data` function in the pi-hole monitor.

```
def fetch_dns_query_data(from_timestamp: int, until_timestamp: int):
    # Load Pi-hole configuration and initialize consumer
    pihole_domain = current_app.config['PIHOLE_DOMAIN']
    auth_token = current_app.config['PIHOLE_AUTH_TOKEN']
    pihole_consumer = PiholeConsumer(pihole_domain, auth_token)
```

Listing 3: Instatiation of PiholeConsumer in the Monitor component

Following this, the `fetch_dns_query_data` function calls the method `get_all_queries` of the `PiholeConsumer`, which was examined above and depicted in Listing 1.

```
query_data = pihole_consumer.get_all_queries(
                from_timestamp, until_timestamp)['data']
```

Listing 4: Pihole consumer method invocation

The response of the consumer's method is a `JSON` object in the form of a Python dictionary datatype. As seen in Listing 4, the response is accessed with the key *data* before it is stored to a variable. This is due to the fact that the returned `JSON` object only has a single attribute containing a list of datapoints represented by a list of its values. An example of the structure of such a response from the Pi-hole API containing a single datapoint is given in Listing 5.

```
{"data": [[
        "1689149492",
        "A",
        "netcom.netatmo.net",
        "192.16x.x.xxx",
        "2",
        "0",
        "3",
        "82",
        "N/A",
        "-1",
        "one.one.one.one#53",
        ""
    ],]}
```

Listing 5: Pi-hole API response for getAllQueries

```
    # Get all active IPs from db
    active_ips = db.session.execute(
        db.select(DeviceConfig.ip_address)
        .where(DeviceConfig.valid_to == None)).scalars().all()
    active_ip_set = set(active_ips)

    dataset = []
    # Process data
    for datapoint in query_data:
        client = datapoint[3]
        # Filter out data from inactive/unregistered clients
        if client not in active_ip_set:
            continue
        timestamp = int(datapoint[0])
        query_type = datapoint[1]
        domain = datapoint[2]
        status = datapoint[4]
        reply_type = datapoint[6]
        dataset.append([timestamp, client, query_type, domain, status, reply_type])

    return dataset
```

Listing 6: Data processing

After obtaining the DNS query data from Pi-hole via the Service Integration API, the raw data is processed before the `fetch_dns_query_data` function returns the resulting data set. When processing the data set, as illustrated in Listing 6, all data points that are not configured as active device IP addresses are discarded and only the relevant attributes of each data point are retained.

The `fetch_dns_query_data` function described above is the approach taken in the Pi-hole monitor module for collecting data from Pi-hole. Various features of the module build on top of this approach by invoking said function in their implementation. One application of this is a job that is designed to be executed at scheduled intervals to continuously monitor the DNS queries of configured devices. The code snippet in Listing 7 illustrates the implementation of the function that executes the business logic of the job. First, the `fetch_query_data_job` function searches the InfluxDB time-series database for the latest entry recorded by a previous run of the job. Based on that, the `fetch_dns_query_data` function is called, using the time interval between the latest record and the current timestamp. Following this, the entries of the returned data set are mapped to a data structure designed to hold InfluxDB measurements, which are comparable to records in a relational database. Finally, the transformed data set is written to the time series database instance. To run the job independently and parallel to the rest of the web application, a Flask-CLI command was created to execute the `fetch_query_data_job` function. The Command-Line Interface (CLI) command listed in 8 is periodically triggered by a cron job that is installed when building the project's Docker image.

```python
def fetch_query_data_job():
    current_app.logger.info('starting job...')
    # Load latest record from influxdb and find timestamp
    influxdb_client = InfluxDBClientWrapper()
    latest_timestamp = influxdb_client.get_latest_timestamp("dns_queries")
    from_timestamp = datetime.now().timestamp() - \
        current_app.config['SCHEDULER_TIMEINTERVAL'] \
        if latest_timestamp == -1 else latest_timestamp
    until_timestamp = int(datetime.now().timestamp())

    # fetch dns query data from pihole
    dataset = fetch_dns_query_data(from_timestamp, until_timestamp)

    # map to influxdb model
    dns_query_measurements = list(map(
        lambda x: DNSQueryMeasurement(x[0], x[1], x[2], x[5], x[3], x[4]), dataset))

    current_app.logger.info(
        f"Writing {len(dns_query_measurements)} new datapoints to influxdb...")

    influxdb_client.store_dns_query_measurements_batch(dns_query_measurements)
```

Listing 7: Fetch query data job

```python
@app.cli.command()
def execute_monitoring_job():
    """Run scheduled job"""
    from app.monitors.pihole_monitor import fetch_query_data_job
    fetch_query_data_job()
```

Listing 8: Command line interface command

4.1.3 Notification Service

The system architecture described in Section 3.3.1 features a Notification Service component in the application's backend. Its responsibilities include generating reports containing visualizations and statistics related to the network behavior of smart home devices and sending them to registered users. To compile these reports, the feature depends on the data made available by the Service Integration API component. It is the objective of the Notification Service to keep users informed about device activities and perhaps enabling them to spot irregularities. The intention is also to raise the awareness of smart home residents of their privacy level through added observability. In the proposed design, the medium used to notify users is email. Different types of notifications may warrant the use of alternative transmission methods, such as instant messaging services. The implementation of the Notification Service component aims to meet the functional requirement number 7 listed in Section 3.2.1. This section examines the creation and transmission of a weekly summary email included in the prototype realization of the Notification Service.

One of the few default dependencies of the Flask framework is the Jinja2 templating engine [54], which allows developers to create dynamic HTML templates. Templates are HTML documents, but extend the syntax of static HTML by allowing variables and expressions that enable the insertion or generation of content when rendering the final HTML document on the server. Jinja templates are not only used to implement the web pages of the prototype's frontend, but also to render the HTML-based email template for the weekly summary notification. Due to inconsistencies in HTML and Cascading Style Sheets (CSS) support across different email clients and other constraints, creating email templates from scratch can prove difficult. To avoid these complexities, the MJML library [43] has been applied to create the layout for the email template. MJML provides a markup language and a component library designed to build responsive emails. The markup can also be combined with templating constructs, such as those used by the Jinja2 engine. After implementing the email layout in MJML markup, the code is transpiled to HTML. For the weekly summary email used in the prototype, this process results in a template containing placeholders for the user's name, a logo, and two plots. Moreover, it embeds the logic to generate a table of the top domains visited by devices. Listing 9 shows a combination of MJML and Jinja syntax that is used to dynamically create the table of top domains. The `mj-table` tag in line 1 is part of MJML's markup and the `for`-loop construct is used by the Jinja engine.

```
1  <mj-table css-class="demTable">
2      <thead>
3          <tr>
4              <th>Smart Device</th>
5              <th>Domain</th>
6              <th>Visits</th>
7          </tr>
8      </thead>
9      <tbody>
10         {% for key in top_dict %}
11         <tr>
12             <td>{{key[0]}}</td>
13             <td>{{key[1]}}</td>
14             <td>{{top_dict[key]}}</td>
15         </tr>
16         {% endfor %}
17     </tbody>
18 </mj-table>
```

Listing 9: Table in the weekly notification email layout in MJML

After implementing the email template, the next step is to provide the content, render it, and add it to a sendable email data type. The prototype's Notification Service module handles these tasks in its `create_weekly_email` function, the first part of which is illustrated in Listing 10.

The function takes an argument of type User, as the generated email has to be sent to a specific email address, and the template contains a variable for the username. It obtains a data set from the Pi-hole Monitor component by calling its `weekly_summary` function

```
1  def create_weekly_email(user: User) -> MIMEMultipart:
2      recipient = user.email_address
3      username = user.username
4
5      # Get the weekly summary from pihole monitor
6      df = weekly_summary()
7      top_domains = df[['client_name', "domain"]] \
8          .value_counts().nlargest(10).sort_values(ascending=False)
9      top_dict = top_domains.to_dict()
10
11     html_content = render_template('emails/weekly-summary.html', \
12         username=username, top_dict=top_dict)
```

Listing 10: Weekly email summary creation (Part 1)

in line 6 of the Listing 10, which returns a pandas DataFrame. The DataFrame is further processed to prepare the data for the top-domain table in the email template. Following this, the HTML content of the email is rendered by providing the email template for the weekly summary, and the values for its `username` and `top_dict` variables to the template engine with the function call in line 10.

```
1  # Creating plots
2  img_1 = figure_to_byte_img(create_pie_chart(df))
3  img_2 = figure_to_byte_img(create_stacked_bar_chart(df))
4
5  # Initalizing images as email attachments
6  logo = MIMEImage(enc_img, 'png')
7  chart1 = MIMEImage(img_1, 'png')
8  chart2 = MIMEImage(img_2, 'png')
```

Listing 11: Weekly email summary creation (Part 2)

After rendering the HTML body, the `create_weekly_email` function prepares the email's attachments. Lines 2 and 3 of the Listing 11 create the plots for the weekly report based on the data contained in the pandas DataFrame that was initialized in Listing 10. Both figures are generated with the help of `plotly.py` [64], a graphing library for Python and then converted to static images in the form of byte strings. The images are then used to initialize the content of two new `MIMEImage` objects in lines 7 and 8 of Listing 11. The `MIMEImage` class is part of the email package in Python's standard library [65], which can be used to create Multipurpose Internet Mail Extension (MIME) datatypes with the media type set to image as defined in RFC 2046 of the Internet Engineering Task Force (IETF) [17].

Finally, Listing 12 shows how all parts are put together to build and return a `MIMEMultipart` message, using the Nofication Service module's `EmailBuilder` class. Adding the images to the message also entails marking each `MIMEImage` with a `Content-ID` header, as can be seen in the code of the the `add_image` method of the `EmailBuilder` in Listing 13. This is important because it allows us to reference the respective image by the given

```
msg = EmailBuilder() \
        .with_subject('Weekly summary') \
        .with_sender(current_app.config["MAIL_USERNAME"]) \
        .with_recipient(recipient) \
        .with_html_content(html_content) \
        .with_text_content(text_content) \
        .add_image(logo, "logo") \
        .add_image(chart1, "chart1") \
        .add_image(chart2, "chart2") \
        .build()

    return msg
```

Listing 12: Weekly email summary creation (Part 3)

`Content-ID` in the email template, which permits the inlining of the email attachment into the HTML body of the message.

```
def add_image(self, image: MIMEImage, content_id: str | None):
    if content_id is not None:
        image.add_header('Content-ID', f'<{content_id}>')
    self.msg.attach(image)
    return self
```

Listing 13: Weekly email summary creation (Part 4)

After generating the weekly summary report for the user, the email is sent to them using a standard email service. For the prototype implementation, it is left to the user to set up an account with an email provider and set the their chosen credentials in the application configuration. The email is sent via the Simple Mail Transfer Protocol (SMTP) [37] using the `smtplib` package [66] of the Python standard library, as illustrated in the Listing 14.

```
send_email(msg: MIMEMultipart):
    try:
        with smtplib.SMTP(current_app.config["MAIL_SERVER"],
                          current_app.config["MAIL_PORT"]) as smtp:
            smtp.starttls()
            smtp.login(current_app.config["MAIL_USERNAME"],
                          current_app.config["MAIL_PASSWORD"])
            smtp.send_message(msg)
    except smtplib.SMTPException as e:
        current_app.logger.error(f'Email not sent to {msg["To"]}. \n Error: {e}')

    current_app.logger.debug(f'Email sent to {msg["To"]}')
    return 'Sent'
```

Listing 14: Weekly email summary creation (Part 5)

4.2 Development and Build Pipeline

In the context of developing the prototype for the proposed system, a continuous integration and delivery (CI/CD) process has been established to streamline the code management, build, and deployment of the project. This section describes the workflows and infrastructures used for these tasks.

The project's source code is managed using Git [21] as a version control system with a remote repository of the code base hosted on GitHub [22]. The source code for the front- and backend of the application is hosted in a mono repository. On the backend, the development relies primarily on Python, using the Flask framework and various libraries. Python virtual environments [67] and the pip package installer [63] are employed to manage the backend dependencies. Frontend development is performed using HTML, Jinja2 templates, CSS and Javascript. For frontend specific libraries included in the project, such as TailwindCSS and MJML, dependency management is carried out using the npm package manager [51]. To enable the usage of npm-installed packages, the project uses Parcel [56] to build and bundle the CSS and Javascript code to be imported into the application's Jinja2 HTML templates.

The prototype web application is built and delivered as a Docker image [10], facilitating its execution in a containerized environment. This image is created in a two-stage build process implemented by the project's Dockerfile. The initial *builder* stage (see Listing 16 in Appendix B) installs nodejs and npm to enable the subsequent installation of npm dependencies. After this, docker copies the project to the builder image and executes the necessary npm commands to generate the frontend source files for the project. The second stage is the actual Docker image that will be published, whereas the builder image is discarded once the build terminates. This final stage, as shown in the code in Listing 17 in Appendix B, initially sets hardcoded environment variables for the production environment and creates the user who runs the application. It then copies the project files to the image and installs cron and the SQLite command-line interface (CLI). Next, it sets up the cron tabs for the monitoring job and the weekly email notification. It then copies the frontend dependencies that where generated in the builder stage to the main image and installs the required Python libraries. Finally, the Dockerfile exposes port 8000 for other containers and defines a boot script to launch the application as an entry point.

The prototype deployment process is implemented through an automated GitHub Actions workflow that builds and deploys the Docker image. The `yaml`-file used to define the workflow can be found in Listing 18 of Appendix B. Triggered upon creating a new release, this workflow checks out the project's Github repository and prepares Docker Buildx to enable multi-platform builds. Following this, it logs into Docker Hub using the credentials stored as repository secrets. The Docker image for ARM architectures is then built, targeting the Raspberry Pi platform, and subsequently pushed to Docker Hub, enabling efficient access and distribution through this channel. This automation optimizes the deployment process and provides users with easy access to the latest version of the web application by simply pulling the image from Docker Hub.

