2019

An efficient framework for privacy-preserving computations on encrypted IoT data

Shruthi Ramesh
Iowa State University

An efficient framework for privacy-preserving computations on encrypted IoT data

by

Shruthi Ramesh

A thesis submitted to the graduate faculty

in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE

Major: Computer Engineering (Secure and Reliable Computing)

Program of Study Committee:
Manimaran Govindarasu, Major Professor
Wensheng Zhang
Neil Gong

The student author, whose presentation of the scholarship herein was approved by the program of study committee, is solely responsible for the content of this thesis. The Graduate College will ensure this thesis is globally accessible and will not permit alterations after a degree is conferred.

.

Iowa State University

Ames, Iowa

2019

DEDICATION

I would like to dedicate this thesis to my family – my parents, Ramesh Rengaswamy and Vijay Ramesh who have always been the constant source of unconditional love, support and encouragement, my brother Dr. Sridharan Ramesh, for always being there whenever I needed moral support, motivation and guidance and to my partner Sudarsanan Krishnan, for the cheers, support and sacrifices throughout this journey.

TABLE OF CONTENTS

Page

LIST OF FIGURES

Page

LIST OF TABLES

Page

LIST OF ALGORITHMS

Page

NOMENCLATURE

IoT	Internet of Things
FHE	Fully Homomorphic Encryption
KMS	Key Manager Server
HCO	Health Care Organization
CIA	Confidentiality, Integrity and Availability
AES	Advanced Encryption Standard
ECC	Elliptic Curve Cryptography
MPC	Multi Party Computation
UDP	User Datagram Protocol
NIST	National Institute of Standards and Technology
HDFS	Hadoop Distributed File System
CloudIoT	Cloud based IoT
IETF	Internet Engineering Task Force
CSP	Cloud Service Providers

ACKNOWLEDGMENTS

This thesis would have not been possible without the help and support of many people. First, I would like to thank my research advisor, Dr. Manimaran Govindarasu for his patience, guidance and encouragement throughout this journey. Thank you for challenging my ideas and allowing me to explore different domains before finding the topic of my interest. Thank you for teaching me the basic tools of conducting a thorough research.

I would also like to thank Dr. Wensheng Zhang and Dr. Neil Gong for taking out time to respond to my emails and for agreeing to be a part of my POS committee. Thank you, Dr. Wensheng Zhang, for always accepting to meet me and for the valuable feedback after every discussion.

I would also like to thank my research group for providing valuable feedback during all the research meetings. I would like to thank my friend Ranjitha for the support and cheers throughout this journey.

ABSTRACT

There are two fundamental expectations from Cloud-IoT applications using sensitive and personal data: data utility and user privacy. With the complex nature of cloud-IoT ecosystem, there is a growing concern about data utility at the cost of privacy. While the current state-of-the-art encryption schemes protect users' privacy, they preclude meaningful computations on encrypted data. Thus, the question remains "how to help IoT device users benefit from cloud computing without compromising data confidentiality and user privacy"? Cloud service providers (CSP) can leverage Fully homomorphic encryption (FHE) schemes to deliver privacy-preserving services. However, there are limitations in directly adopting FHE-based solutions for real-world Cloud-IoT applications. Thus, to foster real-world adoption of FHE-based solutions, we propose a framework called *Proxy re-ciphering as a service*. It leverages existing schemes such as distributed proxy servers, threshold secret sharing, chameleon hash function and FHE to tailor a practical solution that enables long-term privacy-preserving cloud computations for IoT ecosystem. We also encourage CSPs to store minimal yet adequate information from processing the raw IoT device data. Furthermore, we explore a way for IoT devices to refresh their device keys after a key-compromise. To evaluate the framework, we first develop a testbed and measure the latencies with real-world ECG records from TELE ECG Database. We observe that i) although the distributed framework introduces computation and communication latencies, the security gains outweighs the latencies, ii) the throughput of the servers providing re-ciphering service can be greatly increased with pre-processing iii) with a key refresh scheme we can limit the upper bound on the attack window post a key-compromise. Finally, we analyze the security properties against major threats faced by Cloud-IoT ecosystem. We infer that *Proxy re-ciphering as a service* is a practical, secure,

scalable and an easy-to-adopt framework for long-term privacy-preserving cloud computations for encrypted IoT data.

CHAPTER 1. INTRODUCTION TO CLOUD BASED IoT

1.1 Overview

1.1.1 Internet of Things

Internet of Things (IoT) refers to a system where devices with embedded sensors and actuators in the physical world are connected to the Internet through wired or wireless communication channels and can be uniquely identified and addressed. IoT, a vision that was conceived in the early 2000s, has become a reality today, allowing users to connect everything from industrial components like gas turbines, freight goods, industrial heating and ventilation equipment to everyday consumer devices like light bulbs, door lock, washing machines etc., to the Internet. The pervasive presence of the IoT devices around humans has given the devices the ability to measure, infer and even alter the human environment. This has become possible primarily due to several advancements in technologies like wireless networks, embedded sensors, and cloud computing, to name a few, over the years. In a nutshell, an IoT ecosystem offers the following benefits to users, when compared to a traditional IT infrastructure:

i) ***communication***: the ability to collect and transfer information from various sensors like temperature sensor, air purity monitoring sensors, heart rate monitoring sensors etc.

ii) ***remote control****:* the ability to control the environment remotely, like turning on air conditioners at home from the office, remotely administering insulin delivery for a patient, unlocking home door from a car etc.

iii) ***correctness****:* the ability to avoid human errors arising from manual entries during high precision recordings like in heart rate monitoring, supply chain logistics management etc.

iv) ***cost:*** the ability to effectively minimize the cost associated with managing an application infrastructure by means of data, like analyzing sensor data to minimize equipment failures with a planned maintenance, reducing fuel consumption by monitoring drive behavior etc.

Hence, it comes as no surprise that the US National Intelligence Council named IoT as one of the six disruptive technologies that will potentially impact US interests by 2025 [1]. Since a multitude of applications take advantage of IoT, diverse classes of IoT devices have been developed over the years, ranging from wireless sensor networks to fitness bands to smart meters to connected vehicles. These diverse types of devices typically vary in their processing power, cost, buffer space, size, and user interfaces, to name a few aspects. Small devices that are limited in their power, CPU and memory are termed as constrained devices by the IETF [2]. IETF has also roughly classified the constrained devices into three classes ($class - N$ for $N = 0,1,2$) depending on the device capabilities [2] as shown in Table 1.

Table 1. Constrained node classification [2]

Name	**Data size** (KB)	**Code size** (KB)
Class 0	<10	<100
Class 1	~10	~100
Class 2	~50	~250

Class 0 devices typically include sensor-like motes that are very limited in their memory and processing power. They connect to the Internet using other computationally powerful devices like gateways. *Class 1* devices are relatively constrained in processing power and memory and connect to the Internet using lightweight protocol stacks like Constrained Application Protocol over UDP [3]. *Class 2* devices are relatively less constrained and can

support protocol stacks like HTTP, although they are usually designed to use lightweight protocols to improve efficiency and cost.

Figure 1 shows a typical IoT ecosystem where IoT devices capture information and share it to the cloud-based servers via gateways.

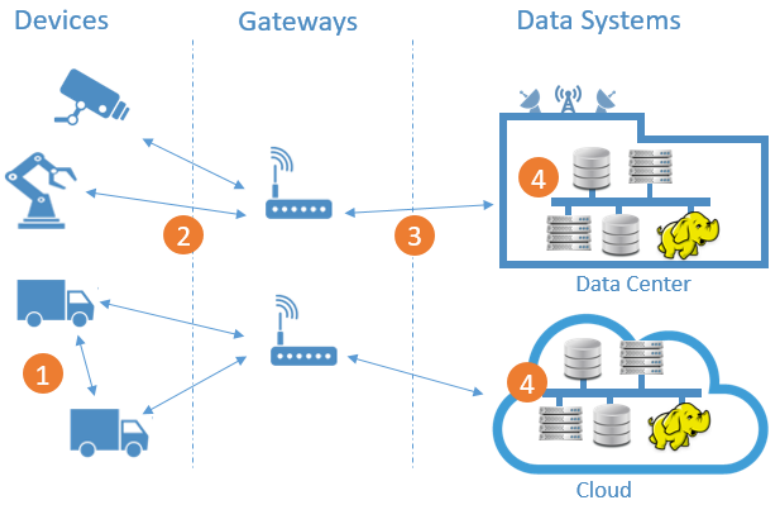

Figure 1. An example of IoT ecosystem

There are two fundamental expectations from IoT applications: data security and data utility. Data security circles around the Confidentiality-Integrity-Availability triad [4] which in the context of IoT device data can be informally and appropriately defined as the follows:

- **Confidentiality**: the data generated by an IoT device is protected by allowing only users (device owner, legitimate recipients, etc.) who can prove their identity and have appropriate privileges to access the data
- **Integrity**: the data generated by an IoT device should not be altered throughout its lifecycle by anyone, including the device owners
- **Availability**: the data generated by an IoT device should be readily available to the appropriate users (device owners, legitimate recipients, etc.) upon request

Data utility refers to using raw data generated by the devices to make intelligent decisions that can help the end-users. With the embedded sensors in IoT devices continuously collecting data about the human environment, device owners are interested in more than just a mere collection and storage of the raw device data in the cloud servers. Rather, they expect useful services like intelligent insights with daily statistics, periodic reports, and alerts from anomaly detection, to name a few. For example, people using fitness bands expect to see their resting heart rate values, number of steps walked in a day, number of hours of deep sleep in a day, the stress levels throughout the day, etc. Thus, it becomes imperative for IoT device vendors to design a back-end infrastructure that can handle and process the device data while meeting the above-mentioned requirements of their consumers.

