POLITECNICO DI TORINO

Master's Degree in Computer Engineering

Politecnico
di Torino

Master's Degree Thesis

Integrated Security for IoT: Methodologies and Certification for Collaborative Smart Devices

Supervisors

Prof. Luca ARDITO

Dr. Andrea SARACINO

Dr. Marco RASORI

Candidate

Oliver Howard GLANVILLE

November 2024

Abstract

The rapid growth of the Internet of Things (IoT) has created both significant opportunities and security challenges. IoT devices are increasingly deployed in dynamic environments, such as smart homes, where securing these devices is critical. These devices often operate with constrained resources, such as limited memory and processing power, making it difficult to enforce complex security policies. This thesis proposes a novel solution to these challenges by leveraging the principles of Security-by-Contract (S×C) to ensure security compliance through Usage Control (UCON) across each stage of an IoT device's lifecycle. The research develops a framework that allows IoT devices to adhere to dynamic security policies, even in resource-constrained environments.

The primary objective of this work is to address the security challenges faced by IoT devices. It integrates S×C principles with UCON, an access control model that guarantees compliance with security contracts. UCON continuously validates device actions against predefined policies, ensuring devices perform only actions that align with security requirements. The novelty of this research lies in demonstrating how S×C compliance can be effectively managed in resource-limited IoT environments, where traditional security models often fail due to insufficient computational resources.

The second objective is to assess the viability of deploying UCON-based S×C enforcement on resource-constrained devices, such as the Raspberry Pi 4, a widely used platform in IoT applications. This research uses a distributed architecture with a Distributed Hash Table (DHT) to manage access control policies across the network. The focus is on testing the performance, scalability, and security of UCON, aiming to evaluate whether it can provide strong security with acceptable performance on lightweight systems.

The experimental setup includes tests for installation, runtime, and revocation performance. Installation tests measure the time required to deploy an IoT application in a smart home environment, while performance tests focus on handling access control requests, with emphasis on policy complexity and system behavior as the number of attributes increases. Revocation tests are critical, as quick revocation is essential for maintaining security in real-time environments. The results show that UCON, even on constrained devices, can handle high volumes of requests while maintaining strong security.

One key finding is that UCON guarantees S×C compliance consistently, even as the complexity of access control policies increases. Despite some performance

degradation as the number of policy attributes grows, the system maintains acceptable response times for both access requests and revocations. This indicates that UCON is a viable solution for securing resource-constrained environments like smart homes.

In addition to validating the UCON approach, the thesis explores a practical application by integrating access control mechanisms into a PiCamera system for video streaming in a smart home. This case study demonstrates how UCON can control access to sensitive resources, ensuring that only authorized users can view or control the video feed, further showcasing the scalability and adaptability of the UCON model in real-world IoT scenarios.

The conclusions confirm that the UCON-based S×C framework is both effective and feasible for ensuring IoT security. The findings indicate that it is possible to maintain S×C compliance even under the limitations of small-scale IoT devices, making this approach suitable for deployment in smart homes. By integrating S×C with UCON, this thesis represents a significant step forward in enhancing the security and reliability of IoT ecosystems, ensuring they can scale securely in complex, dynamic environments.

Acknowledgements

I would like to express my heartfelt gratitude to my family for their unwavering support throughout this journey. To my mother Sonia, my father Clive Howard, my sister Emily, and my grandparents Jean, Maria, Sergio, and Simon, your encouragement and love have meant the world to me.

I am deeply grateful to my girlfriend, Giulia, who has been a constant companion in my life, providing immense support and understanding. I also wish to acknowledge her family, particularly Anna Sophia, for their kindness and encouragement.

A special thanks goes to my lifelong friends Lorenzo and Oleg, as well as to those I have met along this path, Giacomo and Matteo, who I am sure will continue to be part of my life. I would also like to extend my gratitude to all my friends, including those not mentioned here, whose unwavering support has been invaluable during this time.

I would like to express my appreciation to the tutors who supported me in the creation of this project.

Lastly, I extend my sincere gratitude to the Assiste group for providing me with the opportunity to grow as an engineer throughout this journey.

Oliver Howard

Contents

List of Tables

List of Figures

Chapter 1

Introduction

According to a study produced by Statista in collaboration with Transforma Insights in 2024 [1], the number of Internet of Things (IoT) connections worldwide is approximately 18 billion and is projected to reach 32.1 billion by 2030. This data not only provides a clearer understanding of the vastness of the IoT market but also highlights the significant impact that this set of technologies will have in the coming years.

In this context, the security of IoT devices has emerged as a critical concern. With the rapid proliferation of connected devices, ensuring robust security measures has become increasingly challenging. The diverse nature of IoT devices, ranging from simple sensors to complex systems, introduces unique vulnerabilities. Issues such as unauthorized access, data breaches, and the difficulty of enforcing consistent security policies across resource-constrained devices make IoT security one of the most pressing topics in today's technological landscape.

This thesis aims to explore the feasibility of implementing the Security-by-Contract (SxC) paradigm to enforce security policies in restricted environments such as smart homes. The enforcement of these policies is managed by a Usage Control System (UCS). The primary objective is to determine whether robust security measures can be implemented in Internet of Things environments, relying on a constrained hardware platform as the centralized system. This centralized system, which operates on lightweight hardware, will be implemented on a Raspberry Pi 4 with 8 GB of RAM, representing the platform used for the UCS in this study.

1.1 Defining the Internet of Things

Nowadays, the Internet of Things is a hot topic in the technological landscape; however, it is not a new concept. The idea of connecting "things" over the internet began with the advent of the internet itself. For instance, the Trojan Room coffee pot [2], set up at the University of Cambridge in 1991, is often cited as one of the earliest examples of an IoT device. This coffee pot was connected to the Internet to allow users to remotely check its status. Similarly, in 1990, John Romkey created the first internet-connected device, a toaster that could be turned on and off over the Internet [3].

From its inception, the field of Internet of Things has experienced exponential growth and is anticipated to continue expanding in the coming years. To provide a comprehensive overview, it is essential to address the concept of IoT, despite the absence of a universally accepted definition.

The scientific community has proposed various definitions for IoT. One perspective describes it as a network that interconnects ordinary physical objects with identifiable addresses, enabling them to provide intelligent services through traditional information carriers such as the Internet and telecommunication networks [4]. Another view defines IoT as a global network of interconnected objects that are uniquely addressable and based on standard communication protocols [5].

The term "Internet of Things" combines “Internet” and “Things,” encapsulating its core idea. The real value of IoT lies in its ability to connect a diverse range of devices, from everyday objects and embedded sensors to context-aware systems and traditional computing networks. These devices, despite differences in design, protocols, intelligence, and applications, can communicate and integrate to collect, process, and exchange data. This connectivity allows for complex operations and intelligent tasks to be performed both cooperatively and autonomously.

1.2 IoT Architectures and Protocols

The Internet of Things world is characterized by various architectural frameworks designed to manage and integrate its diverse components. Two of the most widely recognized and commonly adopted architectures are the three-layer [6] and five-layer models [7].

Three-Layer Architecture The three-layer architecture is a foundational model in IoT design. It includes the following layers:

1. **Perception Layer** (Physical Layer): This layer is responsible for collecting

data from the physical environment using devices like RFID tags, barcodes, sensors, and cameras. Its main role is to sense and gather data from the real world, converting physical signals into digital information for further processing.

2. **Network Layer**: This layer ensures the secure and efficient transmission of data collected by the perception layer to the application layer. It manages the communication between devices via both wired and wireless methods, using technologies such as Wi-Fi, Zigbee, and cellular networks.

3. **Application Layer**: The application layer processes and analyzes the data received from the network layer, transforming it into actionable information. This layer delivers services to end-users and communicates results back to the perception layer when necessary, closing the feedback loop in IoT systems.

Five-Layer Architecture Building upon the three-layer model, the five-layer architecture introduces two additional layers to enhance functionality and control. This architecture consists of:

1. **Perception Layer**: As in the three-layer architecture, this layer handles the collection of data from the physical world using various sensors and technologies, acting as the interface between the physical and digital realms.

2. **Network Access Layer**: This layer, also known as the access gateway layer, manages communication within local environments. It handles the connection of nodes from the perception layer to the network, processing the initial data and ensuring efficient transmission via base stations, gateways, or other network infrastructure components.

3. **Network Transmission Layer**: This layer ensures end-to-end data transmission across large-scale networks, including satellite, mobile, and optical fiber networks. It handles the neutral access and seamless integration between different communication protocols, ensuring reliable data exchange across diverse networks.

4. **Application Support Layer**: This layer provides support for data processing, storage, and intelligent analysis. It leverages cloud computing, middleware, and database technologies to enable scalable, flexible, and efficient management of IoT data. The application support layer ensures that information from the perception layer is processed and prepared for specific IoT applications.

5. **Application Presentation Layer**: The final layer in the five-layer architecture is responsible for presenting processed data and information to users. This layer focuses on developing user interfaces and applications using multimedia,

virtual reality, and other technologies to ensure seamless interaction between IoT systems and end-users.

1.2.1 IoT Communication Protocols

In addition to the architectural models, IoT relies heavily on a wide range of communication protocols tailored to the specific needs of different applications. These protocols can generally be divided into two main categories: Low Power Wide Area Networks (LPWAN) and short-range networks [8].

LPWAN Protocols LPWAN protocols are designed to cover large distances while consuming minimal power. Two notable protocols in this category are:

- **Sigfox**: Sigfox is a low-power communication technology used for transmitting small amounts of data over distances up to 50 kilometers. It operates on Ultra Narrow Band (UNB) technology, with low data rates (10-1,000 bits per second) and is ideal for applications like smart meters and environmental sensors. Sigfox supports star topology and is commonly used in low-energy IoT applications that require infrequent communication.
- **Cellular**: Cellular networks, such as GSM, 3G, and 4G, provide high-speed connectivity over long distances, making them suitable for applications requiring significant data throughput. However, due to their high power consumption, they are less suitable for low-power IoT devices and M2M communication. Cellular networks are often used in mobile IoT applications, where power sources are not a constraint.

Short-Range Network Protocols Short-range networks are optimized for low power consumption and are typically used in local environments. Some common protocols include:

- **6LoWPAN**: This protocol is widely used in IoT due to its ability to enable IPv6 communication over low-power wireless networks (IEEE 802.15.4). 6LoWPAN supports both star and mesh topologies, offering flexibility in network design, and is well-suited for battery-powered devices.
- **Zigbee**: Zigbee, based on the IEEE 802.15.4 standard, is designed for low-data-rate and low-power applications. It supports various network topologies such as star, mesh, and tree, making it a versatile option for IoT devices requiring extended battery life and secure communication.
- **BLE** (Bluetooth Low Energy): BLE is optimized for low power consumption and short-range communication, often used in IoT applications that require

quick data exchanges over short distances, such as wearables and smart home devices. BLE supports star topologies and can manage a large number of nodes.

- **RFID**: RFID uses radio frequency to communicate between a reader and a tag. It is commonly used for identification and tracking applications, such as in smart shopping and healthcare systems. RFID can operate with both active (battery-powered) and passive tags, with a range that varies depending on the system used.
- **NFC** (Near Field Communication): NFC is a short-range protocol that allows data exchange between devices within a few centimeters. It is often used in mobile payments and access control systems, and supports peer-to-peer communication.
- **Z-Wave**: Z-Wave is another low-power protocol designed for home automation and small-scale IoT networks. It supports mesh topologies and is optimized for small data transfers, making it ideal for applications like smart lighting and energy control.

In summary, the choice of IoT communication protocol depends heavily on the specific requirements of the application, such as range, power consumption, and data rate. Each protocol offers different strengths in terms of topology, security, and performance, ensuring that IoT systems can be tailored to meet the unique demands of various use cases.

1.3 Smart Home Paradigm

The Internet of Things has revolutionized the concept of *smart homes*, where devices and appliances are interconnected through intelligent systems that enhance convenience, efficiency, and security. A *smart home* is defined as a residence equipped with a communication network that allows for remote control, monitoring, and access to various appliances and services, both from within and outside the home [9]. The core components of a smart home are an internal network, intelligent control, and automation systems. The internal network can be based on wired, wireless, or hybrid communication technologies, while intelligent control is often implemented through gateways that manage connectivity between devices and external services [9].

Smart home networks typically use technologies such as powerline communication (PLC), busline systems, and radio frequency (RF). Powerline systems leverage existing electrical wiring to transmit data, making them cost-effective but susceptible to power interruptions and interference. Busline systems, which use dedicated

cables for communication, offer more robust performance but require additional infrastructure. RF-based systems, such as Zigbee and Bluetooth, are widely adopted for their wireless flexibility, though concerns remain over interference and security vulnerabilities [10].

One of the key advantages of smart homes is their ability to automate routine tasks based on user behavior or environmental conditions. For instance, sensors can detect when residents enter or leave a room, adjusting lighting, heating, and other systems accordingly. Moreover, smart homes can provide advanced services such as eldercare and healthcare monitoring, where data from sensors is used to track the well-being of residents and issue alerts when necessary [10]. Additionally, smart home systems can be designed to optimize energy consumption by monitoring and controlling appliances to reduce power usage during peak times, contributing to energy efficiency [10].

While the technology underpinning smart homes has advanced significantly, challenges remain in areas such as data privacy, security, and the complexity of integrating multiple devices and systems. The continuous evolution of AI, sensor technologies, and IoT protocols will be crucial in addressing these issues and further enhancing the capabilities of smart homes.

In summary, the smart home paradigm represents a significant application of IoT, offering residents increased control, convenience, and safety. By leveraging a combination of network protocols and intelligent control systems, smart homes can automate daily activities and provide valuable services.

1.4 Securing Smart Homes: Key Challenges in IoT Security and Privacy

The integration of Internet of Things devices into smart homes brings numerous benefits, such as increased convenience and automation. However, this connectivity also introduces significant security and privacy challenges. Addressing these challenges is crucial to ensuring the safety and privacy of users in a smart home environment.

One of the primary concerns is the security vulnerabilities inherent in IoT devices. Many smart home devices are mass-produced with limited security measures, making them susceptible to cyberattacks. The deployment of similar devices across a network can amplify vulnerabilities; a single compromised device can potentially jeopardize the entire system. Devices often have default or weak passwords and minimal encryption, which can be exploited by attackers to gain unauthorized access to the network [11].

Authentication and access control also pose significant challenges. Many IoT devices use outdated or insufficient authentication methods, leaving them open to unauthorized access and attacks. The diversity of devices and platforms used in smart homes complicates the implementation of consistent security measures. For instance, smart locks, cameras, and lighting systems may each use different authentication methods, creating a fragmented security landscape that is difficult to manage.

Privacy concerns are another major issue. IoT devices continuously collect and transmit vast amounts of personal data, often without adequate protection. This data can include sensitive information such as location, audio, and video recordings. The ubiquitous nature of these devices means that users' data is frequently transmitted over networks that may not be secure, exposing it to potential breaches [11].

A significant challenge to securing smart homes is the lack of standardization across IoT devices. Different manufacturers use proprietary standards, making it difficult to ensure interoperability and implement consistent security measures. Without unified standards, securing a network of diverse devices becomes challenging. Standardization could improve device integration and simplify the application of comprehensive security protocols across various devices [6].

Additionally, the infrequent release of security updates for IoT devices exacerbates these issues. Many devices receive updates irregularly, leaving them vulnerable to newly discovered threats. Manufacturers often prioritize new product development over maintaining security for existing devices, which can leave older devices exposed to potential attacks [11].

These challenges become even more significant when examining the security vulnerabilities across the three layers of the IoT architecture previously outlined in Section 1.2. Each of these layers—perception, network, and application—faces distinct threats that must be addressed.

At the **Perception Layer** the main issues relate to the strength and integrity of the wireless signals used for communication. These signals can be disrupted or intercepted making them vulnerable to physical attacks. Attackers can tamper with the hardware or capture nodes, compromising the entire system's confidentiality through replay attacks or by gaining access to encryption keys via timing attacks. Limited storage capacity, power consumption, and computational ability also make devices in this layer particularly susceptible to threats like denial of service (DoS) attacks, which drain device resources and energy [12].

At the **Network Layer**, the security risks revolve around the possibility of traffic interception and data breaches. The use of remote access mechanisms and the

exchange of information between devices creates vulnerabilities, especially to man-in-the-middle (MITM) attacks. Attackers can eavesdrop on communication channels, steal key material, and compromise the secure exchange of data. The heterogeneous nature of IoT devices further complicates the application of consistent security protocols, as existing network protocols often struggle to accommodate the unique demands of machine-to-machine (M2M) communication in IoT systems [12].

The **Application Layer** is vulnerable due to the lack of unified global policies and standards for IoT application development. The diverse range of authentication mechanisms used across different applications makes it difficult to ensure the privacy of user data. Additionally, the large volume of data shared between connected devices can lead to system overloads, impacting the availability of services. Applications that fail to adequately secure personal data may expose sensitive user information, making it imperative for users to have tools that allow them to control and monitor the data they disclose [12].

In conclusion, while smart home technologies offer numerous advantages, they also present several security and privacy challenges. Addressing these challenges requires a comprehensive strategy that includes enhancing device security, implementing robust authentication and access control measures, improving privacy protections, standardizing protocols, and ensuring regular security updates.

This thesis addresses these critical issues by proposing an innovative solution through the application of the Security-by-Contract paradigm, facilitated by the Usage Control System, which provides an effective solution for managing security in resource-constrained environments, such as smart homes. The research highlights the practical deployment of the UCS on lightweight platforms like the Raspberry Pi, demonstrating its capability to enhance IoT security.

The integration of advanced access control mechanisms within these systems offers a promising approach to safeguarding user data and privacy in the evolving landscape of smart home technology.

1.5 Access Control Challenges in Constrained Devices for Smart Homes

In the context of smart homes, the majority of Internet of Things devices are resource-constrained, meaning they operate with limited computational power, memory, and energy. These limitations stem from the need for IoT devices to remain small, energy-efficient, and cost-effective, but they also introduce significant security challenges. Traditional access control mechanisms, such as Role-Based Access Control (RBAC) or Attribute-Based Access Control (ABAC), which rely on

centralized systems and require substantial processing power, are often not feasible for such devices [13]. Moreover, the dynamic and heterogeneous nature of smart home environments, where devices frequently join or leave the network, complicates the enforcement of security policies in real time.

A critical issue is finding ways to enforce fine-grained policies on constrained devices without overwhelming their limited resources. For example, while ABAC offers greater flexibility in policy management by using attributes rather than explicit user identities, its implementation can demand frequent updates and recalculations of policies, which can be burdensome for constrained devices [13]. Lightweight communication protocols are also essential to reduce the overhead on these devices while maintaining secure interactions. However, many existing solutions rely on computationally expensive cryptographic methods, which may not be suitable for resource-limited devices.

To address these challenges, this thesis proposes the Usage Control System as a more effective approach to securing constrained devices. The UCS enforces security policies dynamically by continuously monitoring and controlling the usage of resources. By leveraging lightweight hardware, such as a Raspberry Pi, the UCS can enforce fine-grained access control policies without overburdening the devices, thus offering a practical solution for managing security in smart homes [13]. This approach ensures that security remains robust while accommodating the limitations of constrained devices, making it a promising solution for addressing the unique challenges of IoT environments.

1.6 Examples of Cyber Attacks on IoT Devices in Smart Homes

The security of smart home IoT devices has been compromised in various high-profile incidents, revealing significant challenges in access control enforcement and the inherent limitations of constrained devices. These cases highlight how vulnerabilities in device security policies, coupled with the limited processing power and memory of IoT devices, can lead to severe security breaches, affecting the privacy and safety of users.

The following examples aim to illustrate the potential risks faced by IoT devices in smart homes, emphasizing how attackers can exploit weaknesses in access control and authorization mechanisms. These real-world incidents underscore the critical need for robust security solutions that can effectively manage access to IoT devices, even in resource-constrained environments.

Ring Camera Breach In December 2019, Ring home security cameras were hacked, with attackers exploiting reused credentials to gain unauthorized access. This incident highlighted the difficulties in enforcing strong access control policies. Despite Ring's efforts to promote multi-factor authentication, the vulnerability stemmed from users' failure to adopt recommended security practices. The breach exposed the limitations of relying solely on passwords for device security, emphasizing the need for improved enforcement of access policies [14].

Z-Wave Downgrade Attack The 2018 discovery of a vulnerability in the Z-Wave protocol, a wireless communication standard used for home automation and IoT devices, underscored the challenges associated with enforcing security standards on such devices. The attack exploited the ability to force devices to downgrade from a secure to an insecure version of the protocol. This flaw, rooted in the support for outdated security mechanisms, revealed how constraints in device capabilities and compatibility issues can undermine security. The incident stressed the importance of consistent enforcement of updated security protocols across all devices [15].

Verkada Camera Incident In March 2021, Verkada experienced a security breach where attackers gained access to customer data through a misconfigured support server. The attackers exploited internal vulnerabilities to access devices and data. This incident demonstrated the risks associated with managing and securing IoT devices when backend support systems are compromised. It highlighted the need for rigorous enforcement of access controls and the challenges of protecting device data from unauthorized access [16].

Casino Hack via IoT Thermometer A notable incident involved hackers targeting a casino through an IoT thermometer in a lobby aquarium. This breach illustrated how seemingly benign IoT devices can become attack vectors if not properly secured. The attackers used the thermometer to infiltrate the network and access sensitive information, showcasing the risks associated with inadequate security measures for IoT devices. It underscored the need for stringent security standards and regulations for all IoT devices to prevent such vulnerabilities [17].

1.7 Overview of My Contributions

This thesis investigates the application of the UCON (Usage Control) framework for ensuring Security-by-Contract (S×C) compliance in smart home environments. My research makes two key contributions that address both the practical and theoretical aspects of deploying this framework on lightweight systems.

Firstly, I propose using the UCON framework to enforce S×C, modeling its components—application (A), contract (C), and policy (P)—in a manner compatible with UCON's principles. Policies and requests follow XACML standards, ensuring that S×C compliance is enforced. The goal is to demonstrate, through practical examples, how the Usage Control System (UCS) can manage real-time access control in dynamic environments, such as smart homes. Rigorous testing validated the UCS's ability to enforce access policies and respond to rapid changes, shedding light on the framework's role in enhancing IoT security.

Secondly, I explore the feasibility of deploying UCON on constrained devices, such as the Raspberry Pi. By testing a modified UCS on these platforms, I show that the framework can maintain effective access control even with limited resources, making it a promising security solution for lightweight IoT devices.

A significant aspect of this work involved evaluating the UCON framework's performance through tests in installation, runtime performance, and access revocation. These tests confirm that the UCON framework can deliver consistent security guarantees, even under limited computational resources, proving that robust access control can be achieved on IoT devices in smart home settings.

The thesis is structured as follows: Chapter 2 reviews existing access control models, focusing on UCON and S×C, and examines related concepts from the SIFIS-Home project.

Chapter 3 describes the methodology for testing the UCS on constrained devices like the Raspberry Pi 4, outlining the experiments to assess installation time, policy enforcement efficiency, and the handling of access control requests, including revocation under real-world conditions.

Chapter 4 presents the experimental results, analyzing how UCON handles complex policies and access revocation. The findings demonstrate the viability of using UCON to enforce S×C compliance in resource-constrained devices, with implications for scalability in dynamic smart home environments.

Chapter 5 explores the practical application of UCON by integrating the UCS with a PiCamera system for video streaming within a smart home. This case study illustrates how UCON can manage access to sensitive resources and demonstrates its flexibility in real-world IoT applications.

