An Efficient Dynamic Key Management Scheme for IoT Devices

by

Vishnu Prasanth Vikraman Pillai

Bachelor of Technology in Computer Science and Engineering,
Mahatma Gandhi University, 2006

A THESIS SUBMITTED IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE DEGREE OF

Master of Computer Science

In the Graduate Academic Unit of Computer Science

Supervisor:	Rongxing Lu, Ph.D., Faculty of Computer Science
Examining Board:	Haruna Isah, Ph.D., Faculty of Computer Science, Zhen Lei, Ph.D., Department of Civil Engineering, Chair Sajjad Dadkhah, Ph.D., Faculty of Computer Science

This thesis is accepted by the
Dean of Graduate Studies

THE UNIVERSITY OF NEW BRUNSWICK

October, 2023

Abstract

The Internet of Things or IoT is a collective term for electronic devices with computing and connectivity. Our proposed dynamic key management scheme is designed for secure group communication of IoT devices. It offers efficient key distribution for a small to medium group of devices in domains such as centralized healthcare systems. Our key management scheme ensures forward secrecy, backward secrecy, and key independence in group communication. The scheme uses binary heap trees and bloom filters for efficient storage, organize and verification of secret keys. It uses polynomial coefficients secured with modular arithmetic to distribute the keys. The proposed implementation of the scheme uses lightweight mathematical operations such as XOR, multiplication, string concatenations, and hashing for devices having limited computing capabilities. The thesis is concluded with the performance analysis of the scheme that demonstrates the suitability of the scheme with similar IoT group communication schemes.

Dedication

I present this thesis as a homage to those individuals who have significantly shaped the course of my journey up to this juncture. My deep gratitude extends to my mother, father, sister, teachers, friends, and even compassionate strangers who have offered steadfast support and nurturing throughout the expedition of my life. I also recognize the challenges posed by adversaries, which have fueled my aspirations for greater heights. It's undeniable that without your persistent encouragement and reminders, the achievements I've attained would have eluded me.

Acknowledgements

I want to express my earnest appreciation to Dr. Rongxing Lu, my supervisor, for his unwavering support, time, and dedication to his students. Dr. Lu's efforts have helped me reshape my technical knowledge and professional values, for which I am forever grateful. I wish to express sincere thanks for the exceptional guidance provided by Dr. Kalikinkar Mandal in his role as an instructor. I extend my gratitude to Dr. Ali Ghorbani from the Canadian Institute of Cybersecurity for his endorsement and to Mohammad Mamun from National Research Council, Canada, for his indispensable mentorship. My heartfelt thanks also go to my family for their support. I sincerely convey my gratitude to the University of New Brunswick for providing me with this invaluable opportunity. I would also like to thank the university staff for their devoted efforts.

Table of Contents

List of Tables

List of Figures

Abbreviations

IoT	Internet of Things
KDC	Key Distribution Center
LKH	Logical Key Hierarchy
$GKMP$	Group Key Management Protocol
PKI	Public Key Encryption
PAC	Polynomial based Access Control Technique
AES	Advanced Encryption Standard
ts, t	Timestamp
μs	Microseconds

Chapter 1

Introduction

1.1 Overview of IoT Key Management

The pervasive influence of technology in nearly every aspect of human life represents one of the most profound contemporary societal transformations. The recent technological advances that made digital devices relatively smaller in size, mobile, and low cost, spawned a new era of computing in a connected environment with smart devices. The increased dependency on smart devices rendered the dire need to collectively address and manage their operations. The involvement of smart devices can be witnessed in diverse application domains [36] including transportation, healthcare, smart homes, and agriculture, and more. This new era of computing in a connected environment of smart devices is christened the Internet of Things or IoT. While any device with computing and connectivity capabilities can be referred to as IoT, it is typically considered that they have low computing capabilities and operate with limited power sources, such as built-in batteries, as seen in devices like smartwatches. It transformed the social sphere with interactive, assistive, informative, and engaging technologies. The advent of IoT also influenced conventional industrial systems, initiated Industry 4.0, and extended the IoT world to include Industrial IoT or IIoT.

Among the many infrastructure enablers of IoT devices, the layer of security is a fundamental entity that establishes trust among a diverse group of communicating devices. As the mobility of IoT devices is becoming a central theme in the new era of automation [42] across various industries handling information, addressing the security of IoT devices has become a new challenge. The anticipated ubiquitousness of IoT devices prompted us to envision an algorithm that aligns with the sustainability goals of the near future. Our research is the continuation of a set of key management methods proposed in the early days of the Internet (detailed in the related works section), a period when algorithms had to be designed for efficiency owing to the limited availability of computers with high computing capabilities. While selecting the desirable features of an existing design, our method further optimizes the efficiency of the referenced methods to suit the dynamicity and real-time requirements of IoT devices.

Data encryption techniques are the de facto solution for securing data in communication. A variety of symmetric encryption techniques are present with proven security and efficiency. Security of the encryption techniques is dependent on how securely the keys are stored and exchanged. The survey by Deogirikar et al. lists a diverse set of attacks on IoT devices due to the leakage of encryption keys [13]. An efficient key management framework is quintessential to ensure the security of IoT devices. A good key management system should limit the access of both keys and the encrypted messages only to its intended recipients for the intended duration. This is ensured by the timely allocation and revocation of session keys. While denying access to unintended recipients, an efficient key ownership method should also mitigate the collusion risk from the excluded users by making the keys random and independent of each other. The major challenge of symmetric key encryption schemes is the lack of efficient methods for key exchange or *rekeying* [1].

[1]Replacing encryption keys with new ones.

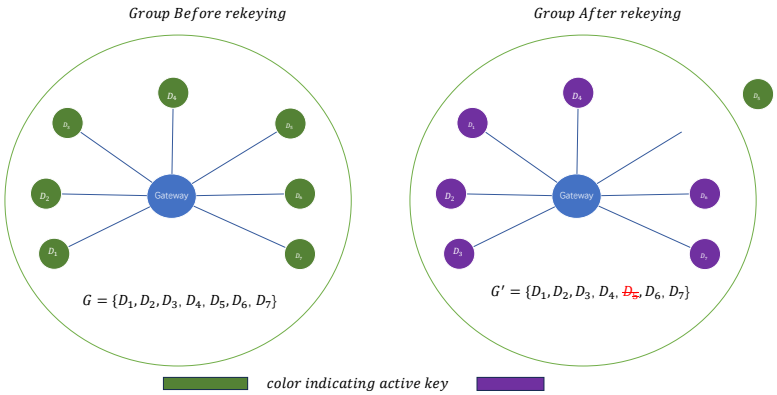

Figure 1.1: Impact of rekeying in a group.

Rekeying methods securely exchange new keys and dynamically create communication groups within a set of trusted devices. The figure 1.1 shows how rekeying transforms the group $\mathcal{G}$ to $\mathcal{G}'$ dynamically and removes the device D_5 from the set of devices $\{D_1, D_2, \cdots, D_7\}$.

The Diffie-Hellman key exchange scheme is a popular and secure symmetric key exchange system. However, it is not a suitable option for a key management system because it cannot be generalized for use in group communication involving more than two members [7]. Another reason to ignore Diffie-Hellman for IoT communications is that is designed for communication among devices with equally high computing capabilities. A typical IoT group communication scenario involves a server with high computational power on one side and many small, low-energy devices on the other side possessing low computational power. A popular key exchange technique is the exchange of encrypted symmetric keys using asymmetric encryption. This technique also involves mathematical exponential operations for large numbers at the receiving end and hence, is not the right fit for IoT devices built for lightweight computations only. The paper by Masanobu Katagi et al. from the Sony Corporation [23] explains why lightweight frameworks are the best choice for IoT device security.

1.2 Motivation and Vision

The world is grappling with climate change due to greenhouse gas emissions, with an increasingly significant share of computers and other digital devices. Alternatives are sought for reducing the carbon footprint in urban transportation, farming, and energy production as well as in various digital ecosystems. Efficient energy consumption is one of the best-proven ways to reduce the carbon footprint. In a world where computing is indispensable, the pathway for energy efficiency is the use of green algorithms [26]. One such example is the replacement of the proof-of-work algorithm with the proof-of-stake algorithm in the blockchain-based cryptocurrency Ethereum in 2021, which had a colossal impact on the Environment [48]. Integration of this algorithm based on the proposition by Sunny King et al. [24] resulted in substantial energy savings. This became a significant motivation that influenced the objectives of our research. Another motivation for our research was the increasing demand for privacy in recent times. The recent surge in devices handling personal data introduced various challenges related to privacy and security. Strong security measures are crucial when dealing with a smartphone containing banking information, whereas a smartwatch connected to social media should ensure privacy by not divulging sensitive health data about the user. The emergence of IoT devices in healthcare is a desirable but vulnerable transformation as it can compromise the privacy of individuals and related security concerns. [14] highlights a third of security breaches among IoT devices happen in the healthcare industry. Nuances in handling individual health data with forward and backward secrecy is a detailed research area [35]. As a result, there is a growing need for reliable, resilient, and robust security frameworks around handling the security and privacy of the information being exchanged.

1.3 A Brief Overview of Contributions

We suggest a dynamic key management system that is practical, efficient, and can exchange keys with minimal communication. The proposed method is a lightweight algorithm that can find applications in fast-paced environments involving trusted IoT device groups. It is suitable for group rekeying and scalable without significantly reducing performance. It has the following features.

- It is an efficient method designed to securely exchange encryption keys in a private network of smart devices ranging from a dozen to a few hundred. It offers forward security and backward security to the data shared by devices based on group membership.

- Our method incorporates many mathematical primitives to achieve security, speed, and low computational and communication expenses, the primary design goals. It uses logical key hierarchies [17] to group devices based on session keys, and uses polynomial coefficients with modular arithmetic.

- Our technique uses Bloom Filter [3] for validating faster validation of probable keys with the server. Timestamps are concatenated with every exchanged message to better defend against attacks, and hashing is used in communication to reduce the exposure of original key values.

- Reducing the number of communications for rekeying with scenario-based communication methods is another feature of the proposed method. The least efficient communication method in a group is unicasting. In our method, the unicast situations are replaced with multicast scenarios by categorizing different use cases and addressing them. We leverage the parent-child relationship among keys in the hierarchical organization by using the XOR operation to derive new keys of individual devices instead of unicasting them from the Gateway

server.

- Our design takes into consideration issues in group key management such as collusion attacks. Wallner et al document various possible attacks in multicast scenarios and recommend hierarchical solutions for key management [47].

1.4 Assumptions Made During Design

There are several assumptions consciously made while writing this thesis that helped us focus on our key management scheme. The scheme assumes that the devices in the group are already authenticated by a protocol such as TLS [46] and protected against any masquerading attacks. The proposed protocol neither focuses on the authorization of devices over active communication channels nor access rights on archived communications. The proposition relies on the established security principles of symmetric key encryption and the frameworks available for security in communication. The scheme primarily focuses on theoretical efficiency and does not address other potential adversarial factors that may impact performance. The limitations of this research are exempt from detailing how the scheme is secured against general security vulnerabilities [51].

1.5 Thesis Organization

After this introductory chapter, the thesis provides a detailed view of cryptographic fundamentals and related works in Chapter 2. The system model outline that describes the structure of the proposal and design goals can be found in Chapter 3. Chapter 4 explains the preliminaries of our scheme that are mandatory in understanding the proposed scheme detailed in Chapter 5. Chapter 6 analyzes the security aspects of the scheme and defends the security goals proposed in Chapter 3. The

performance of the scheme measured after its implementation in Java is presented in Chapter 7. Chapter 8 concludes the thesis with our remarks, observations, and potential future works related to dynamic key management.

Chapter 2

Background and Related Works

This chapter presents a brief overview of the necessity of cryptography fundamentals and its relevance in the communication of IoT devices.

2.1 Cryptography

Cryptography is an active applied research field employed in securing digital data against adversaries using mathematical techniques. In cryptography, the data that needs securing is known as *plaintext*. The encrypted data is called *ciphertext* and can be transferred securely to any intended recipient through open channels. The received *ciphertext* is made legible by a legitimate receiver using a method known as decryption. The encryption and decryption methods are not secured and are available to anyone. The security in the transformation from plaintext to ciphertext is established with a secret number known as the *key*. The content is accessible only to those who possess the key.

The digital data may exist in a state of rest, in use, or in transit [12]. The most vulnerable state for digital data is when it is in transit. While modern electronic inventions made communications efficient for the data in the digital form, the advancements in computing increased the capability of successful interception of digital

communications, especially for the Internet of Things (IoT) devices, thus raising a need for enhanced protection for the data in transit [28] and the availability of devices involved. Hence, the attempt to leverage modern cryptography to guard digital communication among IoT devices started [19]. Modern cryptography for data in transit is shaped around a few fundamental security goals listed below.

- Confidentiality: To limit the data access to an intended group of users. Data Confidentiality is an important aspect of IoT devices deployed in scenarios that handle sensitive data. The data can be personal information, healthcare data, critical industrial data and so on.

- Integrity: Integrity offers verifiable techniques to ensure unauthorized parties have not modified the data. The digital signature is a popular technique for ensuring integrity in the digital world. The IoT devices handling data use a variety of solutions ranging from hashing and message authentication code to Blockchain [27] for ensuring the integrity of communication.