1.1.2 Cloud Computing

The National Institute of Standards and Technology (NIST) has provided the following definition for "cloud computing" capturing its essential and multi-faceted aspects: *"cloud computing is a model for enabling ubiquitous, convenient, on-demand network access to a shared pool of configurable computing resources (e.g., networks, servers, storage, applications and services) that can be rapidly provisioned and released with minimal management effort or service provider interaction."* [5]. Over the years, cloud computing has enabled the realization of new computing paradigms owing to its ability to provide nearly unlimited storage and processing capabilities, in turn hugely impacting the IT industry.

Cloud service providers are companies that leverage cloud computing and offer cloud-based solutions such as Infrastructure as a Service (e.g. Amazon Web Service [6], Microsoft Azure [7]), Platform as a service (e.g. Red hat OpenShift [8]) and Software as a service (Salesforce [9]). Cloud service providers facilitate off-premise storage and processing,

enabling companies to offload the overhead associated with maintaining on-premise infrastructures, managing IT personnel and expanding them as required. Furthermore, the system of virtually distributed, powerful and always up-to-date servers hosted by the cloud service providers can guarantee better resiliency, scalability and security compared to a traditional computing system, while at the same time being cost-effective. These salient features are essentially responsible for the recent transformation in the business models of various industries ranging from transportation to manufacturing to IT companies [10]- [11].

In the following section, the current study provides a high-level functional overview of the state-of-the-art cloud storage and processing services provided by the cloud service providers.

1.1.2.1 Cloud Storage

A typical cloud storage model involves clients storing data (e.g. images, text files) in the cloud server and querying it later when they want to view the data. For example, service likes Amazon S3 [6], Azure storage [7], Google Drive [12] and Dropbox [13]. Clients typically use a secure channel to transfer the data to the cloud to prevent an adversary from eavesdropping on the channel. In case the information shared with the cloud servers is sensitive (e.g. business critical data), clients can encrypt the sensitive and critical data before uploading the data to the cloud servers. An encrypted data, by the virtue of encryption, is protected against eavesdropping over an unsecure wireless channel (security for data in-transit) and against data leakage after storing in the cloud servers (security for data at-rest). Clients can also use encrypted query processing to conceal access patterns from the cloud servers [7, 8] and decrypt the data after receiving.

1.1.2.2 Cloud Compute

By cloud compute, the current study refers to allowing the cloud servers to process data in their servers. Personal assistants like Siri [14] and Alexa [15], to productivity tools like Office Online [16] and Google Docs [12], to big data analytics tools like Hadoop [17], are all examples of compute services provided by cloud service providers. Cloud compute services are generally data driven in that they rely on providing services by processing huge amounts of data shared by the clients. To protect the data during transit and storage, clients typically encrypt the data in their local machine and then upload the data to the cloud servers. When the data is required for computation, the authorized cloud application first decrypts the data and then performs the necessary computation on the data [18]. To protect the result of the computation from an adversary, the cloud application re-encrypts the result before sharing with the clients.

The next section provides a brief overview of IoT ecosystem integration with cloud computing.

1.1.3 Internet of Things and Cloud

In 2011, the number of interconnected IoT devices exceeded the total number of people [19] and by 2025 it is expected to reach a value of over 75 billion [20]. This in turn would substantially increase the volume of data generated. It is thus evident that IoT will soon become one of the main sources of Bigdata [21]. Industrial analysts typically describe the nature of Bigdata using the 4V's: volume (amount of data generated), velocity (rate of data generation), veracity (accuracy of data) and variety (diverse nature of data originating from varied sources) [21]. It can be inferred that a traditional infrastructure is not efficient for long-term storage or processing of such enormous amounts of varied data generated by the IoT

devices. Interestingly, there exists a perfect symbiotic relation between cloud computing and IoT. While IoT generates massive amounts of data, cloud computing can efficiently handle the data throughout its lifecycle.

Several cloud-based IoT applications have been developed over the years that leverage the merits of both the ecosystems. Wearables like Fitbit [22] to smart home appliances like Nest Thermostat [23] and Ring video doorbells [24] to self-driving cars like Tesla [25], are all examples of real-world implementations of Cloud-based IoT applications.

While outsourcing the task of storing and processing the data to the cloud servers can improve efficiency for businesses, it can naturally introduce concerns over the loss of privacy and data security for businesses. Encryption schemes, such as AES [26] can satisfactorily address this concern by allowing businesses/end-users to encrypt the data before outsourcing them to cloud. Thus, this guarantees the privacy of end-users during the storage phase. However, the state-of-the-art encryption schemes do not provide meaningful results when cloud servers perform any computation on the encrypted data. An alternate approach is to allow cloud servers to decrypt the data before performing any computations and re-encrypt the data after the computation. However, it is evident that such a scheme is not privacy-preserving.

1.2 Research Motivation and Objective

The state-of-the-art encryption schemes allow IoT device users to make use of cloud servers to store their encrypted device data. However, it is currently not possible for cloud servers to deliver intelligent insights to the device users while preserving their privacy completely. Fully homomorphic encryption schemes have been proposed in the literature as a solution to tackle this issue. When a data is encrypted under FHE schemes, it is possible for cloud servers to perform computations on it, thereby enabling privacy preserving cloud

computations for encrypted IoT data. However, the FHE schemes proposed so far are computationally intensive and are unsuitable to be employed on an IoT device.

Recent research studies have proposed alternate solutions to assuage this limitation. However, there are still challenges in the practical implementation of these solutions for a resource constrained IoT ecosystem. Fostering real-world adoption of the FHE based privacy-preserving computation schemes for IoT ecosystem requires a practical solution with a modest overhead for all the entities, ranging from the device vendors, the device users to the cloud service providers. Thus, this thesis attempts to answer the question: "*how to help IoT device users benefit from cloud computing without compromising data confidentiality and user privacy?"* by proposing a practical framework called ***Proxy re-ciphering as a service***. It leverages existing schemes such as distributed semi-trusted proxy servers and threshold secret sharing to tailor FHE schemes for IoT applications, thereby enabling long-term privacy-preserving cloud computations for encrypted IoT data.

1.3 Thesis Organization

The rest of the thesis work is organized as follows:

- Chapter 2: Background and literature review of the existing secure and privacy pre-serving computation schemes
- Chapter 3: Proposal of *Proxy re-ciphering as a service* to enable privacy-preserving computations for IoT devices
- Chapter 4: Case study of *Proxy re-ciphering as a service* with application to smart healthcare ecosystem

- Chapter 5: Evaluation of the proposed framework using experimental testbeds and theoretical analysis
- Chapter 6: Conclusion and future work

CHAPTER 2. BACKGROUND AND LITERATURE REVIEW

This chapter focuses on various encryption schemes that are proposed in the literature along with a special emphasis on schemes that allow privacy-preserving computations on the encrypted data. Specific attention is given to the merits and limitations of each of these encryption schemes in the context of IoT. After analyzing each of these schemes, a suitable encryption scheme is chosen for the current study. During this process, various works proposed in the literature are covered that help enable cloud-based privacy-preserving computations on the encrypted data.

The field of End-to-End security encompasses a variety of areas that address various concerns related to computer/network security. Specifically, a data is called End-to-End secure if its CIA triad is protected from the time of its generation to the time of its consumption. In the context of an IoT ecosystem, end-to-end data security essentially means that data is encrypted before it leaves the embedded device and it is not decrypted until it reaches the end-user (typically an application running on the user's smartphone/laptop). The goal of such an implementation is to secure data against the common attacks, such as eavesdropping (confidentiality), man-in-the-middle (integrity, availability) and denial of service (availability) to preserve its CIA triad. Employing End-to-End encryption schemes can satisfactorily guarantee security of a system against eavesdropping attacks.

2.1 Conventional Encryption Schemes

Most real-world applications that aim to protect the confidentiality of their data use conventional encryption schemes. They can be implemented either in the form of a public-key encryption system or as a symmetric key encryption system.

Public key cryptography involves two different keys for encryption and decryption purposes, named public key and private key, respectively. As the name suggests, every participating node/user in a system is given a *public key* that is not a secret and a *private key*, which is a secret and is known only the node/user. A sender node encrypts the data using the public key of the recipient node before sending the data to the recipient. Upon receiving the encrypted data, the recipient node decrypts the data using its private key. One of the main advantages of this system is the key management and the scalability of the scheme.