Chapter 5

Evaluation

The following chapter addresses the evaluation of the application prototype in a live smart home setting. Section 5.1 explains the hardware, software, and networks set up for the assessment. Furthermore, several scenarios for evaluating system performance and experimentally validating its functionalities are outlined in Section 5.2. Section 5.3 presents the results of the experiments to provide an understanding of the viability of the prototype. The chapter ends with a discussion of the results and their implications in Section 5.4.

5.1 Configuration

This section outlines the configuration of hardware, software, and network settings that were employed to test the developed prototype in a productive smart home environment. This setup enabled the evaluation of the system's functionalities and performance.

5.1.1 Hardware

Raspberry Pi: As a platform to host the prototype system, a *Raspberry Pi 3 Model B Plus Rev 1.3* [68] was utilized. The single-board computer was released in Q1 of 2018 and comes with a 1.4 GHz quad-core processor based on an ARMv8 64-bit architecture. It offers 1 gigabyte of synchronous dynamic random access memory (SDRAM) and uses a MicroSD card for permanent storage [12].

Smart home devices: The smart home used as a testing environment for the prototype system provides lighting powered by seven individual Philips Hue smart light bulbs [59]. These lights are managed with a *Hue Bridge* [58], which acts as a hub that communicates with the lights through the ZigBee protocol, connecting them to the network. Furthermore, the smart home uses a Netatmo smart air quality monitor [49]. The device measures temperature, humidity, CO2 levels and noise intensity in ten-minute intervals. It uses WiFi to communicate with the network. Lastly, an

Android-based smart TV box named *Leap-S1* [72] is connected to the smart home via LAN.

Network Infrastructure: The smart home network is interconnected by a standard ISP-issued combined modem-router-gateway. In this case, an *AX7501-B SERIES* device by Zyxel is utilized [77].

5.1.2 Software

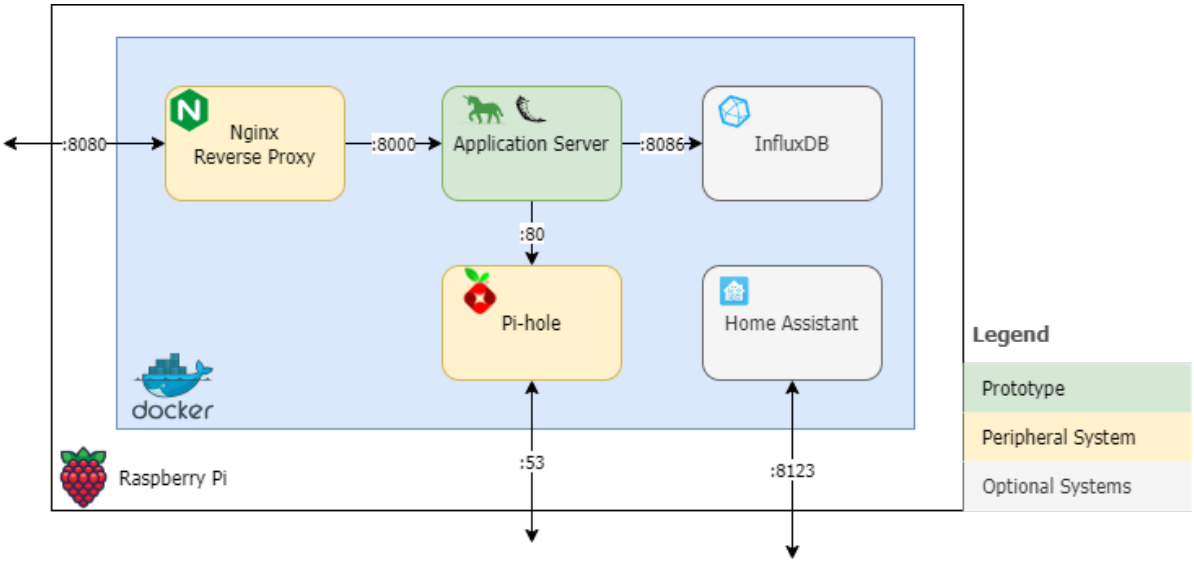


Figure 5.1: Docker containers

The prototype application developed in this thesis and the surrounding software systems it interacts with are hosted in a container environment running on a Raspberry Pi. Containerization is a virtualization method that packages applications, their dependencies, libraries, and settings into individual units known as containers. These containers are self-contained and isolated from the underlying host system, which makes the technology highly portable. In this thesis, Docker [10] is used as a containerization platform to build, deploy, and run applications.

The diagram in Figure 5.1 shows the composition of the Docker containers employed in this assessment. It also illustrates the flow and ports used for communication between containers within the internal Docker network and with the host network. The full docker-compose file, specifying this set-up can be found in Appendix B, divided into Listings: 19, 20, and 21. The central component is the **Application Server** container, which houses the web application prototype. It leverages a Gunicorn WSGI server [28] to run the Flask web application and accept HTTP requests. In front of the application server, a container running an Nginx web server is configured to act as a **reverse proxy**. It handles incoming client requests from the network by directly serving static content for the web application, forwarding requests to the application server, and returning the application server's response. The reverse proxy is configured to soley forward requests from the local network. It thereby helps to shield the web app running on the application server,

which does not directly expose any ports outside the docker environment. Another key part of the system is the **Pi-hole** instance running in a separate docker container. The Application Server instance communicates with the Pi-hole container via API on port 80 using the Docker internal network. Pi-hole also accepts DNS requests from the host network on port 53.

The three container instances described above form the core system around the prototype web application. In an extended configuration, another container running an instance of the **InfluxDB** time series database is included. As the prototype currently uses only one data source to monitor device behavior, it is not dependent on this database to store, merge, and aggregate data from different sources. The necessary data can be queried directly over the Pi-hole API. Additionally, a containerized instance of **Home Assistant** runs on the same Raspberry Pi platform. Home Assistant does not interact with the prototype directly. Still, it is used to test the effects of the policies enforced by the prototype on the platform's ability to communicate with smart home devices and run automation.

5.1.3 Networking

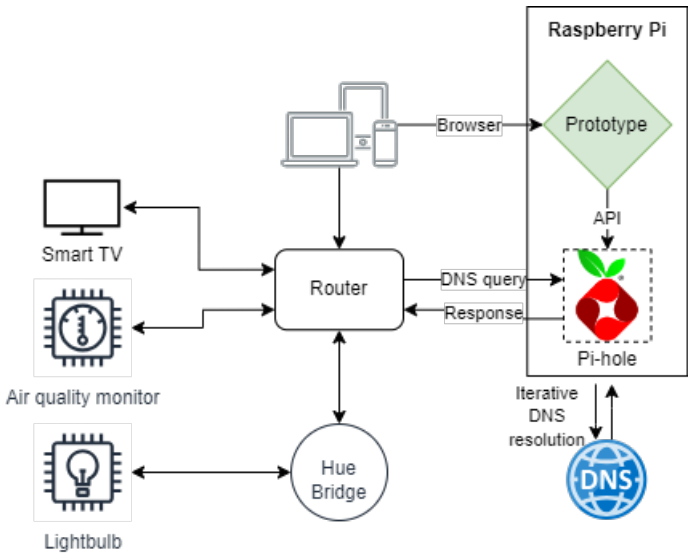


Figure 5.2: Network configuration

A diagram illustrating the network structure used for the hardware discussed in Section 5.1.1 is presented in Figure 5.2. This network was used during the evaluation phase of this thesis. Ethernet cabling connects the Raspberry Pi, smart TV, and Hue Bridge devices to the home router. Hue's smart lightbulbs wirelessly communicate with the Hue Bridge using the ZigBee protocol and are indirectly connected to the smart home network through the bridge. The air quality monitor, on the other hand, is directly connected to the router via Wi-Fi. A key detail in this network structure is the configuration of Pi-hole as the router's primary DNS server. This has the effect that all DNS queries of the connected smart devices are handled by Pi-hole. This enables the use of Pi-hole's DNS sinkhole

mechanism to enforce the policies defined by the user in the prototype application. If Pi-hole does not block a query based on such a policy, it forwards the request to an upstream DNS server to resolve the requested domain.

5.2 Scenarios

To evaluate whether the implementation of the prototype sufficiently satisfies the functional and non-functional requirements defined in Section 3.2, a series of experiments are conducted. This section describes the scenarios that were tested to obtain the necessary qualitative and quantitative data necessary to assess the implementation outcomes.

5.2.1 Monitoring

Our first scenario aims at verifying whether the smart home devices specified by the user and managed in the prototype application are monitored according to the requirements. DNS requests made by devices should be queried at regular intervals via the Pi-hole API. If the application is configured to integrate the time series database module, the collected data should be persisted upon each run of the monitoring task.

To test this, the following steps are performed:

1. Add a new smart home device to the application's database using the web-based admin interface.
2. Let the application run for multiple monitoring cycles.
3. Examine the reporting database table to determine whether the monitoring task has been completed successfully the correct number of times and if the behaviour of the newly configured device was considered during the process.
4. Use the user interface provided by Pi-hole, to retrieve the data for the same time frame and device and compare it to the data stored in the time series database.

The expected result is that the prototype application runs the monitoring job at predefined intervals, according to the configuration of the cron job that triggers it. It is also anticipated that the newly configured device is considered when processing the data returned by the Pi-hole API.

5.2.2 Policy Enforcement

The purpose of this following scenario is to determine whether the policies defined by the user are successfully enforced. This means certain domains are blocked or allowed for a

smart home device, depending on the user's settings. The experiment also aims to verify whether the default policies are applied to newly detected domains.

The following steps are performed in the context of testing policy enforcement:

1. Add a new smart home device using the web-based admin interface. The default policy for this device is set to *allow all* domains.
2. Await the next policy evaluation cycle. Verify that policies have been created with the default behavior for all domains detected in the previous monitoring job for the smart home device in question.
3. Change one of the automatically created policies to block the associated domain.
4. Verify that the policies defined in the prototype application have been successfully installed on the Pi-hole instance.

After executing the steps listed above, the prototype application is expected to have recognized all domains requested by the newly configured smart home device in the previous monitoring cycle. It is also supposed to have created policies for them with the chosen default value of *allow* and one of them was changed to *block*. Furthermore, all policies are expected to be correctly mirrored in the Pi-hole instance's domain list.

5.2.3 Weekly Notification

This test reviews the feature of weekly email notifications on the behavior of registered smart home devices. The aim is to ascertain whether reports are being produced correctly and sent by email in a productive setting.

To verify the weekly notification feature, the following steps are executed:

1. Add a valid Simple Mail Transfer Protocol (SMTP) mail service connection to the application's configuration.
2. Register a user with a valid email address.
3. Add all available smart home test devices to be monitored by the application.
4. Configure allowing and blocking policies for all devices.
5. Verify the receipt of the report by email at the time predefined by the application.
6. Examine the correctness of the data contained in the report.

The anticipated result of this experiment is that a weekly report will be created and dispatched to the relevant email address with accurate information at the right time.

5.2.4 Smart Device Operability

The following scenario aims to determine to what extent shielding smart home devices from the cloud affects the functionality of the devices. For this purpose, we conduct experiments with our smart home devices from Philips Hue and Netatmo. We test the usability of the devices on the one hand via integration with our hub-based smart home platform Home Assistant, and on the other hand with the respective smartphone apps of the manufacturers. The goal is to determine whether blocking domains limits the functionality of the devices.

To assess the operability of smart home devices under the influence of blocking policies we proceed with the following steps:

1. Ensure that the Philips Hue Bridge and the Netatmo air quality monitor are registered in the prototype application.
2. Configure different *allow/block* policy combinations for the domains visited by the respective device.
3. For each of the policy combinations, test whether the specified smart home devices can still be controlled/used via Home Assistant. Furthermore, check if they are still accessible using their respective mobile apps.

The expectation is that Philipps Hue devices will be controllable via Home Assistant, even if all domains are blocked. The reason for this is that the Philips Hue integration for Home Assistant [60] works through local push. This means that Home Assistant is able to provide direct communication with a device and will be alerted when a new state is available, without using a cloud service. The air quality monitor on the other hand depends on the connection to the Netatmo cloud. Home Assistant integrates with Netatmo devices through cloud polling [48], which requires an active Internet connection. Therefore, creating blocking policies for this device is expected to cause limitations in its usability.

5.2.5 Performance Evaluation

In this test scenario, the aim is to evaluate the performance of our prototype deployment on the Raspberry Pi platform. The goal is to measure the resource utilization of the containers running on the physical device to assess the solution's suitability for the platform.

The following step are necessary for this evaluation:

1. Set up the application with a user, all smart home devices and different policies to simulate an average use case.

2. Run a prepared script to measure the resource utilization of the individual container instances while using the application normally over an extended time period to obtain a baseline.

3. Assess the performance by examining the collected data set.

This test scenario is expected to result in a high utilization of available resources for the configuration that includes all containers.

5.3 Results

This section presents the results of the experiments that were outlined in the scenario descriptions of Section 5.2.

5.3.1 Monitoring

To evaluate the monitoring feature of the prototype in a productive setting, the Netatmo air quality monitor device was chosen to run the experiment. After establishing a clean prototype installation on the Raspberry Pi platform, the first step was to register the test device with the application. This was carried out using the web-based user interface, as illustrated in Figure 5.3. The user chooses a name for the smart home device, configures its MAC and IP addresses, and selects a default policy.