The final chapter summarizes the contributions, discusses the implications of the findings, and highlights the potential for UCON-based S×C enforcement in smart homes, paving the way for future developments in secure, scalable IoT systems.

Through these contributions, this thesis not only validates the theoretical framework of S×C but also demonstrates its practical applicability.

Chapter 2

UCON Framework: State of the Art and Applications

This chapter provides a comprehensive examination of the UCON framework, with a particular focus on its application in distributed architectures. It begins by discussing the framework's foundational aspects, including its design principles and the specific challenges it addresses.

The discussion covers the core components and interactions within UCON, highlighting how these elements work together to ensure robust security features. A detailed exploration of the theory behind distributed environments is also presented, offering insights into how UCON integrates within these contexts.

Following the theoretical overview, the chapter transitions to a practical application within the Internet of Things domain, exemplified through an in-depth analysis of the SIFIS-Home project. This case study illustrates how UCON principles are applied in real-world scenarios, detailing the design and development of security systems within the project.

Overall, the chapter aims to bridge theoretical concepts with practical implementations, providing a thorough understanding of both the theoretical underpinnings and real-world applications of UCON in distributed systems.

2.1 Introduction to the UCON Framework

The UCON framework represents a significant advancement over traditional access control systems by addressing their limitations through more dynamic and flexible security mechanisms. Unlike conventional models that assess access rights solely at the time of a request, UCON integrates continuous monitoring and adaptive evaluation to manage and regulate access and usage of resources more comprehensively [18].

Overview of UCON UCON expands upon traditional access control paradigms by introducing continuous monitoring and dynamic evaluation capabilities. Traditional access control mechanisms, such as Mandatory Access Control (MAC), Discretionary Access Control (DAC), Role-Based Access Control (RBAC), and Attribute-Based Access Control (ABAC), generally focus on static, request-time evaluations, where access is granted or denied based on fixed rules [19]. In contrast, UCON incorporates ongoing authorizations and obligations that ensure compliance throughout the entire duration of access. This approach allows UCON to adapt to changing conditions and user behaviors, providing a more flexible and responsive security solution [18].

Design Principles and Objectives The design of the UCON framework is built around several key principles that enhance its flexibility and effectiveness in managing security policies. It introduces three core components: subjects, objects, and rights, supplemented by additional elements such as authorizations, obligations, and conditions [18]. These components work together to create a robust model capable of handling complex security scenarios. For example, UCON can manage policies that involve temporal constraints, such as limiting access based on the duration or frequency of use, which traditional models cannot easily accommodate [19].

The primary objectives of UCON are to provide a more nuanced approach to access control and to address the challenges posed by modern, dynamic environments. By incorporating the ability to enforce and adapt to policies in real-time, UCON improves upon the rigidity of traditional models. This dynamic capability ensures that access rights are not only granted according to predefined rules but are also continuously monitored and adjusted based on real-time data and conditions [18]. As a result, UCON enhances the overall security posture of systems by accommodating complex and evolving security requirements [19].

Problems Addressed by UCON UCON addresses several critical limitations inherent in traditional access control models. Traditional approaches—including

Role-Based Access Control and Attribute-Based Access Control—often fall short in scenarios requiring ongoing compliance with security policies, as they typically evaluate access rights only at the moment a request is made, as discussed in Section 1.5, where the limitations of traditional models are highlighted. For example, UCON can enforce policies that depend on mutable attributes or dynamic conditions—such as ensuring that a subject's access rights remain valid only while certain environmental conditions are met. This ability to continuously monitor and adjust access rights based on real-time changes is a significant improvement over static, request-time evaluations [18].

UCON introduces the concept of mutable attributes—allowing for more dynamic and sophisticated policy enforcement. In particular, it enables the system to update access permissions even during an ongoing interaction, or "active session." An active session refers to a period when a user has been granted access and is currently interacting with a system or resource. If, during such a session, the value of an attribute changes—such as a shift in the security status of a resource—UCON can dynamically revoke access if necessary. This ongoing evaluation of access rights addresses the risks associated with static policies—which may no longer be sufficient in adapting to evolving security threats [19]. By addressing these challenges, UCON provides a more adaptable and secure framework for managing access in complex and variable environments [18].

2.2 Components of the UCON Framework

The UCON framework serves as a comprehensive architecture designed to manage access to resources through a dynamic and continuous approach. At its core lies the Usage Control System, responsible for evaluating and enforcing Usage Control Policies (UCPs). In addition to the UCS, two external components play crucial roles in the overall operation: the Attribute Manager (AM) and the Policy Enforcement Point (PEP).

The following sections present an overview of the framework's architecture, followed by detailed explanations of each component. This includes a distinction between components that form the core UCS and essential external entities that support its functionality.

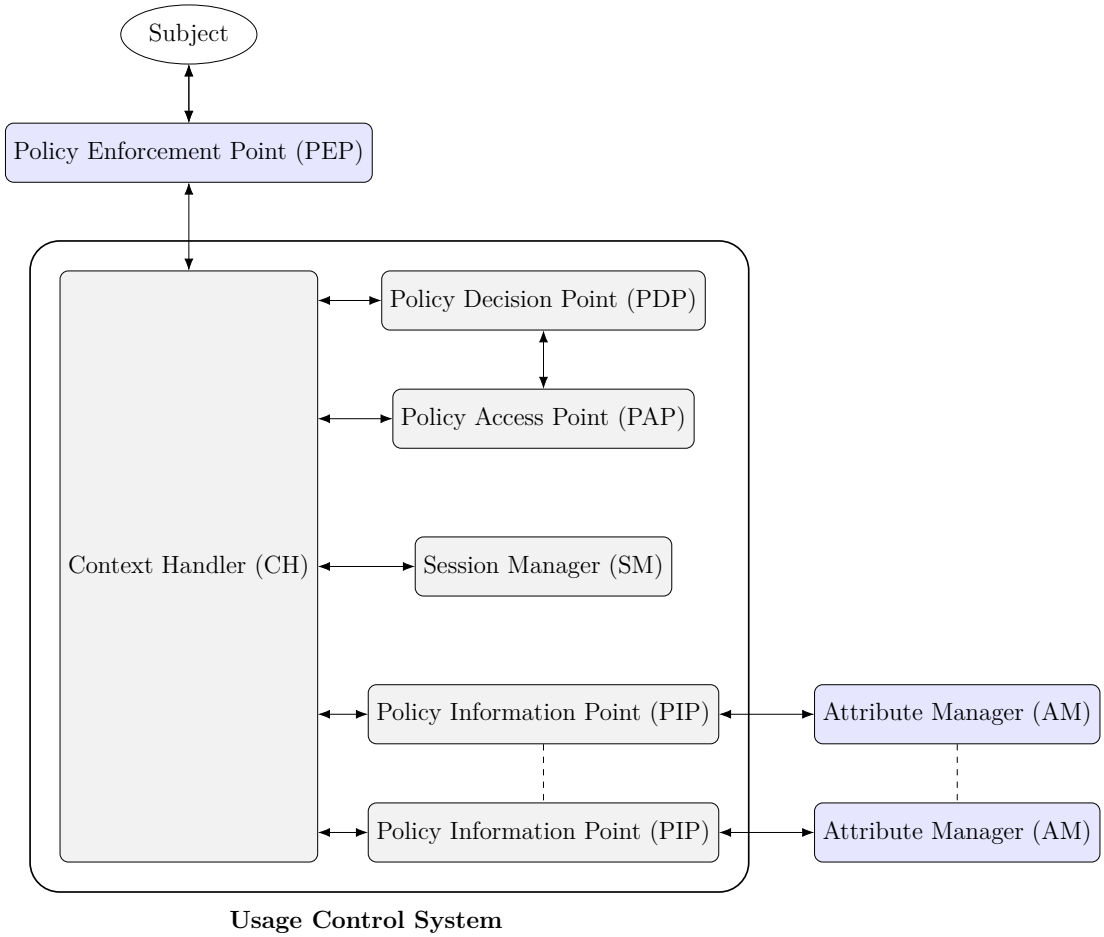

Figure 2.1: UCON Framework architecture.

Figure 2.1 illustrates the interactions between the internal components of the UCS and the external entities. Each component contributes to the effective management of access control and policy enforcement.

Policy Enforcement Point (PEP) The PEP operates outside the UCS and serves as the gateway for controlling access to resources. This component intercepts requests from external subjects and interacts with the UCS to determine whether to permit, deny, or revoke access. When a subject initiates an access attempt, the PEP sends a *tryAccess* request to trigger the authorization workflow. While the PEP plays a crucial role in managing access, the logic for revoking access resides within the UCS. The PEP executes commands issued by the UCS, such as *revokeAccess*, thereby ensuring that the policies enforced by the UCS can be applied continuously throughout an active session [19].

Policy Administration Point (PAP) The PAP forms part of the core UCS and is responsible for managing and storing Usage Control Policies that govern how access is controlled. All policies determining access rights, including pre-conditions, ongoing conditions, and post-conditions, reside here, ensuring that the system always refers to up-to-date rules for decision-making.

Policy Decision Point (PDP) The PDP serves as the decision engine of the UCS. This component evaluates UCON requests against the policies stored in the PAP, determining whether a subject can access a resource. While the PAP is the primary source for these policies, it is worth noting that the trusted PEP could also provide policies for evaluation. Critical for ensuring compliance with the policies defined by the administrator, the PDP evaluates pre-conditions (at the time of the request), ongoing conditions (throughout the session), and post-conditions (after access ends), enabling continuous enforcement of usage policies [19].

Attribute Managers (AMs) AMs operate as external components responsible for providing the UCS with up-to-date information about the attributes of subjects, resources, and environmental conditions. These attributes are crucial for evaluating UCON requests. As independent entities, AMs can be queried or subscribed to when the UCS needs to evaluate conditions based on attribute changes.

Policy Information Points (PIPs) PIPs function as intermediaries between the AMs and the UCS. Positioned within the UCS, they retrieve, subscribe to, and monitor attribute values from the AMs. PIPs supply the *Context Handler* with real-time attribute information, facilitating the dynamic evaluation of access policies. Although PIPs interface with external AMs, their coordination role places them firmly within the UCS.

Session Manager (SM) The SM is responsible for managing the lifecycle of access sessions within the UCS. Each session, uniquely identified and tracked, follows a sequence from the initial access request to its completion or revocation. By maintaining session state, the SM ensures that all actions within the UCON framework are monitored, keeping appropriate records of ongoing and terminated access.

Context Handler (CH) The CH serves as the orchestrator of the UCS, managing the flow of messages between all internal components (e.g., PDP, PIPs) and external ones (e.g., AMs, PEP). This component ensures that UCON requests are evaluated at the correct stages—before, during, and after access—by coordinating

the evaluation of UCPs and updating the SM as required. Its coordination role makes it the central hub of the UCS.

2.2.1 Interaction Among UCON Components: Authorization Workflow

The UCON framework effectively regulates access to resources by facilitating continuous monitoring and enforcement of access rights through its various components. Central to this operation is the interaction between the Policy Enforcement Point, Policy Decision Point, Policy Administration Point, Attribute Managers, Policy Information Points, Session Manager, and Context Handler.

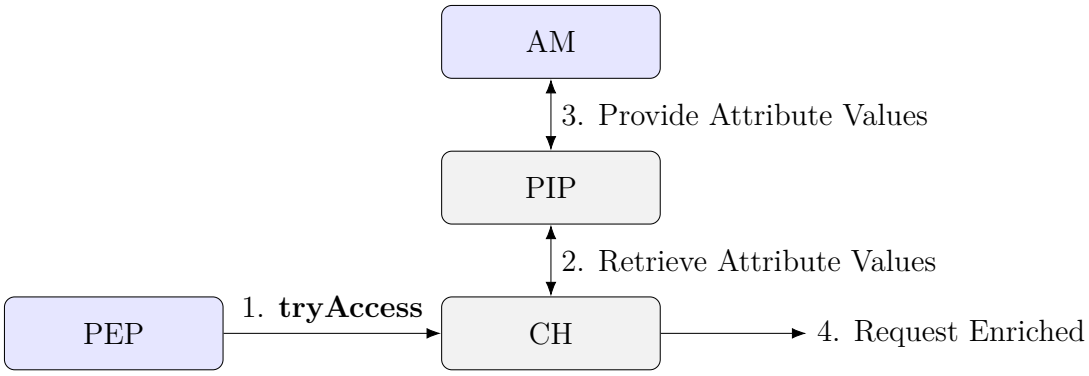

Figure 2.2: Initial request processing: PEP sends tryAccess to CH, attribute retrieval, and policy evaluation

When a subject requests access to a resource, the PEP initiates the process by generating a UCON request and sending a *tryAccess* message to the CH. This request includes necessary identifiers for the subject, resource, and action. Upon receiving this message, the CH enriches the request by retrieving current attribute values from the PIPs, which act as intermediaries between the CH and AMs. This interaction ensures that the most up-to-date information is available for evaluation. We refer to this phase as "fattening," as it involves augmenting the request with additional attribute data.

Next, the CH instructs the PDP to identify an applicable UCP from the PAP. In some cases, the trusted PEP may also provide policies for evaluation, especially when it has locally cached or pre-configured policies relevant to the specific request. The PDP then evaluates the pre-section of the UCP, whether retrieved from the PAP or provided by the trusted PEP, against the enriched UCON request. If this evaluation results in a permit decision, the CH communicates to the SM to create a new session, marking the access status as *TRY_ACCESS*. A *permitAccess* message

is then sent to the PEP, granting the subject access to the resource.

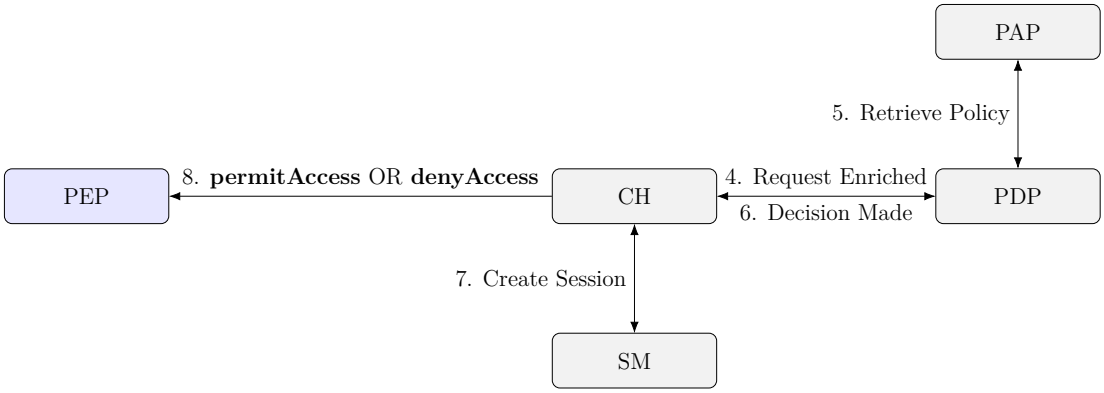

Figure 2.3: Policy evaluation: PDP evaluates policy from PAP or PEP, decision, and session creation

Once access is granted, the PEP sends a *startAccess* message to the CH, which includes the Session ID (SID) that was communicated to the PEP by the UCS in response to the initial *tryAccess* request. The CH then uses the SID to query the SM to retrieve the original access request associated with the session. Once the original request is retrieved, the CH enriches it with updated real-time attribute values, ensuring that the ongoing decision is based on the most current data available.

The enriched request, along with the ongoing section of the UCP, is sent to the PDP for further evaluation. If the ongoing decision results in a permit, the session status is updated to *START_ACCESS*. At this point, the CH subscribes to the relevant PIPs to monitor mutable attributes in real-time, allowing for continuous assessment of attribute changes that may affect the session's access permissions.

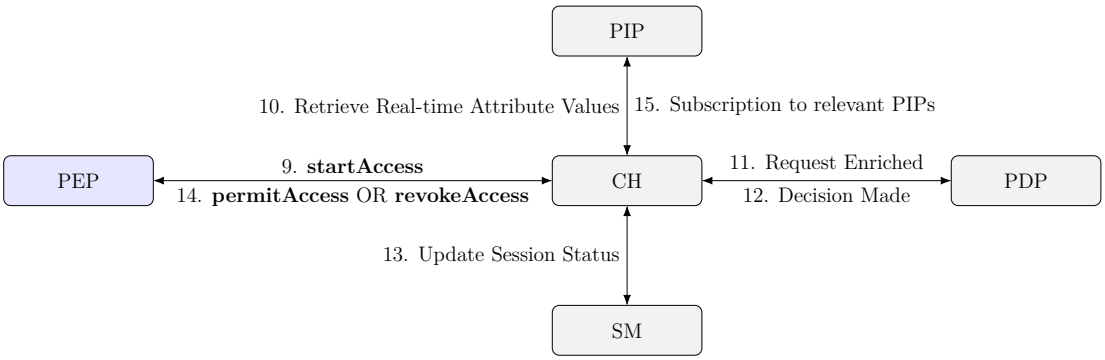

Figure 2.4: Transition from initial request to ongoing session: CH enriches the request with real-time attributes for PDP evaluation

The continuous monitoring capability is a key feature of the UCON framework. If any of the monitored mutable attributes' values change during the session, the CH triggers a policy re-evaluation using the latest data. If the PDP determines that the ongoing conditions are no longer satisfied, a revoke decision is issued. The CH then updates the session status to *REVOKE_ACCESS* and instructs the PEP to revoke access.

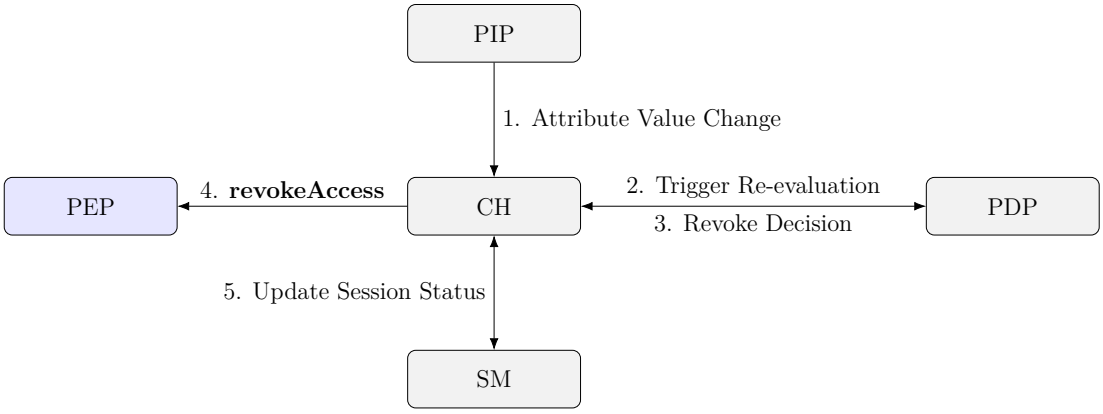

Figure 2.5: Policy re-evaluation during access: Attribute change and potential access revocation

Conversely, if the conditions remain valid, the access grant continues uninterrupted. This dynamic adjustment of access rights exemplifies the UCON framework's superiority over traditional access control mechanisms, which typically assess rights only at the time of the request.

Finally, upon completion of the access, the PEP communicates this to the CH via an *endAccess* message. This message signifies that the session has either concluded naturally or has been revoked due to a policy violation. The CH then instructs the SM to terminate the session, thus ensuring a clear and comprehensive management of access rights throughout the entire process [19].

2.2.2 The Role of XACML in the UCON Framework

The UCON framework leverages the eXtensible Access Control Markup Language (XACML) as a foundational element for specifying and enforcing access control policies. This choice stems from XACML's robust capabilities and its suitability for managing complex and dynamic access scenarios inherent in contemporary systems, including the Internet of Things.

XACML provides a standardized language that facilitates the expression of Usage Control Policies within the UCON framework. The benefits of employing XACML

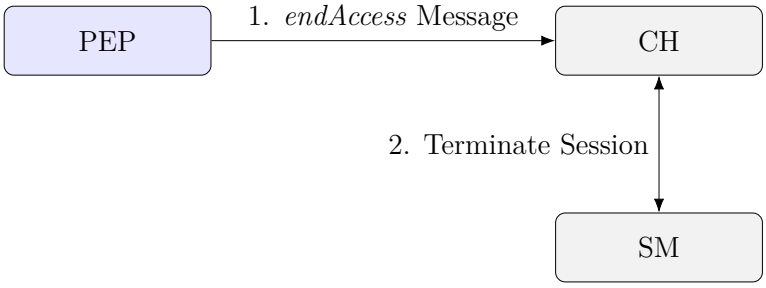

Figure 2.6: Session termination: Communication of access completion and session management

are manifold. Firstly, being a widely recognized standard, XACML enhances interoperability among diverse applications and systems. This interoperability is crucial in scenarios where multiple entities—such as various Attribute Managers and external systems—must collaborate seamlessly. The standardized format allows for consistent policy management and decision-making processes, reducing the complexities associated with integrating disparate systems.

Secondly, the generic nature of XACML enables the deployment of policies across various platforms. A single policy written in XACML can serve multiple applications, thus simplifying policy administration. This characteristic is particularly advantageous in dynamic environments like IoT, where devices and services frequently change. By utilizing a common policy language, organizations can streamline the management of access controls, ensuring that policies are easily adaptable to evolving requirements.

Moreover, XACML supports a distributed approach to policy management. Policies can reference other policies stored remotely, allowing for a modular structure where different users or groups can manage distinct sets of policies. This distributed nature not only facilitates collaborative policy development but also enhances the overall flexibility of the access control model. By integrating results from various policies, XACML provides a comprehensive decision-making framework that reflects the complex interactions present in dynamic systems [20].

Additionally, the power of XACML lies in its support for a wide range of data types and functions, as well as its ability to combine results from different policies effectively. This capability is crucial in scenarios where access decisions depend on various attributes, such as those related to the subject, resource, and environmental conditions. The extensibility of XACML is further complemented by ongoing developments in standards that integrate it with other technologies, such as Security Assertion Markup Language (SAML) and Lightweight Directory Access Protocol

(LDAP). This adaptability positions XACML as a versatile solution for future-proofing access control implementations.

Finally, the flexibility inherent in XACML aligns well with the dynamic nature of modern systems, including the IoT. As environments change—whether through alterations in user roles, resource availability, or contextual factors—XACML's Attribute-Based Access Control (ABAC) model allows for real-time adjustments to access rights. This ensures that access control remains robust and responsive to the needs of the organization, maintaining security while enabling efficient resource use.

In summary, the incorporation of XACML within the UCON framework not only enhances the effectiveness of policy communication and enforcement but also aligns with the evolving landscape of technology, particularly in dynamic environments such as the IoT. By harnessing the capabilities of XACML, the UCON framework is well-equipped to address the complexities of contemporary access control challenges [20].

In addition to the standard XACML features, the UCON framework incorporates U-XACML, an extension specifically designed for expressing Usage Control Policies. U-XACML introduces constructs that address the need for continuous policy enforcement, which is vital for dynamic environments. It allows for conditions to be evaluated at different stages of the access control process through the DecisionTime clause. This clause can specify whether the evaluation occurs at the pre-decision stage (pre) or continuously during access (ongoing) [21].

Furthermore, U-XACML enhances the ObligationExpression element with a similar DecisionTime clause, which defines when obligations should be executed—pre, on, or post the access request. The latter is especially important in usage control scenarios, as it allows for the enforcement of obligations not only before access is granted but also during and after the session, thus ensuring compliance throughout the resource's use.