RSA [27] is one of the most widely used public key cryptographic solutions. The security and the hard problem in this primitive are based on finding prime factors of a composite number. The length of the key used for encryption is a measure of the security of the system. It has been identified that to provide an equivalent level of security compared to AES encryption [26] with 128 bits key, RSA requires 3072 bits of key [28] . This is also shown in Table 2. In addition, it is close to 1000 times slower than the symmetric key counterparts, due to its computational processing power requirements [29]-[30] . This makes asymmetric encryption schemes challenging and computationally intensive for constrained IoT devices.

Elliptic curve cryptography (ECC) [32]-[33] is another public key scheme that is based on elliptic curves over finite fields. It requires smaller key size when compared to RSA and is a better candidate when using public key cryptography for resource constrained IoT devices. The security of ECC is based on the hardness of Elliptic curve discrete logarithm problem [31].

Table 2. Recommended key size in bits by NIST [29]

Symmetric key	RSA key	Elliptic curve key
80	1024	160
112	2048	224
128	3072	256
192	7680	384
256	15360	521

Symmetric key algorithms, on the other hand, perform encryption and decryption using a single key that is pre-shared between a sender and a recipient. Symmetric key encryption schemes are usually less computationally intensive compared to the asymmetric key encryption schemes and require relatively smaller-length key to provide the same level of security guarantees compared to asymmetric key encryption schemes [28].

Currently Advanced Encryption Standard [26] (AES) is one of the most widely deployed symmetric-key encryption algorithm and is accepted as a standard by NIST for government applications and industrial applications [33][34]. Authors in [35] tested and compared the encryption performance of various symmetric key encryption schemes like AES, CLEFIA [36], PRINCE [37], SPECK [38], Camellia [39], Midori [40] under a restricted memory size of 1024 bytes or less of ROM and 128 bytes or less RAM. Their results show that AES was the fastest followed by SPECK. The current work focuses on devices that typically belong to Class 1 or Class 2 category of constrained nodes [discussed in Chapter 1 and shown in **Table 1**]. These devices have around 10-50 KB RAM and 100-500 KB ROM. In most IoT devices, the cryptographic schemes are implemented as a software component rather than a dedicated chip[35]. Therefore, it is practical to choose an algorithm that requires less CPU cycles and a modest amount of ROM. Since the current study requires medical IoT devices to

only perform encryption, following the results of [35], the current study uses AES encryption scheme to encrypt IoT device data before sending to the cloud servers .

AES is by design optimized for low memory, speed and thus has been deployed across platform ranging from 8-bit processors to super power CPU / GPUs [35]. Even x86 architectures have witnessed instruction-set extensions by Intel [33] for improving the performance of encryption and decryption functions. AES is a block cipher that operates on 128-bit blocks of data. The number of rounds (n_r) of processing depends on the size of the key (128/192/256 bits) used in the schemes. As the key length increases, the security of the system increases, along with the number of rounds of processing (n_r=10 for AES-128, 12 for AES-192 and 14 for AES-256). The current study uses AES-128 with $n_r = 10$. AES provides the following modes of operation: Electronic Code Book (ECB), Ciphertext Chain Blocking (CBC), and Counter (CTR).

While AES is energy efficient and provides strong security guarantees, one of the main disadvantages is that it does not provide meaningful results, when performing arithmetic or logical operations on the ciphertext. Figure 2 shows an example of this limitation. It can be seen that when encrypting an integer value of 2 and an integer value of 3, the sum of the encrypted ciphertexts does not yield a value equal to encrypting an integer value of five: $Enc(2) + Enc(3) \neq Enc(5)$.

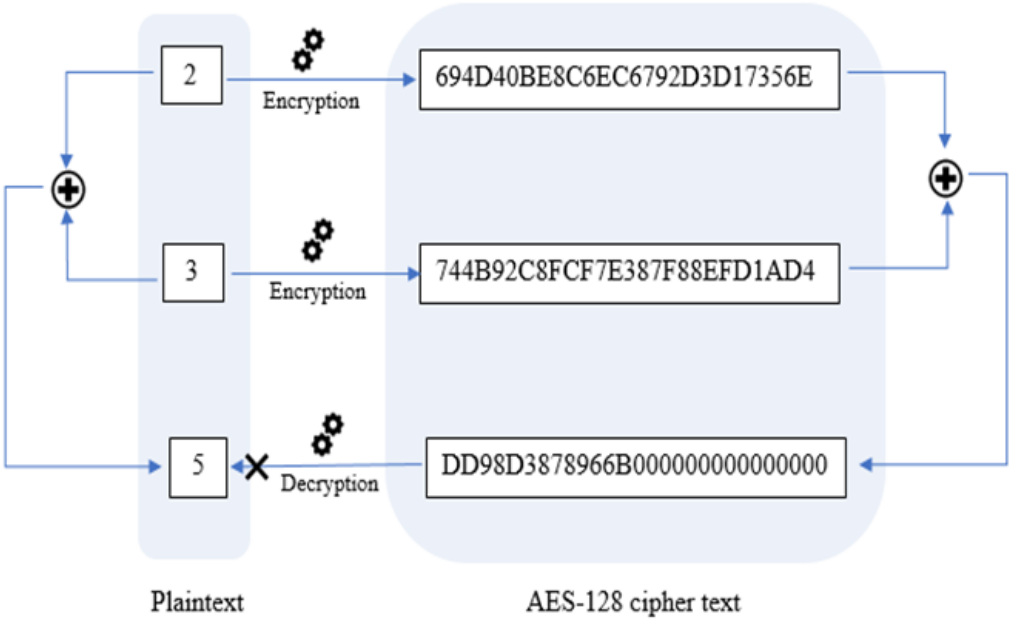

Figure 2. An example of computation on AES-encrypted data

2.2 Homomorphic Encryption Schemes

Homomorphic encryption schemes have been proposed to tackle the problem to provide secure and privacy-preserving computations on encrypted data, while minimizing the communication complexity that secure multi party computation schemes [41]–[43] usually witness. Rivest, Adleman and Dertouzos [44] stated the following around 1978 :

"It remains to be seen whether it is possible to have a privacy homomorphism with a large set of operations which is highly secure"

In other words, the possibility of building an encryption scheme, that is practical and worthwhile to perform arbitrary computations on an encrypted data, without having to decrypt during computation, remained an open question for a long time.

2.2.1 Partially Homomorphic Encryption Schemes

Partially homomorphic encryption schemes are a class of encryption schemes that allow arbitrary number of homomorphic additions or homomorphic multiplications on the

encrypted data. A scheme is *additively-homomorphic* if the result of an addition operation on two ciphertexts followed by a decryption, results in the same value as the addition operation on the two-plaintexts. Similarly, *multiplicatively-homomorphic* scheme is one where the result of a multiplication operation on two ciphertexts followed by a decryption operation gives the same value as the multiplication operation on the two-plaintexts. Table 3 presents the most-widely used partial homomorphic encryption schemes along with their homomorphic properties.

Table 3. Widely known partially homomorphic encryption schemes

Encryption scheme	Homomorphic nature	
	Additive	Multiplicative
RSA[27]		Yes
ElGamal[45]		Yes
Paillier[46]	Yes	
Goldwasser-Micali[47]	Yes	
BGN[48]	Yes	Yes*

*: allows only one multiplication operation

Table 3 shows that RSA and ElGamal schemes allow multiplication operations over the encrypted data while schemes like Galwasser-Micali (GM) and Paillier allow addition operations over the encrypted data. Furthermore, BGN encryption scheme allows an arbitrary number of addition operations on the encrypted data along with one multiplication operation. GM scheme was one of the first schemes to achieve additive homomorphism with probabilistic public key crypto system, giving the highest security guarantees. However, one of the limitations of this scheme was that it allowed only a single bit data as its plaintext input. In

addition, the ciphertexts in this scheme were large, rendering the scheme impractical. Benaloh [49] generalized the GM scheme to extend support for plaintext data of up to a bit-length k . However, it suffered from high cost of decryption [50]. Paillier improved the earlier schemes by allowing encryption of plaintext data up to a bit-length k, while reducing the ciphertext expansion during encryption and at the same time maintaining a reasonable encryption and decryption costs.

CryptDB [51] is seminal research work that focused on access control for SQL queries on encrypted data. The authors implemented a Paillier cryptosystem to compute arithmetic operations like SUM over encrypted data. Authors in [52] proposed *DBMask*, to address the limitations of decrypting data to weaker encryption schemes within the cloud server in CryptDB, thus guaranteeing that the security of the data is not weakened with time. However, both DBMask and CryptDB were designed for web applications, with a fully trusted proxy server intercepting the communication between a client and a server to apply encryption and decryption schemes transparent to the clients. In an IoT application, such an approach can potentially conflict with the security and privacy requirements. In addition, it has been identified that the solutions are computationally intensive with an overhead of 25% [50].

Authors in [50] introduced *Talos*, keeping IoT ecosystems in mind. Their solution does not require a fully trusted proxy server and allowed query processing locally in the end-user device. They improved the efficiency of Paillier encryption scheme to operate on 32-bit integer data, by packing multiple plaintext data in a single ciphertext, thereby allowing parallel operations on multiple plaintexts. Along these lines, *Pilatus* [53] was introduced to overcome some of the limitations of Talos. *Pilatus* further optimized *Talos* to work on larger plaintext data sizes (e.g., 64-bit integers) and introduced data sharing across multiple recipients using

proxy re-encryption. The authors of *Pilatus* demonstrated their work on data from Fitbit [22] servers and on an anonymized data from Awawomen startup [54].