- Availability: Availability is the time-bound assurance of resources. Availability aims to achieve the reliability of the systems in service. This can involve guarding the infrastructure against possible disruptions, unplanned downtime, and deniability of access. Different measures for availability include redundancy, disaster recovery plan, and fault monitoring. Availability is a crucial security goal for systems that rely on real-time data from IoT devices.

- Authentication: It's essential to ensure the entities involved in a communication are known and verified. Authentication procedures provide a verification mechanism for access or communication endpoints. There are various ways to authenticate, such as usernames and passwords, electronic and biometric IDs, and multifactor authentication(MFA). Authentication is an integral part of access control.

2.2 Encryption

The secure exchange of data over a communication channel requires cryptography that involves a set of 3 algorithms -an encryption algorithm to encode the information, a decryption algorithm to decode the information, and a key generation algorithm to generate keys whose possession controls the access to specific encryption or decryption. The data to be encrypted is termed plaintext, and the encrypted data is known as ciphertext. The ciphertext results from 3 components, the plain text, the algorithm, and a secure parameter known as the key.

2.2.1 Encryption Keys

Encryption keys are an integral part of cryptography. The age-old principle from the 19th century that is still relevant in cryptography, known as Kerckhoff's principle [31], emphasizes the security of encryption to be dependent on the key rather than the algorithm used for encryption. The keys are pivotal in encryption, and hence, the security of the encryption often translates to how securely the keys are handled. The encryption techniques are widely grouped into two based on the nature of the keys.

2.2.2 Symmetric Encryption

This type of encryption technique uses a shared key for encryption and decryption. Advanced Encryption Standard, abbreviated as AES [9] is the most popular symmetric encryption technique and is also the example chosen in our proposal due to the abundance of research available on its efficiency on IoT devices [32] and [21]. Other symmetric encryption techniques include DES [8] and PRESENT cipher [4]. Our method suggests using the widely used AES symmetric encryption technique to secure communications in a public network. This thesis uses the following notations

for encryption and decryption of *data* with AES using the secure session key $k_i \in \mathbb{K}$. The equation 2.1 shows an example of how AES algorithms are used.

$$\begin{aligned} AES_{Encryption}(data, k_i) &= ciphertext \\ AES_{Decryption}(ciphertext, k_i) &= data \end{aligned} \tag{2.1}$$

2.2.3 Asymmetric Encryption

Asymmetric Encryption uses a pair of two different keys, unlike symmetric encryption, which uses just one key for both encryption and decryption operations. Though distinct, the keys in the pair are not random but are related mathematically. The intended recipient of encrypted messages is responsible for creating this pair of keys, of which one key will be advertised among entities that the recipient is interested in engaging in communication. Asymmetric encryption is a popular authentication mechanism for Web 3.0 in the form of certificates and digital signatures. The drawback of asymmetric encryption is the requirement of heavy computing using modular arithmetic, making it less desirable for encrypting communication where low-energy devices are involved. Asymmetric encryption still finds its use in the encryption and exchange of symmetric keys.

2.3 Key Management

Key management refers to the lifecycle management of secure keys, including key generation, key storage, key distribution, usage, and expiry of the secure keys used in symmetric encryption. Key managing typically involves a set of algorithms that decides how keys are generated, how symmetric keys are transferred to the intended recipients, how they are stored, and how they are disposed of. The nuances in handling keys are identified and independently addressed using key management de-

pending on the use case. The book titled *Cryptographic Key Management Issues and Challenges in Cloud Services* [6] details the diverse aspects of key management. It includes authentication, authorization of users and devices, protection of command and data from spoofing, integrity to communication by defending against unauthorized modifications, protection of keys from unauthorized disclosure, and outlines the security requirement of the size of the keys to be used. Key management is a vast topic. While we expect a secure key management system to meet all the mentioned criteria, our proposal only addresses a focused part of how keys are distributed and expired in a resource-constrained network.

2.3.1 Key Generation

Depending upon the algorithms we use, keys are generated in various ways. This thesis mentions how our technique generates AES symmetric encryption keys for consumption. Two different methods are adopted for the AES key generation. The first method uses an AES key generation algorithm API. The second method involves XOR operation between existing AES keys to create new AES keys.

2.3.2 Key Storage

Various hardware and software methods are available for secure storage of keys. This includes hardware security modules, "Software-based" key stores or vaults, and trusted platforms. Key storage is beyond the scope of our proposal and not addressed in this thesis.

2.3.3 Key Distribution

Key distribution ensures the safe transfer of keys from one device to another. Various key distribution techniques are present in addition to those mentioned earlier. Our

proposed method uses two types of key distributions - a secure channel key transfer and an encrypted transfer with known keys.

2.3.4 Key Revocation

Revocation or disposal of the keys is required to deny entities access to future communications. In our proposed method, the key revocation is done by creating new keys that will expire the old keys.

2.3.5 Dynamic Key Management

Dynamic key management is key management in a continuously changing group environment called *dynamic groups* where authenticated devices are given temporary access to resources, as opposed to *static groups* where no devices join or leave the group during the group's lifetime. We follow the detailed definition of static and dynamic groups proposed in the Polynomial interpolation-based group communication scheme proposed by Purushothama et al. [39]. This scenario comes in various dynamic networks. Among the mobile devices forming networks, they are collectively called Mobile Ad-hoc Networks or MANET [30]. While computer networks made distributed computing possible [34] and shaped the world of network security, the feature of mobility has made the networks dynamic [38] and introduced the need for the real-time managing of computing devices in networks. The security vulnerability is intrinsic to dynamic networks and requires the incorporation of dynamicity in the associated security layer. The inherent uniqueness in various types of dynamic networks contributed to a diverse set of security solutions, many detailed by Kuhn et al. [25]. For the sake of brevity, our thesis will focus on the dynamic key management alone and will not delve into the intricacies of the dynamicity of networks. For solutions that involve symmetric cryptography, dynamic key management became an integral component of information security in dynamic networks. The objective

of dynamic key management is to effectively manage secure communication channels with the timely distribution and revocation of encryption keys. A detailed survey about dynamic key management can be found in the publishing by Eltoweissy et al. [15]. An efficient key management solution will not only be secure. Still, it will also empower the scalability of the network and optimize the use of scarce resources, particularly energy that extends the mobility of devices involved in communication.

2.4 Related Work

Traditional network architectures, such as the Kerberos protocol, detail the fundamental components involved in secure group communication. The central theme of Kerberos is a trusted entity known as a key distribution center (KDC) for authenticating new devices into group communication. The trusted Gateway server also handles this responsibility in our key management protocol. One of the earliest architecture propositions for managing cryptographic keys for multicast communications can be found in the Group Key Management Protocol by H. Harney et al. [18]. It lists the fundamental requirements for the secure creation and distribution of secret keys. The GKMP refers to the ISO-7498 security service standards for the security aspects of group communication and shortlists the factors relevant to key management as data confidentiality, data authentication, and source authentication. Though not specifically designed for the IoT scenarios involving dynamic devices, the GKMP architecture is designed for the internet; its fundamentals are relevant and draw inspiration from the architectural components in our proposal. It describes the idea of group key, KEK(key encryption key), rekeying, and deletion. The GKMP architecture, however, expects the participants to be trusted as it assumes the individual devices' deletion of the group key is the responsibility. In our design, we presume the participant devices as $honest - but - curious$.

Logical Key Hierarchy (LKH) is a construct that maps all members of a group as leaves of a tree structure, e.g., a balanced binary tree. This LKH data structure is adopted from a few propositions [49], [50], and initially proposed by Wallner et al. [47]. In the LKH tree, the group key is located at the root of the tree, whereas the leaves represent the individual keys of all group members. In addition to the group key and the individual keys, each group member also needs to store all node keys in the path from the leaf key to the root key. If the number of group users is n, then the height of the tree is $h = \log_2 n$, and each user holds $h+1$ keys. When a new user joins or an old user leaves, all keys from the user's leaf key to the root key should be updated. The complexity for key distribution to n users will be $O(\log n)$, which includes some multicast and unicast communications.

Another proposal for hierarchical management of shared keys uses One-way Function trees (OFTs) introduced by Sherman et al. [44]. This scheme does so while achieving forward security and backward security but could not get rid of the unicast scenarios involved. In 2006, Kang et al. [22] optimized the LKH scheme with one-way hash functions for generating new keys and using a node coordinates logic for identifying tree nodes. This method slightly reduced the complexity for nodes leaving the scenario and reduced the storage space required for the keys. Yet, their scheme still requires multicasting to ensure forward security when a user leaves the group and has a logarithmic complexity for computing a replacement key. Purushothama's group scheme [39] proposed in 2013 is another related work. This work also offers key distribution for joining and leaving use cases with logarithmic complexity. The improvement in Purushothama's scheme from the LKH scheme is that it uses Hash functions for secure distribution. The scheme is optimized but still requires multicasting to ensure forward security when a device leaves the network.

In 2013, Piao et al. [37] proposed a polynomial-based key distribution scheme, which needs one re-key message to handle users' joining and leaving. However, the re-key

message includes a polynomial constructed from all n users' secret keys shared with the group controller, which makes its communication cost $O(n)$. In addition, it does not satisfy the key independence; that is, once the group key and $n-1$ individual secret keys are compromised, the rest individual keys can be discovered.

In 2018, AlBakri et al. [2] introduced a hierarchical polynomial-based key management scheme, which follows a similar network structure and uses a trivariate polynomial to distribute keys in bivariate polynomial form using a broadcast key. However, their scheme is not dynamic and cannot handle the events of group users' leaving and joining.

In 2020, Dammak et al. [10] proposed the decentralized lightweight group key management scheme based on the Chinese Remainder Theorem. However, their scheme requires users to own a considerable number of subkeys and involves more computational efforts, including multiple decryptions per device for a single change in the group.

Similar to the abovementioned, our proposed scheme has a binary heap tree-based construct. However, our scheme aims to reduce the communication complexity of the referenced protocols. In our method, when a new user joins the group, the communication cost of re-keying for existing group users is only $O(1)$; when an old user leaves the group, the communication cost of re-keying is also only $O(1)$, where N is the maximum number of users in a group. Using the modular polynomial coefficient broadcasting method, our scheme avoids the need for unicasting while users leave the group.

Chapter 3

Models and Design Goals

This chapter formalizes the proposed system model, security model, and design goals.

3.1 System Model

Similar to many other protocols proposed in domains involving IoT handling personal data [43], [29], our proposed scheme is designed around two type of entities, viz., a group of IoT devices in a home network, and a Server they are connected to. The system model in consideration consists of a small to medium group of IoT devices ranging from half to a few dozen in number, denoted as $\mathcal{D}$ authenticated by a server $\mathcal{S}$. A diverse set of IoT devices are deployed in various usecases. The IoT devices in consideration of this is are generic in nature, and does not expect to have a specific set of properties. However, the general assumption made during this design is that the IoT devices are low-powered sensor devices connected to hub. The local server $\mathcal{S}$ is assumed to have high computing capabilities and is connected to a n uninterrupted powersource. $\mathcal{S}$ acts as a trusted hub for the iternal network communication needs of the IoT devices. $\mathcal{S}$ also acts as a Gateway for the communication of IoT devices with external computers, ensuring the interactions are authenticated and secured. One of the roles of $\mathcal{S}$ when interacting with $\mathcal{D}$ is to manage the encryption keys

used in communication, which is the topic of detailing in this thesis. Our design is suitable for a set of smart devices such as a group of wearable smart healthcare devices in an *Aging In Place* [16] systems setup, *Intelligent intersection management of autonomous vehicles* [33], or in *smart homes* [11]. To illustrate the application of our key management method, we use the exmple of *Aging In Place*. Aging in place refers to the ability of an individual to continue living in their own home or community, typically as they grow older, rather than moving to a care facility or nursing home. It involves making modifications to the living environment to accommodate the changing needs of the elderly person. Installing smart devices is one such modification that acts as a technology enabler for aging in place. Figure 3.1 shows an Aging In Place scenario where our dynamic key management system can be applied to ensure the privacy of healthcare data of individuals.

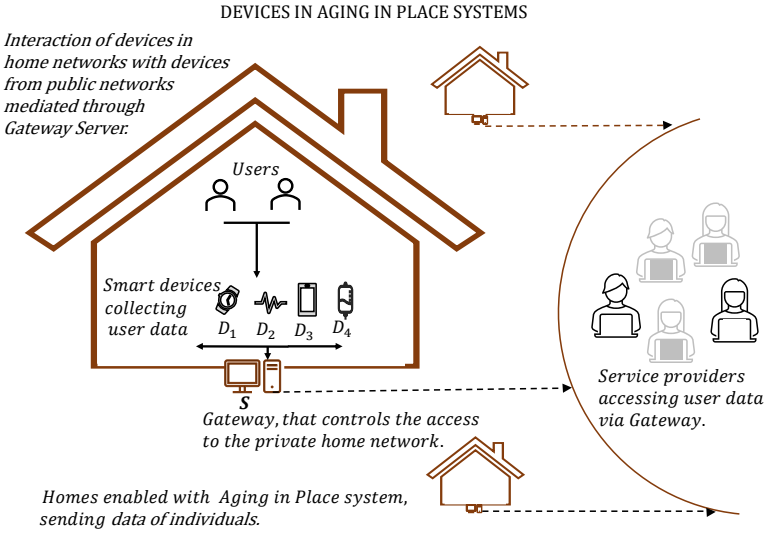

Figure 3.1: An IoT device group under consideration of the key management system

3.1.1 IoT Devices

The group $\mathcal{D}$ consists of a set of devices detailed as $\{D_1, D_2, \cdots, D_n\}$. These are a set of smart IoT devices capable of sensing, processing, storing, and transmitting

data depending on the application environment. In the case of Aging in Place, these devices collect and transmit healthcare data of individuals ranging from blood pressure, activities, heart rate, and any such diagnosable healthcare-related data. Due to the low energy design, their data processing and storage capabilities are limited. The devices are installed and operated under the control and ownership of a server computer known as Gateway server denoted as $\mathcal{S}$.