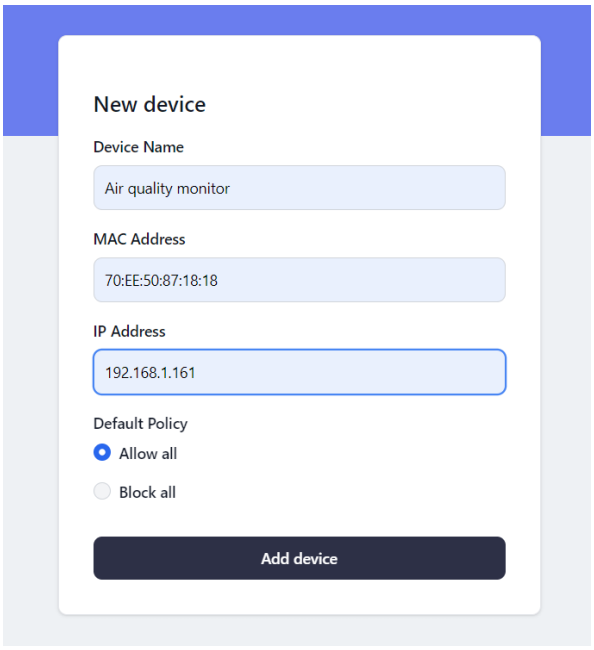


Figure 5.3: Adding a new device

Having registered the device successfully, the next step was to let the monitoring job run multiple times at predefined intervals. In a typical deployment scenario, an hour

between job runs should be ideal, as the amount of data does not become too large and the application performance will not be significantly impacted. To reduce testing time, the application was configured to run a monitoring job every 15 minutes via crontab.

After letting the application run without interference for about an hour, thereby completing the second step of this scenario, we opened a secure shell connection to the Raspberry Pi to examine the results. On the platform, a bash terminal was opened in the Docker container hosting the prototype web application in order to access its relational database with the SQLite command line tool. Table 5.1 shows the results of quering the report table with the following command: `SELECT * FROM MONITORING_REPORT` . Each time a monitoring job is completed successfully and without exceptions, a record is added to this database table. Hence, we can observe that the process was executed four times, as was to be expected after one hour. The table further shows a total of five queries that were made by the air quality monitor during this time. As we have only registered one device with the application, it is clear that the monitoring job must have included this device's DNS queries in its analysis.

data_source	interval_start	interval_end	total_queries	unique_domains	queries_blocked	evt_create
pi_hole	2023-07-25 12:00	2023-07-25 12:15	1	1	0	2023-07-25 12:15
pi_hole	2023-07-25 12:15	2023-07-25 12:30	1	1	0	2023-07-25 12:30
pi_hole	2023-07-25 12:25	2023-07-25 12:45	1	1	0	2023-07-25 12:45
pi_hole	2023-07-25 12:35	2023-07-25 13:00	2	1	0	2023-07-25 13:00

Table 5.1: Report database table after 4 monitoring job executions

Figure 5.4 shows a table listing the datapoints stored in the InfluxDB time series database for the examined time period. This table was generated with the data visualization tools available on the web interface of the InfluxDB instance. The data indicates five DNS query records for one client, which is identified by the air quality monitor's IP address. This is consistent with the information taken from the reporting database table and listed in Figure 5.1. Under the `_value` column of the table in Figure 5.4, we find the domain that the respective client requested.

client	query_type	reply_type	_value	_time
192.168.1.161	A	3	netcom.netatmo.net	2023-07-25 12:15:47 GMT+2
192.168.1.161	A	3	netcom.netatmo.net	2023-07-25 12:25:53 GMT+2
192.168.1.161	A	3	netcom.netatmo.net	2023-07-25 12:35:58 GMT+2
192.168.1.161	A	3	netcom.netatmo.net	2023-07-25 12:46:04 GMT+2
192.168.1.161	A	3	netcom.netatmo.net	2023-07-25 12:56:09 GMT+2

Figure 5.4: Timeseries database

To validate these data points, we use Pi-hole's admin interface to check the query log for the Netatmo air quality monitor. Figure 5.5 shows a screenshot with an extract from the query log for the time frame that was examined, filtering by the IP address of the air quality monitor. Comparing this information with Table 5.1 and the data points in Figure 5.4 confirms that the monitoring job reliably queries Pi-hole's API, stores the data to the time series database, and creates an aggregation in the monitoring report table.

2023-07-25 12:56:09	A	netcom.netatmo.net	192.168.1.161	OK (cache)
2023-07-25 12:46:04	A	netcom.netatmo.net	192.168.1.161	OK (answered by 1.1.1.1#53)
2023-07-25 12:35:58	A	netcom.netatmo.net	192.168.1.161	OK (cache)
2023-07-25 12:25:53	A	netcom.netatmo.net	192.168.1.161	OK (answered by 1.1.1.1#53)
2023-07-25 12:15:47	A	netcom.netatmo.net	192.168.1.161	OK (cache)

Figure 5.5: Pihole query log

5.3.2 Policy Enforcement

In order to assess the policy enforcement scenario outlined in Section 5.2.2, the Philipps Hue Bridge device was added to the prototype application with a default policy of *allow all*. The subsequent monitoring job runs then detected all domains for which the device sent DNS requests to the Pi-hole DNS server. The data was then passed to the Policy Engine component as modeled in the sequence diagram in Figure 3.4 in Section 3.3.1.2. For each domain that did not have an associated policy in the database for the new device, the Policy Engine added a new policy with the default value.

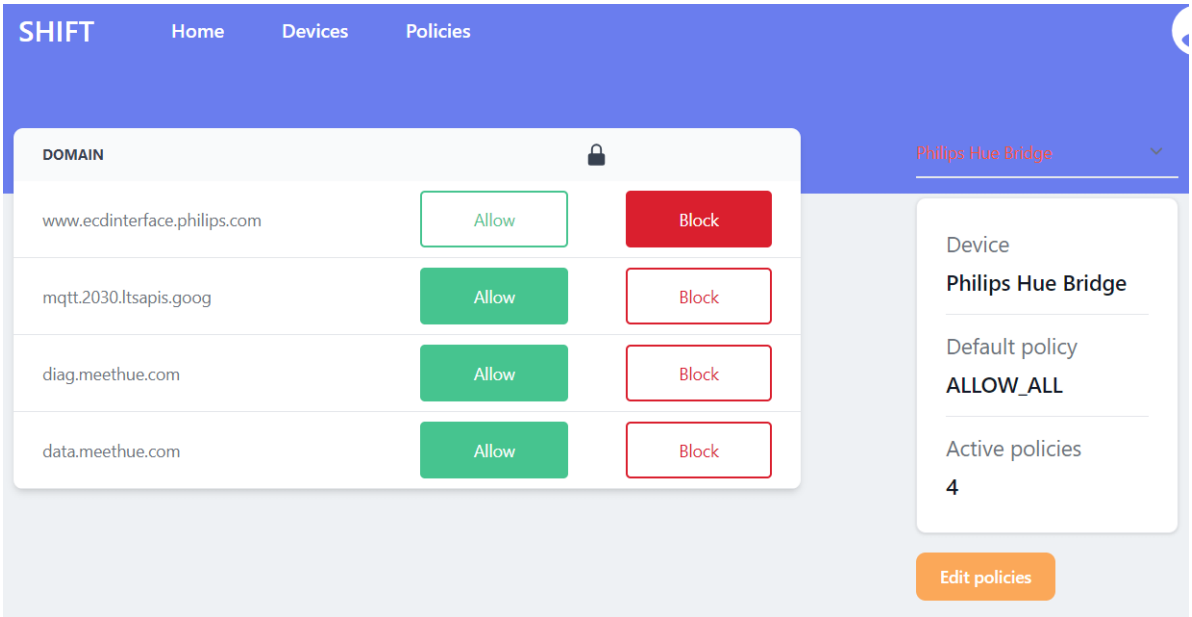


Figure 5.6: Policy dialog for Philips Hue Bridge

The section of a screenshot shown in Figure 5.6 lists the *policies* page of the prototype's UI. With the dropdown menu on the top right side, the desired device can be selected. Hence, the image shows all domains that have a policy linked to the Philips Hue Bridge. The Policy Engine has created policies for all of them with the default *allow* value. To

test both the allow and block cases, one policy was manually changed using the editing feature to *block* the associated domain.

To enforce these user-defined policies, the prototype's Policy Engine installs or updates entries in Pi-hole's domain list. A section of a screenshot shown in Figure 5.7 lists these entries in the domain list for our Pi-hole instance. The list includes all domains for which the prototype created policies after detecting them in a monitoring job run. It encompasses the domains for the Philips Hue Bridge from this scenario and a single one for the air quality monitor from the previous scenario. These policies were translated into domain list rules, either whitelisting the domain for an *allow* policy or blacklisting the domain for a *block* policy. In our test scenario, we have allowed all domains except for *www.ecdinterface.philips.com*, which was then successfully blacklisted in Pi-hole.

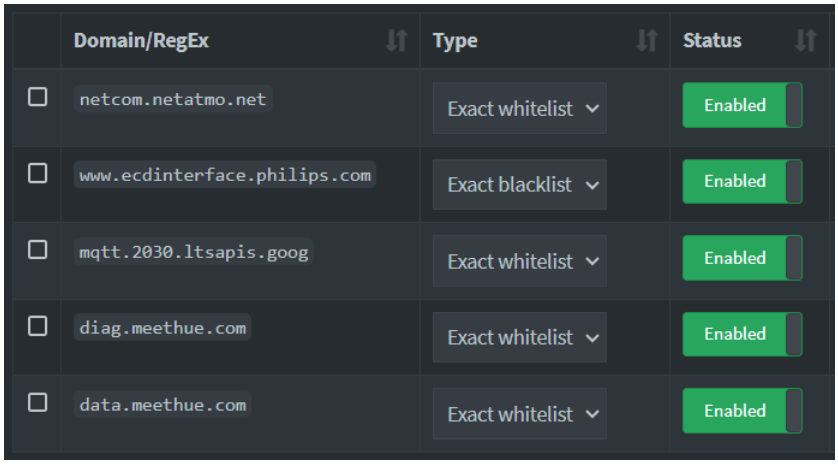


Figure 5.7: Pi-hole domain list

In this evaluation, we observed that the policies defined in the prototype application are consistently transferred to the Pi hole system. The DNS sinkhole functionality of Pi-hole then allows to block those domains that were blacklisted. On the basis of this, the enforcement of policies defined by users of the prototype is ensured. Pi-hole's query log for the blocked domain additionally confirms this, as can be seen in Figure 5.8. After switching the policy to block, the domain list entry in Pi-hole was updated to blacklist the domain, upon which the domain was successfully blocked in subsequent requests.

Time	Type	Domain	Client	Status
2023-07-25 21:26:12	A	www.ecdinterface.philips.com	192.168.1.20	Blocked (exact blacklist)
2023-07-25 21:16:11	A	www.ecdinterface.philips.com	192.168.1.20	Blocked (exact blacklist)
2023-07-25 21:06:11	AAAA	www.ecdinterface.philips.com	192.168.1.20	Blocked (exact blacklist)
2023-07-25 21:06:11	A	www.ecdinterface.philips.com	192.168.1.20	Blocked (exact blacklist)
2023-07-25 20:56:11	A	www.ecdinterface.philips.com	192.168.1.20	OK (answered by one.one.one.one#53)

Figure 5.8: Pi-hole query log

5.3.3 Weekly Notification

An initial step in testing the scenario described in Section 5.2.3 was to configure the SMTP connection to the email service responsible for delivering the notifications generated by the Notification Service component. For this purpose, a new account was created with the address `shift.info@gmx.ch`. GMX [23] was chosen as the email provider for this test, because they allow third party applications to utilize their service to send emails via SMTP. The necessary parameters to use the service for the notification feature consist of the mail server address, a port number, the account's username and password. These arguments were passed to the application through the environment variables defined in a `docker-compose.yml` file for the project. In addition to configuring a sender for the notifications, the scenario required a receiver for these emails. Therefore, a user with a valid email address was registered in the prototype application. All available smart home devices, that is, the air quality control monitor, the smart TV and the Philips Hue Bridge were added to the prototype to be included in the reporting.

The weekly email notification process is activated by a cron job that is set to run at a certain time. To make testing more flexible, the execution times for this job were changed in the crontab of the running Docker container. This had no effect on the results, since the task itself was not changed, and the data taken into account in each execution always included the preceding seven days up to the time of execution.

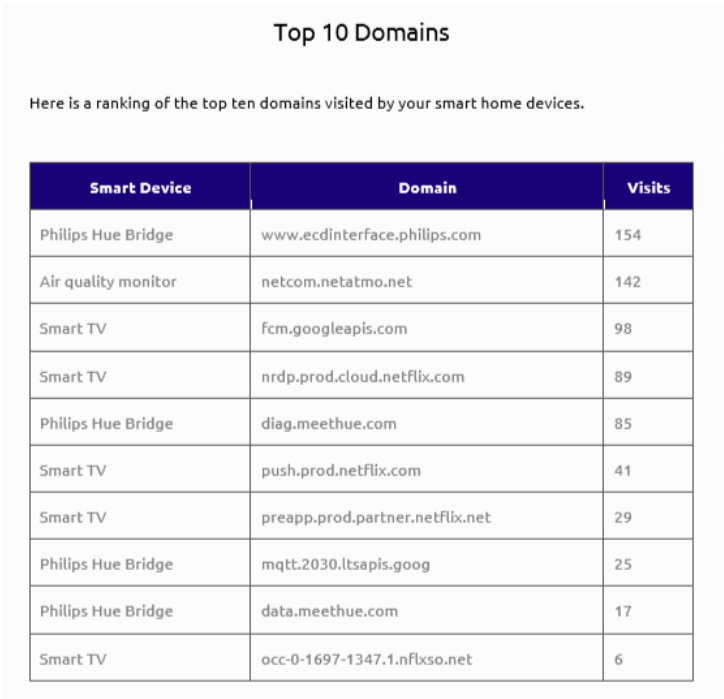
```
1 [2023-07-26 14:30:37 +0200] [1] [ERROR] Worker (pid:55) was sent SIGKILL!
2 Perhaps out of memory?
3 [2023-07-10 14:30:37 +0200] [119] [INFO] Booting worker with pid: 119
4 [2023-07-10 14:31:07 +0200] [1] [CRITICAL] WORKER TIMEOUT (pid:119)
5 [2023-07-10 14:31:08 +0200] [1] [ERROR] Worker (pid:119) was sent SIGKILL!
6 Perhaps out of memory?
7 [2023-07-10 14:31:08 +0200] [156] [INFO] Booting worker with pid: 156
```

Listing 15: Weekly notification log

In the initial run of the tests, we encountered some issues. Weekly email notifications were able to be generated and sent from our development environment. However, the executions initially failed in our productive environment on the Raspberry Pi platform. An excerpt from the log files, which captures the time of an execution of the weekly email notification process, is shown in Listing 15.