Despite its merits, partial homomorphic encryption schemes like Paillier, have a limitation where they can only be used for algorithms that require either addition or multiplication operation (with an exception to BGN) and not a combination of both. However, many practical IoT-based applications may require a combination of both, for example, linear regression, logistic regression, standard deviation, to name a few.

2.2.2 Fully Homomorphic Encryption Schemes

Gentry's Fully homomorphic encryption (FHE) [55] is a seminal work that showed that it is plausible and achievable to design a FHE scheme, answering the open question raised by Rivest, Adleman and Dertouzos . However, the scheme is computationally expensive to apply in real-life due to the *bootstrapping* operations. When a noise-level of a ciphertext reaches the threshold level, any homomorphic operation performed thereafter can potentially result in incorrect results. Gentry addressed this by bootstrapping the ciphertext to transform it into a new ciphertext with lesser noise-levels, thereby enabling further operations on the ciphertext. BGV [56] scheme, based on Learning With Errors, is currently one of the most widely used FHE scheme. It is a leveled FHE scheme that allows evaluation of arithmetic circuits of level L. It does not require bootstrapping and has been optimized to allow practical implementations of FHE based solutions. Libraries like HElib [58]-[59] have been developed using BGV schemes and are available for public use, making FHE accessible to researchers and application developers.

Early works on practical applications of FHE attempted to solve real world problems [59]-[60] , where the clients (laptop like devices) encrypted their data using FHE schemes to allow a computationally powerful third-party server (like a cloud server) to perform

computations on the encrypted data. This was clearly an improvement over to traditional encryption schemes where the cloud servers needed to decrypt data to perform computations. However, these solutions were proposed for traditional file-based applications running on laptop-like devices and are not suitable IoT devices. Furthermore, the computational complexity of this scheme is high, even for traditional devices, as it requires homomorphic encryption before every single data transfer. In addition, the communicational complexity of the scheme is also high due to the size of the ciphertext (for example in [61], the ciphertext-size is $n\,log\,(q)$, where $n = 2048$ and q=2^{258} to allow single multiplication and a large number of additions and the values are larger for applications that require more multiplication operations [61]) resulting in high latencies and bandwidth requirement.

Authors in [35] proposed a possible solution to address this problem. In their scheme, the clients can homomorphically encrypt the AES encryption key once and share it with the server, as a part of setup phase. During the data flow phase, the client can encrypt the data using AES encryption scheme allowing the server to homomorphically evaluate the AES decryption function to transform AES encrypted ciphertext to a homomorphic encrypted ciphertext. Thus, this scheme allows cloud servers to compute on data encrypted under homomorphic scheme, while optimizing the communication complexity. This is discussed in great detail in [62] and is available for application developers along with the HElib library. This solution clearly relieves the client devices from performing computationally intensive task for every single data transfer. Authors in [62] tested their solution on a 4GB RAM Intel Core i5-3320M laptop to evaluate its feasibility and performance. However, this approach is still not suitable for IoT devices and are computationally intensive, even if the device is required to encrypt the AES key under homomorphic encryption scheme just once.

An alternate approach could be to encrypt the key under homomorphic encryption scheme using computationally more powerful devices like a device owner's smartphone. However, there are a couple of limitations in the practical implement of this approach. Although algorithms in HElib are constantly optimized and updated, the authors of [62] have not tested the algorithms in HElib library for their feasibility and efficiency on lower computational devices like IPhones or Android smartphones. In addition, the library is not trivially compatible with Android like environments since they are natively written in C++ and depend on other libraries like NTL [63] and GMP [64]. In addition, device encryption is transparent to end users in some real-world implementations [65]-[66] where device vendors often do not store the encryption keys in the end-user apps running on a smartphone / laptop [67]. Rather, they rely on secure channel communication using HTTPS between a smartphone and the cloud server to download results from cloud servers to the smartphone [65], [68].

Authors in [69]-[70][71] have proposed a novel privacy-preserving remote patient ECG monitoring system. Their work is the first of its kind to show practical feasibility of FHE in terms of computational complexity and the storage costs for a real-world medical application on the cloud. However, they make a few assumptions that might always not be acceptable. They take advantage of a cloudlet (a PC primarily provided by the hospital, in their work) to decrypt the AES encrypted data from the IoT device and re-encrypt it using FHE scheme. This has a few limitations since it requires hospitals to distribute additional hardware to all patients. The other alternative solution requires patients to own a laptop. Although the latter is not a very unreasonable assumption, this can however prevent the mobility of the patients as called out by the authors themselves [69]. Apart from these, such a solution can quickly become a

single point of attack and might not be applicable to real-world implementations like the ones discussed in [22], [65], [68].

CHAPTER 3. PROPOSED SOLUTION-PROXY RE-CIPHERING AS A SERVICE

CloudIoT is a recently introduced paradigm that leverages two complementary technologies: Cloud computing and IoT [72]. Over the years, multitudes of *CloudIoT* applications have been developed, aiming to improve the efficiency of the tasks and the quality of living for the device users. From wearables like Fitbit [22] to secure smart homes appliances like Ring [24] to self-driving cars like Tesla [25], *CloudIoT now* has a pervasive presence among humans. The demand and expectations from the *CloudIoT* applications have also drastically increased, resulting in a highly competitive IoT market. Yet, there are two fundamental expectations from these applications that capture sensitive and personal information: data utility and user privacy. In the contemporary world of IoT, with the complex nature of data usage between multiple service providers that are typically masked away from the end-users, there is a growing concern among device users about receiving utility at the cost of their privacy. For example, an early work on understanding the ecosystem of Fitbit found that the mega dumps consisting of user activity data were sent to an analytics company called *Mixpanel* [68]. Although the Fitbit ecosystem has changed since then, this example illustrates the complicated world of *CloudIoT* with multiple backend service providers that end-users are necessarily not aware of. In a report released by Economist Intelligence Unit in 2018, 74% of the surveyed consumers were concerned that small privacy invasions may eventually result in loss of civil rights [73].

Fully homomorphic encryption schemes (FHE) can be leveraged by cloud service providers to guarantee privacy-preserving data services to device users. Understandably, adopting a new encryption scheme, especially a computationally-intensive and a nascent one, can be challenging for existing *CloudIoT* applications and businesses. A migration from a

functioning workflow to a new FHE based workflow can be practically fostered, only when the cost associated with the transition is modest for all the entities in the IoT ecosystem ranging from the device vendors to the device users to the cloud service providers.

Solutions proposed in the literature have the following limitations when integrating them into a real-world IoT ecosystem : 1) a fully homomorphic encryption scheme is computationally intensive to be deployed on an IoT device 2) a fully trusted cloudlet/proxy server to decrypt AES device data and encrypt it using a FHE data can become a single point of attack and is not truly privacy-preserving , 3) homomorphic encryption of IoT device keys (that are required to transform AES device data to FHE data) demands significant resources for computational purposes from the IoT devices and 4) outsourcing heavy computations to end-user devices like smartphones / laptops is not possible in some real-world implementations where device vendors often do not store the device encryption keys in the end-user apps [67], as discussed in chapter 2 .

Thus, to bridge the gap between the proposed solutions in the literature and their real-world adoption in the IoT ecosystem, the current work proposes **Proxy re-ciphering as a service.** It leverages existing schemes such as distributed semi-trusted proxy servers, threshold secret sharing and FHE to tailor a framework that enables long-term privacy-preserving cloud computations for encrypted IoT data.

3.1 Design Goals

This section will provide a brief overview of the high-level considerations that were given before arriving at the proposed solution. Design goals such as the ones considered form the salient features of the proposed solution.

Security: The data encrypted by an IoT device should not be decrypted by anyone other than the end users (device owners / legitimate recipients)

Availability: The solution should be resilient to any failures (internal or external) to ensure that the service is always available to the IoT device users

Accessibility and deployment readiness: The solution should use existing technologies that are publicly available and easily accessible to application developers

Scalability: The solution should ensure that the storage complexity is acceptable given the large ciphertext size of FHE schemes.

Energy efficiency and Performance: The solution should be lightweight on the IoT devices in terms of computational, communicational and storage complexity

Back-portability: Existing devices that are already operational should be readily able to take advantage of the solution

Transparency: The solution should be adoptable without requiring any additional hardware, for the end-users. Any additional manual/configurational overhead to the device users should be modest.

Practicality: The solution should closely resemble real-world implementations of *CloudIoT* applications, with minimal assumptions.

3.2 Major Cyber Threats

In this section, we will briefly focus on the major threats faced by cloud-based solutions where IoT devices generate data and end-users review the data using their device-companion apps. In the following discussion, the proxy servers are intermediary nodes that intercept IoT device-cloud communication and pre-process the device data before uploading it to the cloud servers.

Threat 1: Cloud database/server compromise: The current study assumes a semi-trusted cloud. This means that the server is honest and performs the algorithms correctly, but it is not trusted with the private data. Compromising cloud servers can give an adversary access to the encrypted data. A rogue administrator can misuse data for monetary incentives.