3.1.2 Gateway Server

The gateway server $\mathcal{S}$ is a computer with an active power source that acts as a portal between a private group of devices and an external network. It accounts for the devices in the private network and is considered as a trusted entity by the devices in $\mathcal{D}$. The Gateway server is the central entity in the security architecture of IoT device networks and is thus responsible for the authentication of IoT devices in a secure way. From the key management perspective, $\mathcal{S}$ is responsible for the initial key assignment, future key distributions, and the expiry of keys among the IoT devices.

3.1.3 Communication Group

The criteria that determine the access of a device to a communication is governed by the *group membership*. An IoT device D_i gains the membership of a coherent group $\mathbb{G}$ by gaining the session key for the encrypted communication. Especially in large installations with thousands of IoT devices, grouping is useful in addressing devices collectively rather than individually for purposes such as data collection, broadcasting messages, etc. In our scenario, we use the group for broadcasting new keys to the member IoT devices.

3.2 Security Model

In our security model, we consider the gateway server $\mathcal{S}$ to be fully trustable, which initializes the whole system, and manages the group keys for all IoT devices in the AiP system. While for the IoT devices $\mathcal{D} = \{D_1, D_2, \cdots, D_n\}$, we consider them as *honest-but-curious*, i.e., they will faithfully follow the protocol, however, when certain situations are satisfied, they will be curious about other unauthorized information. Specifically, the following three properties should be achieved for dynamic key management in our system.

3.2.1 Forward Secrecy:

Forward secrecy [20] protects future communication messages from an exposed old key. Consider a scenario where a device D_i that has access to a valid key K is no longer part of a communication group $\mathcal{G}$. D_i may continue to access the messages it received while it was part of the group $\mathcal{G}$, but the future messages should not be accessible to D_i. This is achieved by invalidating K for future use. The new group $\mathcal{G}$-D_i will be supplied with a new key K_new that is not accessible to D_i.

3.2.2 Backward Secrecy:

Backward secrecy protects the communication history from a new member device with full access. Let us say a device $D_j \notin \mathcal{G}$ is joining the group $\mathcal{G}$ and has access to encrypted message history. It will still not be able to decrypt the message because it lacks the session keys used before its joining. Backward secrecy ensures that the previous keys can neither be accessed nor be derived mathematically. It is made possible with the secure replacement of session keys, one of the key features of our protocol.

3.2.3 Key Independence:

Key independence is a property that ensures one compromised key does not give an advantage to the adversary in compromising more keys. This is achieved through granular control. In addition to the group key, each IoT device is assigned an individual key by the gateway server $\mathcal{S}$ in our system. Key independence requires that even though the group key and some individual keys are compromised, other uncompromised individual keys cannot be discovered.

3.3 Design Goals

Our design goal is to design a secure and efficient dynamic key management scheme for IoT systems, which should achieve the following two properties:

3.3.1 Security

The proposed scheme should achieve forward security, backward security, and key independence properties so as to ensure secure IoT device communications.

3.3.2 Efficiency

One of the key design goals we aim to achieve in our protocol compared to the other related works is efficiency. The proposed scheme is efficient and scalable within a range of a number of devices. For dynamic group key updates due to the user leaving or joining, communication costs and storage overhead between the gateway server and IoT devices should be as low as possible.

Chapter 4

Preliminaries

This chapter details the mathematical constructs that build our scheme. The various factors in the problem statement are addressed with the help of various cryptographic techniques. The techniques used in our proposal include hashing technique, bloom filter, and polynomial equations using modular arithmetic.

4.1 Hashing

Invented with the goal of easy searching in large data, Hans Peter Luhn [45] created this one-way process to generate a unique binary output known as a message digest for a given input data. The fixed-length output of a hashing algorithm is consistent for the same input. It can be used to verify the integrity of sent data at the point of reception in a communication. Hashing algorithms are highly efficient in computing the message digest [40], a popular method for verifying the integrity of communicated messages and stored data. The deterministic nature of the hashing algorithm is utilized in mapping locations in storage and memory, allowing faster search operations. Hash functions are one of the building blocks of Bloom Filter used in our proposition. Hash functions are one-way functions, which means the input value cannot be determined from a hash value. This property of the hash

function is used to enhance the security of our method. In our approach, rather than using actual cryptographic keys, we employ only their hashed values in composing the polynomial. This allows for broadcasting the polynomial without the need for additional encryption. To further enhance security, we concatenate the key input with a timestamp, ensuring that cryptographic keys cannot be identified from the hash, even with an exhaustive approach.

4.2 Bloom Filter

The purpose of Bloom Filter [3] is to verify the presence of an element in a set. A Bloom filter (BF) is a space-efficient data structure designed for the highly efficient querying of the presence of elements. A Bloom filter query only returns a $true$ or $false$. The Bloom Filter is a probabilistic data structure. This means that it can yield positive results with a high accuracy, but not guranteed. The high efficiency of BF comes at the cost of occasional false positive results for a given query. However, the BF never returns a false negative. In its bare form, a BF consists of a few independent hash functions $\mathcal{H} = \{h_1, h_2..h_k\}$ and a considerably large bit array $A\,[1, 2, \ldots N]$ of length N, with all its bits initially set to zero.

4.2.1 Bloom Filter Construction

A BF has a bit array $A[]$ as its central theme for its storage purposes. This array is connected to a set of hash functions $\mathcal{H} = \{h_1, h_2, \cdots, h_k\}$ where each $h_i \in \{h_1, h_2, \cdots, h_k\}$ in is independent of the other. Each hash function can intake a binary integer of random length $\{0, 1\}^*$ and map it to a specific position in the array, setting that bit as 1.

4.2.2 Storing Elements in BF

A BF is capable of storing the presence of multiple elements by setting the bits in the array $A[]$. The storage of the BF is accessed only through its hash functions. Consider $\mathcal{M}$ is a set of numbers with $|m|$ as its cardinality. The BF attempts to store each element m_j from the set $\mathcal{M}$ as follows.

$$\forall m_j \; in \; \mathcal{M}, \forall h_i \; in \; \mathcal{H}, \mathsf{BFStore}(\; A[h_i(m_j)]) = 1 \tag{4.1}$$

After i iterations, an element m_j from $\mathcal{M}$ will be hash-mapped into $k = |\mathcal{H}|$ locations in the array A[]. This unique combination of bits in $A[]$ represents the presence of that specific element. The state change of $A[h_i(m_j)] = 1$ from $A[h_i(m_j)] = 0$ only happens if $A[h_i(m_j)]$ is not accessed before by $\mathsf{BFStore()}$. BF anticipates collisions [5] in their hash function, and it is possible that an attempted bit in the Array A[] is set already.

4.2.3 Querying Elements in BF

An element $m_x \in \mathcal{M}$ can be queried for its presence if $\mathcal{M}$ is mapped in the BF array $A[]$ using a *Check* operation. The BF check takes three inputs - the BF array, the BF hash function set, and the element to be searched. $\mathsf{BFCheck}(A, \mathcal{H}, x)$. The query is executed as

$$\forall h_i \; in \; \mathcal{H}, \mathsf{BFCheck}(\; A[h_i(m_x)]) = 1 \tag{4.2}$$

The check returns positive if all the bits from the hash function mappings of the queried element are set to 1. The $\mathsf{BFCheck}(A, \mathcal{H}, x)$ will return zero if at least one of the bits from the hash map is turned to be zero. $\mathsf{BFCheck}(A, \mathcal{H}, x) = 0$ indicates an absence of the searched element m_x in $\mathcal{M}$. However, $\mathsf{BFCheck}(A, \mathcal{H}, x) = 1$ does not definitively indicate the presence of m_x in $\mathcal{M}$. Instead, it indicates a high

probability of the presence of m_x. This probabilistic nature is due to collisions, an inherent behavior of hash functions.

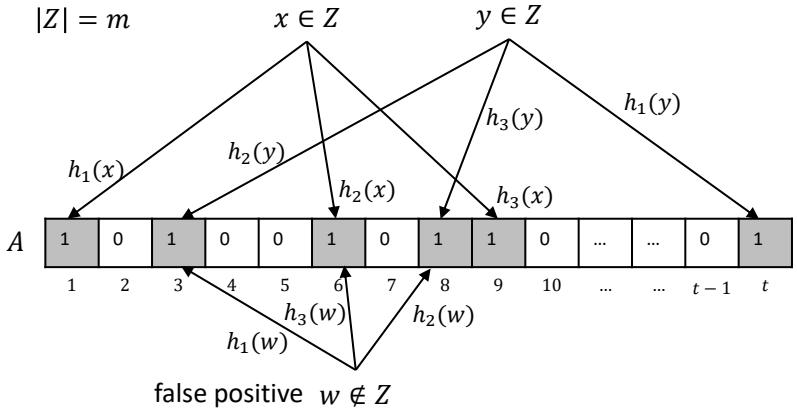

Figure 4.1: An example of BF with $\mathcal{H} = 3$, where $m_x, m_y \in \mathcal{M}$, $m_w \notin \mathcal{M}$, but $\mathsf{BFCheck}(A, \mathcal{H}, w) = 1$.

4.2.4 Controlling the False Positiveness of a BF

The image Fig. 4.1 depicts the probable nature of the BF. The BF query will never result in false negatives but can yield false positives. In our method, this means that there is a chance an IoT device may assume a wrong group key and become isolated from subsequent communications. It's important to address false positives in a BF, which can be mitigated by adjusting the parameters used in its construction. If the element set $\mathcal{M}$ with cardinality m is mapped to a BF array A[] of $t - bits$ long where mappings are uniform and equal probability is assumed while querying, the probability of a query yielding a false positive (fp) can be calculated with following equation,

$fp = (1 - (1 - 1/t)^m)^k \approx (1 - e^{-km/t})^k$.

where k is the number of hash functions,t is the length of BF array, and m is the number of elements that can be stored in the BF. When $k = \ln 2 \cdot \frac{t}{m}$, the false positive can be reduced as $(1 - e^{-km/t})^k = (\frac{1}{2})^k$.

4.3 Polynomial Functions

Our design uses a polynomial function for the secure communication of the encryption keys newly generated by the Gateway Server. The new group key embedded in the sent polynomial coefficients can be resolved only by the intended recipients, who can reconstruct the factored form of the polynomial.

4.3.1 Polynomial-based Access Control Technique

Let p be a large prime with bit length $|p| = \kappa$, e.g., $\kappa = 128$, and $\mathcal{Z} = \{z_1, z_2, \cdots, z_m\}$ be an integer set of size $|\mathcal{Z}| = m$, where each element $z_i \in \mathcal{Z}$ belongs to $\mathbf{Z}_p^*$. The Polynomial based Access Control (PAC) technique enables anyone who holds at least one element in $\mathcal{Z}$ to access a new secret key $s \in \mathbf{Z}_p^*$; while for someone who does not hold any element in $\mathcal{Z}$ cannot access the new secret key s. Concretely, the PAC technique is described as follows.

Polynomial Generation. Given the set $\mathcal{Z} = \{z_1, z_2, \cdots, z_m\}$ and a new secret key s, a polynomial is generated as follows:

$$f(x) = \prod_{z_i \in \mathcal{Z}} (x - z_i) + s \bmod p = \sum_{j=0}^{m} c_j \cdot x^j \bmod p \tag{4.3}$$

where all coefficient c_j, for $j = 0, 1, \cdots, m$, belong to $\mathbf{Z}_p^*$.

Secret Key Access. Given the coefficients (c_m, c_{m-1}, $\cdots$, c_1, c_0) of the polynomial $f(x)$, if someone really has an element $z_\alpha \in \mathcal{Z}$, he can recover the secret key s by computing

$$f(z_\alpha) = \sum_{j=0}^{m} c_j \cdot {z_\alpha}^j \bmod p = s \tag{4.4}$$

From the equation 4.3, it's obvious that if someone does not hold any element in $\mathcal{Z}$, he can obtain the secret key s only with a negligible probability. In this thesis, we will demonstrate how to use the PAC to securely distribute new group keys.