This reveals that the Gunicorn Web Server worker processes were timed out during processing and were consequently terminated before the task could be completed. As a result, no email reports were sent from the prototype. It was discovered that the default timeout for a worker process was set to 30 seconds. As this was not enough time for long running background tasks, this issue was resolved by increasing the request timeout to 180 seconds. Finally, emails were able to be sent automatically, as indicated by the following line of the log file of the associated cron job: `[2023-07-26 20:46:14,645] INFO in wsgi: Sent all weekly notifications`.

The resulting weekly email notification contains two graphics and a table visualizing the configured smart home devices' behavior over the past week. Figure 5.9 shows the table



Top 10 Domains

Here is a ranking of the top ten domains visited by your smart home devices.

Smart Device	Domain	Visits
Philips Hue Bridge	www.ecdinterface.philips.com	154
Air quality monitor	netcom.netatmo.net	142
Smart TV	fcm.googleapis.com	98
Smart TV	nrdp.prod.cloud.netflix.com	89
Philips Hue Bridge	diag.meethue.com	85
Smart TV	push.prod.netflix.com	41
Smart TV	preapp.prod.partner.netflix.net	29
Philips Hue Bridge	mqtt.2030.ltsapis.goog	25
Philips Hue Bridge	data.meethue.com	17
Smart TV	occ-0-1697-1347.1.nflxso.net	6

Figure 5.9: Weekly email notification - Domain table

included in the email from our test scenario. It displays the top ten domains visited by the three smart devices used in this test. The domains are ranked by the number of times the respective device successfully sent a DNS request to resolve the domain.

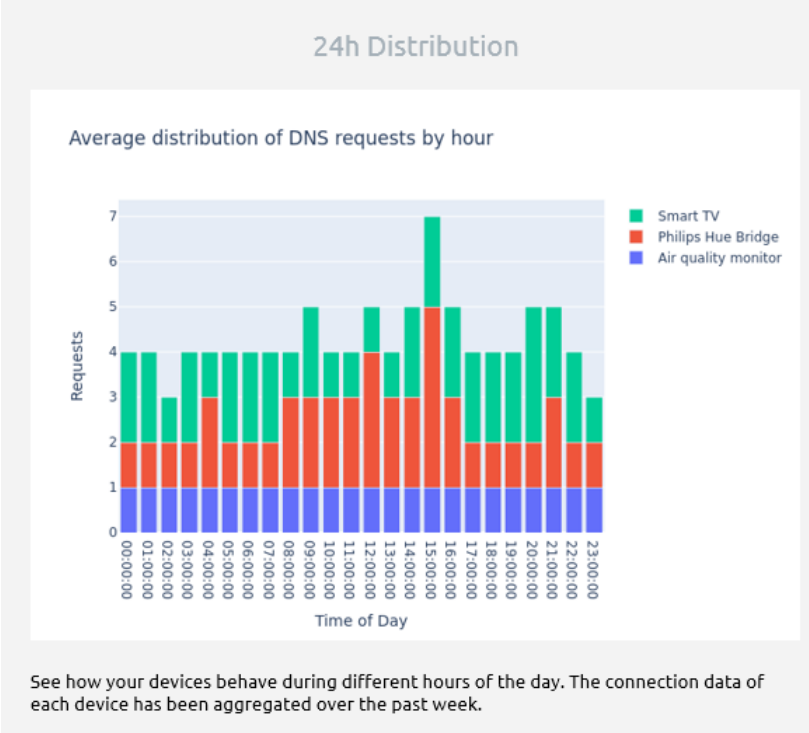


Figure 5.10: Weekly email notification - Stacked bar chart

The weekly report also contains a graph that shows the average number of DNS requests made by each device at each hour of the day. For this purpose, a stacked bar chart was attached to the email and inlined into the layout. The plot resulting from our test run in this scenario is shown in Figure 5.10. This illustration aims to make the behavior of smart home devices during the course of a regular day transparent to the user. An interesting point, which can be seen in the graph from our test run, is that traffic appears to be mostly unrelated to device usage. Although the number of DNS queries increases slightly during the day, it remains relatively constant at night when the devices are not actively used.

5.3.4 Smart Device Operability

We conducted several experiments with the previously added smart home devices from Netatmo and Philips Hue to verify their functionality under policy constraints as described in Scenario 5.2.4. In a first step, all policies for both devices were set to *block* the assiociated domains, as demonstrated by the policy definitions for the Philips Hue Bridge depicted in Figure 5.11.

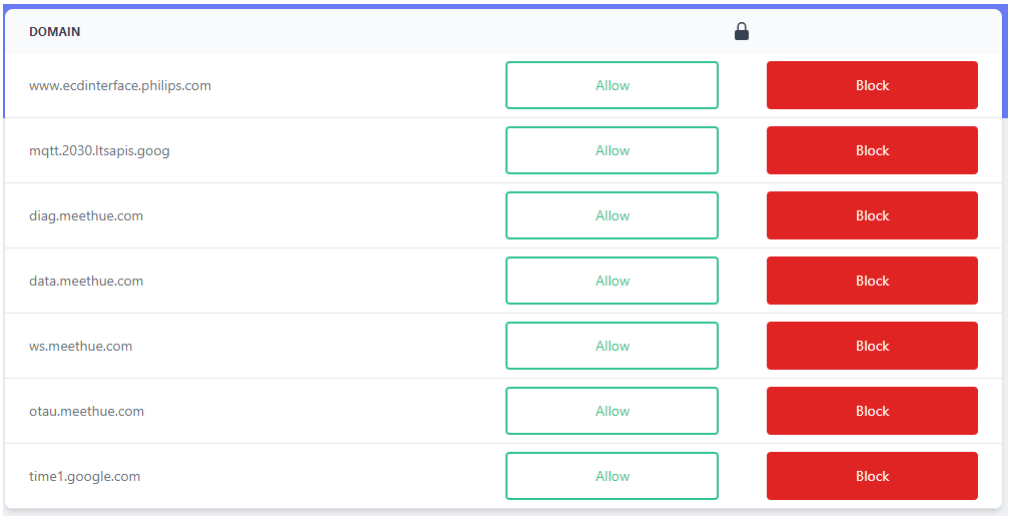


Figure 5.11: Philips Hue policies - All domains blocked

After successfully installing the blocking policies, we tested the data access and device controls for both devices using their Home Assistant integrations, as well as their respective mobile applications. The results of the tests performed are summarized in Table 5.2.

	Philips Hue	**Netatmo**
Home Assistant	Fully operational	Not operational
Mobile App	Operational over Wi-Fi	Not operational

Table 5.2: Operability in total blocking configuration

As expected, Philips Hue lighting was found to be controllable via the Home Assistant platform even without a cloud connection. In the case of the Hue app for Android, we

have observed no restriction on functionality as long as the smartphone is on the same network. However, remote access via the cellular network is no longer possible in this configuration.

In contrast to Philips Hue's Bridge, the Netatmo air quality monitor only calls up a single domain. This is why the device only possessed a single policy that could be set to block. As expected, blocking this domain resulted in no status updates being sent to the cloud server. However, both the Netatmo Home Assistant integration and the manufacturer's Android app *Home Coach* obtain their data from the cloud service. Therefore, the blocking policy rendered the air quality monitor unusable.

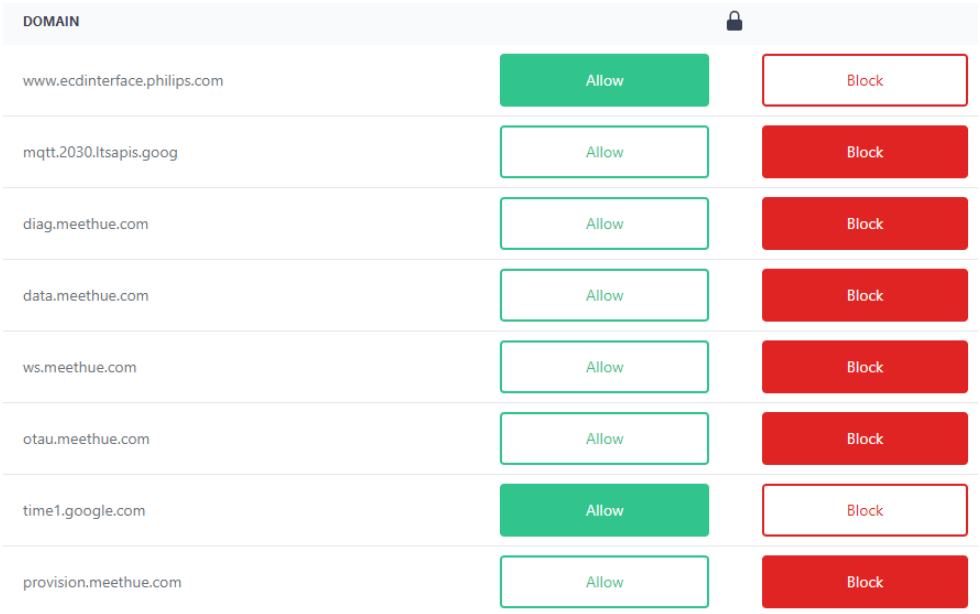


Figure 5.12: Philips Hue policies - Minimal configuration

The next step was to try to determine a minimum configuration for the Philips Hue Bridge. That is, achieving a combination of policies that allow as few domains as possible, while still enabling remote control via the Hue app. Through experimentation, we found that we technically only needed to change one of the existing policies from *block* to *allow* to achieve the desired behavior. Figure 5.12 shows this minimal configuration, allowing us to block most of the domains, while retaining the ability to control the lighting remotely. The domain *www.ecdinterface.philips.com* enables the remote control functionality. Additionally, the policy for *time1.google.com* was changed to allow time synchronization, thus enabling time-based automation.

During our experiments, we made an interesting observation. By setting blocking policies for the Philips Hue Bridge, there was a significant increase in the number of DNS queries that emanated from the device. This effect was not expected. The graph in Figure 5.13 illustrates the discrepancy in requests between the fully blocking and fully allowing policy configurations. In this screenshot of part of the prototype's dashboard, we can observe the decrease in DNS queries from the Hue Bridge after changing all policies from *block* to *allow*. This demonstrates that the number of queries in the setting with all domains blocked is up to 20 times higher than in the setting with none blocked.

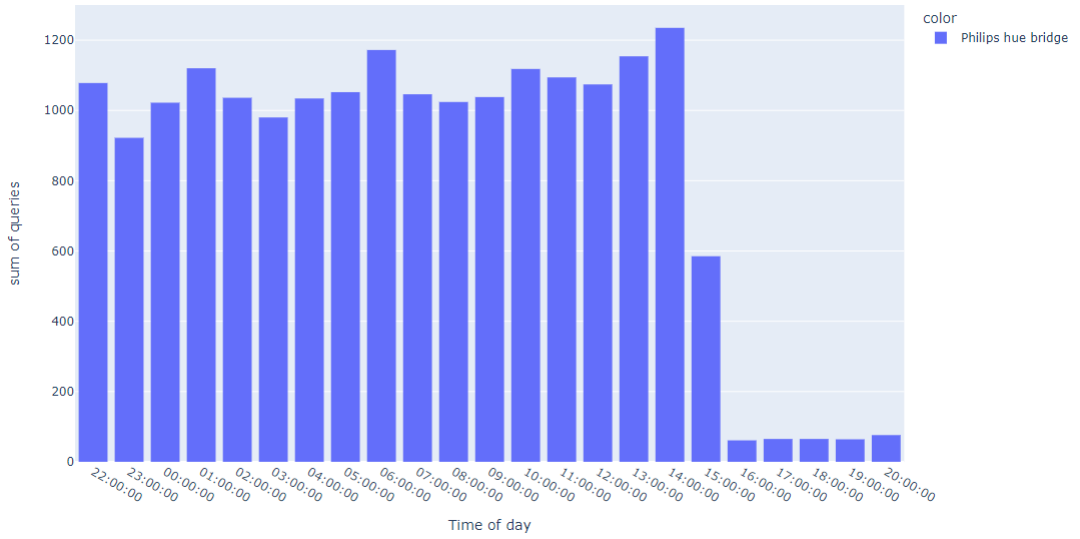


Figure 5.13: Fully blocking vs. fully allowing

5.3.5 Performance Evaluation

In order to measure the resource utilization of each container running on the platform, the bash script displayed in Listing 23 of Appendix B was prepared and deployed on the Raspberry Pi. The script invokes the `docker stats` command from the Docker command line interface every three minutes, reformats the output, and writes the result to a file in comma-separated value (CSV) format.