Threat 2: Proxy database / server compromise: The current study assumes a semi-trusted proxy server. An adversary can be interested in attacking the proxy servers to learn about the device encryption keys.

Threat 3: Byzantine proxy storage servers: A special attack on the proxy servers where an adversary can maliciously control the behavior of nodes/servers in the system resulting in the failure or corruption of data or a combination a both.

Threat 4: IoT device key compromise: An adversary can remotely launch attacks on the resource constrained IoT device to compromise the device key. This is a critical and relevant threat in today's IoT ecosystem[68]-[69]. Most device manufacturers hardcode the encryption keys [75] [65], [66], [74], [76] in the devices during the manufacturing phase. When such a long-term device key is compromised, the *post-compromise security* [77] for the IoT devices can be jeopardized.

Threat 5: Network-based attack: An adversary can attack the communication channel between different participating nodes in the system like the medical device, gateway, proxy servers and the cloud servers. This threat can be launched by a passive or an active attacker. Passive attackers are generally interested in learning about data patterns by launching eavesdropping attack and active attackers are typically interested in intercepting data to launch a man-in-the-middle attack.

3.3 System Architecture

In this section, the key components of the proposed system are discussed:

IoT device

These are embedded devices that fall under Class 1 and class 2 category of constrained devices as defined in Chapter 1 with around 10-50 KB RAM and 100-300 KB ROM. For example, Fitbit devices have around 250 KB flash and 32KB SRAM [74]. They support AES encryption to securely transmit the device generated data wirelessly [65]. They typically do not store information locally for processing and outsource the computations to cloud servers. They communicate only through gateways devices, typically via Bluetooth while other protocols like Zigbee are also supported.

Gateway devices

These are computationally powerful devices compared to the medical devices. They typically function via the device-companion apps running on smartphones (most widely used), laptops (supported configuration), enabling portability for the users. They have enough resources to support full stack HTTP protocols. They pack (typically as a JSON) the data from received from the IoT device before relaying it to cloud servers. They do not perform any other computations on the device data.

Cloud servers

These servers provide storage and compute services for other businesses. They receive device data from the gateway devices. They are computationally powerful and can scale their service when required. When a data is encrypted under FHE scheme, cloud servers can perform computation on the data and share privacy preserving insights with end-users.

Key Manager Server (KMS)

They are maintained by device vendors or OEMs depending on the implementation. They are solely responsible for generating and maintaining the cryptographic keying materials and embedding device specific keys during the manufacturing. These servers are computationally powerful and are assumed to be very secure.

Distributed semi-trusted proxy servers

These are computationally powerful servers that act as an intermediary between businesses and their cloud server providers. The current study employs a total of n proxy servers in a geographical region (across US, for example).

GUI devices

These are devices that display the results of cloud computations to the data consumers. The proposed system currently supports laptops since the authors in [58] have tested the performance on laptop like devices. When an optimized version of the library that is compatible with smartphones is available, the current system can be extended to work on relatively lower computational systems like smartphone devices.

Figure 3 shows proposed system model comprising of the above-mentioned key components along with a brief high-level description of the proposed workflow.

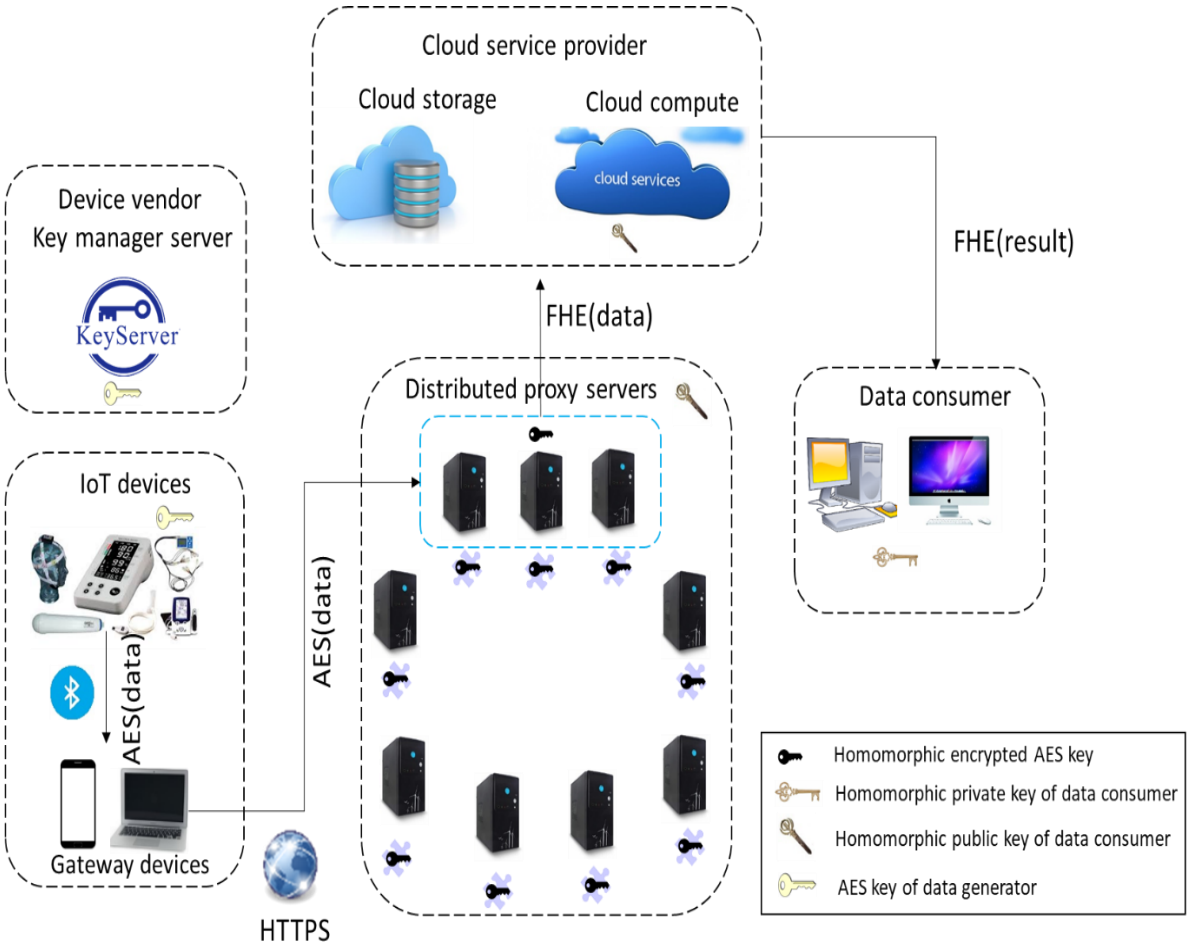

Figure 3. The proposed architecture for Proxy re-ciphering as a service

3.4 Model Assumptions

Given the number and nature of design goals, it was necessary to make few assumptions (listed below) which forms the basis of our system mode. Efforts were taken to ensure that these assumptions are as modest as possible.

1. Cloud servers and proxy servers are semi-trusted. They are honest in performing the algorithms but are not trusted with sensitive data
2. Every entity in the system has a unique and random identifier that is not publicly known.

3. The system chooses m proxy servers that are closest to the cloud servers for every organization, to provide the service. The value of m can differ for each business depending on the load like:
   - proxy server's total capacity
   - maximum server utilization per tenant
   - throughput guaranteed as per the service level agreement
   - incoming traffic load

   A load balancer is assumed to take care of traffic routing between the m proxy servers. Since the proxy servers are essentially cloud based servers, it is assumed that the system always has enough proxy servers to provide service.
4. The knowledge about an IoT device compromise is available to the system.

3.5 Data Distribution Schemes

The current study utilizes distributed proxy framework primarily to build a fault resilient system against corruption and availability of data and services in the system. The following techniques were inspected to select a scheme for distribution of data (which in the current case is device encryption keys encrypted under homomorphic scheme).

Data Replication

The traditional technique to guarantee a robust system is to replicate a server data across multiple servers (let's say, n). This provides a fault resilience [78] up to $n - 1$ servers. However, the secret in the current study is the expanded AES-128 homomorphic device key. The size of homomorphic ciphertexts generated using widely used homomorphic libraries like HElib [57] and SEAL [79] are typically not less than 1MB for one input [80]. Thus, when

considering a practical and a scalable solution, replication (across n servers) is not be the most efficient solution, in terms of storage complexity.

Threshold secret sharing

An informal definition of a threshold secret sharing can be described as follows:

Definition 1: Let k and n be positive integers such that $k \leq n$. A (k, n) Threshold secret sharing scheme is a way to share a secret S with n participants such that any group of at least k participants can pool their shares to compute the value of S, but no group of $k-1$ or fewer should be able to do so.

Thus in a (k, n) threshold secret sharing, a secret is split into n shares by a mutual trusted node (called dealer) and is shared among n participating nodes (called recipients) such that any k or more shares are enough to reconstruct the original secret but a fewer than k shares cannot reproduce the secret. In this scheme, k is called the threshold of secret sharing. In other words, a threshold secret sharing schemes provide a fault resilience up to $n - k$ shares.