To handle mutable attributes effectively, U-XACML introduces the ⟨AttrUpdates⟩ element, which details how and when attribute updates should be managed. This element permits updates at various stages: pre, on, or post the access request. By doing so, U-XACML ensures that access decisions remain valid and relevant, adapting to changes in the context or user status as they occur.

In summary, U-XACML's capabilities for pre, ongoing, and post evaluation of conditions and obligations provide a robust framework for continuous access control within the UCON model, aligning seamlessly with the evolving demands of modern systems, including those within the IoT landscape [21].

2.3 A Distributed Architecture for the UCON Framework

As the Internet of Things continues to expand within industrial settings, managing usage control policies becomes increasingly complex due to the diversity of devices and their functionalities. Traditional centralized models of the UCON framework often face limitations in scalability and fault tolerance, primarily because they are ill-equipped to handle the heterogeneous nature of connected devices in Industrial IoT (IIoT) environments. In response to these challenges, a distributed architecture for the UCON framework has been developed, providing a more robust and flexible solution for managing usage control across multiple devices [22].

This distributed architecture operates on a peer-to-peer (P2P) basis, where each device in the IIoT ecosystem functions as an independent node. Each node is responsible for enforcing usage control policies autonomously, allowing for a decentralized approach that diminishes reliance on a central authority. This design not only enhances scalability but also reduces the risk of single points of failure, which are prevalent in traditional centralized systems. In this framework, every node incorporates key components such as the Policy Administration Point, Policy Decision Point, and Policy Information Points. These components work in tandem to evaluate incoming requests and determine the appropriate course of action based on both local and remote attributes [22].

One of the most significant benefits of this architecture is its capacity to utilize data gathered from various devices within the network. This enables the formulation of more complex and context-aware policies that adapt to real-time conditions. By leveraging a Distributed Hash Table, nodes can efficiently share information about their attributes and states, facilitating a comprehensive view of the operational environment [22]. This capability is crucial for maintaining an updated and accurate assessment of the system's status, which is particularly important in dynamic IIoT environments.

In the context of this thesis, the distributed architecture provides a practical framework for evaluating the performance of the Usage Control System. The study specifically examines the implementation of the UCS on constrained devices such as the Raspberry Pi 4. The distributed nature of the architecture allows for effective management and enforcement of usage control policies even on such lightweight hardware, thereby demonstrating the capability of the UCS to operate efficiently in resource-constrained environments.

The distributed architecture significantly enhances fault tolerance as well. In cases where one or more nodes become unavailable, the remaining devices can

still maintain functionality and ensure the enforcement of control policies. This resilience is a key advantage, as it allows the system to continue operating smoothly even in the face of unexpected disruptions. Furthermore, the architecture supports the replication of critical information across multiple nodes, ensuring that essential data is not lost and that the decision-making process remains uninterrupted.

In addition to these advantages, the distributed UCON framework promotes adaptive decision-making by dynamically redistributing responsibilities among the nodes based on their current capabilities and availability. This adaptability is vital in an IIoT setting, where conditions can change rapidly and the operational context may vary significantly between different devices. As a result, the proposed framework not only supports real-time decision-making processes but also enhances the overall security and compliance of the connected devices across the IIoT landscape [22].

In conclusion, the adoption of a distributed architecture for the UCON framework represents a significant advancement in the management of usage control policies within Industrial IoT settings. By decentralizing the enforcement mechanisms and leveraging a P2P structure, the framework is better equipped to handle the complexities and challenges of modern IIoT environments. This innovative approach ensures a more secure, resilient, and adaptable system capable of meeting the demands of a rapidly evolving technological landscape [22].

2.4 Security by Contract

The Security by Contract (S×C) model is a formal approach designed to ensure that applications and devices behave securely by adhering to predefined security policies. It is based on three core elements: the application or device code (A), the contract (C), and the client policy (P) [21, 23]. The primary goal of S×C is to verify that the behavior of an application complies with the security requirements outlined in the policy.

The contract (C) serves as a formal specification of the security-relevant actions that an application or device is expected to perform. This specification encompasses actions such as system API calls, access to sensitive resources, or any security-related operations during execution. In IoT systems, contracts define both the resources required by devices and those they offer to others within the network [23]. On the other hand, the policy (P) outlines acceptable security behaviors and constraints.

A fundamental aspect of the S×C model is the formal relationship between the

application, contract, and policy, expressed through the following transitive condition:

$$A \preceq C \preceq P \implies A \preceq P$$

This relation indicates that if the application behavior conforms to the contract, and the contract conforms to the policy, then the application is deemed compliant with the policy. This ensures that the security constraints defined by the policy are respected throughout the application's execution [21].

The S×C model employs two key processes to enforce compliance:

- *App-Contract Matching* ($A \preceq C$): This process ensures that the contract accurately reflects the behavior of the application or device, verifying that the application's code adheres to the security specifications defined in the contract.
- *Contract-Policy Matching* ($C \preceq P$): In this phase, the contract is compared to the security policy to verify that the contract satisfies the security conditions imposed by the policy. Only if both matches ($A \preceq C$ and $C \preceq P$) succeed, can it be concluded that $A \preceq P$, meaning the application is secure to execute.

In dynamic environments such as IoT networks, this layered approach allows for adaptable security management. As policies evolve to address new threats or changing operational requirements, the S×C model ensures continuous compliance. When a device attempts to join a network, its contract is evaluated to verify compatibility with the network's security policy before integration. Similarly, if the device's software or contract is updated, these matching processes are repeated to maintain compliance.

In cases where either App-Contract Matching or Contract-Policy Matching fails, a dynamic monitoring mechanism is employed. This monitor continuously tracks the application's behavior at runtime, ensuring that any deviation from the contract or violation of the policy is promptly detected and mitigated. Should the application attempt an unauthorized action, the monitor enforces the necessary security restrictions to prevent the violation. While this introduces some overhead, it is critical for maintaining security integrity, especially in unpredictable environments like IoT systems [21].

In conclusion, the Security by Contract model provides a robust method for managing security in complex and dynamic environments such as IoT networks. By clearly defining contracts that outline application behavior and enforcing policies through both static verification and dynamic monitoring, S×C ensures that applications and devices operate securely, even as security conditions change.

2.5 Practical Applications of UCON Framework: The SIFIS-Home Project

The SIFIS-Home project represents a significant advancement in the development of secure smart home applications, integrating the principles of UCON to ensure user privacy and security. This initiative aims to establish a comprehensive framework that facilitates seamless interaction among various smart devices while maintaining a robust security posture tailored to the unique needs of each household. A critical aspect of this project is its focus on user empowerment, enabling individuals to take an active role in managing their data and the devices within their smart home environment.

At the core of SIFIS-Home is a commitment to enhancing user control over their data and devices. By leveraging UCON's dynamic access control mechanisms, SIFIS-Home allows users to specify privacy preferences in real time. This adaptability not only fosters user confidence but also ensures compliance with emerging privacy regulations that increasingly emphasize user consent and data protection [24].

SIFIS-Home aims to provide a secure-by-design software framework that enhances the resilience of interconnected smart home systems across all stack levels. A fundamental feature of this framework is its provision of robust *development tools*, including a software development kit (SDK) that empowers third-party developers to create applications adhering to the security and privacy standards established by SIFIS-Home. This support is crucial for fostering innovation while ensuring that newly developed applications align with security protocols.

Central to this framework is the *application contract system*, where each application is associated with a formal contract that specifies the security-relevant operations it intends to perform. This contract provides clarity on data access and usage, which is essential for aligning with user-defined policies. By linking these contracts with user preferences, SIFIS-Home enables the dynamic enforcement of security measures, ensuring compliance with individual privacy requirements and overall security standards [24]. The use of *development application programming interfaces (dev-APIs)* further enhances application functionality, allowing secure interactions with IoT devices. Each dev-API is crafted with specific functionalities and security considerations, ensuring that operations demanding exclusive access to resources are well-defined.

Additionally, effective *app lifecycle management* is integral to the SIFIS-Home framework, ensuring that applications are properly managed throughout all operational phases, starting from installation. In this context, only applications that comply with predefined security policies can be installed, providing an initial layer

of security. This lifecycle management also includes monitoring app behavior, implementing updates, and dynamically enforcing security policies based on real-time evaluations. As applications interact with IoT devices, requests are assessed against these policies, enabling immediate responses to potential security threats or deviations from user-defined preferences. This comprehensive approach not only enhances user control and trust among smart home users but also strengthens the overall resilience of smart home systems against cyber threats [24].

In conclusion, the SIFIS-Home project exemplifies the practical application of UCON principles to enhance the security and privacy of smart home environments. By fostering user empowerment, equipping developers with essential tools, and emphasizing transparency and user awareness, SIFIS-Home leads the way in the smart home revolution. It ensures that the promise of technology is fulfilled without compromising user trust, ultimately creating a more secure, private, and user-friendly smart home experience [24].

Chapter 3

Proposed Methodology to Enforce S×C Compliance with UCON on Constrained Devices

This chapter introduces a methodology aimed at enforcing the Security-by-Contract paradigm, in IoT environments, through the use of Usage Control, specifically when implemented on constrained devices like the Raspberry Pi 4. The proposed methodology builds upon established frameworks, such as SIFIS-Home—Section 2.5—and leverages the UCON framework to evaluate the S×C compliance conditions.

The UCON framework is particularly evaluated as a means of aligning application behavior with S×C principles, especially addressing the transitive compliance condition discussed in detail in Section 2.4.

Section 3.1 describes the proposed methodology and illustrates how the S×C components—the application, contract, and policy—are represented within UCON to perform the necessary S×C checks.

Following this, a comprehensive series of tests will assess the UCON framework's performance, security implications, and operational reliability on lightweight devices, with results presented in subsequent chapters. The core goal is to verify UCON's capacity to enforce continuous compliance with security policies in real-time by integrating policy-based checks at both the installation and execution stages.

3.1 UCON as a Mechanism for S×C Compliance

The UCON framework is investigated as a potential method for enforcing Security by Contract compliance within IoT environments, specifically in the context of this thesis, where it is deployed on resource-constrained devices such as the Raspberry Pi 4. A key objective of this investigation is to evaluate how effectively UCON can ensure that applications consistently meet the S×C model's transitive compliance condition $A \preceq C \preceq P \implies A \preceq P$—well explained in Section 2.4—throughout the entire applications lifecycle.

In this context, each application, controlling an IoT device, must include a S×C Contract—provided by the app developer—that describes the application's expected behavior. Specifically, this contract is derived from the dev-APIs embedded within the app code, as detailed in Section 2.5. These dev-APIs represent specific functionalities of the application, categorized based on their security relevance. Within the proposed framework, this contract is translated into XACML requests that capture the security-relevant actions expected from the application. This structured set of requests enables the UCON framework to perform evaluations, aligning the app's contract with applicable security policies to ensure that its behavior complies with the defined security preferences.

To achieve S×C compliance, the UCON framework integrates two primary types of security policies:

- **Installation Policies**: Evaluated during the installation of an application, these policies ensure that only applications that meet specific security requirements are allowed on the system. If an app's contract includes dev-APIs capable of manipulating sensitive resources, the installation policy may block the app if it fails to meet predefined security criteria. This proactive evaluation ensures that contracts align with policies (i.e., $C \preceq P$) before an application is permitted to run.
- **Execution Policies**: These policies are evaluated at runtime and determine whether specific dev-API functions can be executed based on the current system context and mutable attributes. When a dev-API action is invoked, UCON checks it against the active execution policies. If the conditions are not satisfied, the operation is denied. This dynamic enforcement is crucial for maintaining continuous compliance with the S×C model, thereby ensuring that application behavior remains consistent with policy requirements during execution.

By integrating these policies, the UCON framework ensures adherence to contracts, thereby facilitating S×C compliance. During the installation process, the framework

performs contract-policy matching (i.e., $C \preceq P$). Each XACML request embedded in the contract is evaluated against the installation policies within the UCON framework. If a Deny decision is rendered at this stage, it signifies that contract-policy compliance is not met. In such cases, either the application is blocked from installation due to non-compliance, or it is installed with active monitoring to enforce $A \preceq P$ at runtime. In this latter scenario, UCON monitors the specific dev-API associated with the Deny decision, preventing any non-compliant action from executing during runtime.

In cases where an app is installed despite initial policy non-compliance, the UCON framework activates dynamic monitoring mechanisms to ensure that compliance is maintained throughout execution. This setup allows UCON to enforce $A \preceq P$ by selectively monitoring dev-API actions that were flagged during installation as potentially non-compliant. This proactive approach is essential for sustaining real-time adherence to security policies, especially in dynamic environments with frequent updates and changing attributes. As a result, compliance is maintained continuously through a layered strategy that combines static verification at installation with ongoing dynamic monitoring during runtime.

If Contract-Policy Matching ($C \preceq P$) fails, UCON promptly enforces runtime restrictions on the application's behavior to avert any policy violations. This layered enforcement strategy not only safeguards the system's integrity but also ensures that applications operate within the security parameters defined by user preferences.

This layered approach allows for adaptable security management within IoT networks. As the policies evolve to address new threats or changing operational requirements, the S×C model ensures ongoing compliance.

Ultimately, this evaluation will reveal UCON's potential as a model for S×C compliance, ensuring that applications align with user-defined security policies while maintaining minimal performance impact.

3.2 Testing Methodology

Following the explanation of the model presented in this thesis for using the UCON framework as a means to enforce S×C compliance, this section now introduces the testing methodologies employed to evaluate its effectiveness, particularly when UCS is deployed on resource-constrained devices like the Raspberry Pi 4. The remainder of this chapter will outline the methods used to assess UCON's capacity to enforce continuous S×C compliance and dynamically manage security policies under real-world conditions.

The testing methodology includes a series of evaluations to validate core aspects of UCON's performance and security capabilities. These tests were designed to examine UCON's effectiveness in enforcing installation and execution policies, its handling of access revocation, and its operational performance under the resource limitations of a lightweight device. Special focus was given to analyzing trade-offs in performance and resource usage to ensure that UCON's policy enforcement mechanisms functioned within acceptable bounds for both security and user experience.

Through rigorous testing, this methodology aims to identify potential performance bottlenecks and assess UCON's capacity to maintain S×C compliance across various operational stages. The findings from these evaluations provide insight into UCON's suitability for constrained environments, highlighting potential adjustments or optimizations required for effective implementation in IoT systems. The following sections will describe these testing approaches in more detail, presenting the structure and objectives of each test type and providing a comprehensive view of the validation process that establishes UCON's feasibility as a reliable security solution for IoT applications.

3.3 Benefits and Challenges of Deploying UCS on Resource-Constrained Device

The deployment of the Usage Control System on resource-constrained devices presents both unique benefits and challenges. In this context, the Raspberry Pi 4 serves as a representative example of such devices. Understanding the advantages and limitations associated with the Raspberry Pi 4 is essential for effectively managing security in Internet of Things environments. Thus, from this point onward, the discussion will refer to the Raspberry Pi 4 whenever practical aspects related to resource-constrained devices are analyzed throughout the thesis.

3.3.1 Benefits

One of the primary advantages of utilizing the Raspberry Pi 4 for UCS implementation is its low cost and accessibility. The Raspberry Pi 4 is a compact, independent computer that runs various distributions of the Linux operating system, offering flexible programming options tailored to user requirements. It features a powerful Broadcom BCM2711 processor, a quad-core Cortex-A72 (ARM v8) 64-bit SoC operating at 1.5GHz, and substantial RAM of 8GB LPDDR4, allowing efficient handling of multiple processes. This processing power makes it well-suited for developing IoT applications in a rapidly evolving technological landscape [25].

Additionally, the Raspberry Pi 4 supports various peripheral connections via its multiple USB ports (2 x USB 3.0 and 2 x USB 2.0). It also offers advanced connectivity options, including Bluetooth 5.0 and Wi-Fi 802.11ac, which are essential for establishing robust wireless communication in smart home environments. The capability to connect to network devices via Gigabit Ethernet further enhances its functionality and versatility.

Moreover, the Raspberry Pi 4's ability to interface with various sensors and actuators allows for real-time implementation of complex security mechanisms, addressing specific threats prevalent in IoT systems. Its GPIO pins and support for protocols like I2C and SPI facilitate the integration of diverse electronic components, contributing to a rich ecosystem for experimentation and development.

Focusing on the Raspberry Pi 4 allows for exploration of the specific benefits it brings to UCS deployment in resource-constrained environments, providing a clearer understanding of how these devices can effectively support security measures in IoT applications.

3.3.2 Challenges

While deploying the Usage Control System on constrained devices like the Raspberry Pi 4 presents several advantages, it also introduces a range of significant challenges. A primary concern is the inherent overhead associated with utilizing Java in embedded systems. The existing implementation of the UCS is available as part of the SIFIS-Home project[1] and is written in Java. Although Java offers benefits such as platform independence and automatic memory management, its execution on low-power devices often proves demanding. The requirement for a Java Virtual Machine (JVM) can lead to considerable resource consumption, affecting both memory and processing power. This consumption may, in turn, compromise overall system performance, especially in scenarios requiring robust security protocols [26].

Furthermore, the limited processing power and memory capacity of devices like the Raspberry Pi can significantly restrict the effectiveness of security measures. Implementing a comprehensive security framework on such resource-constrained platforms often necessitates trade-offs between the robustness of security features and the consumption of available resources. These trade-offs can inadvertently introduce vulnerabilities, thereby undermining the intended security enhancements [27].

The growing adoption of Java in embedded systems has prompted the development

[1]The usage control engine is available as part of the SIFIS-Home project `https://github.com/sifis-home/usage-control.git`

of specialized hardware solutions aimed at accelerating Java execution. However, these improvements may not fully bridge the performance gap when compared to lower-level languages such as C or C++. Although advancements in compiler technology have resulted in more efficient code generation, certain performance-critical applications may still require assembly language programming, further limiting Java's broader adoption in specific contexts [26].

Garbage collection management in Java also introduces unpredictable latencies, which can be detrimental in embedded systems where real-time responsiveness is crucial. While Java's object-oriented design enhances development and maintenance, it may not align well with applications that demand high performance and quick response times. This challenge is compounded by the industry's limited experience with object-oriented languages, as many practitioners often prefer languages with less runtime and memory overhead, influenced by their previous professional experiences [26].

Another significant challenge pertains to the necessity of effective access control mechanisms. Given that UCS is designed to enforce usage control policies, ensuring the efficient functioning of these mechanisms within the constraints of a Raspberry Pi can be complex. The overhead associated with maintaining secure communications and managing access control can strain the device's capabilities, potentially leading to degraded performance or system failures [27].

In summary, while leveraging Java in embedded systems can enhance code organization and maintenance, the associated challenges warrant careful consideration. Evaluating these factors is essential to ensure that UCS solutions implemented on devices like the Raspberry Pi 4 are both effective and efficient, ultimately advancing security in IoT environments.

3.4 Objectives of the Testing Methodologies

This section delineates the key objectives of the testing methodologies designed to assess the efficacy of the UCON framework as a mechanism for achieving Security by Contract compliance, as discussed in Section 3.1. The primary objective of this research is to determine whether UCON can effectively satisfy S×C compliance within resource-constrained environments, particularly when deployed on devices like the Raspberry Pi 4. To accomplish this overarching goal, several critical sub-evaluations will be conducted:

3.4.1 Adherence to S×C Principles

The methodologies will examine how well UCON adheres to the core principles of the S×C model. This involves ensuring that applications consistently meet the transitive compliance condition $A \preceq C \preceq P \implies A \preceq P$. Evaluating this aspect will establish a baseline for UCON's effectiveness in maintaining security compliance.

3.4.2 Dynamic Monitoring Capabilities

The effectiveness of UCON's dynamic monitoring capabilities will be evaluated, particularly regarding its ability to enforce runtime restrictions and maintain compliance during application execution. Understanding how UCON adapts to runtime conditions will be crucial for its deployment in dynamic environments.

3.4.3 Performance Verification

A crucial objective is to rigorously verify the performance of the UCS when deployed on a Raspberry Pi 4. This involves a comprehensive assessment of how it manages security checks under the constraints imposed by limited resources. By conducting detailed performance measurements, the testing aims to determine whether the system can maintain an acceptable level of responsiveness while effectively enforcing usage control policies. It is essential to evaluate not only the time required for various operations but also the system's overall stability during operation. This analysis will facilitate the identification of potential bottlenecks that could adversely affect user experience or compromise system reliability.

3.4.4 Feasibility Assessment of Security Features

The methodologies are specifically crafted to evaluate the feasibility of implementing robust security features on constrained hardware. Given the challenges associated with utilizing Java—along with its inherent resource overhead—on devices like the Raspberry Pi, the testing will investigate whether the UCS can successfully enforce access control policies without compromising performance. This objective is critical for ensuring that security measures do not lead to unacceptable inconsistency time, especially in scenarios requiring real-time decision-making, where any delay could result in significant security risks. Thus, it is imperative to ascertain that the integration of security features aligns with the operational capabilities of the device.

3.4.5 User Experience Evaluation

User experience is a fundamental aspect of deploying UCS in domestic settings, as it directly influences the adoption and usability of the system. The testing methodologies aim to assess how the performance of the system impacts user interactions during both the installation and operational phases. This includes measuring the response times of access requests and monitoring how users perceive the system's responsiveness during various scenarios.

3.4.6 Identification of Performance Bottlenecks

To effectively address the challenges outlined in the previous section, a significant objective of the testing is to systematically identify any performance bottlenecks that may arise during the operation of the UCS on the Raspberry Pi. This involves conducting a series of tests under varying conditions to uncover inefficiencies in the system's architecture or implementation that could hinder its effectiveness. Understanding these bottlenecks will not only yield valuable insights into potential areas for optimization but also guide future iterations of the UCS to ensure more robust and efficient deployments.

3.4.7 Revocation and Policy Management Analysis

Another critical objective is to analyze the efficiency of revocation processes and policy management within the UCS framework. Given the necessity for effective access control mechanisms in a resource-constrained environment, the testing will evaluate how swiftly and accurately the system can respond to changes in usage control policies and manage revocations. A key concept in this context is the "inconsistency time," which refers to the period between the moment a change in an attribute triggers a re-evaluation and revocation, and the moment the PEP terminates access to the resource. This analysis is essential for ensuring that the UCS can maintain security in real-time, particularly in the context of smart home applications, where unauthorized access can have severe implications for user privacy and safety. The methodologies will explore various scenarios to simulate real-world conditions, providing a comprehensive understanding of the system's capabilities in policy management.

3.4.8 Exploration of Adaptations for Lightweight Systems

Finally, the methodologies will delve into potential adaptations to the UCS that may enhance its performance and security in a lightweight environment. This includes investigating alternative configurations or deployment strategies that optimize resource usage while ensuring the effectiveness of security measures. The objective

is to identify innovative approaches that strike a balance between the need for robust access control and the limitations imposed by the hardware. By exploring these adaptations, the research aims to contribute to the broader discourse on developing effective security solutions for IoT devices, particularly in constrained environments.

In conclusion, the objectives outlined in this section focus on validating the deployment of the UCS on resource-constrained platforms such as the Raspberry Pi 4. By rigorously testing performance, user experience, security features, and management capabilities, this research aims to establish whether the UCS can effectively operate in real-world domestic environments while overcoming the challenges associated with resource constraints. The insights gained from these methodologies will provide a solid foundation for enhancing the security posture of IoT devices and contribute to the development of practical solutions for smart home technologies.

3.5 Test Framework Architecture and Environment Setup

The architecture of the test framework is meticulously designed to evaluate the Usage Control System in a resource-constrained environment, specifically utilizing the capabilities of a Raspberry Pi 4 equipped with 8 GB of RAM. This section outlines the framework's architecture, highlights the software prerequisites, and elaborates on the setup of the testing environment, providing a comprehensive understanding of the components involved and their interactions. The entire designed test framework can be accessed on thesis repository [2].