Chapter 5

The Proposed Scheme

In this chapter, we depict the details of our scheme that builds a secure and efficient dynamic key management scheme. The theme of our scheme is a balanced binary tree of secure AES keys with devices linked to the leaves of the tree, indicating ownership of keys belonging to that branch of the tree. This theme is built based on the Logical Key Hierarchy design proposed by Wallner et al. [47] and formalized by Harney et al. [17]. Our scheme builds the binary tree organization of keys during the system's initialization phase.

5.1 System Initialization

The fully trusted entity *Gateway Server* $\mathcal{S}$ initiates the system by generating secure keys depending on the maximum number of devices it needs to manage keys. For a set of devices $\mathcal{D} = \{D_1, D_2, \cdots, D_n\}$ with maximum cardinality $|\mathcal{D}| = N$, the Server $\mathcal{S}$ generates a set of $2N-1$ AES keys, denoted as $\{\mathcal{K} = (k[1], k[2], \cdots, k[2N-1])\}$. Each secure key k_i is a random binary number in $\mathbf{Z}_p^*$, where p is a large prime number the server generates and stores privately. The prime number p is of a fixed length as decided by the server $\mathcal{S}$. In our proposal, we consider the length of p as $128 - bits$ when represented in binary format. During the initialization phase, $\mathcal{S}$ unicasts a set

of keys to each device D_i through a secure channel. The number of keys distributed to each device D_i is $log(N)$, equivalent to the height of the LKH tree.

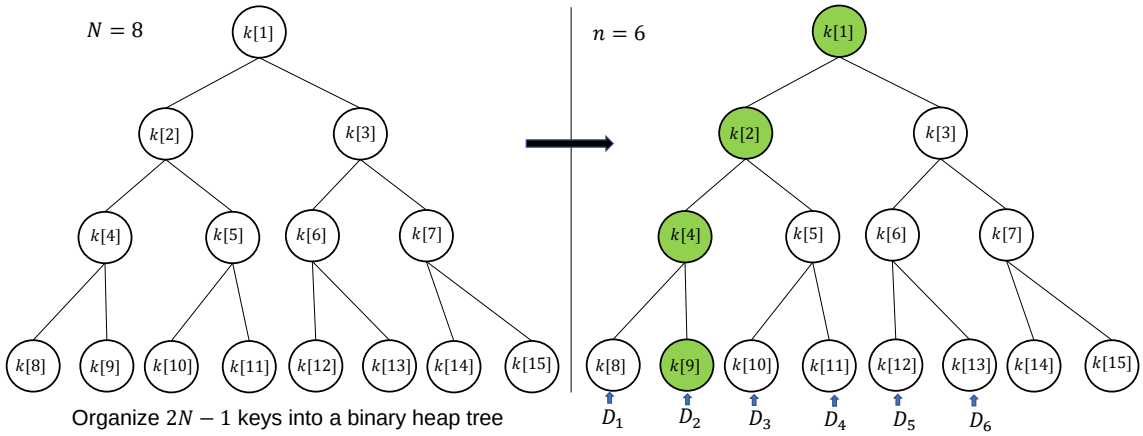

Figure 5.1: A sample key tree with $N = 8$ and $n = 6$. The device D_2 will be assigned the key set $(k[9], k[4], k[2], k[1])$.

The $\mathcal{S}$ considers the devices associated with the LKH tree logically at its leaves and has access to the specific secure key at the corresponding leaf position and the branch of the tree leading to the leaf from the root. For a given device D_i, the corresponding leaf key is $k[N-1+i]$. The set of keys in D_i's possession as decided by the server will be $(k[N-1+i], k[\lfloor (N-1+i)/2 \rfloor], k[\lfloor (N-1+i)/4 \rfloor], \cdots, k[1] = k[\lfloor (N-1+i)/2^w \rfloor])$ when keys are indexed from 1 starting from the root of the tree. For example, in Fig. 5.1, where $N = 8$, $D_i = D_2$ will be assigned a set of keys $(k[N-1+i] = k[9], k[\lfloor (N-1+i)/2 \rfloor] = k[4], k[\lfloor (N-1+i)/4 \rfloor] = k[2], k[\lfloor (N-1+i)/8 \rfloor] = k[1])$. The key $k[1]$ is the key at the root of the LKH and is shared among all the devices in the group, hence known as the *Group Key*. Any broadcast information can be sent by S by encrypting it with $k[1]$. The key $k[N-1+i]$ is shared only to a specific device D_i. This key is used for communicating secretly using unicast messages between the gateway server S and the device D_i. The intermediary keys between the group key and leaf keys are not used for any sub-group communication purposes. They are only used for deriving new keys during group key updates.

The security goals of forward security and backward security are achieved by updating the group key each time an IoT device leaves the group or joins the group. The efficiency of our proposal lies in how the new group key is communicated to the IoT members of the communication group during different scenarios.

5.2 Group Key Update When a New Device Joins the Group

During the initialization phase, the server $\mathcal{S}$ allocates leaf keys for N devices, the maximum number of devices it can support. At first, the server initialized a communication group with several devices $|D| = n$, where $n < N$. In time, a new device D_{n+1} requests S to join the device group $\mathcal{D}$. This initiates a key replacement request, and the server $\mathcal{S}$ handles it the following way.

Assume there are n IoT devices in the group, where $n < N$, and no IoT device previously left the group. When a new IoT device, denoted as D_{n+1}, joins the group, the existing group key needs to be replaced. The following steps will be executed between the gateway server $\mathcal{S}$ and all IoT devices $\mathcal{D}$ for the key update.

Step 1: First step is the new key generation. $\mathcal{S}$ generates a new AES key with $128 - bit$ in length $k[1]_{new} \in \mathbf{Z}_p^*$. and replaces the copy of group key $k[1]$ in its possession. A copy of the old key $k[1]$ will be kept securely.

Step 2: $\mathcal{S}$ The generated new key $k[1]_{new}$ needs to be communicated to the devices in the communication group $\mathcal{D}$. This is achieved through the secure broadcasting of the key. The new key is broadcasted by encrypting it with the old key $k[1]$. i.e. Broadcast the ciphertext C as $C = \text{AES}_{\text{Encrypt}}(k[1], k[1]_{new})$.

Step 3: Every IoT device $D_i \in \mathcal{D}$ receives C and uses the $k[1]$ in their posession to decrypt C and retrieve the $k[1]_{new}$. i.e. $k[1]_{new} = \text{AES}_{\text{Decrypt}}(C, k[1])$. All devices replace their copies of the old group key $k[1]$ with the new key $k[1]_{new}$.

Step 4: $\mathcal{S}$ addresses the key requirements of the newly joined device D_{n+1} using secure channel unicast messaging. First, the new group key $k[1]_{new}$ is shared with the new device. Then $\mathcal{S}$ assigns the leaf key $k[N-1+(n+1)] = k[N+n]$ to the new device along with the keys from the keys of the branch of the key tree leading from root till the leaf $k[N+n]$. The keys assigned to D_{n+1} will be $(k[N+n], k[\lfloor (N+n)/2 \rfloor], k[\lfloor (N+n)/4 \rfloor], \cdots, k[1] = k[\lfloor (N-1+i)/2^w \rfloor])$. This use case assumes the leaves in the key tree before the $N+n$ position are occupied during the initialization. For example, in Fig. 5.2, D_7 will be assigned a set of keys $(k[N+n] = k[14], k[\lfloor (N+n)/2 \rfloor] = k[7], k[\lfloor (N+n)/4 \rfloor] = k[3], k[\lfloor (N+n)/8 \rfloor] = k[1])$. Note that $k[1]$ has already been the updated group key at this step.

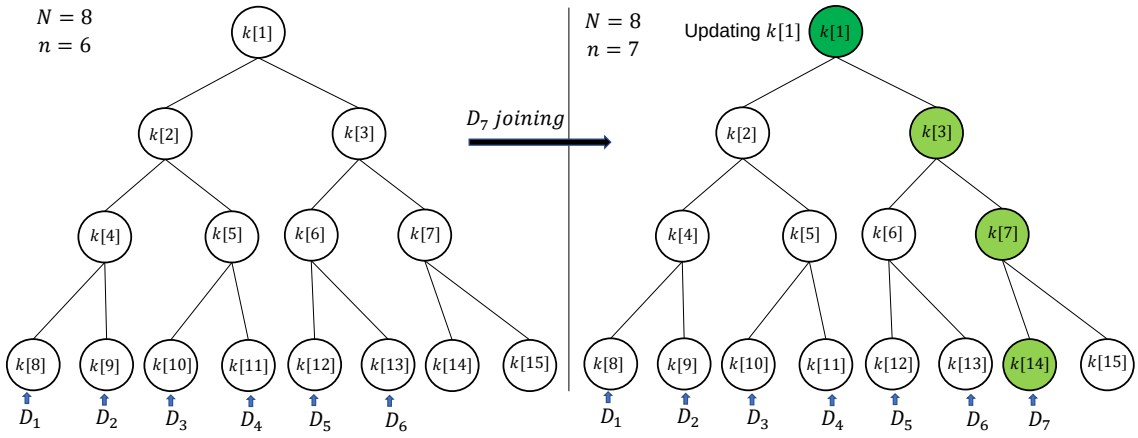

Figure 5.2: Example: The new member D_7 receives keys $(k[14], k[7], k[3], k[1])$.

Also note that, for some cases, before a new IoT device joins the group, if some IoT devices have already left the group, the gateway server S should assign the most left unoccupied leaf key, i.e., $k[N-1+i]$, to the newly joining IoT device, and denotes the newly joining IoT device as the device D_i. After that, following the same steps above, the group key update can be performed by $\mathcal{S}$ and $\mathcal{D}$. See Fig. 5.3 for an example, where a new device will be assigned with the most left unoccupied ID D_5. Then, $k[1]$ will be updated, and D_5 will be assigned with keys $(k[12], k[6], k[3], k[1])$.

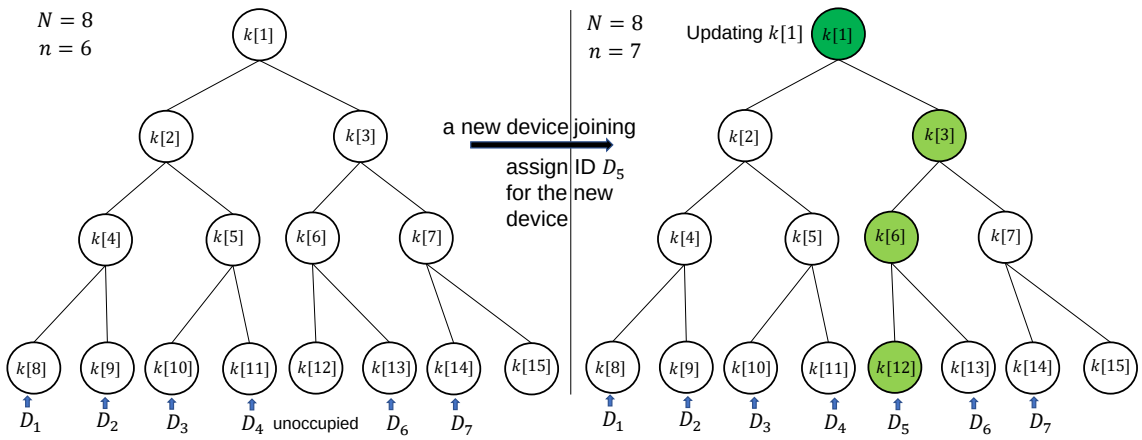

Figure 5.3: The new member D_5 receives keys $(k[12], k[6], k[3], k[1])$.

5.3 Group Key Update When a Device Leaves the Group

In a group of n devices, when a device D_i leaves the group, the above-mentioned broadcasting method is not a possibility. The leaving device also possesses the old group key $k[1]$ used for encrypting the new broadcasted group key $k[1]_{new}$. The following steps will detail how our protocol handles the situation, excluding D_i from future group communications.

Step 1: Expire keys owned by D_i. When $\mathcal{S}$ identifies that the device D_i no longer is a member of the group, $\mathcal{S}$ starts marking the keys present in the key tree and owned by D_i for expiry. This includes the leaf key $k[N-1+i]$ and the branch of keys leading to the key tree root. They are $k[\lfloor (N-1+i)/2 \rfloor], k[\lfloor (N-1+i)/4 \rfloor], \cdots, k[1])$. Figure Fig. 5.4 shows an example of device D_5 leaving the group.