Shamir's secret sharing scheme

Of the many techniques that are available in the literature, the most widely used threshold secret sharing technique was proposed by Shamir [81]. In this (k, n) scheme a secret message $'s'$ is being shared among n participating nodes in the following way:

The dealer constructs a secret polynomial $y = a(x)$ of degree $k - 1$ by randomly choosing coefficients $a_1, a_2, .., a_{k-1}$ from a finite field F and assigns $a_0 = s$.

$$a(x) = s + a_1 x + a_2 x^2 + \cdots + a_{k-1} x^{k-1}$$

Then the dealer picks out n random points on the polynomial, x_i , $1 \leq i \leq n$ and creates shares$(s_1, s_2 , .., s_n)$ to distribute share s_i to recipient r_i, such that

$$s_1 = (x_1, y_1 = a(x_1)) , s_2 = (x_2, y_2 = a(x_2)) , .. , s_n = (x_n, y_n = a(x_n))$$

To reconstruct the secret, a group of k recipients should pool their shares $(s_1, s_2, \ldots\ s_k)$ and construct Lagrange basis polynomials [82] $l_j\,(x), where\ 1 \leq j \leq k$,such that

$$s = \sum_{j=1}^{k} l_j\,(x) y_j$$

It can be inferred that a subset of authorized participants has a k-out-of-n access structure. Shamir's secret sharing scheme is a perfect secret sharing scheme (PSSS) where a subset of non-authorized participants gets *no information* about the secret and thus advantage of a non-authorized subset is same as the advantage of an outsider in computing the correct secret. Here, *no information* is defined in an information-theoretic sense.

Secret sharing schemes in general have one limitation in that the size of the shares are typically at least the size of secret itself. For example, Blakely's [83] have a share size that are k times larger than the size of the original secret. However, Shamir's shares have a *minimal* property in that the shares are only as large as the original secret. This is not a serious limitation in general where the secrets are typically small, like an AES-128 encryption key. However, in the current study, shares of homomorphic encrypted AES keys are distributed to the proxy servers. Thus Shamir's shares with a share-size of $|S|$ for a secret S [84], can incur a high storage complexity.

Krawczyk's secret sharing scheme

Krawczyk's scheme [85] is a space efficient hybrid scheme that utilizes PSSS and Information dispersal algorithm [86] to generate shares close to a size of $\frac{|S|}{k}$ where S represents the original secret. Krawczyk's scheme relaxes the secrecy from an information-theoretic sense to a computational sense, where an adversary has a bounded resource, making it more practical.

Information dispersal algorithm (IDA) proposed by Rabin introduces redundancy into a given data $Data$ before partitioning it into k fragments of size $\frac{|Data|}{k}$, such that, the interpolation of any k fragments will result in the reconstruction of $Data$. Unlike secret sharing schemes, secrecy of the data is inherently not a priority in IDA. In addition, this scheme assumes that the participants behave honestly, returning unaltered fragments for reconstruction. Krawczyk's scheme can be explained as follows:

Distribution scheme

1. Choose a secure private encryption scheme ENC and an encryption key Key.
2. Encrypt the secret S with ENC using Key, let $E = ENC(S, Key)$
3. Use IDA to split the encrypted data E into n fragments: $E_1, E_2, .., E_n$
4. Use PSS to generate n shares of Key : $Key_1, Key_2, .., Key_n$
5. Distribute shares S_i , $1 \leq i \leq n$ to each participant P_i where $S_i = (E_i, Key_i)$

Reconstruction scheme

1. Collect k shares $S_j = (E_j, Key_j)$ from participants P_j , $1 \leq j \leq k$
2. Reconstruct E from collected E_j using IDA
3. Compute Key from Key_j using PSSS
4. Decrypt E with function Dec using Key to recover S, $S = Dec\ (E\ , Key)$

The size of each share received by each participating node is $\frac{|Data|}{k} + |Key|$. Total size of the n shares is therefore $n.(\frac{|Data|}{k} + |Key|)$. Krawczyk's technique is thus a more efficient (since $|Key| << |Data|$) solution for the current study reducing the storage complexity by close to

$^1/_k$.The current study takes advantage of the following cryptographic schemes for data distribution:

IDA: Rabin's IDA scheme

PSSS: Shamir's secret sharing scheme

Private encryption scheme : ChaCha20 [87]

3.6 Proposed Workflow

A high-level rundown of Proxy re-ciphering as a service is shown in Figure 4 and Figure 5.The workflow can be segmented into four phases and the interactions between the entities in each phase is now explained in greater detail.