3.5.1 Software Prerequisites

To effectively run the tests within this framework, a set of software components is required. These include:

- **Java 8**: Essential for executing the UCS, Java provides the runtime environment necessary for the system's functionalities.
- **Maven**: This build automation tool is used to manage project dependencies and facilitate the packaging of the UCS into a runnable JAR file.
- **Python**: Employed for scripting various test scenarios, Python serves as the language for the Policy Enforcement Point (PEP) components that interact with the UCS.

[2]Test Framework respository `https://sssg-dev.iit.cnr.it`

- **Docker**: Utilizing Docker enables the encapsulation of the distributed hash table (DHT) functionalities within containers, ensuring a consistent and isolated execution environment across different machines.

These software components work in unison to establish a robust testing environment that simulates a real-world scenario where the UCS operates in conjunction with various PEPs.

3.5.2 Testing Framework Architecture

The test framework is built upon a distributed architecture designed to optimize the efficiency and reliability of the UCS within a resource-constrained environment. Central to this architecture is the distributed hash table (DHT) component, `sifis-alpine-dht-arm64v8`[3], which enables the UCS to operate seamlessly across multiple devices. This architecture facilitates robust communication and data sharing among various UCS instances, which is essential for managing access control in dynamic environments.

A key feature of this architecture is its use of WebSockets for communication, allowing for real-time interactions between the Policy Enforcement Points and the UCS. This mechanism ensures that the PEPs can make access requests to the UCS, while the UCS promptly communicates its access decisions back to the PEPs based on policy evaluations. Additionally, the DHT supports the rapid dissemination of changes in access control policies across all nodes, ensuring that any updates—such as the addition, removal, or modification of policies—are immediately reflected throughout the system. This responsiveness is crucial for scenarios where immediate enforcement of access control decisions is required across multiple devices.

The testing framework is deployed on a Raspberry Pi 4, configured with Raspberry Pi OS, a lightweight operating system that enhances performance by minimizing resource overhead. This OS is optimized for the hardware specifications of the Raspberry Pi, ensuring that the UCS and PEP components can run efficiently without consuming excessive system resources. The choice of this operating system contributes to the overall stability and responsiveness of the testing environment.

The usage control engine is available as part of the SIFIS-Home project[4].

[3] `https://github.com/sifis-home/libp2p-rust-dht/pkgs/container/sifis-alpine-dht-arm64v8`

[4] `https://github.com/sifis-home/usage-control.git`

3.5.3 Workflow of the Testing Process

The workflow of the testing process follows a systematic approach that emphasizes automation and repeatability. At the outset of each test campaign, the PEP script initializes the UCS, ensuring that it is ready to enforce the defined policies.

Each test campaign is methodically designed to evaluate different aspects of the UCS, such as installation, performance, and the revocation process. By leveraging automation, the testing framework allows for multiple iterations of each test scenario, creating batches that help to minimize the impact of external inefficiencies on the results. This systematic approach not only enhances the reliability of the outcomes but also facilitates a more comprehensive assessment of the UCS's capabilities.

The automated testing scripts are structured to send multiple access requests to the UCS by leveraging the DHT, capturing responses and logging timing data for subsequent analysis. By enabling repeated executions of each test, the framework allows for a robust evaluation that accounts for potential variances in performance due to external factors. This design is essential for isolating the effects of specific changes made to the UCS, ensuring that any observed variations can be attributed to the modifications rather than fluctuations in the testing environment.

Furthermore, while the architecture supports the integration of multiple Raspberry Pi devices, all tests were conducted on a single Raspberry Pi unit, where both the PEP and UCS, along with the DHT, reside. This setup simplifies the testing process while still demonstrating the scalability and effectiveness of the UCS's performance across a distributed architecture. The ability to test in such a flexible environment not only demonstrates the adaptability of the UCS but also highlights its potential for real-world applications in smart homes and other IoT scenarios.

3.5.4 Architecture Overview

In summary, the architecture of the test framework, combined with the lightweight configuration of the Raspberry Pi OS, establishes a robust environment for evaluating the UCS in a resource-constrained setting. The thoughtful orchestration of components, paired with the automated testing processes, ensures a systematic and thorough examination of the system's capabilities. This setup lays a solid foundation for the subsequent analysis of test results, providing insights into the UCS's performance and its interactions within a distributed architecture.

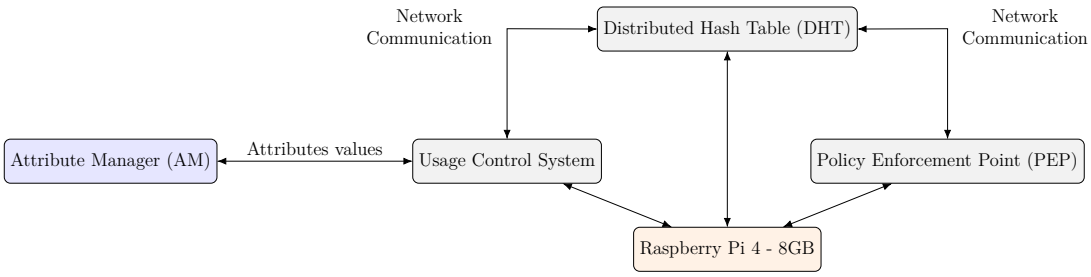

Figure 3.1: Architecture of the Test Framework: Interaction between the UCS, PEP and DHT on the Raspberry Pi

3.6 Testing Categories and Strategies

This section outlines the various testing categories and strategies employed to assess the performance and security of the User Control System (UCS) deployed on the Raspberry Pi 4. Each testing category serves a distinct purpose, ensuring a thorough evaluation of the system's capabilities and limitations. The focus here will be on the *installation tests*, which are critical for understanding the overhead and performance of the UCS during its initial setup and operation.

3.6.1 Installation Tests

The installation tests are designed to evaluate the overhead introduced during the application installation process, focusing on two key aspects: the efficiency of the installation procedure and the performance of the policy enforcement mechanism. These tests are pivotal in assessing the feasibility of deploying the User Control System on resource-constrained devices such as the Raspberry Pi 4, particularly in ensuring compliance with Security by Contract principles.

By validating the alignment between application contracts and corresponding policies, these tests confirm that the UCS effectively enforces compliance, thereby enhancing security in domestic settings. This evaluation is crucial for identifying performance bottlenecks and optimization opportunities that contribute to user satisfaction and overall system usability.

The tests were structured with the following key goals in mind:

- **Evaluating Installation Time Overhead:** This goal measures the time required to install an application, particularly in smart home environments where IoT devices are managed. A fast and efficient installation process is vital for user adoption, as long installation times could lead to dissatisfaction and hinder the system's practicality.

- **Assessing Contract-Policy Matching Time:** Another core objective is to evaluate the UCS's policy enforcement efficiency. In this context, it is important to remind that the contract is composed of multiple XACML requests, each representing a distinct security-relevant operation the application intends to perform. Therefore, the matching process between contract and policy involves verifying each individual request. This metric is critical for user experience, as delays in processing these requests may prevent users from promptly performing their desired actions.

To rigorously assess these metrics, several scripts and automated processes were implemented to minimize manual intervention. This includes scripts designed to generate policies and requests, allowing users to specify test directories and numbers (maximum 10) for both elements. This automation ensures consistency across test iterations and facilitates large-scale testing under uniform conditions, which is essential for generating reliable data.

Python Script as Policy Enforcement Point (PEP). A central component of the testing process is a Python script which simulates the Policy Enforcement Point (PEP) interacting with the UCS. This script registers itself with the UCS, initializes it with the predefined policies, and manages the *tryAccess* and *startAccess* requests during the installation phase.

The main tasks performed by the script include:

- **UCS configuration:** The script initializes the UCS by registering policies and preparing the system to process incoming requests.
- **Request Management:** It sends a predefined number of *tryAccess* requests, recording the time at which the first request is dispatched. For each *tryAccess* response evaluated as *Permit*, it immediately sends a corresponding *startAccess* request and logs the time when the last *startAccess* response is received.
- **Result Logging:** Timing data is logged for later analysis, providing insights into the total time taken for policy evaluation and request handling.

Following is the Python code—accessible on thesis repository—that illustrates the functionality of the PEP from a conceptual point of view. This is included in this thesis as a foundational element for subsequent test scenarios, requiring only minor modifications to adapt to different testing contexts.

```
1 import json
2 import websocket
3 from pathlib import Path
4 import base64
5 import uuid
6 import datetime
```

```
7  import argparse
8  import sys
9  import time
10
11 # WebSocket URI for connection
12 websocket_uri = "ws://localhost:3000/ws"
13 req_number = 1         # Number of requests to send
14 pip_number = 1         # Number of PIPs to add
15 pip_counter = 0        # Counter for added PIPs
16 policy_counter = 0     # Counter for added policies
17 t_i = 0                # Start timestamp
18 t_f = 0                # End timestamp
19 req_completed = 0      # Completed requests count
20 verbose = False        # Verbose output flag
21 policy_dir = ""        # Directory for policies
22 request_dir = ""       # Directory for requests
23 output_dir = ""        # Directory for output results
24 nexec = 1             # Execution number
25
26 def on_message(ws, message):
27     # Handle incoming messages from the WebSocket
28         if "ucs-command" in message:
29                 print("\nReceived message from the ucs:")
30                 parsed = json.loads(message)
31                 print("type: " +
                   ↪  parsed["Volatile"]["value"]["command"]["value"]["message"]["purpose"])
32                 if verbose:
33                         print(json.dumps(parsed, indent=2))
34                         print("\n--------------------------------\n")
35
36                 # Handle error response
37                 if parsed["Volatile"]["value"]["command"]["value"]["message"]["purpose"] ==
                   ↪  "ERROR_RESPONSE":
38                         exit(parsed["Volatile"]["value"]["command"]["value"]["message"]["description"])
39
40                 # Handle try response
41                 if parsed["Volatile"]["value"]["command"]["value"]["message"]["purpose"] ==
                   ↪  "TRY_RESPONSE":
42                         if
                           ↪  parsed["Volatile"]["value"]["command"]["value"]["message"]["evaluation"]
                           ↪  != "Permit":
43                                 exit("TryAccess evaluation is not Permit.")
44                         session_id =
                           ↪  parsed["Volatile"]["value"]["command"]["value"]["message"]["session_id"]
45                         start_access(session_id)
46
47                 # Handle start response
48                 if parsed["Volatile"]["value"]["command"]["value"]["message"]["purpose"] ==
                   ↪  "START_RESPONSE":
49             if parsed["Volatile"]["value"]["command"]["value"]["message"]["evaluation"] !=
               ↪  "Permit":
50                                 exit("StartAccess evaluation is not Permit.")
51                         global req_completed
52                         req_completed += 1
53                         if req_completed == req_number:
54                                 global t_f
55                                 t_f = int(datetime.datetime.now().timestamp()*1000)
56                                 save_results()
57
58                 # Handle register response
```

```
59                  if parsed["Volatile"]["value"]["command"]["value"]["message"]["purpose"] ==
                    ↪  "REGISTER_RESPONSE":
60                          if parsed["Volatile"]["value"]["command"]["value"]["message"]["code"] !=
                            ↪  "OK":
61                                  print("Unable to register the PEP")
62                                  exit()
63                          add_policies()
64
65                  # Handle add policy response
66                  if parsed["Volatile"]["value"]["command"]["value"]["message"]["purpose"] ==
                    ↪  "ADD_POLICY_RESPONSE":
67                          global policy_counter
68                          policy_counter += 1
69                          if policy_counter == num_policies:
70                                  print("All Policies added.")
71                                  print("We can start adding PIPs...")
72                                  add_pips()
73
74                  # Handle add PIP response
75                  if parsed["Volatile"]["value"]["command"]["value"]["message"]["purpose"] ==
                    ↪  "ADD_PIP_RESPONSE":
76                          global pip_counter
77                          pip_counter += 1
78                          if pip_counter == pip_number:
79                                  print("All PIPs added (" + str(pip_number) + ")")
80                                  print("We can start making the requests...")
81                                  time.sleep(5) # Give UCON time to save the state
82                                  make_requests()
83
84  def on_error(ws, error):
85              print(error)
86
87  def on_close(ws, close_status_code, close_msg):
88              print("### Connection closed ###")
89
90  def on_open(ws):
91          time.sleep(10)  # Wait before sending the register command
92          print("### Connection established ###")
93          print("[ " + websocket_uri + " ]")
94          register()
95
96  def print_and_send(json_out):
97      # Print and send the JSON output
98          if verbose:
99                  print("Message sent:")
100                 print(json.dumps(json_out, indent=2))
101         ws.send(json.dumps(json_out))
102
103 def register():
104     # Prepare registration request for the PEP
105     ws_req = {
106                 "RequestPubMessage": {
107                         "value": {
108                         "timestamp": int(datetime.datetime.now().timestamp()*1000),
109                         "command": {
110                     "command_type": "pep-command",
111                     "value": {
112                         "message": {
113                             "purpose": "REGISTER",
114                             "message_id": str(uuid.uuid1()),
115                             "sub_topic_name": "topic-name-the-pep-is-subscribed-to",
```

```
116                             "sub_topic_uuid": "topic-uuid-the-pep-is-subscribed-to"
117                         },
118                         "id": "pep-websocket_client",
119                         "topic_name": "topic-name",
120                         "topic_uuid": "topic-uuid-the-ucs-is-subscribed-to"
121                     }
122                 }
123             }
124         }
125     }
126     print("\n---------- REGISTER ------------\n")
127     print_and_send(ws_req)
128 
129 def make_requests():
130     # Start making access requests
131     global t_i
132     t_i = int(datetime.datetime.now().timestamp()*1000)
133     for x in range(1, req_number + 1):
134         try_access(x)
135 
136 def try_access(n):
137     # Prepare and send a TryAccess request
138     num = str(n)
139     request_file = Path(f'{request_dir}/request_{num}.xml')
140     if not request_file.exists():
141         print(f"Request file {request_file} does not exist.")
142         return
143     request = request_file.read_text()
144     if verbose:
145         print("XACML request used:")
146         print(request)
147     b = base64.b64encode(bytes(request, 'utf-8'))
148     request64 = b.decode('utf-8')
149 
150     ws_req = {
151                 "RequestPubMessage": {
152                             "value": {
153                                 "timestamp": int(datetime.datetime.now().timestamp()*1000),
154                                 "command": {
155                                             "command_type": "pep-command",
156                                             "value": {
157                                                 "message": {
158                                                             "purpose": "TRY",
159                                                             "message_id": str(uuid.uuid1()),
160                                                             "request": request64,
161                                                             "policy": None
162                                                 },
163                                                 "id": "pep-websocket_client",
164                                                 "topic_name": "topic-name",
165                                                 "topic_uuid":
                                                 ↪  "topic-uuid-the-ucs-is-subscribed-to"
166                                             }
167                                 }
168                             }
169                 }
170     }
171     print("\n--------- TRY ACCESS -----------\n")
172     print_and_send(ws_req)
173 
174 def start_access(s_id):
175     # Prepare and send a StartAccess request
```

```
176     ws_req = {
177         "RequestPubMessage": {
178             "value": {
179                 "timestamp": int(datetime.datetime.now().timestamp()*1000),
180                 "command": {
181                     "command_type": "pep-command",
182                     "value": {
183                         "message": {
184                             "purpose": "START",
185                             "message_id": str(uuid.uuid1()),
186                             "session_id": s_id
187                         },
188                         "id": "pep-websocket_client",
189                         "topic_name": "topic-name",
190                         "topic_uuid": "topic-uuid-the-ucs-is-subscribed-to"
191                     }
192                 }
193             }
194         }
195     }
196     print("\n-------- START ACCESS ----------\n")
197     print_and_send(ws_req)
198 
199 def add_policies():
200     # Add all policies defined in the directory
201         for x in range(1, num_policies + 1):
202                 add_policy(x)
203 
204 def add_policy(n):
205     # Prepare and send an AddPolicy request
206     num = str(n)
207     policy_file = Path(f'{policy_dir}/policy_' + num + '.xml')
208     if not policy_file.exists():
209         print(f"Policy file {policy_file} does not exist.")
210         return
211     policy = policy_file.read_text()
212     if verbose:
213         print("XACML policy used:")
214         print(policy)
215     b = base64.b64encode(bytes(policy, 'utf-8'))
216     policy64 = b.decode('utf-8')
217 
218     ws_req = {
219                 "RequestPubMessage": {
220                         "value": {
221                 "timestamp": int(datetime.datetime.now().timestamp()*1000),
222                 "command": {
223                     "command_type": "pap-command",
224                     "value": {
225                         "message": {
226                             "purpose": "ADD_POLICY",
227                             "message_id": str(uuid.uuid1()),
228                             "policy": policy64,
229                             "policy_id": "policy_" + num
230                         },
231                         "id": "pap-web_socket",
232                         "topic_name": "topic-name",
233                         "topic_uuid": "topic-uuid-the-ucs-is-subscribed-to"
234                     }
235                 }
236             }
```

```
        }
    }
    print("\n--------- ADD POLICY -----------\n")
    print_and_send(ws_req)

def add_pips():
    # Add all PIPs defined
    print("Running add_pips()")
    for x in range(1, pip_number + 1):
        print("Running add_pip")
        add_pip_reader(x)

def add_pip_reader(n):
    # Prepare and send an AddPIP request
    num = str(n)
    attribute_id = "urn:my-namespace:1.0:environment:attribute-" + num
    category = "urn:oasis:names:tc:xacml:3.0:attribute-category:environment"
    data_type = "http://www.w3.org/2001/XMLSchema#string"
    attribute_value = "attribute-value-" + num
    file_name = "attribute-" + num + ".txt"
    refresh_rate = 1000

    ws_req = {
                "RequestPubMessage": {
            "value": {
                "timestamp": int(datetime.datetime.now().timestamp()*1000),
                "command": {
                    "command_type": "pip-command",
                    "value": {
                        "message": {
                            "purpose": "ADD_PIP",
                            "message_id": str(uuid.uuid1()),
                            "pip_type": "it.cnr.iit.ucs.pipreader.PIPReader",
                            "attribute_id": attribute_id,
                            "category": category,
                            "data_type": data_type,
                            "refresh_rate": refresh_rate,
                            "additional_properties": {
                                attribute_id: file_name,
                                file_name: attribute_value
                            }
                        },
                        "id": "pip-reader-" + num,
                        "topic_name": "topic-name",
                        "topic_uuid": "topic-uuid-the-ucs-is-subscribed-to"
                    }
                }
            }
        }
    }
    print("\n------ ADD PIP READER -------\n")
    print_and_send(ws_req)

def save_results():
    # Save execution results to the output directory
    global req_number, nexec
    if output_dir:
        Path(output_dir).mkdir(parents=True, exist_ok=True)
        file_path = Path(f"{output_dir}/out{req_number}{nexec}.txt")
        with open(file_path, "a+") as f:
            f.write(f"t_f = {t_f}\n")
```

```
298             f.write(f"t_i = {t_i}\n")
299         print(f"Results saved to {file_path}")
300
301 if __name__ == "__main__":
302     # Command-line argument parsing
303     parser = argparse.ArgumentParser(description='FindPolicy batch test.')
304     parser.add_argument('-v', '--verbose', action='store_true', help="Enable verbose output")
305     parser.add_argument('-r', '--requests', type=int, default=1, help="Number of requests to
        ↪ send")
306     parser.add_argument('-p', '--pips', type=int, default=1, help="Number of PIPs to add")
307     parser.add_argument('--policies', type=int, default=1, help="Number of policies to add")
308     parser.add_argument('--policy-dir', type=str, required=True, help="Directory containing the
        ↪ policies")
309     parser.add_argument('--request-dir', type=str, required=True, help="Directory containing the
        ↪ requests")
310     parser.add_argument('--output-dir', type=str, required=True, help="Directory to save the
        ↪ output results")
311     parser.add_argument('--nexec', '--number-execution', type=int, default=1, help="Execution
        ↪ number for output diffs")
312     args = parser.parse_args()
313     req_number = args.requests
314     pip_number = args.pips
315     verbose = args.verbose
316     num_policies = args.policies
317     policy_dir = args.policy_dir
318     request_dir = args.request_dir
319     output_dir = args.output_dir
320     nexec = args.nexec
321     # Initialize WebSocket connection
322     ws = websocket.WebSocketApp(websocket_uri,
323                                 on_open=on_open,
324                                 on_message=on_message,
325                                 on_error=on_error,
326                                 on_close=on_close)
327     ws.run_forever()
```

Bash Script for Automating Test Executions. To streamline the testing process, a Bash script was developed to automate repeated executions of the Python script under different test conditions. Automation is essential for ensuring both efficiency and consistency across multiple iterations, eliminating the need for manual intervention and reducing the possibility of human error. The decision to use Bash scripting was made to avoid unnecessary overhead on the Raspberry Pi system.

The Bash script handles the following tasks:

- **Automating Multiple Test Runs:** By repeatedly executing the Python script, the Bash script allows data to be collected across various test scenarios, such as varying numbers of requests and policies. This repeated execution ensures statistically significant results by averaging out any temporary system anomalies, such as CPU spikes or memory usage fluctuations.
- **Initializing the DHT:** At the beginning of the batch test, the script initializes the DHT to ensure a completely reset testing framework, allowing for

comprehensive assessments of the interactions between the components.

- **Resetting the UCS:** Before each new test run, inside the batch test, the script resets the UCS to ensure that no residual memory overhead or cached data from previous runs impacts the current iteration. This guarantees that each test starts from a clean state, ensuring data integrity.
- **Managing Temporary Files:** Output from each test run is stored in temporary files, which are later processed and analyzed. This method allows for easy tracking of results and helps identify anomalies or irregularities in the data.