Step 2: Replace expired keys with new keys. $\mathcal{S}$ needs to replace the set of keys that is expired due to D_5 leaving. $\mathcal{S}$ generates a new AES $128-bit$ key $k[1]_{new}$ to replace the group key $k[1]$ in the key tree. The remaining expired keys are now targeted for replacement. However, they are not replaced by generating new AES keys. Instead, the expired keys are XORed with the new group key $k[1]_{new}$

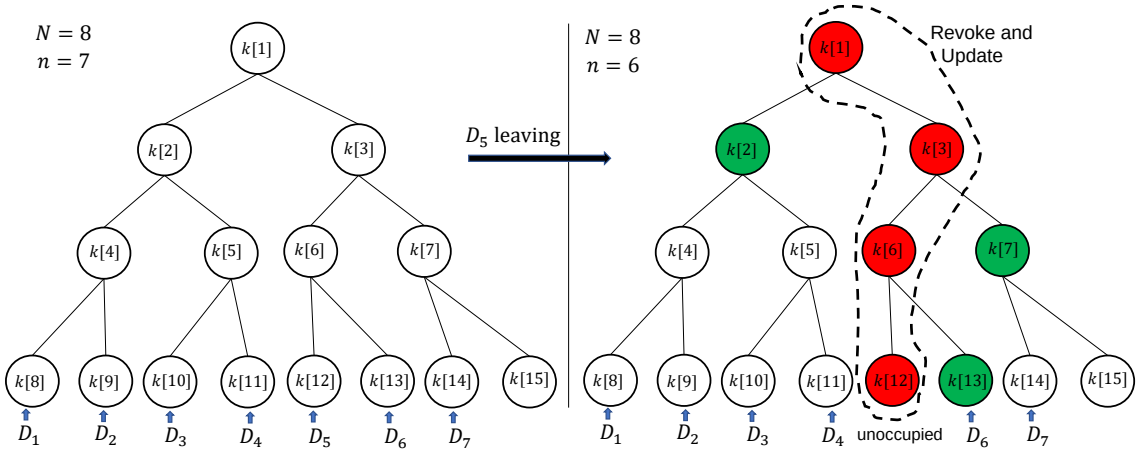

Figure 5.4: Example: The new member D_7 receives keys $(k[14], k[7], k[3], k[1])$.

to generate the new keys in those positions. The updated key set will look like $k[\lfloor (N-1+i)/2 \rfloor] \oplus k[1]_{new}, k[\lfloor (N-1+i)/4 \rfloor] \oplus k[1]_{new}, \cdots, k[1]_{new})$.

Step 3: Identify subgroup roots for communication. This step details the communication of the new group key to the updated group $\mathcal{D}|\{D_i\}$. Broadcast is insecure in this scenario as it cannot exclude D_i. The alternate option is unicasting the new group key to all other members in $\mathcal{D}$. However, this is highly inefficient and has $O(n)$ complexity. Hence, in our design, $\mathcal{S}$ takes an alternative approach by communicating the new group key to $\mathcal{D}|\{D_i\}$ by embedding it in a polynomial with the intermediate keys from the key tree. First, $\mathcal{S}$ starts identifying the root of the subgroups of the new group $\mathcal{D}|\{D_i\}$. This set is created by finding the sibling keys of the keys except the Group key previously owned by D_i. A sibling key is a key in a tree with the same parent and can be identified by dividing the key's position with a divisor as 2. The sibling key of k_i is either k_{i+1} or k_{i-1}. If divisions yield the same result (same parent key) for both keys, we can conclude that the keys are siblings. This logic is applied to the expired keys of D_i to identify the sibling set $K_{D_i} = (k[N-1+i], k[\lfloor (N-1+i)/2 \rfloor], k[\lfloor (N-1+i)/4 \rfloor], \cdots, k[1])$.

Step 4: Securing the sibling keys with cryptographic hash and timestamp. The keys used in the polynomial are first processed with the following logic. $\mathcal{S}$

intakes the sibling set K_{D_i} and generates a hash set $\mathcal{Z} = z_w, z_w - 1, \cdots, z_1$, where each element $z_i \in \mathcal{Z}$ is generated by hashing each element $k_{D_i} \in K_{D_i}$ using the timestamp ts.

$$z_i = H(k_{D_i}||ts) \tag{5.1}$$

Step 5: Generate the polynomial coefficients. The hash set is used by $\mathcal{S}$ to generate a polynomial of degree $|\mathcal{Z}|$ where each z_i will serve as a solution to the polynomial that will yield the new group key $k[1]$. It means that the generated polynomial can be resolved only by members of $\mathcal{D}$ who possess one of the $z_i \in \mathcal{Z}$. The polynomial created will be

$$f(x) = \prod_{i=1}^{w}(x - z_i) + k[1] \bmod p \tag{5.2}$$

The above polynomial can further be resolved into its coefficients as

$$f(x) = \sum_{j=0}^{w} c_j \cdot x^j \bmod p \tag{5.3}$$

The coefficients are intended for broadcasting to the devices in the group $\mathcal{D}$.

Step 6: Create the bloom filter. The $\mathcal{S}$ needs to build a verification mechanism for validating the success of polynomial resolution attempts. This is done by creating a Bloom Filter and storing the coefficients' presence. It executes the following algorithm.

$$BFStore(k[1]||ts) \tag{5.4}$$

$\mathcal{S}$ then broadcasts the BF along with the coefficients set and ts to all devices in the group $\mathcal{D}$.

Step 6: Resolving the broadcasted polynomial. Every $D_i \in \mathcal{D}$ will receive

the broadcast and will attempt to resolve the polynomial with all the keys they possess except the root key. Each device will make $w = log(n) - 1$ attempts to resolve the polynomial. After verifying with BF, the successful attempts will result in updating the specific key and every key below in the hierarchical order using the XOR operation.

When the broadcast is received by the Device D_i, it picks up the keys in its possession except $k[1]$ as $(k[N-1+i], k[\lfloor (N-1+i)/2 \rfloor], k[\lfloor \lfloor (N-1+i)/4 \rfloor], \cdots, k[\lfloor \lfloor (N-1+i)/2^w \rfloor])$. It then generates a set $\mathcal{Z}'$ by hashing each key against the ts received from the broadcast.

$$\begin{aligned} \mathcal{Z}' = (&H(k[N-1+i]||ts), \\ &H(k[\lfloor (N-1+i)/2 \rfloor]||ts), \\ &H(k[\lfloor \lfloor (N-1+i)/4 \rfloor]||ts), \\ \cdots, &H(k[\lfloor \lfloor (N-1+i)/2^w \rfloor]||ts)) \end{aligned} \tag{5.5}$$

The device D_i proceeds to find the new key by attempting to resolve the polynomial using elements in the $\mathcal{Z}'$.

D_i will generate a maximum of w probable keys. Whenever a probable key $k_{probable}[1]$ is created after resolving a polynomial, D_i will verify them with the Bloom Filter as follows.

$$BFCheck(k_{probable}[1]||ts) \tag{5.6}$$

If the bloom filter confirms the key is valid, D_i will replace its $k[1]$ with $k_{probable}[1]$. Then, it will hierarchically perform XOR operations to derive the new key. That is, $k_{new}[\lfloor (N-1+u)/2^w \rfloor] = k[\lfloor (N-1+u)/2^w \rfloor] \oplus k[\lfloor (N-1+u)/2^{w-1} \rfloor]$.

If the Bloom filter returns 0 for every member in $\mathcal{Z}'$, the key change does not target the device D_i. Hence, it will continue to use its existing keys.

Chapter 6

Security Analysis

This chapter analyzes the proposed scheme against the defined security goals and will explain how the scheme achieves backward security, forward security, and key independence.

6.1 Backward Security

The goal of backward security is protecting communication history from newly joining devices in a group. Let D_x be a device that joins the group $\mathcal{D}$ with a group key k_{tn} where t indicates the time when D_x joined. The new device may gain access to the message history storage location where messages are encrypted with multiple AES keys $k_{tn-1}, k_{tn-2}, \cdots$ where $tn-1, tn-2, \cdots$ indicates the timepoints before tn; i.e., before the joining of D_x. However, it cannot decrypt the encrypted message history as it does not possess the encryption key used for encrypting the messages. It is impossible to derive $k_{tn-1}, k_{tn-2}, \cdots$ from k_{tn} due to the randomness property of AES key generation. It is possible for D_x to eavesdrop the broadcast ciphertext and $C = \text{AES}(k_{tn-1}, k_{tn})$. D_x also gains k_{tn} legitimately. However, one $plaintext, ciphertext$ pair is insufficient to launch a cryptanalysis using a known plaintext attack model [41] against the secure AES. Thus, we can conclude that this

scheme ensures backward compatibility.

6.2 Forward Security

Forward security assures the confidentiality in a group from a device D_x, which is no longer part of the group for the communications that occurred after D_x has left the group. This is made possible by expiring the list of keys D_x had access to$((k[N-1+x], k[\lfloor (N-1+x)/2 \rfloor], k[\lfloor (N-1+x)/4 \rfloor], \cdots, k[1]))$ when it was the member of the group $\mathcal{D}$. The new key is communicated using a polynomial $f(x) = \prod_{j=1}^{w}(x - z_i) + k[1] \bmod p$, built from the sibling keys of the D_x key list except for the root key. However, D_x can only guess its sibling keys with a negligible probability greater than $\frac{\lg N}{2^{128}}$, which is equivalent to a random choice. Hence, we can conclude this scheme assures forward security.

6.3 Key Independence

Our scheme achieves the key independence property by ensuring a secured key cannot be compromised even when multiple related keys are known. The key independence can be examined at the different stages of key handling. During the key generation, the AES key generator generates keys randomly. For a given set of generated group keys $\mathcal{K} = \{k_1, k_2, \cdots, k_{t_1}, k_t\}$, the revelation of any random set of keys ranging from $1 \ldots k_n - 1$ also cannot compromise the security of k_t because of the randomness in key generation. During the new group key distribution, our scheme uses hashing, concatenation, and modular arithmetic operations on individual keys when the polynomial is formed. Our scheme constructs the polynomial as $f(x) = \prod_{i=1}^{w}(x - z_i) + k[1] \bmod p$, where each $z_i = H(k_i||ts)$. The keys are not directly used. This ensures that the individual keys used in constructing the polynomial stay irrecoverable for an unintended recipient. This is an advantage in

our scheme compared to Piao et al.'s scheme [37], where the polynomial is constructed with individual keys. In the Piao's scheme, the polynomial is constructed $P = \prod_{i=1}^{n}(x - k_i) + K_t$, where k_t is the group key and $k_i = k_1, k_2, \cdots, k_n$ represents the individual device keys. It is easy to recover k_n from this polynomial when the keys $\{k_1, k_2, \cdots, k_{n-1}\}$ are known.

With the highlighted security analysis, it is evident that the security model in our proposed scheme protects encryption keys from revealing each other, thereby achieving key independence. It also achieves forward and backward security by ensuring the time-bound access of private information based on group membership duration.

Chapter 7

Performance Evaluation

7.1 Performance Comparison

In this section, we compare the performance of our proposed scheme in terms of re-keying costs and storage overhead. Specifically, we compare our proposed scheme with LKH [49], [17], and Piao et al.'s scheme [37], as our proposed scheme is closely related to them. We consider $2^{w-1} < n < N = 2^w$, i.e., $\lceil \log_2 n \rceil = \log_2 N = w$, the large prime p and the timestamp ts are 128 bits, and the bloom filter size is also 128 bits in our proposed scheme, which can achieve a much lower false positive rate. For the fairness of comparison, we consider all key sizes in three schemes to be 128 bits. Table 7.2 shows the comparison of the number of re-keying messages and the size of each message after joining/leaving for the three schemes. In terms of the number of re-keying messages, both Piao et al.'s [37] scheme and our scheme are more efficient than LKH [49], [17], as the number of messages after joining (multicast and unicast) and leaving is all 1. In addition, in terms of the size of the message, our scheme is better than Piao et al.'s scheme. For storage overhead, we compare the three schemes in Table 7.1. From the table, we can see that our scheme has a little higher storage cost at the gateway server. Since the gateway server is powerful in both

storage and computation, the storage cost is not a big issue. On the IoT device side, our scheme has the same overhead as LKH. Although Piao et al.'s scheme has less overhead, the communication costs for re-keying, as we analyzed above, are much higher than LKH and our scheme.

Table 7.1: Comparison of the storage overhead of the three schemes

Schemes	Gateway server	Each IoT device
LKH	$128 * (2n - 1)$	$128 * (\log_2 n + 1)$
Piao et al.'s scheme	$128 * (n + 1)$	$128 * 2$
Our proposed scheme	$128 * (2N - 1)$	$128 * (\log_2 N + 1)$

Table 7.2: Comparison of the number of re-keying messages and the size of each message after join/leave

Schemes	The number of re-keying messages and the size (in bit) of each message					
	Join				Leave	
	Multicast		Unicast			
	number	size of each message	number	size of each message	number	size of each message
LKH [49, 17]	$\log_2 n - 1$	128	$\log_2 n + 1$	128	$2 \log_2 n$	128
Piao et al.'s scheme [37]	1	$128 * (n + 2)$	1	128	1	$128 * n$
Our proposed scheme	1	128	1	$128 * (\log_2 N + 1)$	1	$128 * (\log_2 N + 3)$

7.2 Execution Results

Here, we present the readings and results from the tests conducted on the program built based on the proposed scheme. The tests run in a laptop computer powered by a 64bit Intel(R) Core(TM) i7-8650U CPU @ 1.90GHz 2.11 GHz processor running the Windows 11 Pro operating system. The PC has 8GB RAM. The program is developed in Java using Visual Studio Code editor and is designed to run on PCs with lower performance and configurations.

The program is compiled in Visual Studio Code using Java plugins. The source

code uses only basic data structures and can be rewritten to other object-oriented languages running in different environments without major structural changes. The source code is provided in Appendix A of this document. The program's *Main()* method invokes two methods.

• **AddDevice()** This method adds a new device to the communication group and simultaneously broadcasts the new group key encrypted with the previous group key. In our test application, up to 8 devices can be added. Each device has a specific slot(position) allocated when the device is added to the group, starting from the lowest available position.