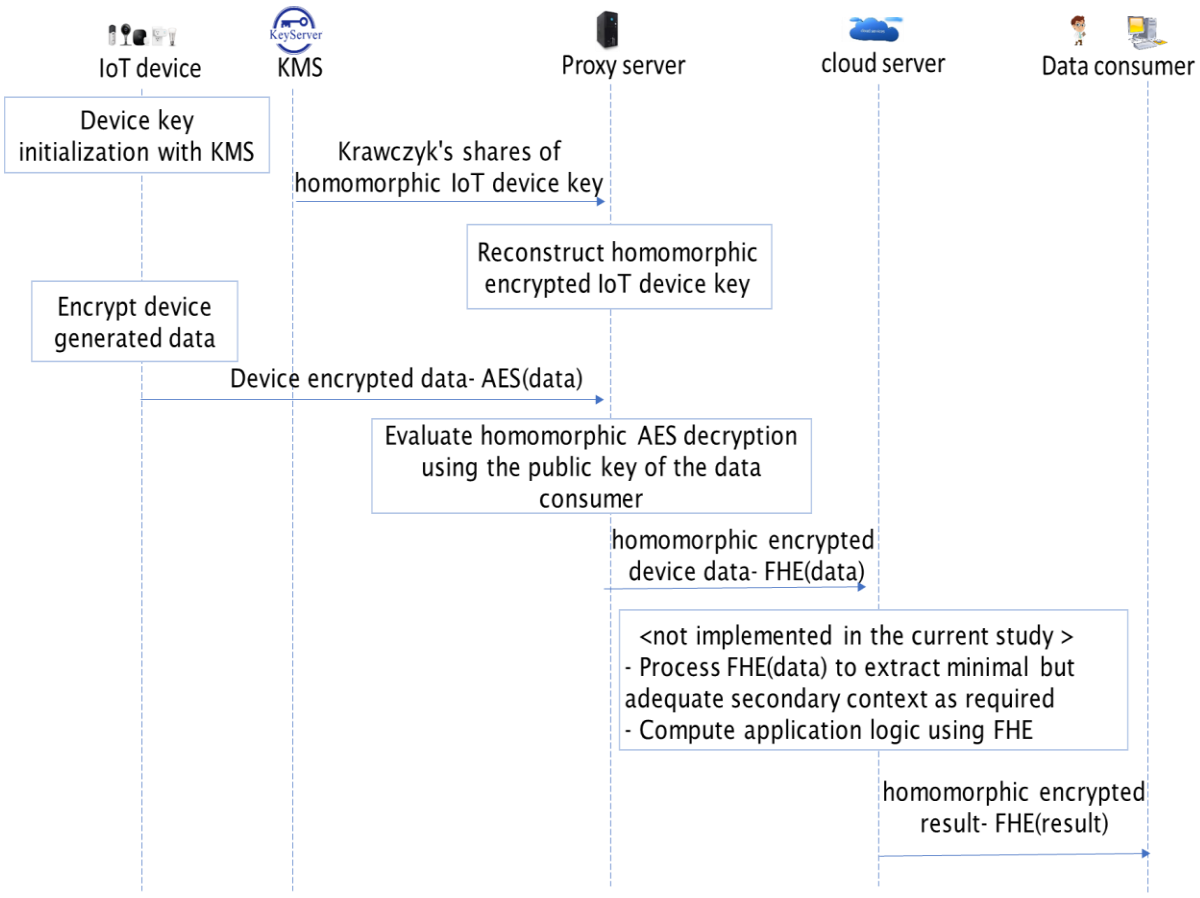

Figure 4. High level workflow of Proxy re-ciphering as a service-Phase I-III

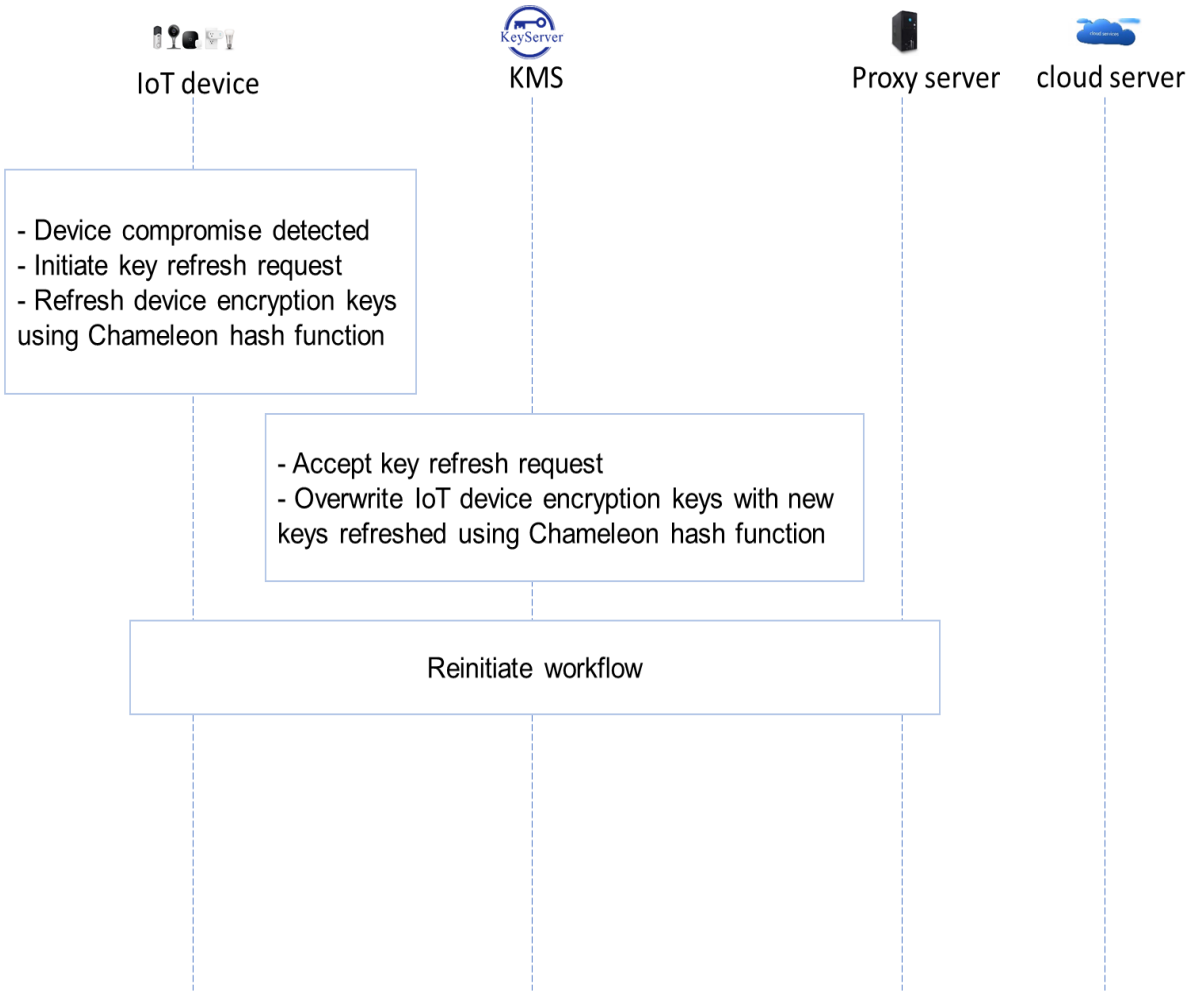

Figure 5. High level workflow of Proxy re-ciphering as a service-Phase IV

Phase I: System Initialization

Values for (k, n) are chosen for the entire system. A device-vendor generates a 128-bit AES device key (let's say k_{dev}) and initializes it in the IoT device. Every user in the system is provided with a unique homomorphic public-private key pair (pk_i, sk_i). The context information and the keys required for homomorphic encryption schemes are serialized and communicated to the proxy servers and the cloud compute servers. The system chooses m proxy servers for every participating business.

Phase II: Device setup

IoT device user (let's say, $user_i$) begins authenticating the account and registering the IoT device (let's say $device_{id}$) with the cloud service provider through the gateway device. After authenticating, the proxy servers are assigned as the new endpoint to the gateway device, by the cloud server. The gateway device now starts to share the data from $device_{id}$ with the cloud server through the proxy servers. During its connection with the gateway device, proxy server requests the key manager server for the device encryption key for $user_i$, encrypted under homomorphic scheme, by sharing public key pk_i as shown in Algorithm 1.

Algorithm 1
Secure sharing of IoT device key

Where: KMS
Input: $user_i$ and pk_i
Output: shares of device key k_{dev} encrypted under homomorphic scheme

1. Fetch AES device key k_{dev} corresponding to $user_i$

2. Encrypt AES device key k_{dev} using homomorphic encryption scheme:
   $\text{hom\_}k_{dev} <- \text{encryptAESkey}(k_{dev}, pk_i)$

3. Generate (k, n) shares for $\text{hom\_}k_{dev}$ using Krawczyk's scheme:
   Encrypt $\text{hom\_}k_{dev}$ using ChaCha20 with a random key $sskey$
   $E_i = Enc(\text{hom\_}k_{dev}, sskey)$
   Generate Shamir's shares for $sskey$
   Generate fragments for E_i using Rabin's IDA
   $shares_i = (E_i, sskey_i)$

4. Distribute $shares_i$ to n cloud proxy servers
5. Delete $\text{hom\_}k_{dev}$

Algorithm 1. Secure sharing of encrypted IoT device key

KMS runs Algorithm 1 as shown above, upon receiving a request from the proxy server. Homomorphic encryption of k_{dev} and generation of Krawczyk's shares were done using libraries provided in the literature ([57] ,[88] respectively). At the end of Algorithm 1, the KMS deletes hom_k_{dev}. This ensures that there is no additional storage-overhead for the KMS after executing Algorithm 1. This is an important step to ensure that it is practically possible for existing KMS infrastructures to run this algorithm with a modest computational and storage overhead. At the end of Algorithm 1, shares of hom_k_{dev} are distributed to n proxy servers.

Algorithm 2
Secure retrieval of IoT device key

Where: 'm' proxy servers providing service to $user_i$
Input: key shares $shares_i$=$(E_i, sskey_i)$
Output: reconstructed encrypted key hom_k_{dev}

1. Pool m shares : $shares_1$, $shares_2$, … $shares_m$
2. Request $n - m$ servers to contribute shares
3. Wait until at least $k - m$ shares ($shares_{m+1}, shares_{m+2}, \ldots shares_k$) are received
4. Reconstruct hom_k_{dev} using Krawczyk's scheme:
   Reconstruct E using interpolation
   Reconstruct $sskey$ using Lagrange's interpolation
   Decrypt E using ChaCha20 encryption with $sskey$
   hom_k_{dev} $= Dec(E, sskey)$
5. Store hom_k_{dev} with $user_i$

Algorithm 2. Secure reconstruction of encrypted IoT device key

Upon receiving the encrypted key shares, the m proxy servers providing service to $user_i$ run Algorithm 2 as shown above. To provide re-ciphering service to $user_i$, the m servers require hom_k_{dev}. . Hence, the m servers wait until at least other $k - m$ proxy servers provide

their shares $shares_j, 1 \leq j \leq k - m$. When at least k shares are pooled, each proxy server independently reconstructs hom_k_{dev} with Krawczyk's reconstruction scheme. The current study utilize the libraries from literature ([88]) to reconstruct hom_k_{dev}. After executing Algorithm 2, the m proxy servers have hom$_k_{dev}$ for $user_i$.

Phase III: Data flow

At this stage $device_{id}$ becomes functional and is ready to use. The device begins to record data (let's call it $data$) and encrypts it using an encryption scheme (let's say Enc) and k_{dev}. Upon encryption, $device_{id}$ transmits the encrypted data securely to the gateway device. When a proxy server receives the encrypted IoT device data from the gateway device, it is required to transform the ciphertext from Enc scheme to FHE scheme. This is possible by exploiting the homomorphic property of the FHE scheme. Additional details are discussed in [62] . The following paragraphs aim to explain the implementation of this homomorphic property for the current study.

For the sake of explanation, let us call the encrypted data transmitted by the IoT device as $data'$. Then $data' = Enc\ (data, k_{dev})$. According to homomorphic property of FHE, for any function f:

$$f\left(HE_{pk}\ (x), HE_{pk}\ (y)\right) = HE_{pk}((f(x, y))$$

Where, HE_{pk} () represents the fully homomorphic encryption under the public key pk.

Substituting $f = Dec, x = data'$ and $y = k_{dev}$ to evaluate a decryption function Dec corresponding to the encryption scheme Enc, produces:

$$LHS = Dec\left(HE_{pk}\,(data'), HE_{pk}\,(k_{dev})\right) => Dec\left(HE_{pk}\,(Enc(data, k_{dev})), \text{hom\_}k_{dev}\right)$$

$$RHS = HE_{pk}\left(Dec(data', k_{dev})\right) => HE_{pk}(Dec(Enc\,(data, k_{dev}), k_{dev}))$$

The RHS equation is $HE_{pk}(Dec(Enc\,(data, k_{dev}), k_{dev}))$, where the value within the parenthesis of $HE_{pk}\,(\ldots)\ is\ Dec(Enc\,(data, k_{dev}), k_{dev})$. This is a standard decryption function, yielding $data$ as the output. Thus, it can be inferred that $RHS = HE_{pk}(data)$. Now, equating LHS and RHS gives:

$$Dec\left(HE_{pk}\,(Enc(data, k_{dev})), \text{hom\_}k_{dev}\right) = HE_{pk}(data)$$

The above equations play a vital role in the current application as it explains that in order to transform $data$ encrypted by $device_{id}$ using Enc with k_{dev} to a FHE scheme at the proxy server, the proxy server is required to perform i) homomorphic encryption of $device_{id}$ generated ciphertext $data' = ENC(data, k_{dev})$ with pk and a ii) homomorphic evaluation of decryption function Dec corresponding to Enc with $\text{hom\_}k_{dev}$.Thus the computational overhead of proxy servers that affects the system parameters like utilization, throughput etc., depends on the cost of homomorphic encryption of $data'$ and homomorphic evaluation of Dec .