The following Bash script is included not only to demonstrate the implementation of the testing framework but also because it serves as a crucial component that can be adapted for subsequent test types, thereby emphasizing its relevance within the context of this thesis.

```
#!/bin/bash
# Enable to debug:
# set -x
# Default values
MAX_REQUEST=10
MIN_REQUEST=1
VERBOSE=false

# Function to print help message
print_help() {
    echo "Usage: $0 [-r requests] [-t times] [--td test-dir] [-v] [--ps python-script] [--ucsdht
    ↪  UCSDht-jar-path] [UCS_OPTIONS] [-h]"
    echo
    echo "Options:"
    echo "  -r, --requests             Number of requests to send in a single test execution"
    echo "  -t, --times                Number of times to run the complete test"
    echo "  --td test-dir              Path to the root directory of the tests"
    echo "  -v, --verbose              Enable verbose output"
    echo "  --ps python-script         Name of the Python script to run for a single test"
    echo "  --ucsdht UCSDht-jar-path   Path to the UCSDht.jar file"
    echo "  UCS_OPTIONS                Additional options for the UCS execution"
    echo "  -h, --help                 Show this help message"
    exit 1
}

# Parse command-line arguments
OPTIONS=$(getopt -o r:t:v --long requests:,times:,td:,verbose,ps:,ucsdht:,help -- "$@")
if [ $? -ne 0 ]; then
    echo "Error parsing options"
    exit 1
fi
eval set -- "$OPTIONS"

# Initialize variables with default values
req=""
tms=""
TEST_DIRECTORY=""
PYTHON_SCRIPT=""
```

```
38 UCSDHT_JAR_PATH=""
39 UCS_OPTIONS=""
40
41 while true; do
42     case "$1" in
43         -r|--requests) req="$2"; shift 2;;
44         -t|--times) tms="$2"; shift 2;;
45         --td) TEST_DIRECTORY="$2"; shift 2;;
46         -v|--verbose) VERBOSE=true; shift;;
47         --ps) PYTHON_SCRIPT="$2"; shift 2;;
48         --ucsdht) UCSDHT_JAR_PATH="$2"; shift 2;;
49         -h|--help) print_help; shift;;
50         --) shift; break;;
51         *) echo "Internal error!"; exit 1;;
52     esac
53 done
54
55 # Collect remaining arguments as UCS_OPTIONS
56 UCS_OPTIONS="$@"
57
58 # Validate parameters
59 if [ -z "$req" ] || [ -z "$tms" ] || [ -z "$TEST_DIRECTORY" ] || [ -z "$PYTHON_SCRIPT" ] || [ -z
   ↪  "$UCSDHT_JAR_PATH" ]; then
60     echo "Error: All parameters must be specified."
61     print_help
62 fi
63
64 if [ "$req" -lt "$MIN_REQUEST" ] || [ "$req" -gt "$MAX_REQUEST" ]; then
65     echo "Incorrect parameter values. See help for usage."
66     exit 1
67 fi
68
69 # Directories for request and policy outputs
70 REQUEST_DIR="$TEST_DIRECTORY/requests"
71 POLICY_DIR="$TEST_DIRECTORY/policies"
72
73 # Clean up and create output directories
74 rm -rf "$TEST_DIRECTORY/UCSOutputs" "$TEST_DIRECTORY/PEPOutputs"
75 mkdir "$TEST_DIRECTORY/UCSOutputs" "$TEST_DIRECTORY/PEPOutputs"
76
77 # Run the test
78 docker_cmd_dht="docker run -d --network host ghcr.io/sifis-home/sifis-alpine-dht-arm64v8
   ↪  --shared-key a789c015f35fef58fa18f4c36ace5fa4b466a1900709fe64fd32f69aa14de289"
79 lxterminal -e bash -c "$docker_cmd_dht; exec bash"
80 sleep 5
81
82 # Execute tests
83 for i in $(seq $tms); do
84     for j in $(seq $req); do
85         echo "START PEP [tms: $i - req: $j - pip 1 - Npolicy: $j]"
86
87         # Start the Python script for PEP
88         if [ "$VERBOSE" = true ]; then
89             lxterminal -e python $PYTHON_SCRIPT -v -r $j -p 1 --policies $j --policy-dir
                ↪  $POLICY_DIR --request-dir $REQUEST_DIR --output-dir $TEST_DIRECTORY/PEPOutputs
                ↪  --nexec $i &
90         else
91             lxterminal -e python $PYTHON_SCRIPT -r $j -p 1 --policies $j --policy-dir $POLICY_DIR
                ↪  --request-dir $REQUEST_DIR --output-dir $TEST_DIRECTORY/PEPOutputs --nexec $i &
92         fi
93
```

```
94          echo "START UCS"
95          # Run the UCS with provided options
96          java -jar $UCSDHT_JAR_PATH --hard-reset $UCS_OPTIONS 2>&1 | grep "INFO" >
            ↪  "$TEST_DIRECTORY/UCSOutputs/out${i}${j}.txt" &
97
98          sleep 50
99
100         echo "STOP PEP: $PYTHON_SCRIPT"
101         # Stop the PEP Python script
102         pkill -9 -f "python $PYTHON_SCRIPT -${VERBOSE:+v} -r $j -p 1 --policies $j"
103
104         echo "STOP $UCSDHT_JAR_PATH"
105         pkill -9 -f "java -jar $UCSDHT_JAR_PATH --hard-reset"
106     done
107 done
108
109 # Clean up Docker containers
110 docker stop $(docker ps -q)
111 sleep 10
112 docker rm $(docker ps -aq)
113
114 # Call the combination_outputs script
115 echo "Calling combination_outputs script"
116 lxterminal -e python $TEST_DIRECTORY/combination_outputs_installation_testBatch.py -r $req -t $tms
    ↪  --td $TEST_DIRECTORY &
117 sleep 180
118 echo "Installation batch test concluded"
```

Throughout the installation tests, several configurations were used to simulate different real-world scenarios. These configurations varied the number of requests and policies to examine how the UCS scales and performs under different loads.

Key configurations included:

- **Varying Numbers of Requests and Policies:** The number of requests ranged from 1 to 10, with the number of policies always set equal to the number of requests. This ensured that each request was matched to a unique policy, avoiding any caching optimizations that could artificially enhance performance.
- **Repetition for Statistical Accuracy:** Each test configuration was executed 20 times to ensure statistically reliable results. This repetition helped eliminate the influence of random factors such as system load or background processes, ensuring a clear understanding of system performance under different conditions.

3.6.2 Running Tests

The process of enforcing security policies during runtime involves continuously verifying that the actions executed by the application code A conform to the client policy P. This enforcement mechanism is critical to ensuring that the behavior of

the application, as defined by its contract C, remains within the boundaries of the policy's security requirements. Whenever a monitored dev-API is attempted at runtime—as explained in Section 2.5—the Usage Control System must evaluate the action against the policy to either permit or deny it.

The running tests are designed to assess how effectively the UCS enforces these security policies during runtime, especially as the number of attributes involved in the evaluation increases. As policies become more complex, due to a rise in the number of attributes, the system's resource requirements also grow. Therefore, it becomes essential to evaluate how efficiently the UCS can handle this complexity while ensuring that performance remains within acceptable limits.

These tests focus on assessing the performance impact caused by the number of Policy Information Points, which correspond to the attributes in the policies. Unlike installation tests, which measured the efficiency of installing applications, these running tests analyze the authorization flow of an application already installed in a smart home environment. Specifically, they examine the timing required to evaluate an access request made by the application at runtime in order to perform an action on a resource, ensuring that the action complies with the policy P.

As the number of attributes in the policy increases, the complexity of the evaluation process escalates. This is due to the need for the UCS to involve a greater number of PIPs, each of which corresponds to an individual attribute in the policy. Consequently, each PIP must interface with an Attribute Manager responsible for handling the specific attribute's values. This interaction can lead to significant stress on the system during policy evaluations, thereby increasing the overall workload on the platform.

To isolate the effects of policy complexity on performance, each running test evaluates a single request against varying numbers of PIPs. This focused approach allows for a clear assessment of how the system's performance metrics change as more attributes are processed during policy evaluation without the interference of additional requests, which could introduce overhead and complicate the results.

Key metrics, such as response times for different phases of request handling—specifically *tryAccess* and *startAccess*—are analyzed. Monitoring these times is crucial, as any substantial increase in response time as the number of PIPs grows could indicate areas for optimization.

To run these running tests efficiently, adaptations were made to existing automation scripts to handle the increased number of PIPs. The initialization of the UCS with the correct policy setup is performed while ensuring that the timing of each phase is logged for analysis. Automation is employed to streamline the testing process, enabling multiple configurations to be tested without manual intervention.

This consistency across test iterations is critical for gathering reliable data across different scenarios.

The automation process allows for rapid test execution and provides the ability to repeat tests across different configurations, ensuring that the results are statistically significant. Each configuration executes multiple times to account for potential anomalies, such as CPU spikes or variations in memory usage, which could otherwise skew the results.

In this series of tests, the number of PIPs varies from 1 to 50, with each PIP representing an attribute in the policy. The decision to keep the number of requests fixed at one per test allows for a focus exclusively on the impact of policy complexity. To summarize, in this batch, each test includes:

- A single request sent to the UCS.
- A policy utilized in the test with a varying number of attributes, corresponding to the number of PIPs involved in the evaluation.
- Evaluation of every request against a corresponding policy, ensuring that each *tryAccess* request results in a *Permit*, and each *startAccess* request also results in a *Permit*.

3.6.3 Revocation Tests

Revocation tests play a critical role in verifying the effectiveness of access control mechanisms within the system, ensuring that any changes in access rights are communicated and enforced promptly. This is especially vital for maintaining the security integrity of applications operating in real-time environments, where delays in access revocation could expose the system to unauthorized entry and significant vulnerabilities.

A key performance metric evaluated during these tests is the latency between the moment an attribute's value is altered to one that violates predefined access policies and the time when the application receives the revoke signal from the User Control System. This interval is known as *inconsistency time* and is essential for assessing how quickly the system can respond to changes in access permissions. Timely revocation is crucial for mitigating unauthorized access and maintaining robust security protocols, as it ensures that access rights remain coherent with the established policies.

In the context of smart home applications, the ability to dynamically adjust access rights in response to real-time events is critical for security. For example, if a user's status changes, the system must swiftly revoke any previously granted access to prevent unauthorized actions. Understanding the performance of the

revocation process thus contributes to evaluating the overall security posture of the application. By ensuring rapid response times for revocation, the reliability of the UCS is bolstered, fostering user trust in its security measures.

To implement these tests, adaptations were made to existing automation scripts to incorporate revoke logic and log the timing associated with the reception of revoke signals. The simulation of this behavior involved executing a typical authentication flow, where the system initially grants access based on specific attribute values. Following this, the value of a designated attribute is changed to one that triggers the revocation of access rights. The timing of both the attribute change and the reception of the revoke signal are recorded, facilitating a comprehensive analysis of the revocation process.

To create a realistic testing environment, varying numbers of Policy Information Points—ranging from 1 to 50—were utilized, reflecting the complexity of the associated policies. Each PIP corresponds to an individual attribute that impacts access decisions. This setup mirrors the methodology employed in previous batches, ensuring consistency in testing while allowing for a focused analysis of the revocation mechanism. By maintaining a single request per test, the influence of additional overhead is minimized, thereby isolating the performance implications of policy complexity on the revocation process.

In practical terms, the simulation involved manipulating the last attribute of the ongoing section of the policy, specifically to eliminate any potential optimizations that could be inadvertently introduced by the UCS. This ensures that the revocation behavior is tested under controlled conditions, providing a reliable measurement of the system's performance in real-world scenarios.

3.7 Proposed Modifications to the User Control System

In the pursuit of enhancing the authentication flow and overall performance of the User Control System, several strategic modifications were implemented and rigorously tested. This section details these modifications, explaining their significance, the reasoning behind their feasibility, and the expected impact on the system's security posture. The overarching aim is to assess how these changes can affect performance while maintaining robust security measures.

One primary modification involves disabling the Distributed Hash Table update mechanism, which refers to the upload of the overall state of the UCS across the network. This state encompasses critical information such as the database containing ongoing sessions, active policies, and Policy Information Points. While

DHT updates facilitate a distributed operational model and improve data consistency across nodes, they may introduce performance overhead unsuitable for resource-constrained devices like the Raspberry Pi. Disabling DHT updates may allows for greater efficiency in processing requests, streamlining the UCS's overall operation. Although this trade-off limits the system's fully distributed capabilities and need to be fully evaluated.

Another key modification is the removal of the journaling functionality within the UCS when writing to the SQLite database. The UCS implements journaling during write operations to ensure that transactions are completed successfully before changes are finalized. However, this functionality can introduce performance overhead that is less practical for constrained environments. Eliminating journaling allows the UCS to operate more efficiently, although this may affect robustness during power outages. This change strikes a balance between functionality and efficiency, acknowledging that smart home devices may face intermittent power reliability issues. While the reduction in update frequency could impact resilience, it may lead to improved system responsiveness and resource management.

A crucial modification involves leveraging the trusted nature of the Policy Enforcement Point within the architecture. As a fully trusted entity, the PEP can send applicable policies directly within the *tryAccess* requests. This change eliminates the need for the UCS to conduct potentially time-consuming policy searches, facilitating a faster authorization process. The PEP's trustworthiness ensures that it can accurately assess which policies are relevant to a given access request without compromising security.

Regarding security, these modifications are designed to maintain robust access control mechanisms. While performance enhancements are essential, ensuring security remains a top priority. The ability to send applicable policies directly within requests is expected to have minimal impact on the attack surface. Additionally, the decision to forgo DHT updates is informed by the need to enhance efficiency in the system's operation, particularly for lightweight devices. This approach aims to uphold strong security measures without overloading constrained resources.

In summary, these proposed modifications aim to enhance the UCS's performance and efficiency while prioritizing security. The interplay between these factors is crucial for achieving optimal outcomes in lightweight systems. This thesis seeks to demonstrate that it is possible to refine the UCS's authentication flow while adhering to rigorous security standards.

3.8 Testing Proposed Modifications

Testing the proposed modifications in the User Control System (UCS) was conducted to evaluate the changes made to enhance authorization flow and overall performance. The focus of these tests was to collect data that would help understand the effects of the modifications on system operations.

To assess the impact of disabling the Distributed Hash Table (DHT) update mechanism, we performed tests aimed at measuring the efficiency of request processing. This involved monitoring the response times for user authentication requests before and after the modification. The data collected during these tests was intended to elucidate the trade-offs associated with reduced distributed capabilities in the context of a smart home environment.

Tests were designed to analyze the effects of removing the journaling functionality on system operations. This was particularly important during the authorization flow, where writes to the database occur. Since journaling only takes place during these write operations to the SQLite database, understanding its impact on performance is crucial for evaluating the efficiency of the modified UCS.

The testing framework also explored the integration of policies directly within the *tryAccess* requests, leveraging the trusted nature of the Policy Enforcement Point (PEP). This aspect of the testing focused on measuring the time taken to process authentication requests when applicable policies were included, compared to previous methods that required policy searches. By collecting this data, we aimed to understand how this modification influenced the speed of the authentication process.

In summary, the tests conducted on the proposed modifications aimed to gather valuable data regarding the performance and efficiency of the UCS, focusing on the specific changes made and their implications for access control mechanisms.

Chapter 4

Methodologies for Data Collection and Results Evaluation

This chapter outlines comprehensive methodologies for data collection and evaluation, forming the foundation for assessing the system's performance across various test scenarios. Emphasis is placed on systematic techniques for gathering data from the tests described in the previous chapter, ensuring reliability and thoroughness in the results.

Initially, the chapter elaborates on the automated tools and scripts used in the data collection process. These tools are designed to maintain consistency throughout testing, minimizing manual input and reducing human error. Automation streamlines test execution and facilitates the replication of scenarios under uniform conditions, which is essential for data validity. Specific scripts and tools utilized will be detailed, demonstrating their functionality within the overall testing framework.

The chapter will also outline key metrics and evaluation criteria employed to assess system performance, including installation efficiency, runtime performance, and revocation effectiveness. The rationale behind selecting these metrics will be discussed, ensuring alignment with the testing framework's objectives and the practical requirements of smart home environments.

Configurations and scenarios simulated during testing are designed to reflect real-world conditions, enabling a realistic assessment of the system under various loads and complexities. The diversity of test conditions, including variations in the number of requests and policies, will be examined, contributing to a comprehensive

understanding of the system's capabilities and limitations.

Transitioning from data collection to processing and analysis, both quantitative and qualitative aspects will be covered to facilitate a holistic understanding of system performance. This section will clarify the analytical techniques employed, illustrating how data transforms into actionable insights that inform optimization decisions.

Additionally, the methodologies will evaluate the proposed modifications to the User Control System to determine their potential as enhancements for constrained devices. Insights will be gathered on which modifications may improve performance while ensuring security, and which may introduce vulnerabilities or be deemed unsuitable.

In summary, this chapter serves as a crucial component of the thesis, articulating a structured approach to data collection and evaluation that underpins the findings and conclusions drawn in subsequent sections.

4.1 Data Collection Methodologies

This thesis employed various data collection methodologies to evaluate the performance of the User Control System across different operational scenarios. The primary goal was to systematically gather relevant data points to gain insights into the system's efficiency and performance characteristics.

Central to the data collection process were the outputs from the Policy Enforcement Point, which tracked and recorded key events during UCS operations from the user perspective. As requests were processed, timestamps for actions such as request submissions and response receptions were collected. This information allowed for the calculation of the time taken for different operations, providing a clear overview of system performance.

A significant aspect of this research involved testing combinations of operational parameters. By varying configurations of policies and request attributes, a comprehensive dataset was generated. These combinations formed the foundation for the output data collected from individual tests within each batch, enabling a nuanced understanding of the UCS's behavior under different conditions.

Data was gathered selectively from both the PEP and the UCS, focusing on distinct performance aspects. The PEP provided insights into request-response cycles, while UCS data offered a deeper understanding of internal processing and policy evaluations. This dual approach ensured thorough documentation of both external interactions and internal dynamics.

To enhance the robustness of data collection, multiple execution iterations were conducted for each test scenario, with a total of 10 or 20 executions performed per batch. This methodology allowed for averaging results, minimizing the impact of outliers and ensuring statistical validity.

Data verification was crucial to maintaining the integrity of the collected information. Temporary files were created to log outputs from the UCS, confirming the correctness of test executions. This step validated the findings, ensuring conclusions were based on reliable data.

In addition to timing data, the collection process focused on capturing operational metrics, such as the number of policies evaluated and attributes linked to each request. Cataloging these metrics provided a holistic view of the UCS's performance under varying conditions and configurations.

For analysis, the collected data was exported to CSV format for further examination. Tools like Microsoft Excel were utilized to organize, visualize, and interpret the information, facilitating the identification of trends and significant findings within the dataset.

Overall, the data collection methodologies were designed to enable an in-depth analysis of UCS performance while ensuring that the gathered data was relevant and reliable. By structuring the process around specific operational combinations, critical performance metrics were uncovered, informing future optimizations and enhancements to the UCS.

4.2 Test Results for the Usage Control System

This section presents the results from the tests conducted on the Usage Control System. These tests, designed to assess the system's performance under different conditions and across various stages of the authorization process, were thoroughly outlined in section 3.6.

The results focus on key aspects such as installation, runtime performance, and revocation handling. By analyzing the collected data, insights are provided into the system's functionality, its response to varying loads, and its overall stability. These findings form the basis for the subsequent analysis and evaluation in the following section.

4.2.1 Installation Tests

The installation tests conducted on the Usage Control System aimed to measure the overhead introduced during the application installation phase, specifically focusing

on the time elapsed between the submission of the first *tryAccess* request and the receipt of the last *startAccess* response. Two distinct modes were tested to capture variations in performance, each designed to reflect different approaches to processing authorization requests.

The primary objective of these tests was to evaluate the UCS's performance on a constrained device, such as the Raspberry Pi 4. Given the constraints of such devices, it was important to explore whether sending requests in different orders would yield varying performance outcomes. The rationale behind testing both modes lies in the nature of request processing in the UCS, where requests may arrive in a non-sequential manner. Both tested modes respect the UCON flow, ensuring that the system's access control logic remains consistent regardless of the order in which requests are processed.

In the first mode (Mode 1), the *startAccess* request for a given action was sent immediately after receiving a *Permit* response for the corresponding *tryAccess*. This mode simulates a scenario where each request is handled in real time as soon as its authorization is confirmed, regardless of whether additional *tryAccess* requests are still pending. This approach allows for an analysis of the immediate performance impact of processing each request individually.

The second mode (Mode 2), by contrast, involved submitting all *tryAccess* requests consecutively before initiating any *startAccess* actions. This separation between the *tryAccess* and *startAccess* phases enabled a clearer analysis of how the system handles batches of requests. By isolating the two phases, it was possible to observe whether the UCS exhibited performance improvements or degradations as the number of *tryAccess* requests increased.

This distinction is crucial, as sending multiple *tryAccess* requests together may prevent the UCS from unnecessarily reloading components required for processing subsequent *startAccess* requests. In scenarios where components, such as those responsible for creating session IDs, are retained in cache, this could lead to delays if the requests are processed out of order. By executing all *tryAccess* requests first, potential caching benefits can be leveraged, thus avoiding performance degradation that might occur if each request is processed immediately as it is authorized.

Both testing modes involved the use of automated scripts to ensure consistency in test execution. Policies were generated and matched with corresponding *tryAccess* requests, with up to 10 requests per test, and the process was repeated 20 times to ensure statistically reliable results. Each test was configured to match each request to a unique policy, ensuring that no caching or optimization techniques artificially enhanced the results.

Table 4.1 and Table 4.2 present the average times measured in each mode, as

the number of requests increases. These results provide a clear understanding of how the system's performance scales with an increasing number of authorization requests.

Number of Requests	Avg. Time for Complete All Requests [s]	Standard Deviation %
1	5.40	1.77
2	9.34	2.58
3	13.28	2.58
4	16.47	2.61
5	19.36	2.63
6	21.59	3.53
7	23.92	3.60
8	25.88	3.70
9	27.77	2.86
10	28.97	2.34

Table 4.1: Average timing results for the complete authorization flow during the installation process based on the number of requests - Mode 1

Number of Requests	Avg. Time TryAccess Phase [s]	Avg. Time StartAccess Phase [s]	Avg. Time for Complete All Requests [s]	Standard Deviation %
1	3.05	2.37	5.42	0.53
2	5.49	3.92	9.41	0.47
3	7.85	5.53	13.38	0.49
4	9.80	6.60	16.40	0.55
5	11.99	7.72	19.72	1.22
6	13.63	8.07	21.70	3.04
7	15.64	8.88	24.51	3.48
8	17.32	9.24	26.56	1.86
9	18.78	9.57	28.35	0.90
10	19.21	9.83	29.04	0.27

Table 4.2: Average timing results for the complete authorization flow during the installation process based on the number of requests - Mode 2

The graphical representation in Figure 4.3 illustrates the differences in timing between the two modes, providing a clearer comparison of the system's behavior as the number of requests increases.