• **RemoveDevice()** This method broadcasts the new group key using polynomial coefficients and modular arithmetic while removing the requested device from the group. The memory consumption for *RemoveDevice()* method was assessed by removing the devices from the last to the first.

The program is run on automated mode for 100 cycles with a 1ms sleep timer between the method calls. The observations of the executions were summarized by calculating the average and are presented below.

7.2.1 CPU Usage

The CPU usage for the *AddDevice()* and *RemoveDevice()* operations were measured in microseconds. The results for CPU consumption were very consistent. The *AddDevice()* operation consumed between $200 - 400\mu s$, depending upon the devices already present in the group. Except for the first device, the incremental time taken for adding a new device in the group was in the range of $40 - 50\mu s$.

The graph in figure 7.1 shows increasing time due to the simulated IoT objects in a single PC. In the use case scenario with real IoT devices, the computation of the new key by decryption will be distributed among different processors. Hence, the graph will be flatter and less inclined.

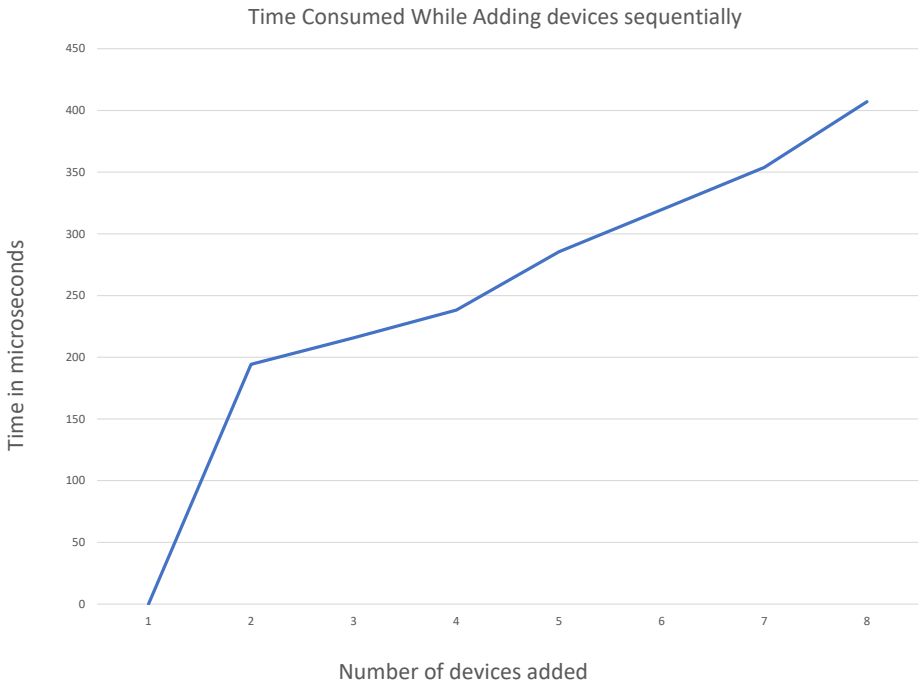

Figure 7.1: Time consumption while devices join the group.

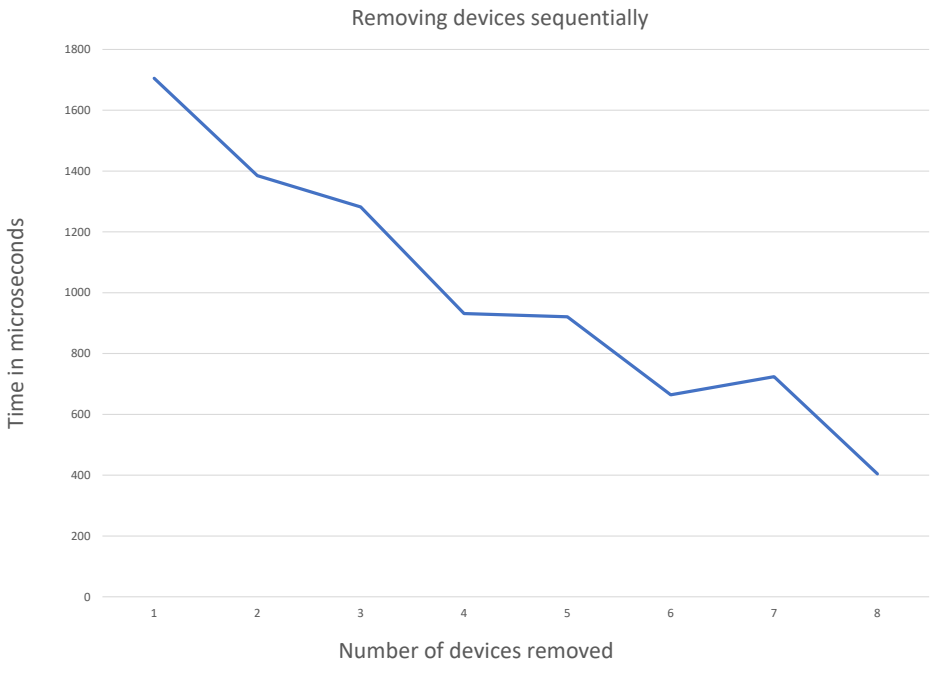

Figure 7.2: Time consumption while devices leave the group.

The highest CPU usage during the removal of the device was $1700\mu s$, and the lowest was $400\mu s$. As observed in figure 7.2, the time consumed by *RemoveDevice()* is directly proportional to the remaining devices—the lesser the number of devices, the lesser the polynomial resolutions per device. *RemoveDevice()* consumes more time than the add operation due to the same reasons stated in the memory assessment. This calculation will also be distributed among the individual devices in a real scenario, averaging the calculation cost per device as $400\mu s$.

Chapter 8

Conclusions and Future Works

8.1 Conclusion

In this thesis, we have proposed an efficient dynamic key management scheme for IoT systems. Our proposed scheme is characterized by employing a binary heap tree, bloom filter, and polynomial-based technique to achieve secure and efficient dynamic key management. Security analysis shows that our scheme can achieve desirable security requirements, i.e., forward security, backward security, and key independence. Since our scheme relies on AES encryption alone and does not use public key cryptography, it is secure from many associated vulnerabilities [**?**]. In addition, a detailed performance evaluation also indicates our scheme is more efficient than the referenced works. Our implementation demonstrated the scheme's efficiency. One of the limitations of our scheme is the increasing complexity of the number of devices that can be used. We hope to address this limitation in our future works.

8.2 Future Work

Our scheme proposes the key replacement when group members change. There are other reasons to replace the group key, such as a long session—our scheme in the future aims to address scenarios other than group membership change that require key replacements. The performance evaluation indicated the need for optimizing the proportionally large memory consumption on key replacement during the removal of devices. Though the measures are IoT-friendly, the model's scalability will not be as efficient as it is. This is something we would like to address in our future work. The degree of the polynomial is the primary reason that limits the scalability. One way to handle the scalability is by horizontally scaling the model using multiple key trees. In our future work, we also seek alternatives for trust-based dynamic key management. One of the few desirable traits for a key management algorithm would be a distributed approach instead of the proposed centralized method for key management. Our key management is pivoted around the trusted gateway server. A trust-less mechanism may better cater to real-time needs. Another alternative we would like to use is an alternative encryption mechanism. The proposed method ensures the use of 128-bit AES encryption. While AES is efficient, other encryption schemes may also fit into the security and privacy requirements and are worth exploring. How the current implementation performs against larger AES keys is also worth looking into. The plan to approach future work is best done after receiving the measurement of this scheme implemented on a real scenario, including actual IoT hardware devices.

Bibliography

[1] *Java Program to Implement Bloom Filter - Sanfoundry — sanfoundry.com*, [Accessed 09-08-2023].

[2] Ashwag Albakri, Mahesh Maddumala, and Lein Harn, *Hierarchical polynomial-based key management scheme in fog computing*, (2018), 1593–1597.

[3] Burton H. Bloom, *Space/time trade-offs in hash coding with allowable errors*, **13** (1970), no. 7.

[4] Andrey Bogdanov, Lars R Knudsen, Gregor Leander, Christof Paar, Axel Poschmann, Matthew JB Robshaw, Yannick Seurin, and Charlotte Vikkelsoe, *Present: An ultra-lightweight block cipher*, (2007), 450–466.

[5] Andrei Broder and Michael Mitzenmacher, *Network Applications of Bloom Filters: A Survey*, Internet Mathematics **1** (2003), no. 4, 485 – 509.

[6] Ramaswamy Chandramouli, Michaela Iorga, and Santosh Chokhani, *Cryptographic key management issues and challenges in cloud services*, (2014).

[7] Tzu-Chiang Chiang and Yueh-Min Huang, *Group keys and the multicast security in ad hoc networks*, (2003), 385–390.

[8] Don Coppersmith, *The data encryption standard (des) and its strength against attacks*, IBM journal of research and development **38** (1994), no. 3, 243–250.

[9] Joan Daemen and Vincent Rijmen, *Aes proposal: Rijndael*, (1999).

[10] Maissa Dammak, Sidi-Mohammed Senouci, Mohamed Ayoub Messous, Mohamed Houcine Elhdhili, and Christophe Gransart, *Decentralized lightweight group key management for dynamic access control in iot environments*, IEEE Transactions on Network and Service Management **17** (2020), no. 3, 1742–1757.

[11] Subir Das, Yoshihiro Ohba, Mitsuru Kanda, David Famolari, and Sajal K Das, *A key management framework for ami networks in smart grid*, IEEE Communications Magazine **50** (2012), no. 8, 30–37.

[12] Dorothy E Denning and Peter J Denning, *Data security*, ACM computing surveys (CSUR) **11** (1979), no. 3, 227–249.

[13] Jyoti Deogirikar and Amarsinh Vidhate, *Security attacks in iot: A survey*, (2017), 32–37.

[14] Mohamed Elhoseny, Navod Neranjan Thilakarathne, Mohammed I Alghamdi, Rakesh Kumar Mahendran, Akber Abid Gardezi, Hesiri Weerasinghe, and Anuradhi Welhenge, *Security and privacy issues in medical internet of things: overview, countermeasures, challenges and future directions*, Sustainability **13** (2021), no. 21, 11645.

[15] M. Eltoweissy, M. Moharrum, and R. Mukkamala, *Dynamic key management in sensor networks*, IEEE Communications Magazine **44** (2006), no. 4, 122–130.

[16] Sheik Mohammad Mostakim Fattah, Nak-Myoung Sung, Il-Yeup Ahn, Minwoo Ryu, and Jaeseok Yun, *Building iot services for aging in place using standard-based iot platforms and heterogeneous iot products*, Sensors **17** (2017), no. 10, 2311.

[17] H. HARNEY, *Logical key hierarchy protocol*, IETF Internet Draft (1999).

[18] Hugh Harney and Carl Muckenhirn, *Rfc2094: Group key management protocol (gkmp) architecture*, (1997).

[19] Wan Haslina Hassan et al., *Current research on internet of things (iot) security: A survey*, Computer networks **148** (2019), 283–294.

[20] Xiaobing He, Michael Niedermeier, and Hermann De Meer, *Dynamic key management in wireless sensor networks: A survey*, Journal of network and computer applications **36** (2013), no. 2, 611–622.

[21] Chung-Wen Hung and Wen-Ting Hsu, *Power consumption and calculation requirement analysis of aes for wsn iot*, Sensors **18** (2018), no. 6, 1675.

[22] Qiaoyan Kang, Xiangru Meng, and Jianfeng Wang, *An optimized lkh scheme based on one-way hash function for secure group communications*, (2006), 1–4.

[23] Masanobu Katagi, Shiho Moriai, et al., *Lightweight cryptography for the internet of things*, sony corporation **2008** (2008), 7–10.

[24] Sunny King and Scott Nadal, *Ppcoin: Peer-to-peer crypto-currency with proof-of-stake*, self-published paper, August **19** (2012), no. 1.

[25] Fabian Kuhn and Rotem Oshman, *Dynamic networks: models and algorithms*, ACM SIGACT News **42** (2011), no. 1, 82–96.

[26] Loïc Lannelongue, Jason Grealey, and Michael Inouye, *Green algorithms: quantifying the carbon footprint of computation*, Advanced science **8** (2021), no. 12, 2100707.

[27] Bin Liu, Xiao Liang Yu, Shiping Chen, Xiwei Xu, and Liming Zhu, *Blockchain based data integrity service framework for iot data*, (2017), 468–475.

[28] Kelvin Ly and Yier Jin, *Security challenges in cps and iot: From end-node to the system*, (2016), 63–68.

[29] Mehedi Masud, Gurjot Singh Gaba, Karanjeet Choudhary, M. Shamim Hossain, Mohammed F. Alhamid, and Ghulam Muhammad, *Lightweight and anonymity-preserving user authentication scheme for iot-based healthcare*, IEEE Internet of Things Journal **9** (2022), no. 4, 2649–2656.

[30] Sadiya Mirza and Sana Zeba Bakshi, *Introduction to manet*, International research journal of engineering and technology **5** (2018), no. 1, 17–20.