To obtain a baseline of the performance metrics, the script described above was executed over a period of twelve hours during regular operations. For this initial assessment, the following containers were running continuously: the prototype flask application, the InfluxDB time series database, the Nginx reverse proxy server, and the Home Assistant instance. Upon termination of the test script, the collected data were loaded into a Python script, cleaned, and outliers were removed using the z-score method. Following this, several plots were created to illustrate the data set.

The upper plot in Figure 5.14 shows the relative CPU usage of each of the containerized applications. It can be seen from the graph that apart from a few spikes in the course of the Pi-hole instance, the CPU load is in a relatively low range. On the basis of this observation, the Raspberry Pi's processing power seems to be sufficient to run our application in this configuration. Furthermore, no performance problems were noticed while using the application during this period.

Memory utilization of the docker containers during the baseline measurement is shown in the second plot of Figure 5.14. The plot displays a dedicated line for the relative memory usage of each container, as well as a line representing the sum over all container instances. In contrast to the values observed for CPU usage, the total memory loads on the system were relatively high. The largest contributors to these values were the Home Assistant instance and the prototype container. A constant load between 10 and 15% caused by the prototype application is rather at the high end for a lightweight application. However,

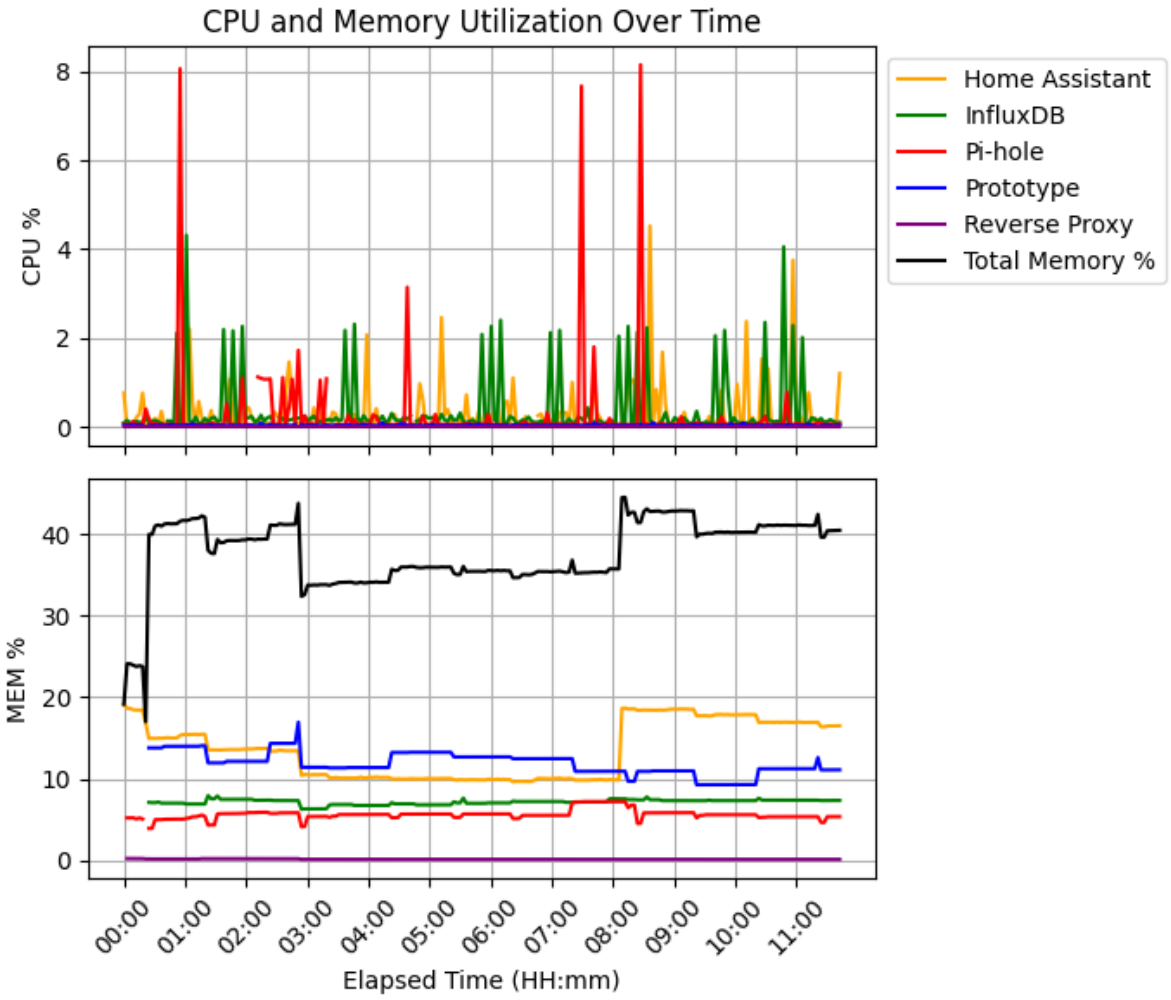


Figure 5.14: Relative CPU and memory usage of docker containers

there were also no major fluctuations in the course of the measurements, indicating a certain level of stability.

During the experiments on the productive Raspberry Pi system, there were situations where the available random access memory (RAM) on the platform was not sufficient. This led to several events in which the system became unresponsive and had to be restarted. To counteract this, a swap file with a higher capacity was configured. This significantly reduced the load on the RAM. Although access to the virtual memory of the swap file is much slower, no major performance losses were observed during the testing.

To get a more complete picture of memory utilization, additional data was collected. In addition to the relative values for the individual containers, the total load on the Raspberry Pi was measured at the same time. The plot in Figure 5.15 shows the course of RAM and swap utilization as well as their respective upper limits. To illustrate the share of the running containers in the RAM load, the aggregated value over all instances is also shown. Since it is difficult to associate the swap memory used with a specific process, the contribution of the Docker containers could not be shown here.

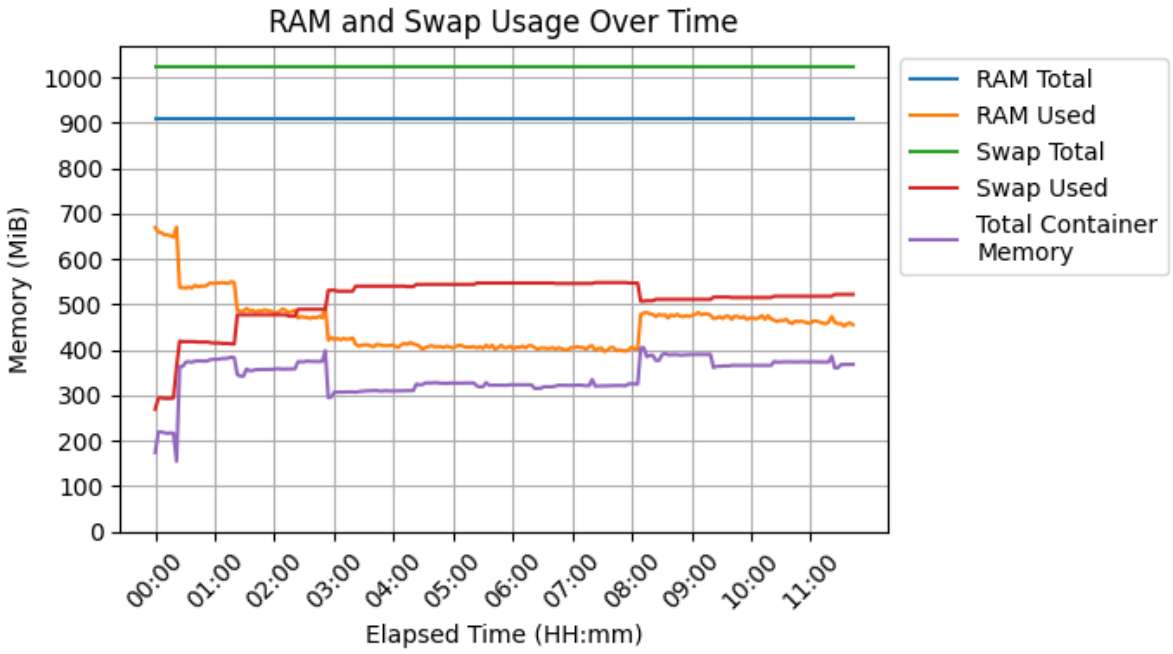


Figure 5.15: Absolute system memory and swap utilization

5.4 Discussion

The core of this thesis includes the design and implementation of a prototype. This prototype permits users to manage smart home devices and define simple policies to allow or block domains, which is enforced by integrating a DNS sinkhole. Additionally, users are kept informed about the behavior of their devices. The evaluation results indicate that the goals of this thesis have been achieved to a satisfactory extent.

In the context of the evaluation phase of this thesis, a series of experiments were conducted to assess the viability of specific features of the prototype implementation and to collect performance-related data. The evaluation presented in this chapter is structured into experiment scenarios that encompass different aspects of the prototype system. The initial three scenarios outlined in Section 5.2 focused on monitoring, policy enforcement, and email notifications, which are the primary features of the application. These scenarios aimed to evaluate and demonstrate the respective functionalities in a productive setting, that is, in a live smart home environment. The purpose of the fourth scenario was to then test possible limitations in the usability of smart home devices due to policy enforcement. The last scenario was designed to assess the suitability of the developed solution for the Raspberry Pi platform based on the given resource constraints.

The first test case described in Section 5.2.1 sought to validate the prototype's ability to monitor DNS requests of smart home devices. The execution of the steps necessary to test this scenario was documented in Section 5.3.1. It indicates that the prototype allows users to add new smart home devices via the user interface. Furthermore, the results reveal that monitoring jobs, which receive data about DNS queries of the devices via the Pi-hole API, are automatically executed at regular intervals. Finally, it shows that the

collected data points are reliably stored in a time series database. These results confirm the fulfillment of the functional requirements 1, 4 and 5 defined in Section 3.2.1.

In this thesis, the implementation of the prototype involved Pi-hole as the only data source for the network behavior of smart home devices. For this reason, the use of a time series database is not strictly necessary. However, with the vision of a more comprehensive system using multiple data sources in mind, the use of a time series database makes sense. It facilitates the consolidation of the collected data in a uniform format and store it for later processing, for example, to build reports or create data visualizations. In the presented case, though, one is able to query the Pi-hole API at any given time for this data. It is not very useful to save the data twice at this stage. That is why the prototype was designed not to rely on this storage medium. Nevertheless, the feature was implemented - mainly as a proof-of-concept for an extended version - and can be activated via configuration.

The goal of the second experiment, as described in the second scenario 5.2.2, was to assess the capabilities of the system to set and enforce simple *allow/block* policies. These features have been implemented to meet functional requirements 8, 9 and 10 as defined in Section 3.2.1. The test results documented in Section 5.3.2 confirm that the prototype application meets the specified requirements. The documentation asserts that policies are automatically created for newly detected domains with the configured default values of the respective smart home devices. Furthermore, it was proven that users can easily make changes to policies created via the user interface to influence behavior. It was also verified that both the automatically generated policies and changes made by the user are propagated from the prototype to the Pi-hole system. Through the DNS sinkhole function of Pi-hole, user-defined blocking policies could be successfully enforced.

It can be argued that this protoype provides another interface to interact with Pi-hole. This is true to some extent, as the functionalities of the prototype rely heavily on it. Pi-hole even offers a much richer set of features and configurations, making it a powerful tool. However, it is not simple to use and geared more towards tech-savvy users. The prototype developed, on the other hand, is aimed at average smart home owners who want more transparency and control over their privacy. Furthermore, it is designed to allow the integration of additional mechanisms for enforcing privacy policies beyond the Pi-hole.

In contrast to Pi-hole, the prototype offers the function of automatically generated reports about the behavior of smart home devices, which are sent to the user via email. The test scenario outlined in Section 5.2.3 was designed to test this feature. The results presented in Section 5.3.3 affirm that the protoype is capable of generating a weekly report and sending it to the user as an email notification. The implementation of this feature meets the functional requirement number 7 defined in Section 3.2.1. Testing this scenario revealed some pitfalls that could subsequently be resolved. Nevertheless, these difficulties highlight the challenges of implementing a solution for a platform with limited resources.

In addition to the integrated dashboard in the Web UI, weekly email notifications provide an important source of information for users about the activities of their smart home devices. Therefore, they play a key role in achieving one of the main goals of this thesis. The statistics and visualizations contained in the notifications make the behavior of the

devices in the smart home observable, and thus create increased transparency regarding the impact on privacy.

In the fourth test scenario outlined in Section 5.2.4, the influence of policies on the usability of certain smart home devices was measured. A hub-based smart home environment powered by the Home Assistant platform as well as the native smart phone applications of the respective device manufacturers were utilized to perform experiments in this scenario. The goal was to determine how blocking certain domains would affect the ability of these tools to control devices and access their data.

The results presented in Section 5.3.4 illustrate that the method by which smart home device integrations are implemented is critical. It determines the extent to which communication with cloud servers can be influenced without severely restricting functionality. In most cases, this will be determined by the possibilities manufacturers offer developers to integrate their products into a platform like Home Assistant. The decisive factor here is whether communication with devices is supported in the local network or whether there is only a cloud interface. Moreover, it was found that the blocking policies on one of the test devices resulted in a surge in the number of DNS queries observed. This, in turn, raises new questions and problems. For instance, it would be essential to investigate the impact of this increased network traffic on the available bandwidth. Furthermore, this increase in requests distorts the statistics and data visualizations applied in the prototype to make the behavior of smart home devices observable. Since this effect was only noticed during the evaluation, it was not considered in the application design.