Figure 4.1: Graphical representation of average timing results for the complete authorization flow during the installation process based on the number of requests - Mode 1

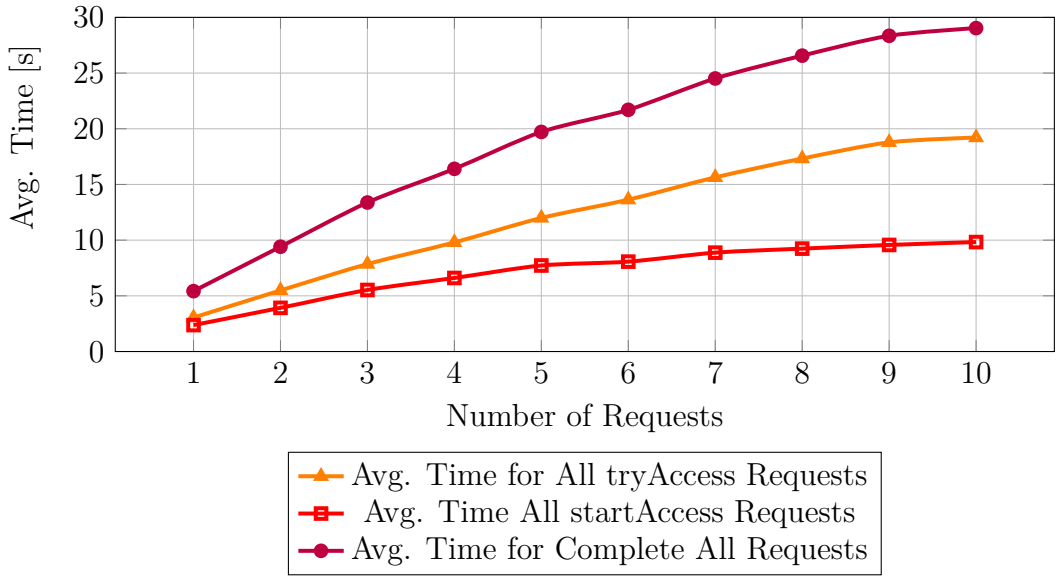

Figure 4.2: Graphical representation of average timing results for the complete authorization flow during the installation process based on the number of requests - Mode 2

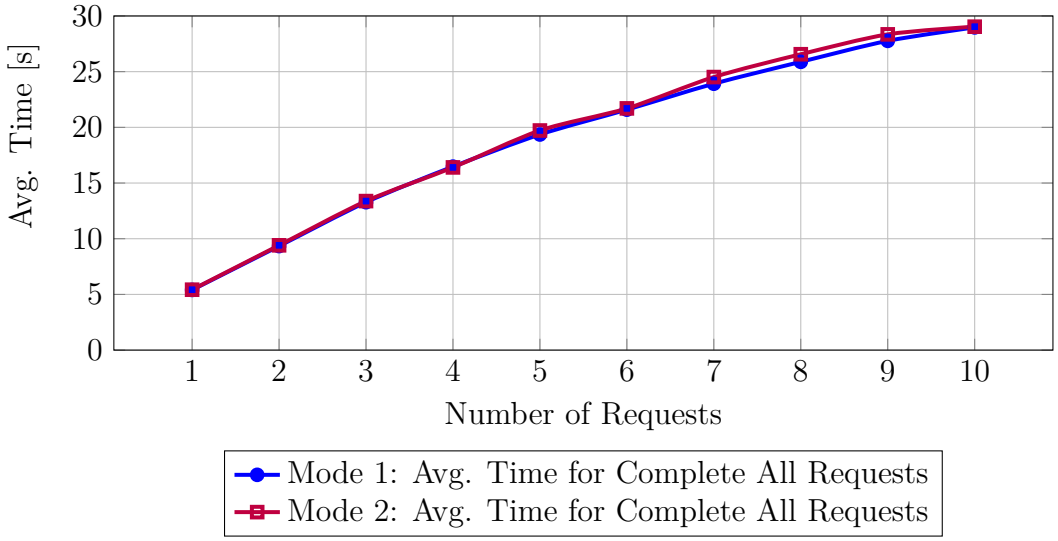

Figure 4.3: Comparative graphical representation of average timing results for complete requests during the installation process - Modes 1 and 2

In addition to the aggregated data for the two modes, the tests also enabled tracing of the individual steps within the authorization flow for each request. By analyzing the UCS's internal logs, it was possible to capture detailed timing information for each step in the authorization process. This includes the initial submission of the *tryAccess* request, the policy evaluation, the issuance of the *Permit* decision, and the subsequent interactions that occur after the *startAccess* request is made by the PEP.

Figure 4.4 presents a graphical representation of the detailed timing breakdown for a single authorization flow, providing deeper insights into where potential delays might occur within the system.

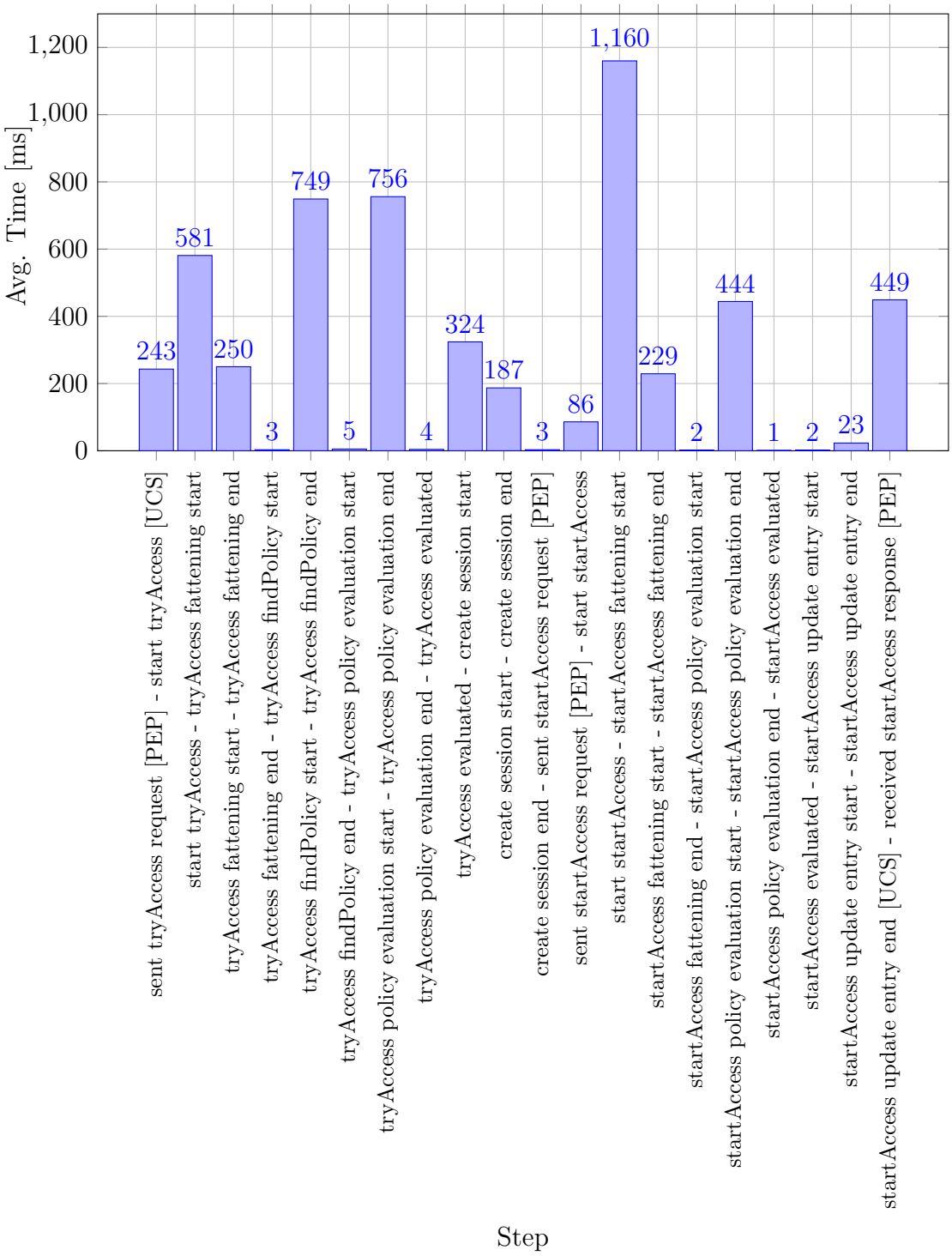

Figure 4.4: Graphical representation of the detailed timing breakdown for an individual authorization flow

This graphical representation helps identify any inefficiencies or bottlenecks within the authorization process.

4.2.2 Running Tests

The running tests were designed to evaluate the runtime performance of the Usage Control System when processing requests for an application that is already installed. These tests focus specifically on measuring the overhead introduced during the runtime authorization phase, analyzing the system's behavior as the complexity of the policies increases.

In these tests, each request follows the standard authorization flow: an initial *tryAccess* request is submitted, resulting in a *Permit* decision, followed by a corresponding *startAccess* request, which also results in a *Permit*. However, unlike the installation tests, the primary focus here is on how the UCS handles the evaluation of policies that include varying numbers of attributes within their ongoing sections.

Each policy is linked to a number of Policy Information Points (PIPs), where each PIP corresponds to a specific attribute that must be evaluated. To investigate how the number of attributes in the ongoing section of each policy affects performance, the tests were conducted by varying the number of attributes from 1 to 50. This approach allows for a detailed analysis of the UCS's scalability and efficiency when handling policies of increasing complexity. Each test consists of a single request, ensuring that the observed performance reflects the impact of policy complexity alone, without interference from additional request overhead.

All PIPs are of the same type, with each PIP obtaining the value of its corresponding attribute by reading from a file located within the device. Automated scripts were employed to ensure consistent test execution and accurate logging of performance metrics, including the response times for the *tryAccess* and *startAccess* phases. These metrics provide insights into how the UCS responds to the increasing workload imposed by the evaluation of multiple attributes found in the ongoing section of each policy. The number of PIPs involved in each test plays a critical role, as each additional attribute introduces a new layer of complexity, requiring interactions with the relevant Attribute Manager responsible for processing that attribute's values.

Table 4.3 presents the average execution times measured during the running tests, with varying numbers of PIPs involved. These results provide valuable insights into how the system's runtime performance evolves as the complexity of the policy attributes increases.

To provide a clearer understanding of the performance degradation as the number of

Number of PIPs	**Avg. Time [s]**	**Standard Deviation %**
1	5.04	3.91
2	5.25	3.89
3	5.33	3.29
4	5.38	4.30
5	5.56	3.95
6	5.58	3.29
7	5.53	3.96
8	5.56	5.13
9	5.67	4.53
10	5.70	4.33
11	5.64	3.12
12	5.74	3.94
13	5.84	6.10
14	5.89	4.94
15	6.08	2.91
16	6.01	3.67
17	6.01	4.00
18	6.09	4.05
19	6.07	5.02
20	5.96	3.92
21	6.07	4.71
22	6.14	2.49
23	6.03	3.39
24	6.03	4.34
25	6.08	3.26
26	6.12	4.15
27	6.20	2.16
28	6.30	2.84
29	6.25	5.75
30	6.29	3.10
31	6.32	3.85
32	6.36	3.55
33	6.40	4.90
34	6.33	4.37
35	6.40	5.68
36	6.37	2.93
37	6.51	4.49
38	6.70	4.06
39	6.57	4.63
40	6.55	6.42
41	6.46	5.08
42	6.58	4.26
43	6.21	5.60
44	6.11	4.95
45	6.19	4.69
46	6.16	3.83
47	6.27	2.90
48	6.33	4.41
49	6.32	4.20
50	6.35	3.94

Table 4.3: Average runtime performance results based on the number of attributes in the policy

PIPs increases, Figure 4.5 visualizes the average time for processing a single request and the related standard deviation percentages. This graphical representation highlights how the system's performance is impacted as the policy complexity increases.

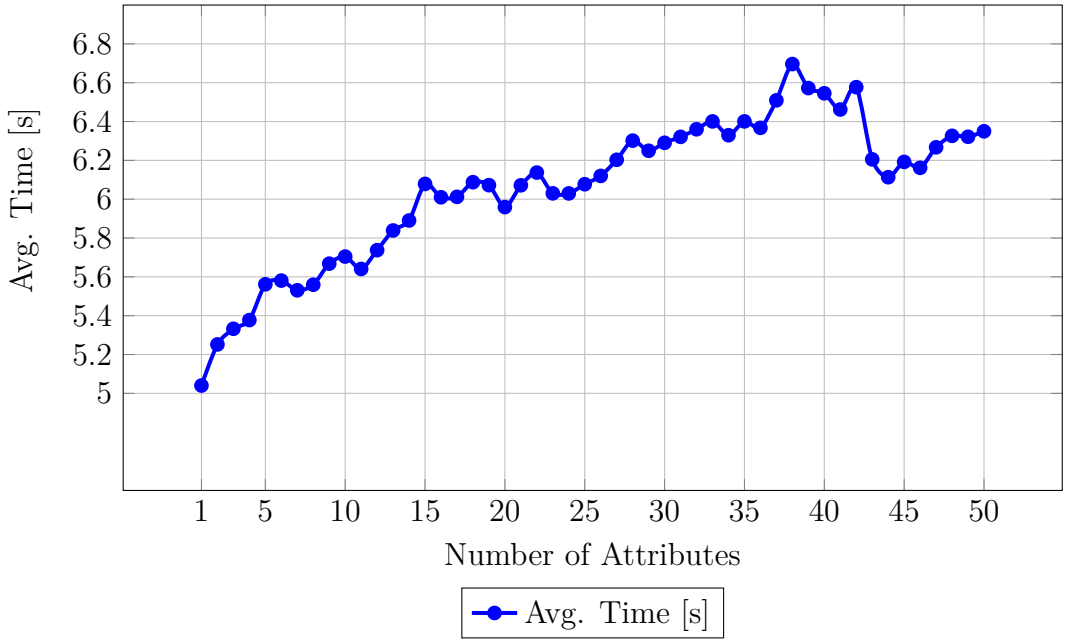

Figure 4.5: Graphical representation of average execution time for varying number of attributes during running tests

4.2.3 Revocation Tests

The revocation tests aimed to evaluate the efficiency of the Usage Control System in managing the revocation of previously granted permissions. These tests are crucial for understanding how quickly the system responds to changes in access rights, particularly when an attribute's value is modified in a way that violates established policies.

The focus is on measuring the inconsistency time, which refers to the period between the alteration of an attribute's value and the Policy Enforcement Point receiving the revoke signal from the UCS. Each test follows a standard workflow: access is initially granted based on specific attribute values, after which a designated attribute's value is intentionally changed to trigger the revocation process. Timestamps for both the attribute modification and the receipt of the revoke signal at the PEP are recorded for analysis.

Testing involved varying the number of Policy Information Points from 1 to 50, corresponding to different policy attributes. This range facilitates an examination of how revocation performance is influenced by the number of attributes evaluated. By concentrating on a single request per test, we isolate the impact of attribute complexity and minimize extraneous overhead.

Table 4.5 presents the average inconsistency times measured during the revocation tests, illustrating the latency experienced as the number of PIPs increases and providing insights into the UCS's responsiveness in enforcing access control across different policy complexities.

Number of PIPs	Avg. Time [s]	Standard Deviation %
1	1.52	12.48
2	1.48	8.14
3	1.47	29.82
4	1.59	20.33
5	1.61	22.11
6	1.69	23.28
7	1.47	28.25
8	1.49	24.19
9	1.52	22.97
10	1.70	22.15
11	1.66	21.37
12	1.66	19.26
13	1.70	13.30
14	1.72	19.88
15	1.69	10.34
16	1.64	14.96
17	1.77	13.40
18	1.99	5.30
19	2.12	5.48
20	2.56	28.20
21	2.85	6.59
22	2.90	4.67
23	3.18	4.90
24	3.38	4.35
25	3.79	4.74
26	3.92	5.03
27	4.23	4.48
28	4.33	3.30
29	4.62	3.42
30	5.27	2.49
31	5.75	3.65
32	5.88	4.42
33	6.24	7.84
34	5.98	5.05
35	6.25	7.64
36	6.08	4.45
37	6.43	6.26
38	9.22	4.57
39	9.53	3.50
40	9.68	3.10
41	9.95	4.50
42	9.86	5.26
43	10.38	6.43
44	6.58	4.71
45	6.75	8.25
46	7.01	5.66
47	10.59	9.26
48	10.18	6.06
49	10.55	5.71
50	9.98	6.06

Table 4.4: Average revocation performance results based on the number of attributes in the policy

To illustrate the performance degradation as the number of PIPs increases, Figure 4.6 visualizes the average revocation times for processing a single revocation request. This graphical representation highlights the impact of policy complexity on the system's responsiveness.

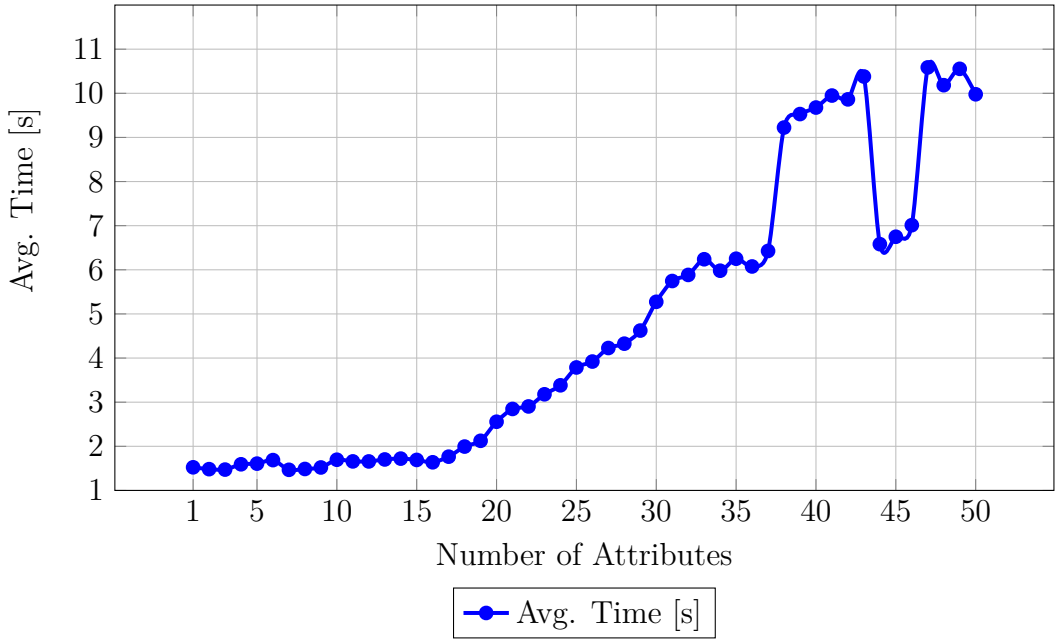

Figure 4.6: Graphical representation of average execution time for varying number of attributes during revocation tests

4.3 Analysis of Usage Control System Test Results

The analysis of the test results for the Usage Control System deployed on constrained devices, specifically a Raspberry Pi 4, reveals several performance challenges that need to be addressed. The performance was evaluated through three types of tests: installation, running, and revocation. These results offer detailed insights into the system's limitations and strengths, particularly in environments with limited resources, highlighting areas where targeted optimizations and adjustments are necessary for improved usability and performance.

The installation tests showed that executing the system in the two modes tested—processing all *TryAccess* requests before *StartAccess*, or handling them out-of-order—did not result in any significant differences in performance. This demonstrates that the UCS can handle different request patterns consistently, even on constrained devices. However, the tests revealed a notable overhead in terms of installation time, particularly during specific phases of the process. This suggests that the UCS, in its unmodified state, may introduce usability challenges when deployed on lightweight systems. Users could experience delays during the installation of applications, which might negatively impact the overall efficiency of system

rollouts in real-world applications where swift deployment is critical. Although these installation-related performance issues do not directly affect the system's operational phase, they indicate potential bottlenecks that, if left unaddressed, could compromise user satisfaction and delay initial system availability.

Nevertheless, despite these initial overheads, the system displayed a consistent and predictable degradation in performance as the number of requests increased, with the performance drop remaining within expected percentages. This trend indicate that the system retains a certain degree of scalability, even when operating in environments with limited resources. The ability of the UCS to handle an increasing volume of requests without suffering disproportionate slowdowns is an encouraging sign. It suggests that, provided the installation overhead is carefully managed and optimized, the UCS can be effectively deployed in resource-constrained environments. This level of predictable scalability is important for ensuring the system's usability in applications.

The performance tests, which examined the system's behavior as the number of attributes in the policies increased, revealed an expected decline compared to the baseline performance, but not significantly in percentage terms. These results suggest that the system is optimized for environments where policies are highly dynamic and involve frequent evaluations of multiple attributes. The ability to maintain relatively stable performance as policy complexity increases clearly indicates that the UCS can handle complex policy management even under resource constraints, making it a viable option for scenarios where dynamic access control is critical, such as in IoT or smart environments.

In contrast, the revocation tests exposed a more concerning performance issue. A substantial degradation in revocation times was observed as the number of attributes increased, with a staggering 654 % increase in the time required to complete a revocation when comparing tests with 1 attribute to those with 50 attributes. This raises serious security concerns, as slow revocation times could lead to unauthorized access being maintained far longer than intended. In environments dealing with sensitive data or critical infrastructure, this delay could have severe consequences, as access to resources that should have been revoked immediately might persist, potentially leading to data breaches or unauthorized use of critical systems. The inability to revoke access promptly undermines the reliability of the UCS in dynamic contexts, where permissions need to be adapted or revoked in real time. This issue highlights a significant weakness in the current implementation of the UCS, and addressing it will be essential to ensure that the system can perform effectively in situations where security and timely access revocation are paramount.

Overall, the results from these tests highlight both the strengths and weaknesses

of the UCS when deployed on constrained devices. The installation phase demonstrated that, while some overhead is present, the system maintains a good level of scalability, handling increasing request volumes in a predictable manner. The running tests showed that, the system performs exceptionally well under more complex policy conditions, managing multiple attributes with surprising efficiency. This indicates that the UCS is well-suited to environments where access control policies are dynamic and complex. However, the severe degradation in revocation times represents a critical flaw, as delayed revocations could lead to unauthorized access, particularly in sensitive and time-critical environments.

Addressing these challenges, particularly the revocation performance issue, is crucial to ensuring the UCS can function reliably and securely in resource-constrained settings. Once these performance issues are resolved, the UCS could offer a robust solution for managing access control in lightweight systems. Its ability to handle complex, attribute-intensive policies with reasonable scalability makes it an attractive candidate for deployment in environments such as smart homes, where constrained devices are prevalent, and dynamic access control is often required. These results lay the foundation for future optimizations, suggesting that, with targeted improvements, the UCS could become a reliable and effective solution for access control in a wide range of constrained device environments.

4.4 Test Results for the Modified Usage Control System

The proposed modifications to the Usage Control System were subjected to thorough testing to evaluate their impact on performance, efficiency, and security. The tests aimed to assess improvements in request processing speed, system resilience, and the dynamic handling of access control policies, while comparing the system's behavior before and after the modifications.

This section details the testing process for each modification, offering a comparative analysis of the system's performance prior to and following the changes. The focus is on examining the specific effects on the authentication flow and other key operational metrics.

4.4.1 Performance Impact of Disabling Distributed Hash Table Updates

The decision to disable Distributed Hash Table updates represents a significant reduction in the distributed functionality of the Usage Control System. While DHT updates facilitate synchronization across multiple nodes, they introduce notable performance overhead, particularly in resource-constrained environments such as the Raspberry Pi 4 platform, on which the UCS is deployed. Given the limited computational and memory resources of such devices, it becomes crucial to assess whether a more centralized architecture, with DHT updates disabled, could provide a better trade-off between performance and functionality.

The modification aimed to eliminate the DHT update overhead during the request handling flow by disabling the StatusWatcher component, which is responsible for monitoring changes in the UCS local database and triggering DHT updates. By doing so, the system sacrifices its distributed nature in favor of a more lightweight operation, potentially better suited for environments with limited resources.

To evaluate this modification, the performance of the UCS was analyzed in various test scenarios, including installation tests, running tests, and revocation tests. The analysis compared the behavior of the system before and after disabling the DHT updates, focusing on metrics such as request processing time, authentication flow latency, and overall system responsiveness.

In the following sections, graphical representations of the test results are presented, offering a comparative analysis of the system's performance before and after the modification, with particular attention to installation, running, and revocation phases.