[31] Sasa Mrdovic and Branislava Perunicic, *Kerckhoffs' principle for intrusion detection*, **Supplement** (2008), 1–8.

[32] Pedro Sanchez Munoz, Nam Tran, Brandon Craig, Behnam Dezfouli, and Yuhong Liu, *Analyzing the resource utilization of aes encryption on iot devices*, (2018), 1200–1207.

[33] Elnaz Namazi, Jingyue Li, and Chaoru Lu, *Intelligent intersection management systems considering autonomous vehicles: A systematic literature review*, IEEE Access **7** (2019), 91946–91965.

[34] Sean W O'Malley and Larry L Peterson, *A dynamic network architecture*, ACM Transactions on Computer Systems (TOCS) **10** (1992), no. 2, 110–143.

[35] Suryakanta Panda, Samrat Mondal, Rinku Dewri, and Ashok Kumar Das, *Towards achieving efficient access control of medical data with both forward and backward secrecy*, Comput. Commun. **189** (2022), no. C, 36–52.

[36] Yusuf Perwej, Kashiful Haq, Firoj Parwej, M Mumdouh, and Mohamed Hassan, *The internet of things (iot) and its application domains*, International Journal of Computer Applications **975** (2019), no. 8887, 182.

[37] Yanji Piao, Jonguk Kim, Usman Tariq, and Manpyo Hong, *Polynomial-based key management for secure intra-group and inter-group communication*, Comput. Math. Appl. **65** (2013), no. 9, 1300–1309.

[38] Warren B Powell, Patrick Jaillet, and Amedeo Odoni, *Stochastic and dynamic networks and routing*, Handbooks in operations research and management science **8** (1995), 141–295.

[39] B. R. Purushothama and B. B. Amberker, *Secure group and multi-layer group communication schemes based on polynomial interpolation*, Secur. Commun. Networks **6** (2013), 735–756.

[40] Ronald Rivest, *The md5 message-digest algorithm*, (1992).

[41] D Rossi, M Omana, C Metra, and ML Valarmathi, *Cryptanalysis of simplified-aes encrypted communication*, International Journal of Computer Science and Information Security (IJCSIS) **13** (2015), no. 10.

[42] Kazhan Othman Mohammed Salih, Tarik A Rashid, Dalibor Radovanovic, and Nebojsa Bacanin, *A comprehensive survey on the internet of things with the industrial marketplace*, Sensors **22** (2022), no. 3, 730.

[43] Manasha Saqib, Bhat Jasra, and Ayaz Hassan Moon, *A lightweight three factor authentication framework for iot based critical applications*, Journal of King Saud University - Computer and Information Sciences **34** (2022), no. 9, 6925–6937.

[44] A.T. Sherman and D.A. McGrew, *Key establishment in large dynamic groups using one-way function trees*, IEEE Transactions on Software Engineering **29** (2003), no. 5, 444–458.

[45] Hallam Stevens, *Hans peter luhn and the birth of the hashing algorithm*, IEEE spectrum **55** (2018), no. 2, 44–49.

[46] Ramao Tiago Tiburski, Leonardo Albernaz Amaral, Everton de Matos, Dario F. G. de Azevedo, and Fabiano Hessel, *Evaluating the use of tls and dtls protocols in iot middleware systems applied to e-health*, (2017), 480–485.

[47] Debby Wallner, Eric Harder, and Ryan Agee, *Key management for multicast: Issues and architectures*, (1999).

[48] Moritz Wendl, My Hanh Doan, and Remmer Sassen, *The environmental impact of cryptocurrencies using proof of work and proof of stake consensus algorithms: A systematic review*, Journal of Environmental Management **326** (2023), 116530.

[49] Chung Kei Wong, Mohamed G. Gouda, and Simon S. Lam, *Secure group communications using key graphs*, (1998), 68–79.

[50] ———, *Secure group communications using key graphs*, IEEE/ACM Trans. Netw. **8** (2000), no. 1, 16–30.

[51] Sen Xu, Manton Matthews, and Chin-Tser Huang, *Security issues in privacy and key management protocols of ieee 802.16*, Proceedings of the 44th Annual Southeast Regional Conference (New York, NY, USA), ACM-SE 44, Association for Computing Machinery, 2006, p. 113–118.

Appendix A

Code

This part of the thesis presents the implementation source code of the proposed scheme. This program is compiled in VS Code using Java plugins. The code is modularized into the following files.

A.1 AES.Java

Contains the module responsible for creating, encrypting, decrypting and XOR of AES Keys. It ensures that no keys are beyond 16 bytes of length during any of the operations and truncates the last byte if any key becomes more than 16 bytes. This code uses Java APIs related to AES.

```
import java.math.BigInteger;
import java.security.NoSuchAlgorithmException;
import java.security.SecureRandom;
import java.util.Arrays;
import java.util.Base64;
import java.util.Random;
import javax.crypto.Cipher;
import javax.crypto.KeyGenerator;
import javax.crypto.SecretKey;
import javax.crypto.spec.IvParameterSpec;
import javax.crypto.spec.SecretKeySpec;

public class AES
{
```

```
String strArray;
IvParameterSpec ivp;
Random random;
int keyLength;
final static int AESbytelength=16;
final static int offset =0;
boolean foundkey;

byte [] keyinBytes;
public AES(BigInteger p, int keyLength)
{
byte[] iv = new byte[AESbytelength];
new SecureRandom().nextBytes(iv);
ivp = new IvParameterSpec(iv);
this.keyLength = keyLength;
}

public SecretKey AESKeyGen
(int kz) throws  NoSuchAlgorithmException,
   IllegalArgumentException
{
try
{  //sometimes, AES generate 17bytes key and it crashes during
//decryption. this try-catch block avoids the AESEncryption
//AESDecryption crashing for 17byte keys
SecretKey key, truncatedkey;
KeyGenerator keygen;
keygen = KeyGenerator.getInstance("AES");
keygen.init(keyLength);
key = keygen.generateKey();
keyinBytes=key.getEncoded();
BigInteger bkey = new BigInteger(keyinBytes);
bkey = bkey.mod(Gateway.p);
byte[] newkeyinBytes= new byte[16];
Arrays.fill( keyinBytes, (byte) 1 );
newkeyinBytes = Arrays.copyOf(keyinBytes,keyinBytes.length
    <newkeyinBytes.length? keyinBytes.length: newkeyinBytes.
       length);
truncatedkey =new SecretKeySpec(newkeyinBytes,offset,
                        AESbytelength, "AES");
if(truncatedkey.getEncoded().length>16)
    System.out.printf("Warning:␣Long␣key␣after␣truncating");
return truncatedkey;
}
catch (Exception e)
{
System.out.println("Exception␣:␣offset␣values␣="+","+offset+"
   ,\n"
```

```
                                +AESbytelength+","+keyinBytes.
                                   length);
KeyGenerator keygen2= KeyGenerator.getInstance("AES" );
return  keygen2.generateKey();
}
}

public static String AESencryptECB
 ( SecretKey content, SecretKey key )  throws Exception
{
byte [] plaintext = content.getEncoded();
Cipher cipher = Cipher.getInstance( "AES/ECB/PKCS5Padding" );
cipher.init (Cipher.ENCRYPT_MODE,  key );
byte[] cipherText = cipher.doFinal( plaintext );
return Base64.getEncoder().encodeToString( cipherText );
}

public static SecretKey AESDecryptECB
( String ciphertext, SecretKey key )  throws Exception
{
Cipher cipher = Cipher.getInstance( "AES/ECB/PKCS5Padding" );
cipher.init (Cipher.DECRYPT_MODE,  key );
byte[] plaintext = cipher.doFinal(Base64.getDecoder().
                                        decode(ciphertext));
SecretKey decryptedkey = new SecretKeySpec(plaintext, "AES");
if(decryptedkey.getEncoded().length>16)
System.out.printf("Warning:␣Long␣key␣after␣AESDecryptECB");
return decryptedkey;
}

static SecretKey XORAESKeys
(SecretKey key1, SecretKey key2)  throws
   NoSuchAlgorithmException
{
SecretKey newkey;
byte [] b1 = key1.getEncoded();
byte [] b2 = key2.getEncoded();
int maxlength =b1.length>b2.length? b1.length: b2.length;
byte [] b3 = new byte[maxlength];
if(b1.length < b2.length)
{
for (int i=0; i<b1.length ;i++)
    b2[i] = (byte) (b1[i]^b2[i]);
b3 = b2;
}
else
{
for (int i=0; i<b2.length ;i++)
```

```
    b1[i] = (byte) (b1[i]^b2[i]);
b3 = b1;
}

BigInteger bkey = new BigInteger(b3);
bkey = bkey.mod(Gateway.p);
newkey = new SecretKeySpec(bkey.toByteArray(), "AES");
if(newkey.getEncoded().length>16)
{
System.out.printf
("Warning:Length␣XORAESKeys␣=␣%d%n", newkey.getEncoded().
   length);
newkey= truncateKey(newkey.getEncoded());
System.out.printf("Truncated␣length␣%d%n",
                            newkey.getEncoded().length);
}
return newkey;
}
static SecretKey truncateKey(byte[] key)
{
byte[] truncatedkey= new byte[AESbytelength];
System.arraycopy(key, 0, truncatedkey, 0,
                                          AESbytelength-1);
SecretKey newkey =new SecretKeySpec(truncatedkey,0,
                            AESbytelength, "AES");
if(newkey.getEncoded().length>16)
System.out.printf("Warning:␣Long␣key␣after␣truncateKey");

return newkey;
}
}
```

A.2 BloomFilter.java

This file contains reused publicly available source code [1]. The bloom filter size can be adjusted for efficiency, reducing false positive errors.

```
/*
Java Program to Implement Bloom Filter.This
code is taken from https://www.sanfoundry.com
/java-program-implement-bloom-filter/
 */
import java.util.*;
import java.security.*;
```

```
import java.math.*;
import java.nio.*;

/* Class BloomFilter */
class BloomFilter
{
    private byte[] set;
    private int keySize, size;
    private MessageDigest md;

    /* Constructor */
    public BloomFilter()
    {
        //setSize = capacity;
        set = new byte[524288]; //testcode: = new byte[4096];
        keySize = 128;
        size = 0;
        ResetBloomFilter();

    }
    /* Function to clear bloom set */
    public final void ResetBloomFilter()
    {
        Arrays.fill(set, (byte)0);
        size = 0;
        try
        {
            md = MessageDigest.getInstance("MD5");
        }
        catch (NoSuchAlgorithmException e)
        {
            throw new IllegalArgumentException("Error␣:"+
                    "MD5␣Hash␣not␣found");
        }
    }

    /* Function to get size of objects added */
    public int getSize()
    {
        return size;
    }
    /* Function to get hash - MD5 */
    private int getHash(int i)
    {
        md.reset();
        byte[] bytes = ByteBuffer.allocate(4)
                                        .putInt(i).array();
        md.update(bytes, 0, bytes.length);
```

```
        return Math.abs(new BigInteger(1, md.digest())
                        .intValue()) % (set.length - 1);
    }
    /* Function to add an object */
    public void add(Object obj)
    {
        ResetBloomFilter();
        int[] tmpset = getSetArray(obj);
        for (int i : tmpset)
            set[i] = 1;
        size++;
    }
    /* Function to check is an object is present */
    public boolean contains(Object obj)
    {
        int[] tmpset = getSetArray(obj);
        for (int i : tmpset)
            if (set[i] != 1)
                return false;
        return true;
    }
    /* Function to get set array for an object */
    private int[] getSetArray(Object obj)
    {
        int[] tmpset = new int[keySize];
        tmpset[0] = getHash(obj.hashCode());
        for (int i = 1; i < keySize; i++)
            tmpset[i] = (getHash(tmpset[i - 1]));
        return tmpset;
    }
}
```