The final test scenario, described in Section 5.2.5, focuses on performance. Its objective was to determine whether the proposed system is suitable for operation on a SoC platform with limited resources. For this purpose, the CPU and memory utilization of the individual containerized applications were examined. Additionally, the total memory load on the hardware was measured.

The performance evaluation's results documented in 5.3.5 indicate that a SoC platform such as Raspberry Pi can effectively support the prototype together with the remaining containerized applications with satisfactory CPU performance and stable memory utilization. Implementing a swap file mitigated RAM-related issues, ensuring system responsiveness and reliability during testing.

Chapter 6

Final Considerations

In this concluding chapter, Section provides a summary of the work that was carried out in the course of this thesis 6.1. Section 6.2 then outlines the conclusions drawn from this work. This includes a consideration of whether the set goals have been achieved and which factors have contributed significantly to the outcome. In addition, obstacles encountered during the course of the work and how they were overcome are discussed. It also addresses the limitations of the presented results. Section 6.3 lists the contributions made through the work on this thesis. Finally, Section 6.4 discussess opportunities for future research and development based on the outcomes.

6.1 Summary

This thesis proposes a novel software system to improve privacy protection in smart homes. The system is designed to run on a system-on-chip (SoC) platform and to fit seamlessly into an existing smart home environment. The idea is to monitor the communication of smart home devices, make the resulting data transparent to residents, and restrict devices' access to the Internet according to user-defined policies. Different ways of collecting data and enforcing policies are possible, *e.g.* by integrating different interfaces and tools. The vision is to have an extensible framework that allows for a combination of several such mechanisms. To create a proof of concept for this proposal, a prototype was realized within the scope of this thesis. The approach chosen was to monitor smart home devices' DNS queries and enforce simple *allow/block* policies via a DNS sinkhole.

To establish the theoretical context for this thesis, the first step was to explore relevant fundamental concepts through a literature review. A definition of the term *smart home* was given and its significance as an application of the Internet of Things as well as its composition and characteristics were examined. Additionally, the security goals, vulnerabilities, and threats for smart home environments were analyzed. Of particular interest in the context of this work was the review of existing research on smart home privacy. The issues associated with gathering personal information were discussed, as well as the views and preferences of consumers. Moreover, in order to provide a basis for working

with Pi-hole, a brief overview of the functionalities of DNS systems and DNS sinkholes was compiled.

The literature review then progressed to the study of related work. The main focus was placed on existing research on the topic of privacy-enhancing technologies (PET). Different privacy preservation mechanisms were introduced and analyzed on the basis of technologies already developed or proposed in research papers. As a result of the insights gained, a summary comparison of these PETs was derived to obtain an overview of the state-of-the-art. This research has served as inspiration for the design of the proposed hub-centric data minimization PET. Finally, an overview of the hardware and software tools used for the realization of the practical part of this thesis was given.

Afterward, a documentation of the design of the proposed system is featured. This includes the description of an application scenario that abstracts the smart home environment into layers and shows how the proposed system is incorporated into it. The requirements for the implementation of the prototype were derived from the defined thesis goals and previous research. The design also contains a specification of the software architecture. The system comprises a frontend, a backend, and a database model, which in turn have been structured into individual components, each of which form a self-contained feature of the system. Its frontend is designed to include an administration interface and a dashboard. On the backend, a lightweight web server handles the main application logic, such as monitoring device communications and aggregating reports for user notifications. In addition, the surrounding systems with which the application interacts were specified. Most notably, it defines the Pi-hole integration, which enables the monitoring of DNS queries and the enforcement of blocking policies.

A prototype of the designed system was implemented to allow for validation of the general concept. The main part of the prototype was written in Python using the Flask framework to create a lightweight web application. Building the prototype's backend included the development of an integration component to communicate with Pi-hole, a monitor component to observe DNS queries via Pi-hole, and a policy engine to enforce user-defined policies. Furthermore, a weekly email notification service, a routes module to handle incoming HTTP requests, and a (data) models component for interactions with the respective databases were added. For the application's frontend, Flask's templating system was used to generate dynamically created HTML pages. In addition, custom Javascript was added to make these pages interactive. The layout and design of the user interface are powered by TailwindCSS and Flowbite. For the implementation of the dashboard of the UI, Plotly Dash was used to create interactive data visualizations. To streamline the development process, a continuous integration and delivery (CI/CD) infrastructure has been set up. It serves the purpose of managing the project's source code, building the application as a docker image and deploying it to the remote Docker Hub repository.

During the evaluation phase, the protoype was deployed in a live smart home environment. Experiments were conducted to assess the prototype and gather data on its viability. The evaluation included test scenarios focused on verifying the functionality of the monitoring, policy enforcement, and email notification features. Additionally, the operability of smart home devices was investigated under restricting policies. Therefore, different policy settings were configured to test whether they had an impact on device accessibility via

the Home Assistant platform or through manufacturer-specific mobile apps. Furthermore, the suitability of the solution for a resource-constrained SoC platform was examined by collecting and analyzing performance data in the productive deployment.

6.2 Conclusions

A prototype software system to enhance user privacy in smart home environments was successfully designed, implemented and evaluated, thereby meeting the goals that were outlined for this thesis. By observing their DNS queries, this prototype is able to monitor the network traffic of smart home devices. The collected data is processed to make the communication behavior of the devices transparent to users. Furthermore, the prototype allows users to define simple *allow/block* policies to control the devices' access.

The integration of Pi-hole has made it possible to implement monitoring jobs that regularly fetch data about DNS requests from smart home devices. Furthermore, its DNS sinkhole feature has permitted to successfully enforce user-defined policies in order to block specific domains. One limitation that must be mentioned here is that only traditional DNS queries can be captured via this integration. DNS over HTTPS (DoH) requests, for example, do not use Pi-hole as a DNS server and therefore cannot be blocked this way.

To create more transparency with respect to the communication of smart home devices, a dashboard was added to the user interface. In addition, a weekly report is generated and sent to the user via email. Both features include data visualizations and statistics on the behavior of these devices. Through these channels, useful information can be provided to the user, revealing insights into the privacy implications of using these technologies.

The prototype also met the prerequisite that the proposed system must operate on a SoC platform. Performance evaluations carried out on the Raspberry Pi hardware confirm this. Since memory swapping had to be applied to ensure stable operation, there may still be potential for optimization. Given that the heaviest load was caused by Home Assistant, it might be reasonable to run this part on a separate hardware. However, more recent versions of the Raspberry Pi offer increased memory capacities, which could render this step obsolete.

Tests conducted with Home Assistant have shown that the developed prototype can be used to disconnect the cloud connections of individual devices in a hub-centric smart home. The mode of communication used between the smart home platform and the individual device largely determines whether restricting its access to the Internet limits its functionality. For platform integrations that use local communication, there should be no major limitations. Integrations that are solely cloud-based are inherently dependent on this connection. However, in the experiments conducted, it was observed that for certain devices, not all of the frequently accessed domains were mandatory for the functionality of cloud-driven services such as remote control.

Whilst the prototype successfully accomplished the set objectives, certain challenges were encountered during its development and evaluation process. One such challenge was that

the current Pi-hole HTTP API does not provide the ability to add or modify whitelist or blocklist entries. A new Representational State Transfer (REST) API is under development and would provide this capability, but is not stable enough at this time. This difficulty was overcome by accessing Pi-hole's database directly for this task. Another challenge was that features that had been developed and successfully tested in the local environment did not behave in the same way when deployed on the Raspberry Pi platform. Often, these difficulties were related to missing configurations or permissions in the Docker base image. Moreover, a different web server and an additional reverse proxy were used in the production setting, which also presented pitfalls. Since the process of manually building and deploying the prototype was cumbersome, new builds were only rolled out at irregular intervals during the initial development phase. Therefore, problems were often detected later. However, by adding a build and deployment pipeline, this process could be automated. This reduced the intervals between feature implementation and testing on the target system, resulting in a more efficient development process.

Lastly, the scope of this thesis initially included not only the integration of Pi-hole but also an integration of Home Assistant via an API. However, during the discovery phase of the work, it was found that this was not necessary to achieve the objectives of the thesis. Nevertheless, Home Assistant was chosen as the platform for our smart home environment in our application scenario. Thanks to its offline capability and local smart device integrations, the platform was ideally suited to complement the prototype's setup.

6.3 Contributions

The following contributions were made within the framework of this thesis:

- Design and implementation of a prototype software system. Its features include monitoring DNS queries from smart home devices, and using the collected data to inform users about the communication behavior of devices. Furthermore, the prototype can enforce user-defined policies to allow or block certain domains.

- Evaluation of the prototype in a live smart home environment to assess the viability of the examined approach to enhance residents' privacy.

- Releasing the source code of the prototype to the public and making it accessible on Github. Publishing the project under an open source license is intended to invite further research and collaborative development within the open source community [14].

- The prototype Docker image is made publicly available on Docker Hub, facilitating easy deployment for test and research purposes [13].

6.4 Future Work

This project has successfully developed a prototype aiming to lay the foundation for a more comprehensive privacy protection solution for smart home environments. However, to achieve a more mature system suitable for wider use, further research and development is required.

First, it would be essential to conduct more comprehensive testing with a diverse selection of smart home devices. The number of different devices available for experimentation in this thesis was limited. Therefore, the results may not be fully representative, given the complexity and variety of smart home configurations.

To improve the user experience, a feature should be added to automatically detect smart home devices on the network. This could be implemented through ARP scanning. The monitoring and policy enforcement mechanisms of the system could be expanded by implementing alternative methods. For example, additional network monitoring techniques, such as ARP spoofing and deep packet inspection, could be used to get a more complete overview of communications inside a smart home. A valid approach would be to integrate additional existing tools that already offer these capabilities through APIs.

To serve different use cases and a wider range of user preferences, the system could furthermore be extended to support more sophisticated policy types. It could even be considered to incorporate machine learning to automatically suggest tailored policies based on device behavior and general user preferences. Additionally, further research could aim to identify the type of data collected and transmitted by smart home devices. Integrating this information into the system could greatly increase transparency and help users make informed decisions. Lastly, to increase the accessibility and usability of the system, a Home Assistant plugin could be developed to enable direct interaction with the system on the Home Assistant platform. Having most of the functions of the smart home in one place could greatly improve the user experience.

Bibliography

[1] Ado Adamou Abba Ari et al. "Enabling privacy and security in Cloud of Things: Architecture, applications, security & privacy challenges". In: *Applied Computing and Informatics* (July 2020).

[2] Muhammad Raisul Alam, Mamun Bin Ibne Reaz, and Mohd Alauddin Mohd Ali. "A Review of Smart Homes-Past, Present, and Future". In: *IEEE Transactions on Systems, Man, and Cybernetics, Part C (Applications and Reviews)* 42.6 (Apr. 2012), pp. 1190–1203.

[3] Amazon.com Inc. *Understand Smart Home Security Skills*. 2023. URL: `https://developer.amazon.com/en-US/docs/alexa/device-apis/overview-smart-home-security.html` (visited on June 5, 2023).

[4] Apple Inc. *Developing Apps and Accessories for the Home*. 2023. URL: `https://developer.apple.com/apple-home/` (visited on Aug. 8, 2023).

[5] Noah Apthorpe et al. "Keeping the Smart Home Private with Smart(er) IoT Traffic Shaping". In: *Proceedings on Privacy Enhancing Technologies* 2019.3 (2019), pp. 128–148.

[6] Nazmiye Balta-Ozkan et al. "The development of smart homes market in the UK". In: *Energy* 60 (Oct. 2013), pp. 361–372.

[7] Guy Bruneau. *DNS Sinkhole*. Tech. rep. SANS Institute, Aug. 2010.

[8] Haotian Chi et al. *PFirewall: Semantics-Aware Customizable Data Flow Control for Smart Home Privacy Protection*. 2021. URL: `https://arxiv.org/abs/2101.10522`.

[9] European Union Agency for Cybersecurity. *DNS Sinkhole - ENISA*. URL: `https://www.enisa.europa.eu/topics/incident-response/glossary/dns-sinkhole` (visited on Apr. 3, 2023).

[10] Docker, Inc. *Docker: Accelerated, Containerized Application Development*. URL: `https://www.docker.com/` (visited on July 15, 2023).

[11] P.J. Eby. *PEP 3333 - Python Web Server Gateway Interface v1.0.1*. Sept. 2010. URL: `https://peps.python.org/pep-3333/` (visited on July 10, 2023).

[12] eLinux.org. *RPi HardwareHistory*. Jan. 2023. URL: `https://elinux.org/RPi_HardwareHistory` (visited on July 20, 2023).

[13] Elliott Wallace. *shipp - Dockerhub*. URL: `https://hub.docker.com/r/elliottwallace/shipp` (visited on Aug. 9, 2023).

[14] Elliott Wallace. *shipp - Smart Home Integrated Privacy Protection*. URL: `https://github.com/elduwa/shipp` (visited on Aug. 9, 2023).

[15] Müge Fazlioglu. *IAPP privacy and Consumer Trust Report - Executive Summary*. Mar. 2023. URL: `https://iapp.org/resources/article/privacy-and-consumer-trust-summary/` (visited on Apr. 20, 2023).

[16] Python Software Foundation. *Welcome to Python.org*. 2023. URL: https://www.python.org/ (visited on July 10, 2023).

[17] Ned Freed and Dr. Nathaniel S. Borenstein. *Multipurpose Internet Mail Extensions (MIME) Part Two: Media Types*. RFC 2046. Nov. 1996. URL: https://www.rfc-editor.org/info/rfc2046.

[18] Mario Frustaci et al. "Evaluating Critical Security Issues of the IoT World: Present and Future Challenges". In: *IEEE Internet of Things Journal* 5.4 (Oct. 2018), pp. 2483–2495.