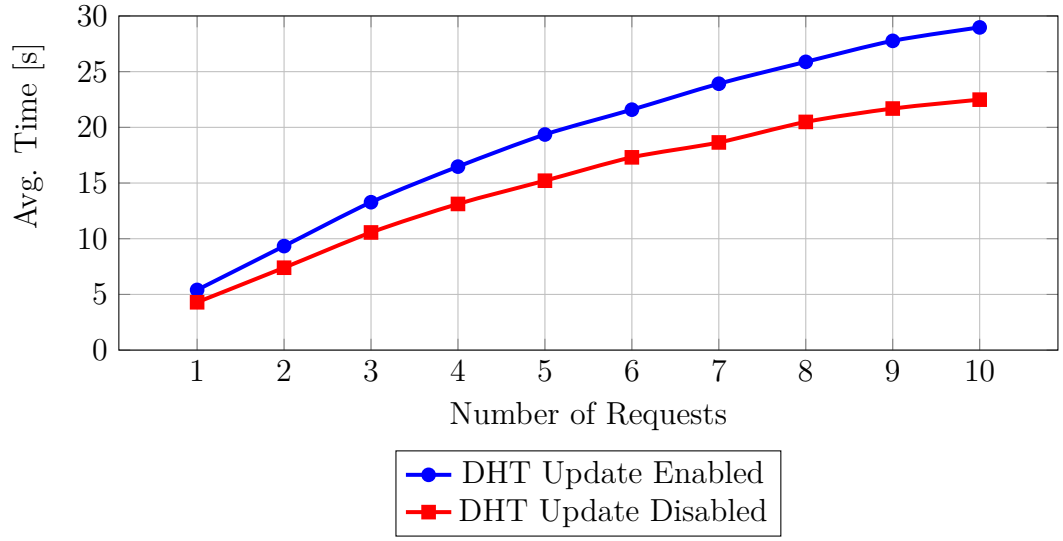

Figure 4.7: Comparative graph of average timing results during the installation process with DHT update enabled and disabled

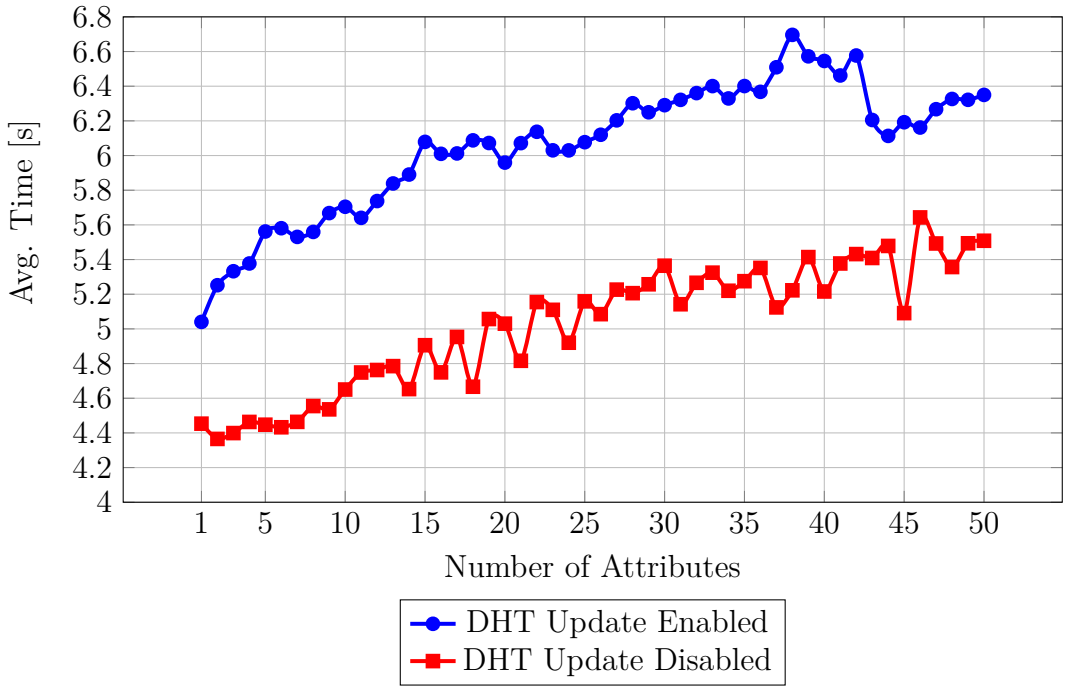

Figure 4.8: Comparative graph of average execution time for varying number of attributes during running tests with DHT update enabled and disabled

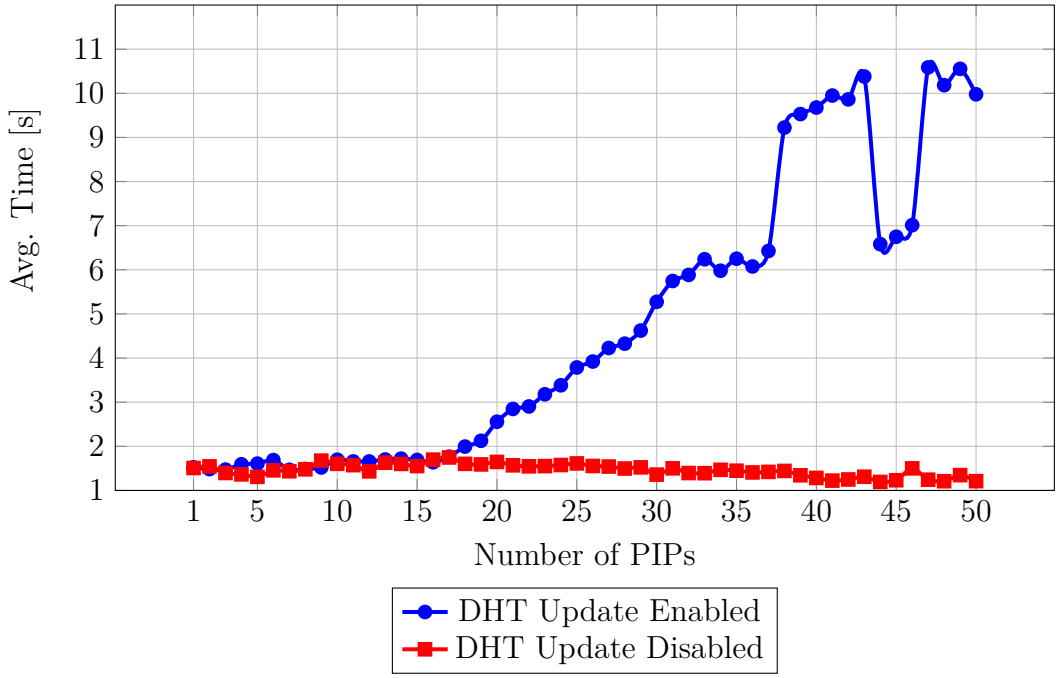

Figure 4.9: Comparative graph of average execution time for varying number of attributes during revocation tests with DHT update enabled and disabled

4.4.2 Performance Impact of Removing Journaling

Another key modification within the Usage Control System is the removal of the journaling functionality. This change aims to reduce system functionality and resilience, particularly in the face of potential power failure errors. While journaling enhances system resilience by capturing updates frequently, it can introduce performance overhead, making it less practical for resource-constrained environments such as the Raspberry Pi 4 platform, where the UCS is deployed.

The decision to disable journaling is rooted in the recognition that smart home devices may encounter intermittent power reliability issues. By eliminating the journaling process, the UCS can operate more efficiently, albeit at the cost of robustness during unexpected power outages. This modification underscores the necessity of balancing functionality and efficiency, as the removal of frequent updates may impact the system's ability to recover from failures. However, it may lead to improved system responsiveness and resource management, critical factors in resource-limited environments.

Furthermore, this change allows for a more lightweight system while maintaining a distributed architecture. This enhancement could potentially open the door to

broader applications in smart home environments, enabling higher connectivity and integration among devices.

To evaluate the impact of removing the journaling functionality, the performance of the UCS was analyzed across various test scenarios, similar to the approach taken when disabling DHT updates. The tests included installation tests, running tests, and revocation tests. The performance metrics considered included request processing time, authorization flow latency, and overall system responsiveness.

In the subsequent sections, graphical representations of the test results will be presented, illustrating the comparative analysis of the UCS's performance before and after the modification. This analysis will focus on how the removal of journaling affects the system's efficiency, particularly during the installation, running, and revocation phases. The findings will provide insights into whether the trade-off in resilience is justified by the gains in performance and resource utilization.

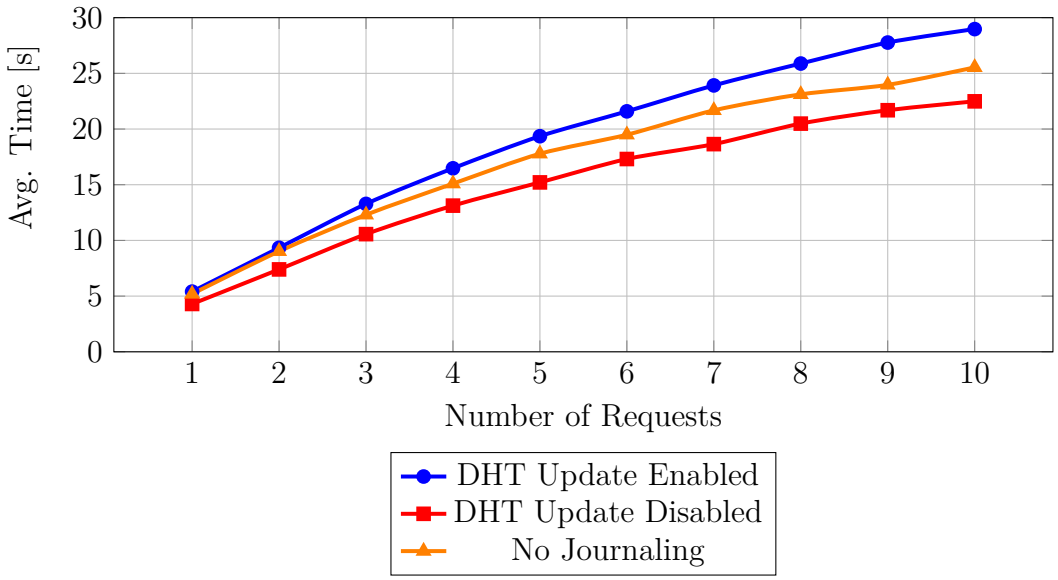

Figure 4.10: Comparative graph of average timing results during the installation process with DHT update enabled, disabled, and with journaling removed

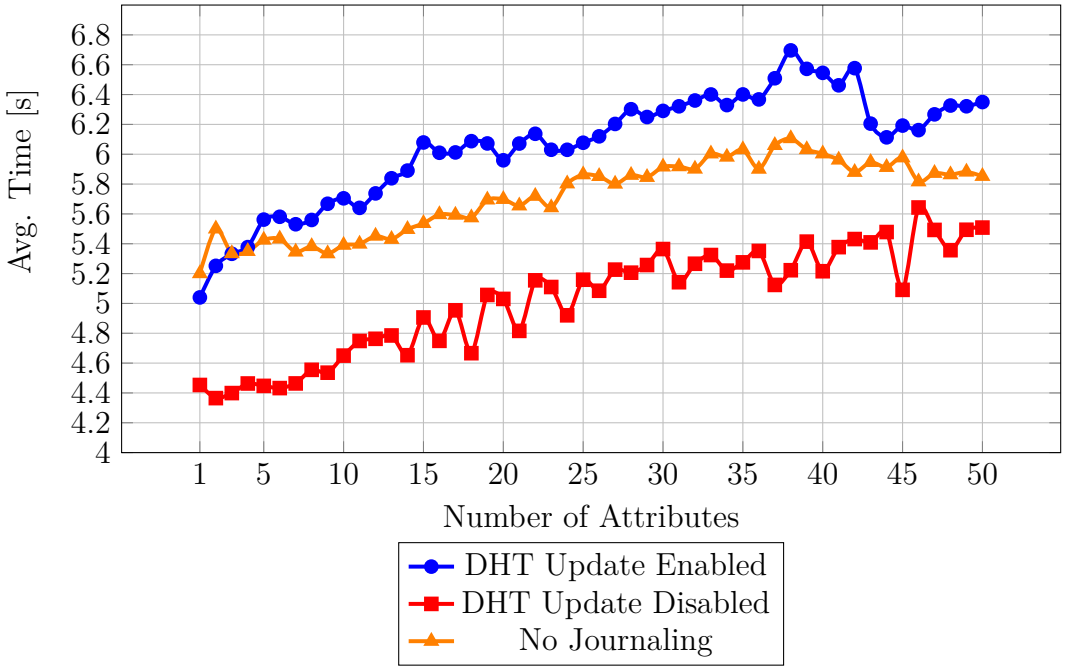

Figure 4.11: Comparative graph of average execution time for varying number of attributes during running tests with DHT update enabled, disabled, and with journaling removed

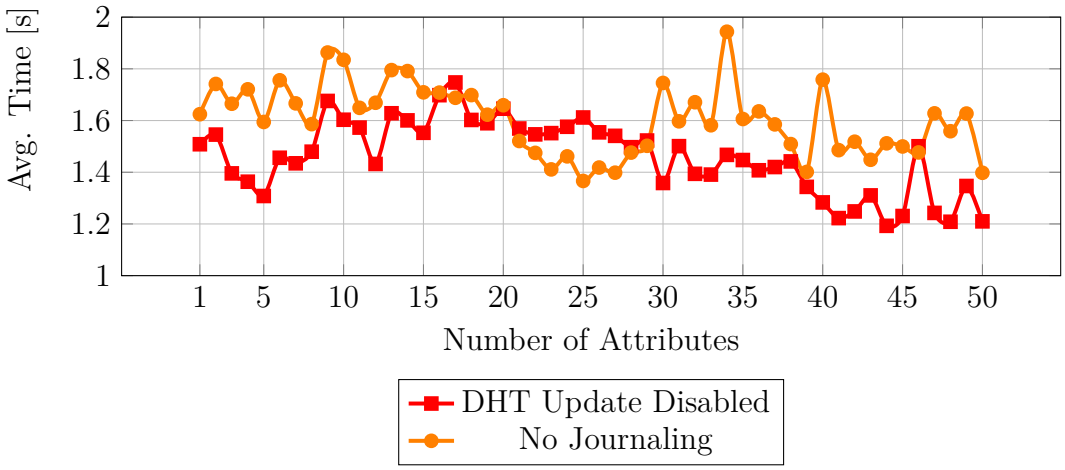

Figure 4.12: Comparative graph of average execution time for varying number of attributes during revocation tests with DHT update disabled and with journaling removed

4.4.3 Performance Impact of Direct Policy Integration in *tryAccess* Requests

Integrating policies directly within the *tryAccess* requests marks a significant improvement in the architecture of the Usage Control System. This modification leverages the trusted nature of the Policy Enforcement Point (PEP) to reduce the operational overhead associated with policy searches.

In standard implementations, the UCS conducts time-consuming searches for applicable policies, introducing latency in the request handling process. By allowing the PEP to embed relevant policies directly into the *tryAccess* requests, this overhead is eliminated, streamlining the authorization flow and improving system performance.

Although using the PEP as a trusted component raises concerns about increasing the attack surface, this trade-off is justified when considering the goal of deploying the UCS on constrained devices, such as the Raspberry Pi 4. In addition to performance gains, reducing the operations on limited hardware also enhances system security by minimizing vulnerabilities related to resource exhaustion. This approach improves both performance and functionality by simplifying policy management without compromising security.

The performance tests in this section were conducted using the *No Journaling* variant of the UCS, which was chosen after prior tests demonstrated its superiority in terms of both performance, functionality, and security. As a result, only results with this variant are presented here.

Although other configurations, such as *DHT Update Disabled* or the original UCS, could have been evaluated, the notable benefits of *No Journaling* made it the most suitable foundation for further enhancements. While the integration of policies directly into *tryAccess* is an independent optimization, this study focuses on the most promising configuration to maximize performance, security, and functionality in resource-limited environments.

To accurately evaluate the efficiency of this implementation, it was first necessary to assess the impact of searching for an applicable policy with a growing number of policies, referred to as the *findPolicy* operation. In this case, the complexity is represented by the number of policies loaded into the system, with a fixed scenario where the applicable policy is always the last one in the set. This setup ensures that no unintended optimizations influence the results and provides a consistent metric for evaluating performance.

The *findPolicy* operation represents a key part of the standard UCS request handling process and, on average, accounts for 757 ms—approximately 24.82 % of the total

handling time for a single *tryAccess* request with the standard UCS implementation. Understanding how this time scales with the number of policies is essential to properly gauge the performance gains from removing this step in the modified approach. By quantifying the search time under varying conditions, it becomes possible to measure the extent of the optimization achieved through direct policy integration.

Table 4.5 presents the results of the tests conducted, illustrating the time taken by the *findPolicy* operation as the number of policies increases. This data provides a baseline for comparing the performance improvements realized by bypassing this step in the optimized implementation.

Number of Policies	Avg. Time [s]	Standard Deviation %
1	0.71	0.01
2	0.80	0.01
3	0.81	0.01
4	0.73	0.01
5	0.90	0.01
10	1.05	0.01
11	0.74	0.01
12	0.71	0.01
13	0.80	0.01
14	0.86	0.01
15	0.81	0.01
20	0.88	0.01
21	0.90	0.01
22	0.92	0.01
23	0.80	0.01
24	0.83	0.02
25	0.90	0.01
30	0.91	0.01
31	0.91	0.01
32	1.01	0.01
33	1.00	0.01
34	0.99	0.01
35	1.05	0.02
36	0.91	0.01
37	0.87	0.01
38	0.98	0.01
39	0.95	0.02
40	0.92	0.01
41	0.88	0.01
42	0.93	0.01
43	0.89	0.01
44	0.95	0.01
45	0.91	0.01
46	1.04	0.01
47	1.00	0.01
48	0.96	0.01
49	0.90	0.01
50	0.92	0.01

Table 4.5: Average *findPolicy* performance results based on the number of loaded policies, ensuring that the applicable policy is always the last in the set

As observed from the data, the *findPolicy* operation is impacted only minimally, with negligible fluctuations (less than 200 ms) as the number of initialized policies increases. This indicates that the overhead associated with policy searching does not significantly degrade performance. However, it represents approximately 30 % of the time required for handling a *tryAccess* request, so removing this step could

positively affect the overall efficiency of the system when deployed on constrained devices.

To further assess the robustness of this evaluation, a stress test was conducted using the modified version of the Usage Control System, where the *findPolicy* operation was removed. In this scenario, consecutive *tryAccess* requests were processed, with all requests sent by the Policy Enforcement Point (PEP) already containing the applicable policy. This configuration was designed to simulate a real-world usage scenario, allowing for a comparative analysis against the unmodified UCS implementation from the perspective of the PEP.

The results of this stress test are presented in Table 4.6 , highlighting the differences in performance between the modified UCS variant (with the *findPolicy* operation removed) and the standard implementation, focusing on request handling times and overall system responsiveness under load.

Number of Requests	**Avg. Time TryAccess Phase [s]**	**Avg. Time StartAccess Phase [s]**	**Avg. Total Time [s]**	**Standard Deviation %**
1	2.87	2.09	4.96	4.10
2	5.38	3.40	8.78	2.23
3	7.50	4.89	12.39	2.02
4	9.70	5.73	15.43	1.75
5	11.70	6.59	18.29	2.33
6	13.48	6.89	20.38	3.63
7	15.41	6.93	22.34	4.68
8	16.98	7.62	24.61	3.33
9	18.03	7.23	25.26	3.73
10	19.95	7.65	27.60	3.74

Table 4.6: Average timing results for the complete authorization flow using the modified UCS version with the *findPolicy* operation removed, based on the number of requests

Figure 4.13 presents a graphical comparative version of the same data between this implementation and the one prior to the modification. This visual representation will help clearly illustrate the performance variations achieved through the direct integration of policies in the access requests.

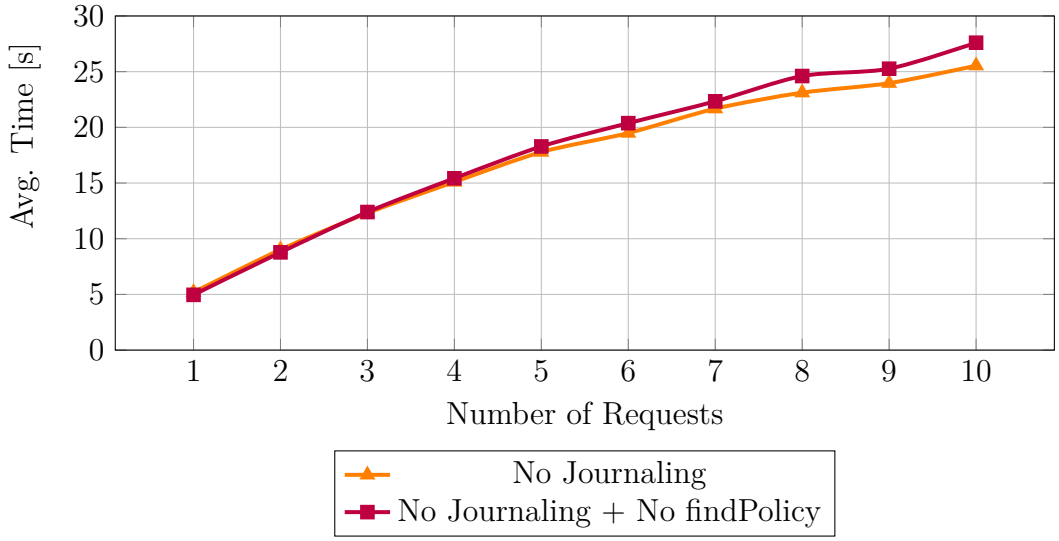

Figure 4.13: Comparative graph of average timing results during the stress tests with No Journaling and No Journaling + No findPolicy

To explain why, even though the UCS no longer needs to invest time in searching for the applicable policy, there is no significant change in processing time from PEP prospective, as shown in Figure 4.13, it is important to note that the PEP is no longer limited to merely creating the *tryAccess* request and sending it to the UCS. In the modified architecture, the PEP now assumes a more active role by selecting the applicable policy and managing the Base64 encoding of the identified policy to be inserted inside the request before transmission. This additional step introduces complexity.

Furthermore, upon receiving a new *tryAccess* request, the UCS is responsible for accurately decoding the Base64-encoded request, enabling it to handle and process the request as intended. This new workflow introduces a layer of complexity.

In order to provide a comprehensive overview of the impact of the modification on overall performance, data from the UCS has been collected and analyzed. This analysis aims to identify which steps in handling requests have the greatest impact and to quantify the overall effect on request processing.

A comparative view of the steps of the request handling process that have been substantially impacted is presented in Figure 4.14.

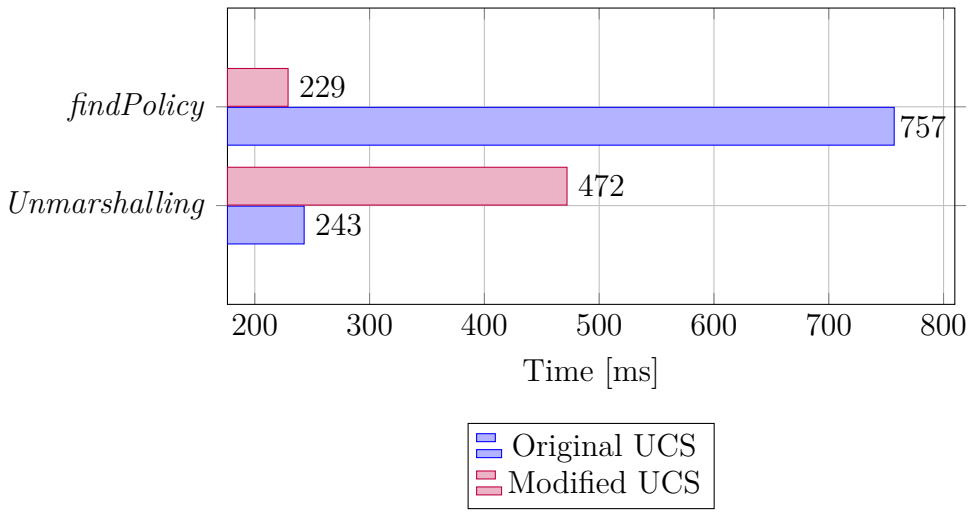

Figure 4.14: Comparison of Request Handling Steps Before and After Modification

As shown in Figure 4.14, the unmarshalling of the received request——converting the XML-formatted request into a Java object, useful for subsequent steps——takes longer than before, reducing the efficiency gained from eliminating the findPolicy operation.