A.3 Device.java

Handles data structure that stores the group of devices as a list. A maximum of each 8 devices can be contained in the device list. Each device contains a list 4 of keys from the Key Heap tree at the gateway server.

```
import java.math.BigInteger;
import java.security.MessageDigest;
import java.security.NoSuchAlgorithmException;
import javax.crypto.SecretKey;
import javax.crypto.spec.SecretKeySpec;
```

```
public class Device
{
SecretKey[] devicekeys;
MessageDigest md;
BigInteger p = Gateway.p;

public Device()
{
devicekeys = new SecretKey[3];
}

boolean ReplaceGroupKeyatdevice(String encryptedkey)
                        throws Exception
{
SecretKey newgroupKey = AES.AESDecryptECB(encryptedkey,
            (SecretKey) this.devicekeys[0]);
if (newgroupKey != null) {
UpdateDeviceKeys(newgroupKey);
return true;
}
return false;
}

private BigInteger getHash(BigInteger input)
{
try
{
md = MessageDigest.getInstance("MD5");
}
catch (NoSuchAlgorithmException e)
{
throw new IllegalArgumentException
      ("Error : MD5 Hash not found");
}
md.reset();
byte[] bytes = input.toByteArray();
md.update(bytes, 0, bytes.length);
return new BigInteger(md.digest());
}

boolean AttemptToResolvePolynomialandReplaceKeys(
BigInteger[] Poly, BloomFilter bf) throws
   NoSuchAlgorithmException
{

for (int k = Gateway.treelevels; k >= 0; k--)
{
```

```
SecretKey key = (SecretKey) this.devicekeys[k];
BigInteger ts = Poly[3];
BigInteger K = getHash
      (new BigInteger(key.getEncoded()).add(ts));
BigInteger param1 = K.modPow(new BigInteger("3"), p);
                    //3=Gateway.treelevels
BigInteger param2 = (K.modPow(BigInteger.TWO, p).
                  multiply(Poly[0]).mod(p)).mod(p);
BigInteger param3 = (K.multiply(Poly[1])).mod(p);
BigInteger param4 = Poly[2].mod(p);
param1 = param1.add(param3);
param2 = param2.add(param4);
BigInteger probableGroupKey = param1.subtract(param2);
probableGroupKey = probableGroupKey.mod(p);
if (probableGroupKey.signum() > 0)
probableGroupKey = probableGroupKey.subtract(p);

probableGroupKey = BigInteger.ZERO.subtract(probableGroupKey);

if (bf.contains(probableGroupKey))
{
if (probableGroupKey.toByteArray().length > Gateway.keyLength)
{
System.out.printf("Warning:␣"+
            "Long␣key␣after␣BF%n.␣Program␣will␣crash");
}
SecretKey receievedGroupkey = new SecretKeySpec
      (probableGroupKey.toByteArray(), "AES");
UpdateDeviceKeys(receievedGroupkey);
return true;
} // if not, try another key from the keylist.

} // end for
return false;
}// end function

public static String print(byte[] bytes)
{
StringBuilder sb = new StringBuilder();
sb.append("[␣");
for (byte b : bytes)
{
sb.append(String.format("0x%02X␣", b));
}
sb.append("]");
return sb.toString();
}
```

```
public void UpdateDeviceKeys(SecretKey groupkey)
                  throws NoSuchAlgorithmException
{
this.devicekeys[0] = groupkey;
for (int n = this.devicekeys.length - 1; n >= 1; n--)
{
SecretKey subkey = AES.XORAESKeys(this.devicekeys[n],
                                  this.devicekeys[n - 1]);
this.devicekeys[n] = subkey;
}
}
}// end class
```

A.4 Gateway.java

The Gateway server module maintains the key heap tree. The key heap tree is created during program initialization, and keys are replaced with new ones each time a device is added or removed from the device list. The gateway server also issues keys for the newly added device.

```
//Variables are made public to optimize memory utilization
// in the repeated testing scenario.
import java.math.BigInteger;
import java.security.MessageDigest;
import java.security.NoSuchAlgorithmException;

import javax.crypto.SecretKey;

public class Gateway
{
private static Gateway instance=null;
SecretKey[]  Heaparray;
AES aes;
int N;//Number of devices or leaf nodes.
int totalkeys; //Number of total nodes in the heap.
public static BigInteger p=
new BigInteger("3401615785904725136076687606536554462O3");
public static int treelevels=3; //4 levels : 0,1,2 & 3
public static int keyLength =128;
SecretKey nextGroupkey;
Long time = System.currentTimeMillis();
BigInteger ts = new BigInteger(time.toString());
BigInteger[] siblingkeys= new BigInteger[treelevels];
```

```
BigInteger[] poly = new BigInteger[treelevels+1];//+1 for
   timestamp
MessageDigest md;

private Gateway() throws NoSuchAlgorithmException
{
aes = new AES(p, keyLength);
N = (int) Math.pow((double)2, (double)treelevels);
totalkeys = 2*N-1;
CreateHeap();
}
//singleton instance returning method.
public static synchronized Gateway getinstance()
        throws NoSuchAlgorithmException
{
if (instance==null)
instance = new Gateway();
return instance;
}
private final void CreateHeap()
        throws NoSuchAlgorithmException
{
Heaparray = new SecretKey[totalkeys];
for(int i =0;i<totalkeys;i++)
{
SecretKey newSkey =(SecretKey)aes.AESKeyGen(keyLength);
Heaparray[i] =newSkey;
}
}

String getEncryptedNextGroupKey() throws Exception
{
//replace the existing Group key
SecretKey encryptionKey=Heaparray[0];
String encryptednextGroupkey =
        AES.AESencryptECB(nextGroupkey, encryptionKey);
return encryptednextGroupkey;
}

void setnextGroupkey() throws Exception
{
nextGroupkey=(SecretKey)aes.AESKeyGen(keyLength);
}

void UpdateHeap() throws NoSuchAlgorithmException
{
Heaparray[0]=nextGroupkey;
int j=0;
```

```
//Replace the other keys in the heap tree with XOR logic
for(int i =totalkeys-1;i>=1;i--)
{
j=i-1;
Heaparray[i]=AES.XORAESKeys(Heaparray[i],Heaparray[j/2]);
}
}

SecretKey[] GetKeysforNewlyAddedDevice
( int nPosition) throws NoSuchAlgorithmException
{
SecretKey keylist[]  = new SecretKey[4];
    //copy one key from each level
nPosition+=N;//converting list position to leaf node.

for (int k=0; k<=treelevels;k++)
{
int KeyPos = nPosition/ ((int)Math.pow((double)2, (double)k));
keylist[treelevels-k]= Heaparray[KeyPos-1];
}//end for
return keylist;
}
private BigInteger getHash(BigInteger input)
{
try
{
    md = MessageDigest.getInstance("MD5");
}
catch (NoSuchAlgorithmException e)
{
    throw new IllegalArgumentException("MD5␣Hash␣Error");
}
md.reset();
byte[] bytes = input.toByteArray();
md.update(bytes, 0, bytes.length);
return new BigInteger(md.digest());
}

BigInteger[] Get3rdDegreePolynomial(int nPosition) throws
   Exception
{
BigInteger bnewGroupkey = new BigInteger (nextGroupkey.
   getEncoded());
//copy one key from each level
nPosition+=N;//converting list position to leaf node.

for (int k=0; k<treelevels;k++)
{
```

```
int KeyPos = nPosition/ ((int)Math.pow((double)2, (double)k));
KeyPos =((KeyPos/2) == (KeyPos-1)/2) ? (KeyPos-1) : (KeyPos+1)
   ;
siblingkeys[k]= getHash(
    new BigInteger(Heaparray[KeyPos-1].getEncoded()).add(ts));
}//end for

if(siblingkeys.length ==treelevels)
{
poly[0]=siblingkeys[0].
        add(siblingkeys[1].add(siblingkeys[2])); //z1+z2+z3
poly[0]=poly[0].mod(p);
BigInteger temp1=(siblingkeys[0].multiply(siblingkeys[1]));
BigInteger temp2=(siblingkeys[0].multiply(siblingkeys[2]));
BigInteger temp3=(siblingkeys[1].multiply(siblingkeys[2]));
poly[1] = temp1.add(temp2.add(temp3));//z1z2+z2z3+z1z3
poly[1]=poly[1].mod(p);
poly[2]=siblingkeys[0].multiply(siblingkeys[1]).
                        multiply(siblingkeys[2]);//k+z1z2z3
poly[2]=poly[2].add(bnewGroupkey);
poly[2]=poly[2].mod(p);
            poly[3]=ts;
}
return poly;
}

BloomFilter GetBloomFilter()
{
BloomFilter bf = new BloomFilter();
BigInteger content = new BigInteger(nextGroupkey.getEncoded())
   ;
content = content.mod(p);
bf.add(content);
return bf;
}

}
```

A.5 Keymanager.java

Contains the application main that issues command to gateway and device on adding or removing of devices.

```
//Application Main. Editor used: Microsoft VS Code.
```

```
//Application Main. Editor used: Microsoft VS Code.

import java.math.BigInteger;
import java.util.Scanner;

public class KeyManager
{
static Device[] devicelist;
static Gateway gwyServer;
static final int maxdevices =8;
static long memadd,timeadd;
static Runtime runtime;
static long usedMemoryBefore, usedMemoryAfter;
static String addmem, removemem;
static String addtime, removetime;
static long start, end;
static boolean nret=false;

public static void main(String[] args) throws Exception
{
devicelist = new Device[maxdevices];
gwyServer = Gateway.getinstance();
Scanner input;
int choice=0;
runtime = Runtime.getRuntime();
input = new Scanner(System.in);
System.out.println("Measures␣Time␣&␣Memory.");
System.out.println("Press␣1␣and␣wait␣1␣minute.");
choice = input.nextInt();
addmem = new String();
removemem = new String();
addtime = new String();
removetime = new String();
switch (choice)
{
case 1 :
for (int i=0; i<101; i++)
{
for (int j=0; j<devicelist.length;j++)
{
    AddDevice(j);
}

for (int j=devicelist.length-1; j>=0;j--)
{
    RemoveDevice(j);
}
addmem +="\n";
```

```
addtime +="\n";
removemem +="\n";
removetime +="\n";
}

System.out.println("MEMORY USAGE");
System.out.println("ADD");
System.out.print(addmem);
System.out.println("REMOVE");
System.out.print(removemem);
System.out.println("TIME USAGE");
System.out.println("ADD");
System.out.print(addtime);
System.out.println("REMOVE");
System.out.print(removetime);

break;
default:
break;
}
input.close();
}

static boolean AddDevice( int nPos) throws Exception
{
usedMemoryBefore = runtime.totalMemory()
    - runtime.freeMemory();
start = System.nanoTime();

int emptyslotes=0;
//Get a new groupkey encrypted with current groupkey.
                    //Broadcast it to all devices.
gwyServer.setnextGroupkey();

String encryptedkey = gwyServer.getEncryptedNextGroupKey();


for (int i=0; i<devicelist.length; i++)
{
if(devicelist[i]!=null)//replace group key only
                // if there is at least one device existing.
{
nret = devicelist[i].ReplaceGroupKeyatdevice(encryptedkey);
}
else
{
emptyslotes+=1;
}
```

```
usedMemoryAfter = runtime.totalMemory() - runtime.freeMemory()
   ;
}
if(emptyslotes==devicelist.length)//first device.
nret=true;
//Device keys update successful. Update heap, add new device.
if(nret)
{
gwyServer.UpdateHeap();
Device newdevice = new Device();
newdevice.devicekeys= gwyServer.GetKeysforNewlyAddedDevice(
   nPos);
devicelist[nPos]=newdevice;
}

end = System.nanoTime();
memadd = usedMemoryAfter-usedMemoryBefore;
memadd = memadd/1000;//kb
addmem = addmem+",␣"+ memadd;
timeadd = (end-start);
timeadd =timeadd/1000;//microseconds
addtime = addtime+",␣"+ timeadd;
Thread.sleep(1);
return nret;
}

static boolean RemoveDevice(int nPosition) throws Exception
{
usedMemoryBefore = runtime.totalMemory()
    - runtime.freeMemory();
start = System.nanoTime();

int remainingdevicescount = 0;
gwyServer.setnextGroupkey();
for (int i=0; i<devicelist.length;i++)
{
if(devicelist[i]!=null)//finds the first empty slot
remainingdevicescount++;
}
if(devicelist[nPosition]==null)
return false;

if(remainingdevicescount==0)//if we are removing
// the last device, no group key communication needed.
{
return false;//do nothing
}
else
```

```
{

devicelist[nPosition]= null;//Remove the device
BigInteger [] polyCoeffients = gwyServer.
                    Get3rdDegreePolynomial(nPosition);
BloomFilter bf = gwyServer.GetBloomFilter();//returns next
   Gkey bf.
int replaced=0, count=0;
for (int i=0; i<devicelist.length; i++)
{
if (devicelist[i]!= null)
{   count++;
if(devicelist[i].
        AttemptToResolvePolynomialandReplaceKeys(
           polyCoeffients, bf))
        replaced++;
    usedMemoryAfter = runtime.totalMemory() - runtime.
       freeMemory();
}
}
if(replaced!=count)
System.out.printf("Error:␣Only␣%d␣out␣of␣"+
                           "%d␣devicekeys␣updated%n",replaced
                              , count);
gwyServer.UpdateHeap();

}


end = System.nanoTime();
memadd = usedMemoryAfter-usedMemoryBefore;
removemem = removemem+",␣"+ memadd;
timeadd = (end-start);
timeadd =timeadd/1000;
removetime = removetime+",␣"+ timeadd;
Thread.sleep(1);
return true;
}
}
```

Vita

Candidate's full name: Vishnu Prasanth Vikraman Pillai

Bachelor of Technology in Computer Science and Engineering, Mahatma Gandhi University, Kerala, India, 2006.

Master of Computer Science, University of New Brunswick, Fredericton, Canada, Expected May 2024.

Publications:

1. Vishnu Prasanth Vikraman Pillai, Zeming Zhou, Songnian Zhang, Rongxing Lu and Mohammad Mamun. An Efficient Dynamic Key Management Scheme for IoT Devices in Aging in Place Systems. 2023 IEEE/CIC International Conference on Communications in China (ICCC), August 2023

Posters:

1. Vishnu Prasanth Vikraman Pillai, Zeming Zhou, Songnian Zhang, Rongxing Lu and Mohammad Mamun. An Efficient Dynamic Key Management Scheme for IoT Devices in Aging in Place Systems, Research Exposition 2023, Fredericton, Canada, April 14, 2023.