[19] Dimitris Geneiatakis et al. "Security and privacy issues for an IoT based smart home". In: *2017 40th International Convention on Information and Communication Technology, Electronics and Microelectronics (MIPRO)*. May 2017, pp. 1292–1297.

[20] George Wright. *Amazon to pay $25m over child privacy violations*. July 2023. URL: https://www.bbc.com/news/technology-65772154 (visited on Aug. 6, 2023).

[21] Git. *Git*. URL: https://git-scm.com/ (visited on July 15, 2023).

[22] GitHub. *GitHub*. URL: https://github.com/ (visited on July 15, 2023).

[23] GMX. *GMX E-Mail*. URL: https://www.gmx.ch/mail/ (visited on July 25, 2023).

[24] Google. *A Helpful Home is a Private Home*. URL: https://safety.google/nest/ (visited on June 5, 2023).

[25] Kirsten Gram-Hanssen and Sarah J. Darby. ""Home is where the smart is"? Evaluating smart home research and approaches against the concept of home". In: *Energy Research & Social Science* 37 (Mar. 2018), pp. 94–101.

[26] Jayavardhana Gubbi et al. "Internet of Things (IoT): A vision, architectural elements, and future directions". In: *Future generation computer systems* 29.7 (Sept. 2013), pp. 1645–1660.

[27] Kamal Gulati et al. "A review paper on wireless sensor network techniques in Internet of Things (IoT)". In: *Materials Today: Proceedings* 51 (2022), pp. 161–165.

[28] Gunicorn. *Gunicorn*. URL: https://gunicorn.org/ (visited on July 20, 2023).

[29] Badis Hammi et al. "Survey on smart homes: Vulnerabilities, risks, and countermeasures". In: *Computers & Security* 117, 102677 (June 2022).

[30] Jin-Hee Han, YongSung Jeon, and JeongNyeo Kim. "Security considerations for secure and trustworthy smart home system in the IoT environment". In: *2015 International Conference on Information and Communication Technology Convergence (ICTC)*. Oct. 2015, pp. 1116–1118.

[31] Home Assistant. *Awaken Your Home*. 2023. URL: https://www.home-assistant.io/ (visited on Aug. 8, 2023).

[32] Danny Yuxing Huang et al. "IoT Inspector: Crowdsourcing Labeled Network Traffic from Smart Home Devices at Scale". In: *Proc. ACM Interact. Mob. Wearable Ubiquitous Technol.* 4.2 (June 2020).

[33] Fortune Business Insights. *Smart Home Market Size, Share & COVID-19 Impact Analysis, By Device Type (Safety and Security Devices, Energy and Water Control, Climate Control, Lighting Control, Consumer Electronics), By Housing Type (Multifamily Dwelling, Single Family Dwelling), and Regional Forecast, 2023-2030*. July 2023. URL: https://www.fortunebusinessinsights.com/industry-reports/smart-home-market-101900 (visited on July 27, 2023).

[34] Haojian Jin et al. "Peekaboo: A Hub-Based Approach to Enable Transparency in Data Processing within Smart Homes". In: *2022 IEEE Symposium on Security and Privacy (SP)*. May 2022, pp. 303–320.

[35] Andrei Kazlouski, Thomas Marchioro, and Evangelos Markatos. "I Just Wanted to Track My Steps! Blocking Unwanted Traffic of Fitbit Devices". In: *Proceedings of the 12th International Conference on the Internet of Things*. IoT '22. Nov. 2023, pp. 96–103.

[36] Felix Klement, Henrich C. Pöhls, and Korbinian Spielvogel. "Towards privacy-preserving local monitoring and evaluation of network traffic from IoT devices and corresponding mobile phone applications". In: *2020 Global Internet of Things Summit (GIoTS)*. IEEE. June 2020, pp. 1–6.

[37] Dr. John C. Klensin. *Simple Mail Transfer Protocol*. RFC 5321. Oct. 2008. URL: https://www.rfc-editor.org/info/rfc5321.

[38] Neil Klingensmith, Younghyun Kim, and Suman Banerjee. "A Hypervisor-Based Privacy Agent for Mobile and IoT Systems". In: *Proceedings of the 20th International Workshop on Mobile Computing Systems and Applications*. HotMobile '19. Feb. 2019, pp. 21–26.

[39] Nikos Komninos, Eleni Philippou, and Andreas Pitsillides. "Survey in smart grid and smart home security: Issues, challenges and countermeasures". In: *IEEE Communications Surveys & Tutorials* 16.4 (Apr. 2014), pp. 1933–1954.

[40] Grafana Labs. *Grafana Labs - Open Source*. 2023. URL: https://grafana.com/oss/ (visited on May 8, 2023).

[41] Josephine Lau, Benjamin Zimmerman, and Florian Schaub. "Alexa, Are You Listening? Privacy Perceptions, Concerns and Privacy-Seeking Behaviors with Smart Speakers". In: *Proc. ACM Hum.-Comput. Interact.* 2.CSCW (Nov. 2018), pp. 1–31.

[42] Wenda Li et al. "Motivations, barriers and risks of smart home adoption: From systematic literature review to conceptual framework". In: *Energy Research & Social Science* 80, 102211 (Oct. 2021).

[43] Mailjet. *MJML - The responsive email framework*. URL: https://mjml.io/ (visited on July 13, 2023).

[44] Anna Maria Mandalari et al. "Blocking without Breaking: Identification and Mitigation of non-essential IoT Traffic". In: *Proceedings on Privacy Enhancing Technologies* 2021.4 (2021), pp. 369–388.

[45] P. Mockapetris and K. J. Dunlap. "Development of the Domain Name System". In: *Symposium Proceedings on Communications Architectures and Protocols*. SIGCOMM '88. Aug. 1988, pp. 123–133.

[46] Hooman Mohajeri Moghaddam et al. "Watching You Watch: The Tracking Ecosystem of Over-the-Top TV Streaming Devices". In: *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*. CCS '19. Nov. 2019, pp. 131–147.

[47] Nataliia Neshenko et al. "Demystifying IoT Security: An Exhaustive Survey on IoT Vulnerabilities and a First Empirical Look on Internet-Scale IoT Exploitations". In: *IEEE Communications Surveys & Tutorials* 21.3 (Apr. 2019), pp. 2702–2733.

[48] *Netatmo - Home Assistant*. URL: https://www.home-assistant.io/integrations/netatmo/ (visited on July 29, 2023).

[49] Netatmo (Legrand). *Smart Indoor Air Quality Monitor | Netatmo*. URL: https://www.netatmo.com/smart-indoor-air-quality-monitor (visited on July 20, 2023).

[50] Lily Hay Newman. *Hacker Lexicon: What Is Sinkholing?* WIRED. Jan. 2018. URL: https://www.wired.com/story/what-is-sinkholing/ (visited on Apr. 3, 2023).

[51] npm, Inc. *npm*. URL: https://www.npmjs.com/ (visited on July 15, 2023).
[52] OpenAPI Initiative. *OpenAPI Specification v3.1.0*. Feb. 2021. URL: https://spec.openapis.org/oas/latest.html (visited on July 12, 2023).
[53] Pallets. *Design Decisions in Flask - Flask Documentation (2.3.x)*. 2023. URL: https://flask.palletsprojects.com/en/2.3.x/design/ (visited on July 10, 2023).
[54] Pallets. *Jinja - Jinja Documentation (3.1.x)*. 2023. URL: https://jinja.palletsprojects.com/en/3.1.x/ (visited on July 13, 2023).
[55] Pallets. *Welcome to Flask - Flask Documentation (2.3.x)*. 2023. URL: https://flask.palletsprojects.com/en/2.3.x/ (visited on July 10, 2023).
[56] Parcel. *Parcel*. URL: https://parceljs.org/ (visited on July 15, 2023).
[57] European Parliament and Council of the European Union. *Regulation (EU) 2016/679 of the European Parliament and of the Council of 27 April 2016 on the protection of natural persons with regard to the processing of personal data and on the free movement of such data, and repealing Directive 95/46/EC (General Data Protection Regulation) (Text with EEA relevance)*. Apr. 27, 2016. URL: https://data.europa.eu/eli/reg/2016/679/oj (visited on Aug. 1, 2023).
[58] Philips Hue. *Hue Bridge*. URL: https://www.philips-hue.com/en-my/p/hue-bridge/8719514342644 (visited on July 20, 2023).
[59] Philips Hue. *Smart lighting*. URL: https://www.philips-hue.com/en-my (visited on July 20, 2023).
[60] *Philips Hue - Home Assistant*. URL: https://www.home-assistant.io/integrations/hue/ (visited on July 29, 2023).
[61] Pi-hole. *Pi-hole - A black hole for Internet advertisements*. URL: https://github.com/pi-hole (visited on July 12, 2023).
[62] Pi-hole. *Pi-hole: Network-wide Ad Blocking*. 2023. URL: https://pi-hole.net/ (visited on Feb. 20, 2023).
[63] *pip - The Python Package Installer*. URL: https://github.com/pypa/pip (visited on July 15, 2023).
[64] Plotly, Inc. *plotly/plotly.py: The interactive graphing library for Python*. URL: https://github.com/plotly/plotly.py (visited on July 15, 2023).
[65] Python Software Foundation. *email - An email and MIME handling package*. July 2023. URL: https://docs.python.org/3.11/library/email.html (visited on July 15, 2023).
[66] Python Software Foundation. *smtplib - SMTP protocol client*. July 2023. URL: https://docs.python.org/3.11/library/smtplib.html (visited on July 15, 2023).
[67] Python Software Foundation. *venv - Creation of virtual environments*. July 2023. URL: https://docs.python.org/3/library/venv.html (visited on July 15, 2023).
[68] Raspberry Pi. *Raspberry Pi 3 Model B+*. URL: https://www.raspberrypi.com/products/raspberry-pi-3-model-b-plus/ (visited on July 20, 2023).
[69] Reuters. *Amazon's Ring doorbell was used to spy on customers, FTC says in privacy case*. May 2023. URL: https://www.theguardian.com/technology/2023/may/31/amazon-ring-doorbell-spying-ftc (visited on Aug. 6, 2023).
[70] Biljana L. Risteska Stojkoska and Kire V. Trivodaliev. "A review of Internet of Things for smart home: Challenges and solutions". In: *Journal of Cleaner Production* 140.3 (Jan. 2017), pp. 1454–1464.

[71] Leong Yee Rock, Farzana Parveen Tajudeen, and Yeong Wai Chung. "Usage and impact of the internet-of-things-based smart home technology: a quality-of-life perspective". In: *Universal Access in the Information Society* (Nov. 2022).

[72] STRONG. *LEAP-S1*. URL: `https://uk.strong-eu.com/products/ip-tv-receivers/leap-s1/` (visited on July 20, 2023).

[73] Madiha Tabassum, Tomasz Kosiński, and Heather Richter Lipford. ""I Don't Own the Data": End User Perceptions of Smart Home Device Data Practices and Risks". In: *Proceedings of the Fifteenth USENIX Conference on Usable Privacy and Security*. SOUPS'19. Aug. 2019, pp. 435–450.

[74] M. Vimalkumar et al. "'Okay google, what about my privacy?': User's privacy perceptions and acceptance of voice based digital assistants". In: *Computers in Human Behavior* 120, 106763 (July 2021).

[75] Igor Zavalyshyn, Nuno O Duarte, and Nuno Santos. "HomePad: A Privacy-Aware Smart Hub for Home Environments". In: *2018 IEEE/ACM Symposium on Edge Computing (SEC)*. Oct. 2018, pp. 58–73.

[76] Igor Zavalyshyn et al. "SoK: Privacy-enhancing Smart Home Hubs". In: *Proceedings on Privacy Enhancing Technologies* 2022.4 (2022), pp. 24–43.

[77] Zyxel. *AX7501-B SERIES*. URL: `https://service-provider.zyxel.com/global/en/products/fiber-oltsonts/10g-active-fiber/hgus/ax7501-b-series` (visited on July 20, 2023).