4.5 Analysis of Modified System Test Results

The analysis of the modified Usage Control System reveals several key performance improvements following changes to its architecture, while also highlighting certain trade-offs. The modifications introduced—disabling Distributed Hash Table updates, removing journaling functionality, and directly integrating policies into *TryAccess* requests—were extensively tested for installation, runtime, and revocation performance on a resource-constrained device like a Raspberry Pi 4. Each modification impacts the system differently, enhancing performance or reducing functionality, but overall, the system retains its usability and security.

Disabling DHT updates had a significant effect on system performance. By removing the synchronization overhead associated with DHT updates, the UCS experienced improved installation times and a more efficient architecture, shifting from a distributed to a single-node model. During runtime tests, the system showed not only faster base performance but also a reduced degradation rate as the number of attributes in the policies increased. This indicates that the UCS is more efficient at handling requests under load when DHT updates are disabled.

Most notably, the revocation phase saw substantial gains. Previously, the overhead

from DHT updates across distributed nodes resulted in excessive delays for revoking permissions, compromising system responsiveness and security in real-time applications. In contrast, the modified UCS consistently completed revocations in under two seconds, regardless of policy complexity. This improvement is critical in environments such as smart homes, where timely revocation is essential. While this modification sacrifices distributed scalability, it enhances performance significantly for centralized or small-scale deployments.

The removal of journaling functionality also contributed to improved performance but with different implications. Journaling was primarily used for ensuring data integrity during power failures and introduced substantial performance overhead due to frequent updates. By removing journaling, the UCS exhibited faster installation and running times, similar to the effects seen with DHT updates disabled. While the architecture remains distributed, the frequency of updates to the DHT is reduced, leading to a more streamlined and responsive system, albeit at a slight cost to fault tolerance.

Both modifications—disabling the DHT updates and removing journaling—led to significant improvements in revocation performance. Previously, the system's overhead in managing updates hampered timely access revocation. The modified UCS demonstrates that removing some features is essential for maintaining secure and prompt access control environments prioritizing performance. This balance between efficiency and resilience is especially beneficial in resource-constrained settings.

In comparing the two versions, the DHT-disabled configuration offered superior performance, with faster installation and running times than the journaling-disabled version. The overhead from maintaining the DHT proved more detrimental to performance than that introduced by journaling. However, the trade-offs remain significant: disabling DHT updates enhances performance but sacrifices the ability to scale across distributed nodes, whereas removing journaling improves performance while retaining a degree of fault tolerance.

Direct integration of policies into *TryAccess* requests marks a significant change in the interaction between the PEP and the UCS. In the original system, the PEP simply created *TryAccess* requests, while the modified architecture requires the PEP to actively select and embed policies into these requests. Although this adds complexity to the PEP's role, increasing the attack surface, it significantly reduces the workload of the UCS.

Upon receiving the *TryAccess* request, the UCS decodes the embedded Base64-encoded policy, streamlining the process by eliminating costly policy lookups. This new workflow has resulted in improved response times: *TryAccess* operations

average 2.33 seconds, with *StartAccess* operations even faster at about 1.2 seconds. The reduction in time between these two operations highlights increased efficiency from the decreased overhead in policy management.

While this modification adds some burden to the PEP, it ultimately benefits the overall system by offloading tasks from the UCS, allowing it to focus on decoding and evaluating policies rather than searching for them. This division of labor becomes particularly advantageous in smart home environments, where the number of requests and PEPs can grow exponentially. Although the end-user experience may remain largely unchanged, the reduced workload on the UCS can lead to better overall system performance during high demand.

4.6 Final Considerations

The analysis of the modified Usage Control System highlights its potential for efficient deployment on constrained devices like the Raspberry Pi 4. The modifications implemented—including the disabling of Distributed Hash Table updates and the removal of journaling functionality—have led to significant performance improvements, making the UCS a viable option for resource-limited environments.

Among the variations tested, the final architecture without journaling has proven to be the most favorable, striking an optimal balance between functionality, performance, and security in access control. By eliminating journaling, the system reduces overhead, which significantly enhances performance relative to the original version, where journaling is enabled.Additionally, this configuration maintains an acceptable level of system-wide resilience, particularly when compared to the alternative architecture in which DHT updates are also disabled. While the removal of journaling reduces the system's ability to withstand sudden interruptions—reducing local resilience—it achieves a middle ground by preserving the DHT updates, which ensure consistent management of access permissions across the network. This balance minimizes the potential impact of security vulnerabilities in individual nodes on the overall system while optimizing performance for constrained environments.

The modified UCS demonstrates improved installation and running times and ensures timely revocation of access permissions, consistently achieving revocations in under two seconds, regardless of policy complexity. This capability underscores its suitability for dynamic environments like smart homes, where rapid adjustments to access control are critical.

Moreover, the testing frameworks validate UCON as an effective mechanism for achieving Security by Contract compliance. By managing app-to-contract and contract-to-policy matching, UCON ensures applications align with predefined

security contracts, which is crucial in environments with rapidly changing security requirements.

The evaluation of UCON's dynamic monitoring capabilities has revealed its effectiveness in enforcing runtime restrictions, ensuring continuous alignment with user-defined security policies throughout the application's lifecycle.

Ultimately, the UCS in its modified form offers a robust solution for managing access control in constrained settings. With targeted optimizations, it can effectively handle complex policies and high volumes of access requests, making it an attractive choice for modern IoT applications. The findings from this research affirm that UCON can significantly enhance compliance with the S×C model, paving the way for a more secure and adaptable IoT ecosystem.

Chapter 5

PiCamera IoT Application with UCS on Constrained Devices

This chapter presents a practical example of enforcing Security-by-Contract compliance in IoT applications through the UCON framework. The proposed approach is applied to a common IoT device, the PiCamera, operating within the constraints of a Raspberry Pi platform. The primary objective is to demonstrate the effectiveness of the UCS framework in managing and enforcing real-time access control policies for video streaming, addressing essential security requirements in smart home environments.

For a practical reference, the application code implementing this example is available in the thesis repository[1]. This example serves as a final demonstration that utilizes the UCON framework for enforcing a monitored dev-API at runtime.

The approach assumes control over the methods available to application developers, i.e., the *dev-APIs*, used to construct the application. These *dev-APIs* are defined within the SIFIS-Home project, as explained in Section 2.5. A valid contract maker is also presumed to be accessible, allowing the creation of contracts, defined in Sections 2.4 and 3.1, by extracting pertinent information from the *dev-APIs* in the application code.

The chapter presents the application and its development, structured to detail

[1]Thesis repository: `https://sssg-dev.iit.cnr.it/marco-rasori/sxc-ucon-tests`

each stage of the application life cycle, including installation, policy enforcement during video capture, and revocation of access rights as managed by the UCON framework.

5.1 Application Development and Deployment

In the context of enforcing Security-by-Contract compliance using the UCON framework, the PiCamera application exemplifies a practical implementation designed for real-time access control in video streaming. This application operates seamlessly on a Raspberry Pi platform, leveraging the capabilities of the PiCamera while adhering to essential security requirements in smart home environments.

To facilitate effective control over the PiCamera, the application utilizes *FFmpeg*, a powerful multimedia framework that enables video processing and encoding. This allows users to capture high-quality video streams with customizable settings such as framerate, resolution, and quality. Additionally, the application employs *WebSocket* for real-time communication between the user and the User Control System (UCS).

The design incorporates hypothetical developer APIs, which encapsulate key functionalities and integrate a Policy Enforcement Point within the *dev-API* code. This integration allows the application to make real-time requests to the UCS, ensuring dynamic policy evaluations.

The APIs are:

- ***start_PiCamera_streaming***: This API initiates the video streaming process, allowing users to begin capturing video with configurable parameters on the storage location they prefer, with unique video files.
- ***update_default_settings***: This API enables adjustments to the default video settings, with modifications taking effect upon the next restart of the video streaming.
- ***update_current_settings***: This API facilitates the modification of current streaming settings by stopping the current stream, updating the video streaming settings, and restarting the recording. This API requires that the PiCamera is already streaming.
- ***stop_PiCamera_streaming***: This API stops the current video streaming process, ensuring that the PiCamera is no longer capturing video. It safely terminates the streaming session by killing the FFmpeg process responsible for the video capture and updates the application state to reflect that the camera is no longer running.

Additionally, the application is designed for easy deployment across various devices through containerization. By utilizing *Docker*, the application can be built and run on a Raspberry Pi 4 using the following command, ensuring compatibility with the ARM architecture:

```
docker build --platform linux/arm64 -t picamera-app $PATH
```

During the deployment phase, it is crucial that the application possesses a valid contract specifying the security-relevant actions it is permitted to perform, as explained in Section 2.4. This contract is derived from the *dev-APIs* and acts as a manifest detailing the application's intended security operations. Features provided by the contract maker may include:

- ***Starting the video stream***: This means that the application is able to control and initiate video streaming through the IoT device.
- ***Modifying video settings***: This means that the application can modify video settings, such as resolution, frame rate, and quality, in both static and dynamic ways.
- ***Stopping the video stream***: This means that the application can terminate the current video streaming process, ensuring that the IoT device is no longer capturing video.

5.2 Application Life-Cycle

The objective of this section is to outline the relevant stages in the application life cycle, which involves several critical phases: installation, execution, and the management of access permissions. The aim is to illustrate how the Usage Control System can effectively enforce Security-by-Contract compliance in a practical application.

To set the stage for the discussion, the primary subjects involved are identified:

- **User**: The individual who interacts with the application to manage video streaming.
- **Application**: The PiCamera application designed for video capture and streaming.
- **Smart Home**: The environment equipped with IoT devices, including the PiCamera.
- **User Control System**: The system responsible for managing permissions and enforcing security policies.

In this context, it is assumed that the user, or more specifically, the smart home administrator, has defined policies within the UCON framework that may affect the life cycle of the application, because they pertain to actions that may be executed through the various *dev-APIs* utilized in the application's development. For this section, it is hypothesized that the user has defined policies that state:

- An application that controls video recording must not be installed if it does not allow the user to specify where videos should be stored. This policy ensures that users are aware of how their video data will be managed.
- An application that controls video recording must not be installed if it overwrites any existing video files. This ensures that users do not accidentally lose important recordings.
- An application that controls video recording may not register videos with a resolution exceeding 1920x1080 to prevent excessive memory usage.
- An application that controls video recording may not lower the quality below a certain threshold, fixed at 15, during night hours (23:00-06:00). This prevents poor-quality recordings when lighting conditions are less than optimal, ensuring that any video captured at night remains usable. If this condition cannot be respected, the recording should be stopped to prevent memory usage of useless video.
- An application that controls video recording may not operate if the device's battery level is below 20 %. This prevents interruptions in recording due to sudden power loss.

5.2.1 Installation Phase

During installation, the UCS verifies Security-by-Contract compliance by performing Contract-Policy matching, which must be satisfied to allow the application installation. In the context of the PiCamera application, installation can be successfully completed in the smart home environment because it fulfills all specified requirements outlined by the user-defined policies.

The *start_PiCamera_streaming* dev-API, for instance, enables the application to initiate video capture while adhering to policies regarding video storage and file management. This API ensures that users can specify the storage location for video files and prevents overwriting existing recordings, aligning with the policies that safeguard user data.

However, while the application may install successfully, it is imperative for the UCS to activate monitoring. This is necessary because certain actions taken by the application during execution could potentially violate the defined policies. For

instance, because the application may allow changes to video recording quality and resolution settings, it is crucial to enforce restrictions, such as not permitting resolutions above 1920x1080 or quality levels below the specified threshold during night hours.

The UCS's monitoring role becomes critical in these scenarios to ensure that the application complies with all user-defined policies throughout its operation. This proactive approach helps mitigate risks associated with non-compliance, thereby maintaining the integrity and security of the smart home environment.

5.2.2 Running Phase

Once the PiCamera application is installed, it enters the running phase, where it actively engages in video capture and streaming according to the parameters set by the user. During this phase, the application utilizes the previously defined *dev-APIs* to perform its functionalities, while the UCS continuously monitors the application's compliance by performing Application-Policy matching. This ensures that the behavior of the application, as defined by its contract, remains within the boundaries of the policy's security requirements.

As the application operates, it may attempt to modify settings related to video capture, such as resolution and quality through the *update_current_settings* dev-API. When such a dev-API is invoked, the PEP asks for permission from the UCS, which evaluates these modifications against the established policies. For example, if the application tries to change the resolution to 2560x1440, the UCS will intervene, performing a corrective action that may involve stopping the video recording and logging the incident for compliance tracking.

5.2.3 Potential Revocation Phase

In cases where the application is found to be non-compliant with user-defined policies during the running phase, the UCS may initiate a potential revocation phase. This phase involves reassessing the application's permissions, which may lead to the suspension or complete revocation of access rights to certain resources.

The revocation process is triggered by changes in certain attributes that the application utilizes during its operations. For instance, if the application, which has started the recording with acceptable values, according to user-defined policies, during the session, attempts to change resolution exceeding the allowed 1920x1080 limit, the UCS could revoke its permission to record video. The user would then be informed of this action and the reasoning behind it, allowing them to make informed decisions regarding the application's usage.

Similarly, if the application continues to lower the video quality below the permitted threshold during night hours, the UCS might temporarily disable the application until compliance can be ensured. This could require the user to review the settings or reset the application before access is restored.

In addition, if the device's battery level falls below 20 %, the UCS will revoke video recording permissions to prevent interruptions due to sudden power loss. To delve in practical actions taken by the UCS in this phase, consider the scenario when the user initiates video capture via the *start_PiCamera_streaming* dev-API. At this moment, the PIP retrieves the current battery level, and the UCS evaluates the policy. If the battery level is above 20 %, access is granted, allowing the recording to proceed. As the application runs, the UCS continuously monitors the battery level, re-evaluating policies when changes occur, provided there is an active session with an associated battery-level condition. If the battery level drops below the threshold of 20 %, the UCS revokes recording permissions, preventing unauthorized operation. If the battery level remains above the threshold, recording continues without interruption. In either case, the user receives a notification, allowing them to manage device resources effectively.

Overall, the potential revocation phase acts as a safeguard, ensuring that the application adheres to established security requirements while allowing users to address any compliance issues that arise during operation.

5.3 Considerations

In conclusion, the analysis of the application life cycle in this practical example illustrates how the UCON framework serves as an efficient mechanism for enforcing Security-by-Contract compliance. The structured approach to installation, running, and potential revocation underscores the framework's capacity to maintain security and integrity in IoT environments, such us the smart homes.

Chapter 6

Conclusions

This thesis has critically examined the security challenges faced by IoT devices throughout their operational life cycle, highlighting the significance of effective security mechanisms. Central to this research is the validation of the Usage Control System within the UCON framework as an efficient tool for achieving Security-by-Contract compliance in smart home environments.

The findings demonstrate that the UCS is adept at managing access control policies, effectively addressing the dynamic security requirements inherent in smart homes. By aligning applications with predefined security contracts, the UCS ensures robust protection against potential vulnerabilities, thereby reinforcing the overall security posture of IoT devices in domestic settings.

Additionally, this research establishes that deploying the UCON framework on lightweight systems, such as the Raspberry Pi, is feasible and practical. The modified version of the UCS, as discussed in the thesis, facilitates this deployment without compromising performance or security.

In summary, the modified UCS emerges as a strong solution for access control management in constrained environments, bridging the gap between functionality, performance, and security. The insights gained from this study contribute significantly to the body of knowledge in IoT security, offering a foundation for further exploration and development of more secure and adaptable IoT solutions in smart home contexts.

Bibliography

[1] Statista and Transforma Insights. *Number of Internet of Things (IoT) connections worldwide from 2022 to 2023, with forecasts from 2024 to 2033*. Accessed: September 2024. 2024 (cit. on p. 1).

[2] Quentin Stafford-Fraser. *Trojan Room Coffee Pot Biography*. Accessed: September 2024. 2024 (cit. on p. 2).

[3] John Romkey. «Toast of the IoT: The 1990 Interop Internet Toaster». In: *IEEE Consumer Electronics Magazine* 6 (Jan. 2017), pp. 116–119. DOI: 10.1109/MCE.2016.2614740 (cit. on p. 2).

[4] Hua-Dong Ma. «Internet of Things: Objectives and Scientific Challenges». In: *Journal of Computer Science and Technology* 26.6 (2011), pp. 919–924. DOI: 10.1007/s11390-011-1189-5 (cit. on p. 2).

[5] Maarten Botterman. *Internet of Things: an early reality of the Future Internet*. sn, 2009 (cit. on p. 2).

[6] Sarah A. Al-Qaseemi, Hajer A. Almulhim, Maria F. Almulhim, and Saqib Rasool Chaudhry. *IoT architecture challenges and issues: Lack of standardization*. 2016. DOI: 10.1109/FTC.2016.7821686 (cit. on pp. 2, 7).

[7] Chang-Le Zhong, Zhen Zhu, and Ren-Gen Huang. *Study on the IOT Architecture and Gateway Technology*. 2015. DOI: 10.1109/DCABES.2015.56 (cit. on p. 2).

[8] Shadi Al-Sarawi, Mohammed Anbar, Kamal Alieyan, and Mahmood Alzubaidi. *Internet of Things (IoT) communication protocols: Review*. 2017. DOI: 10.1109/ICITECH.2017.8079928 (cit. on p. 4).

[9] Li Jiang, Da-You Liu, and Bo Yang. *Smart home research*. 2004. DOI: 10.1109/ICMLC.2004.1382266 (cit. on p. 5).

[10] Liyanage C. De Silva, Chamin Morikawa, and Iskandar M. Petra. «State of the art of smart homes». In: *Engineering Applications of Artificial Intelligence* 25.7 (2012). Advanced issues in Artificial Intelligence and Pattern Recognition for Intelligent Surveillance System in Smart Home Environment, pp. 1313–1321. ISSN: 0952-1976. DOI: https://doi.org/10.1016/j.engappai.2012.

05.002. URL: https://www.sciencedirect.com/science/article/pii/S095219761200098X (cit. on p. 6).

[11] Lo'ai Tawalbeh, Fadi Muheidat, Mais Tawalbeh, and Muhannad Quwaider. «IoT Privacy and Security: Challenges and Solutions». In: *Applied Sciences* 10.12 (2020). ISSN: 2076-3417. DOI: 10.3390/app10124102. URL: https://www.mdpi.com/2076-3417/10/12/4102 (cit. on pp. 6, 7).

[12] Rwan Mahmoud, Tasneem Yousuf, Fadi Aloul, and Imran Zualkernan. «Internet of things (IoT) security: Current status, challenges and prospective measures». In: *2015 10th International Conference for Internet Technology and Secured Transactions (ICITST)*. 2015, pp. 336–341. DOI: 10.1109/ICITST.2015.7412116 (cit. on pp. 7, 8).

[13] Shantanu Pal. «Limitations and Approaches in Access Control and Identity Management for Constrained IoT Resources». In: *2019 IEEE International Conference on Pervasive Computing and Communications Workshops (PerCom Workshops)*. 2019, pp. 431–432. DOI: 10.1109/PERCOMW.2019.8730651 (cit. on p. 9).

[14] Neil Vigdor - The New York Times. *Somebody's Watching: Hackers Breach Ring Home Security Cameras*. Accessed: September 2024. 2019 (cit. on p. 10).

[15] Swati Khandelwal - The Hacker News. *Z-Wave Downgrade Attack Left Over 100 Million IoT Devices Open to Hackers*. Accessed: September 2024. 2018 (cit. on p. 10).

[16] Verkada. *Summary: March 9, 2021 Security Incident Report*. Accessed: September 2024. 2021 (cit. on p. 10).

[17] Oscar Williams-Grut - Business Insider. *Hackers once stole a casino's high-roller database through a thermometer in the lobby fish tank*. Accessed: September 2024. 2018 (cit. on p. 10).

[18] Antonia M. Reina Quintero, Salvador Martínez Pérez, Ángel Jesús Varela-Vaca, María Teresa Gómez López, and Jordi Cabot. «A domain-specific language for the specification of UCON policies». In: *Journal of Information Security and Applications* 64 (2022), p. 103006. ISSN: 2214-2126. DOI: https://doi.org/10.1016/j.jisa.2021.103006. URL: https://www.sciencedirect.com/science/article/pii/S221421262100212X (cit. on pp. 14, 15).

[19] Marco Rasori, Andrea Saracino, Paolo Mori, and Marco Tiloca. «Using the ACE framework to enforce access and usage control with notifications of revoked access rights». In: *International Journal of Information Security* 23.5 (2024), pp. 3109–3133. ISSN: 1615-5270. DOI: 10.1007/s10207-024-00877-1. URL: https://doi.org/10.1007/s10207-024-00877-1 (cit. on pp. 14–17, 20).

[20] Hany F Atlam, Madini O Alassafi, Ahmed Alenezi, Robert John Walters, and Gary B Wills. *XACML for Building Access Control Policies in Internet of Things*. 2018 (cit. on pp. 21, 22).

[21] Alessandro Aldini, Antonio La Marra, Fabio Martinelli, and Andrea Saracino. «Ask a(n)droid to tell you the odds: probabilistic security-by-contract for mobile devices». In: *Soft Comput.* 25.3 (Feb. 2021), pp. 2295–2314. ISSN: 1432-7643. DOI: 10.1007/s00500-020-05299-4. URL: https://doi.org/10.1007/s00500-020-05299-4 (cit. on pp. 22, 24, 25).

[22] Antonio La Marra, Fabio Martinelli, Paolo Mori, and Andrea Saracino. «A Distributed Usage Control Framework for Industrial Internet of Things». In: *Security and Privacy Trends in the Industrial Internet of Things*. Ed. by Cristina Alcaraz. Cham: Springer International Publishing, 2019, pp. 115–135. ISBN: 978-3-030-12330-7. DOI: 10.1007/978-3-030-12330-7_6. URL: https://doi.org/10.1007/978-3-030-12330-7_6 (cit. on pp. 23, 24).

[23] Alberto Giaretta, Nicola Dragoni, and Fabio Massacci. «IoT Security Configurability with Security-by-Contract». In: *Sensors* 19.19 (2019). ISSN: 1424-8220. DOI: 10.3390/s19194121. URL: https://www.mdpi.com/1424-8220/19/19/4121 (cit. on p. 24).

[24] Joni Jämsä. *SIFIS-Home*. Accessed: 2024-10-05. 2020. URL: https://www.sifis-home.eu/ (cit. on pp. 26, 27).

[25] Mirjana Maksimović, Vladimir Vujović, Nikola Davidović, Vladimir Milošević, and Branko Perišić. «Raspberry Pi as Internet of things hardware: performances and constraints». In: *design issues* 3.8 (2014), pp. 1–6 (cit. on p. 32).

[26] Øyvind Strøm, Kjetil Svarstad, and Einar J. Aas. «On the Utilization of Java Technology in Embedded Systems». In: *Design Automation for Embedded Systems* 8.1 (Mar. 2003), pp. 87–106. ISSN: 1572-8080. DOI: 10.1023/A:1022344203816. URL: https://doi.org/10.1023/A:1022344203816 (cit. on pp. 33, 34).

[27] Stan Kurkovsky and Chad Williams. «Raspberry Pi as a Platform for the Internet of Things Projects: Experiences and Lessons». In: *Proceedings of the 2017 ACM Conference on Innovation and Technology in Computer Science Education*. ITiCSE '17. Bologna, Italy: Association for Computing Machinery, 2017, pp. 64–69. ISBN: 9781450347044. DOI: 10.1145/3059009.3059028. URL: https://doi.org/10.1145/3059009.3059028 (cit. on pp. 33, 34).