Abbreviations

ARP	Address Resolution Protocol
ARM	Advanced RISC Machine
API	Application Programming Interface
CI/CD	Continuous Integration and Continuous Delivery
CIA	Confidentiality, Integrity and Availability
CLI	Command-Line Interface
CPU	Central Processing Unit
CSS	Cascading Style Sheets
CSV	Comma-Separated Value
DDoS	Distributed Denial of Service
DNS	Domain Name System
DoH	DNS over HTTPS
DoS	Denial of Service
GDPR	General Data Protection Regulation
HTML	Hyper Text Markup Language
HTTP	Hypertext Transfer Protocol
HTTPS	Hypertext Transfer Protocol Secure
IETF	Internet Engineering Task Force
IoT	Internet of Things
IP	Internet Protocol
ISP	Internet Service Provider
LAN	Local Area Network
MAC	Media Access Control
MIME	Multipurpose Internet Mail Extension
MITM	Man-In-The-Middle-Attack
PEP	Python Enhancement Proposal
PET	Privacy Enhancing Technology
RAM	Random-Access Memory
REST	Representational State Transfer
SDRAM	Synchronous Dynamic Random-Access Memory
SMTP	Simple Mail Transfer Protocol
SoC	System-on-Chip
TLD	Top-Level Domain
UI	User Interface
UML	Unified Modeling Language
URL	Uniform Resource Locator

WSGI	Web Server Gateway Interface
WSN	Wireless Sensor Networks

List of Figures

3.1 Overview of application scenario 18
3.2 System Components 20
3.3 ER-Diagram 22
3.4 UML Sequence Diagram 23

5.1 Docker containers 36
5.2 Network configuration 37
5.3 Adding a new device 41
5.4 Timeseries database 42
5.5 Pihole query log 43
5.6 Policy dialog for Philips Hue Bridge 43
5.7 Pi-hole domain list 44
5.8 Pi-hole query log 44
5.9 Weekly email notification - Domain table 46
5.10 Weekly email notification - Stacked bar chart 46
5.11 Philips Hue policies - All domains blocked 47
5.12 Philips Hue policies - Minimal configuration 48
5.13 Fully blocking vs. fully allowing 49
5.14 Relative CPU and memory usage of docker containers 50
5.15 Absolute system memory and swap utilization 51

List of Tables

2.1 Related work 14

5.1 Report database table after 4 monitoring job executions 42

5.2 Operability in total blocking configuration 47

List of Listings

1 Snippet taken from the PiholeConsumer class . . . 26
2 The send_request method of the Query class . . . 27
3 Instatiation of PiholeConsumer in the Monitor component . . . 28
4 Pihole consumer method invocation . . . 28
5 Pi-hole API response for getAllQueries . . . 28
6 Data processing . . . 29
7 Fetch query data job . . . 30
8 Command line interface command . . . 30
9 Table in the weekly notification email layout in MJML . . . 31
10 Weekly email summary creation (Part 1) . . . 32
11 Weekly email summary creation (Part 2) . . . 32
12 Weekly email summary creation (Part 3) . . . 33
13 Weekly email summary creation (Part 4) . . . 33
14 Weekly email summary creation (Part 5) . . . 33
15 Weekly notification log . . . 45
16 Dockerfile - Builder stage . . . 70
17 Dockerfile - Final stage . . . 71
18 Github actions worklow - Build and deployment . . . 72
19 Docker compose file (Part 1) . . . 73
20 Docker compose file (Part 2) . . . 74
21 Docker compose file (Part 3) . . . 75
22 Pi-hole API excerpt . . . 76
23 Performance measuring script . . . 77

Appendix A

Contents of the Repository

The source code of the protoype developed during the course of this thesis is hosted in a Github repository at: `https://github.com/elduwa/shipp`.

The project has been tentatively named *shipp*, which stands for *Smart Home Integrated Privacy Protection*. Since this is still a prototype that is intended to be further developed by the open source community, the name is subject to change. Therefore, aside from this Appendix, the thesis does not explicitly reference the prototype's name.

The Github repository includes the following content:

- **Instructions:** The `README.md` file of the repository includes instructions for the installation and configuration of the prototype. In addition, some usage examples are presented. Moreover, directions for setting up a development environment are listed.
- **Web Application:** The source code for the prototype Flask web application is contained in the *app* directory, in addition to the `wsgi.py` and `config.py` modules in the root directory of the repository.
- **Docker:** The repository contains a Dockerfile that hosts the instructions for building the prototype's Docker image. It also provides a `docker-compose.yml` template for the set-up of Docker containers for all subsystems, including the prototype, Pi-hole, InfluxDB, Home Assistant and Nginx.
- **Dependencies:** A `requirements.txt` file lists all the Python dependencies required by the prototype. Similarly, a `package-lock.json` file lists all npm dependencies.

Appendix B

Code

```
FROM arm64v8/python:3.11-slim-bullseye as builder

WORKDIR /builder

USER root

RUN apt-get update && apt-get install -y curl
RUN curl -fsSL https://deb.nodesource.com/setup_18.x | bash -
RUN apt-get install -y nodejs

COPY package*.json ./

RUN npm install

COPY . .

# Build the frontend dependencies
RUN npm run build
RUN npm run build-mail
```

Listing 16: Dockerfile - Builder stage

```
FROM arm64v8/python:3.11-slim-bullseye

ENV FLASK_APP wsgi.py
ENV FLASK_ENV production
RUN adduser --disabled-password --gecos "" server_runner
WORKDIR /opt/webapp
USER root

COPY --chown=server_runner:server_runner app app
COPY --chown=server_runner:server_runner migrations migrations
COPY --chown=server_runner:server_runner wsgi.py config.py boot.sh requirements.txt ./

RUN apt-get update && apt-get install -y cron nano sqlite3

RUN echo "0 * * * * cd /opt/webapp/ && . .venv/bin/activate \
    && . ./project_env.sh \
    && flask execute-job >> /opt/webapp/logs/pihole_job.log 2>&1" \
    >> /etc/cron.d/webapp-cron \
    && echo "30 12 * * 0 cd /opt/webapp/ && . .venv/bin/activate \
    && . ./project_env.sh && flask execute-weekly-notifications \
    >> /opt/webapp/logs/weekly_email_job.log 2>&1" >> /etc/cron.d/webapp-cron \
    && crontab -u server_runner /etc/cron.d/webapp-cron \
    && mkdir -p /opt/webapp/logs \
    && touch /opt/webapp/logs/pihole_job.log \
    /opt/webapp/logs/weekly_email_job.log \
    && chown server_runner:server_runner /opt/webapp/logs/pihole_job.log \
    /opt/webapp/logs/weekly_email_job.log \
    && chmod u+s /usr/sbin/cron

RUN chown -R server_runner:server_runner /opt/webapp
RUN chmod u+x /opt/webapp/boot.sh

USER server_runner

RUN mkdir -p /opt/webapp/data/rel_db
RUN mkdir -p /opt/webapp/data/pihole_etc

# Copy built files from the builder stage
COPY --from=builder --chown=server_runner:server_runner /builder/app/static/dist \
    /opt/webapp/app/static/dist

RUN python -m venv .venv
RUN .venv/bin/pip install -r requirements.txt

EXPOSE 8000

ENTRYPOINT ["./boot.sh"]
```

Listing 17: Dockerfile - Final stage

```yaml
name: Build and Push Docker Image

on:
  release:
    types:
      - created

jobs:
  build-and-push-image:
    runs-on: ubuntu-latest

    steps:
      - name: Check out code
        uses: actions/checkout@v3

      - name: Set up Docker Buildx
        uses: docker/setup-buildx-action@v2

      - name: Log in to Docker Hub
        uses: docker/login-action@v2
        with:
          username: ${{ secrets.DOCKERHUB_USERNAME }}
          password: ${{ secrets.DOCKERHUB_TOKEN }}

      - name: Build and push Docker image for ARM architecture
        uses: docker/build-push-action@v4
        with:
          context: .
          push: true
          tags: ${{ secrets.DOCKERHUB_USERNAME }}/shipp: \
            ${{ github.event.release.tag_name }}, \
            ${{ secrets.DOCKERHUB_USERNAME }}/shipp:latest
          platforms: linux/arm64
```

Listing 18: Github actions worklow - Build and deployment

```yaml
services:
  prototype:
    container_name: shipp
    image: elliottwallace/shipp:latest
    expose:
      - 8000
    volumes:
      - static_volume:/opt/webapp/app/static
      - sqlite-data:/opt/webapp/data/rel_db
      - pihole-etc:/opt/webapp/data/pihole_etc
    environment:
      - SECRET_KEY=<your_secret_key>
      - API_SECRET_KEY=<your_api_secret_key>
      - SQLITE_URL=sqlite:////opt/webapp/data/rel_db/sqlite.db
      - INFLUXDB_ACTIVE=true
      - INFLUXDB_URL=http://influxdb:8086/
      - INFLUXDB_AUTH_TOKEN=<your_influxdb_auth_token>
      - INFLUXDB_ORG=home
      - INFLUXDB_BUCKET=communications
      - PIHOLE_DOMAIN=<your_pihole_domain>
      - PIHOLE_AUTH_TOKEN=<your_pihole_auth_token>
      - PIHOLE_DB_URL=sqlite:////opt/webapp/data/pihole_etc/gravity.db
      - MAIL_SERVER=<your_mail_server>
      - MAIL_PORT=<your_mail_port>
      - MAIL_USERNAME=<your_mail_username>
      - MAIL_PASSWORD=<your_mail_password>
      - SCHEDULER_TIMEINTERVAL=3600
      - TZ=<your_timezone>
    restart: unless-stopped

  nginx:
    container_name: nginx_reverse_proxy
    image: nginx:stable-bullseye
    depends_on:
      - prototype
    ports:
      - "8080:80"
    volumes:
      - ./nginx/nginx.conf/:/etc/nginx/nginx.conf
      - ./nginx/templates:/etc/nginx/templates
      - static_volume:/usr/src/app/static
    environment:
      - LOCAL_NETWORK_IP_RANGE=<your_local_ip_range>
    restart: unless-stopped
```

Listing 19: Docker compose file (Part 1)

```
  influxdb:
    container_name: influxdb
    image: influxdb:2.7.1-alpine
    ports:
      - "8086:8086"
    volumes:
      - influxdb-data:/var/lib/influxdb2
      - influxdb-config:/etc/influxdb2
    environment:
      - DOCKER_INFLUXDB_INIT_MODE=setup
      - DOCKER_INFLUXDB_INIT_USERNAME=<your_username>
      - DOCKER_INFLUXDB_INIT_PASSWORD=<your_password>
      - DOCKER_INFLUXDB_INIT_ORG=home
      - DOCKER_INFLUXDB_INIT_BUCKET=communications
      - DOCKER_INFLUXDB_INIT_RETENTION=1m
      - DOCKER_INFLUXDB_INIT_ADMIN_TOKEN=<your_influxdb_auth_token>
    restart: unless-stopped

  pihole:
    container_name: pihole
    image: pihole/pihole:latest
    ports:
      - "53:53/tcp"
      - "53:53/udp"
      - "80:80/tcp"
      - "443:443/tcp"
    environment:
      - "TZ:<your_timezone>"
      - "WEBPASSWORD:<your_password>"
      - "FTLCONF_LOCAL_IPV4:<raspberry_pi_ip>"
      - "PIHOLE_DNS_:1.1.1.1;1.0.0.1"
      - "REV_SERVER:true"
      - "REV_SERVER_TARGET:<local_router_ip>"
      - "REV_SERVER_CIDR:<your_local_ip_range>"
    volumes:
      - "pihole-etc:/etc/pihole"
      - "pihole-dnsmasq:/etc/dnsmasq.d"
      - "./resolv.conf:/etc/resolv.conf"
    logging:
      driver: "json-file"
      options:
        max-size: "10m"
        max-file: "3"
    restart: unless-stopped
```

Listing 20: Docker compose file (Part 2)

```
  homeassistant:
    container_name: homeassistant
    image: ghcr.io/homeassistant/home-assistant:stable
    restart: unless-stopped
    network_mode: host
    privileged: true
    volumes:
      - homeassistant-config:/config
      - /etc/localtime:/etc/localtime:ro

volumes:
  sqlite-data:
  influxdb-data:
  influxdb-config:
  pihole-etc:
  pihole-dnsmasq:
  homeassistant-config:
  static_volume:
```

Listing 21: Docker compose file (Part 3)

```yaml
openapi: 3.0.0
info:
  title: Pi-hole API
  description: This is an unofficial OpenAPI 3.0 specification for the Pi-hole API.
  version: 1.0.0
servers:
  - url: http://pi.hole/admin/
    description: The default URL for the Pi-hole API
paths:
  /api.php:
    get:
      summary: Main API endpoint
      description: Returns a JSON object with data.
      parameters:
        - name: getAllQueries
          in: query
          description: The getAllQueries 'endpoint' (needs parameters below)
          required: false
          schema:
            type: integer
            default: 1
        - name: from
          in: query
          description: from timestamp
          required: false
          schema:
            type: string
            format: date-time
        - name: until
          in: query
          description: until timestamp
          required: false
          schema:
            type: string
            format: date-time

# 1) Define the key name and location
components:
  securitySchemes:
    ApiKeyAuth:
      type: apiKey
      in: query
      name: auth
# 2) Apply the API key globally to all operations
security:
  - ApiKeyAuth: []
```

Listing 22: Pi-hole API excerpt

```bash
#!/bin/bash
# Run for 12 hours (43200 seconds) with the command: nohup bash /path/to/script &
timeout_duration=42300
function log() {
    echo "$(date '+%Y-%m-%d %H:%M:%S') - $1" | tee -a output.log
}
function cleanup() {
    log "Script execution completed."
    exit 0
}

trap cleanup SIGTERM SIGINT
current_timestamp=$(date '+%Y-%m-%d_%H-%M')
mkdir -p datasets

csv_file="datasets/container_stats_${current_timestamp}.csv"
echo "Timestamp,CONTAINER ID,Name,CPU %,MEM %,MEM USAGE / LIMIT,NET I/O,BLOCK I/O, \
PIDs,SysMem Total,SysMem Used,SysMem Available,Swap Total,Swap Used" > "$csv_file"

log "Script started. Output will be logged in output.log"
end_time=$(( $(date +%s) + timeout_duration ))

while true; do
    # Get the current Unix timestamp
    c_timestamp=$(date '+%Y-%m-%d %H:%M:%S')
    if [ "$(date '+%s')" -ge "$end_time" ]; then
        cleanup
    fi
    # Execute docker stats command
    docker_stats_output=$(docker stats --no-stream --format " \
    {{.Container}},{{.Name}},{{.CPUPerc}},{{.MemPerc}}, \
    {{.MemUsage}},{{.NetIO}},{{.BlockIO}},{{.PIDs}}")
    if [ $? -eq 0 ]; then
        # Get memory and swap usage
        memory_info=$(free -h | awk 'NR==2 {print $2","$3","$7}')
        swap_info=$(free -h | awk 'NR==4 {print $2","$3}')
        while read -r line; do
            echo "$c_timestamp,$line,$memory_info,$swap_info" >> "$csv_file"
        done <<< "$docker_stats_output"
        log "Data collected successfully."
    else
        log "Error collecting data: $docker_stats_output"
    fi
    sleep 180 # Collect data every 3 minutes
done 2>&1 | tee -a output.log
```

Listing 23: Performance measuring script