AES encryption in CTR mode is used as Enc scheme for the IoT device. This is chosen after comparing the performance of homomorphic evaluation of AES encryption in block cipher mode and stream cipher mode as explained in detail in Chapter 5.1. Although usage of stream ciphers instead of AES encryption in the stream cipher mode might give better performance, AES encryption is still chosen so that existing real-world implementations[65], especially the operational devices can adopt the proposed solution without requiring major changes.

Two key points to note when using AES encryption in the counter mode compared to block cipher mode are

i) The decryption function in CTR mode involves i) an *encryption* the counter values to generate a key stream and ii) a XOR operation of the keystream with the ciphertext. It is important to make advantage of this fact, because homomorphic evaluation of decryption function is typically slower (60% slower in [73]) than the homomorphic evaluation of the encryption function. Hence, with AES encryption in counter mode, it is enough to encrypt the counter values using homomorphic encryption scheme and XOR the homomorphic ciphertext with the device generated ciphertext, to transform the data from AES scheme to FHE scheme.

ii) The decryption function described above can be distinctly separated into two stages, a device-data dependent stage and a device-data independent stage, owing to its inherent construction mechanism. Typically, AES encryption is not computationally intensive for cloud servers that have abundant resources and specialized hardware. Hence this division is typically not mandatory. However, when performing a homomorphic evaluation of AES, it is critical to take advantage of this feature as it greatly improves the throughput of proxy servers, as discussed in Chapter 5.

Algorithm 3

Proxy re-ciphering

Where: Cloud Proxy server

Input: AES encrypted data $data' = Enc(data, k_{dev})$

Output: Homomorphic encrypted data

Offline pre-processing:

1. Fetch homomorphic encrypted key for $user_i$: hom_k_{dev}
2. Generate Homomorphic key stream using [3]:

$$HE_{pk}(counter) \coloneqq \text{HE-}AES\,(counter, \text{hom\_}k_{dev})$$

Online processing:

1. Encode AES encrypted data as homomorphic plaintext $encoded_{data'}$
2. XOR homomorphic encoded AES data with homomorphic encrypted keystream:

$$HE_{pk}(data) \coloneqq FHEAddConstant(HE_{pk}(counter), encoded_{data'})$$

3. Store $HE_{pk}(data)$ in cloud compute DB
4. Delete $data'$

Algorithm 3. Re-ciphering during data flow

When the proxy server receives AES ciphertext $data'$ from the gateway device, it runs Algorithm 3 shown above to transform $data'$ from AES scheme to FHE scheme. As discussed above, offline phase is the data independent phase and can be processed by the proxy servers before receiving $data'$. This is possible because, in most IoT applications, the synchronization interval between the IoT device and the servers are pre-defined [68]. During the online processing phase, the proxy server transforms $data'$ from AES-CTR scheme to FHE scheme by adding the homomorphic keystream $HE_{pk}(counter)$ with the encoded ciphertext

$encoded_{data'}$. The operations are performed in the current study are in Galois Field $GF(2)$ [62], [89] and hence a XOR operation is equivalent to an addition. After executing Algorithm3, the proxy server uploads the IoT device data encrypted under homomorphic encryption scheme $HE_{pk}(data)$, to the cloud servers, for secure computation. Optionally, if required, the proxy server can upload the metadata for IoT device's original ciphertext $data'$ to the cloud storage DB server, for auditing/logging.

Understandably, transforming every raw device data into homomorphic ciphertexts at the proxy servers can cause concerns about blow-up in the storage and bandwidth for cloud service providers. However, sharing homomorphic ciphertexts actively discourages cloud service providers from long-term storage of large amounts of raw device data, unnecessarily. Instead, it prompts the application developers to a) transform the raw homomorphic ciphertexts received from the proxy servers into a secondary useful knowledge/context that has minimal, yet adequate information required for the application and b) delete the rest of the raw data. In fact, studies like [110]- [111] have shown that end-user acceptance of IoT applications increase with continuous purging of raw device data by the service providers.

Phase IV: Dynamic key refresh

This is an additional phase that is initiated when i) a device compromise is detected where an adversary learns about the device encryption keys or ii) at a pre-defined frequency (6 months, 1 year etc.) to regularly refresh the encryption keys used by a device.

Unlike typical client-client or client-server key exchange protocols, the keys required for the current study are not ephemeral keys (like session keys) but are long-term secure device

specific encryption keys, between two known entities. Such a key refresh protocol is typically expected to guarantee the following properties:

- Uniqueness: the encryption keys should be unique for every device – KMS pair
- Correctness: the encryption keys should be correctly synchronized between the IoT device and KMS
- Security: the encryption keys should not be shared over the wireless channel
- Randomness: the encryption keys should have high entropy
- Post-compromise secrecy: the advantage of an adversary with a stolen key in learning about future/newly generated data should be minimal
- Latency: the communication complexity and the computation complexity should be minimal

The current work explores a new way of generating synchronized symmetric encryption keys with the help of chameleon hash functions [92]. Unlike regular hash functions , under specific conditions, Chameleon hash functions can produce hash collisions $CHAM(m_1, hk, r_1) = CHAM(m_2, hk, r_2)$, for $m_1, m_2 \in Z_q^*$, $m_1 \neq m_2$ and $r_1, r_2 \in Z_q^*$ where hk is the *hashing* key,. To generate a collision, chameleon hash functions require knowledge of a special *trapdoor* key x, such that manipulating the values of r_1, r_2 using: $r_2 \coloneqq \frac{m_1 + x.r_1 - m_2}{x} \; q$ can guarantee $CHAM(m_1, hk, r_1) = CHAM(m_2, hk, r_2)$ [92].

The gateway device runs Algorithm 4 when it receives a key request from the device. The current work does not focus on implementing the logic to verify a key request. Libraries from literature ([93]) are used to generate chameleon hashes and chameleon collisions in Algorithms 4, 5 and 6. The gateway generates a random id ($rand_{id}$) using a computationally secure pseudo random generator (CSPRNG).The gateway XORs $device_{id}$ with $rand_{id}$ for the IoT device to produce $m_1 = \text{XOR}(\ device_{id},\ rand_{id})$. It sets $m_2 = \ KMS_{id}$, where KMS_{id} is the identifier for the KMS responsible for $device_{id}$. Thus for $m_1 \neq m_2$, the gateway device finds two random and independent values $r_1\ and\ r_2$, using the following equation :

$$r_2 \coloneqq \frac{m_1 + x.r_1 - m_2}{x}\ q$$

where x is the trapdoor of the chameleon hash function (stored securely at the gateway) and r_2 is called the *chameleon-collision* [92].

Algorithm 4

Finding chameleon collision at gateway

Where: IoT gateway device

Input: trapdoor key $tk = x \in Zq, KMS_{id}, device_{id}$, prime number q such that $p = kq + 1$ where p is random large prime number

Output: Chameleon collision r_2

1.Verify key refresh request

2. Generate a random number $rand_{id}$ using CSPRNG

3. One time pad the $device_{id}$ with $rand_{id}$, $m_1 =$ XOR($device_{id}$, $rand_{id}$)

4. Generate random number r_1 for hashing m_1

5. Send r_1 and $rand_{id}$ to IoT device

6. Store KMS_{id} as m_2

7. Find a Chameleon collision r_2 using $m_1 + x.\ r_1 = m_2 + x.\ r_2\ mod\ q$

8. Send r_2 to KMS

Algorithm 4. Finding chameleon collision at gateway

Algorithm 5

Dynamic key generation at IoT device

Where: IoT device

Input: random prime numbers $p,\ q$ such that $p = kq + 1$, hash key $hk = y = g^{x} mod\ p$, random numbers $rand_{id}, r_1$

Output: Chameleon hash as encryption key

1. One-time pad of $device_{id}$ with $\boldsymbol{rand\_id}$ to maintain entropy

$$m_1 = \text{XOR}(\ device_{id},\ rand_{id})$$

2. Compute Chameleon hash using [5]

$$key' = CHAM(m_1, hk,\ r_1)$$

3. Store $\boldsymbol{key'}$ as encryption key

Algorithm 5. Dynamic key generation at IoT device

At the end of Algorithm 4 , the gateway device shares $rand_{id}$ and r_1 with the IoT device and r_2 with the KMS such that the IoT device and the KMS can run Algorithm 5 and Algorithm 6 respectively. After receiving $rand_{id}, r_1$ from the gateway device, the IoT device runs Algorithm 5. At the end of the algorithm, the device generates a key (let's call it key') where

$$key' = CHAM(m_1, hk,\ r_1) = g^{m1} y^{r1}\ mod\ p\ .$$

Algorithm 6

Dynamic key generation at KMS

Where: KMS

Input: random prime numbers $p,\ q$ such that $p = kq + 1$, hash key $hk = y = g^x mod\ p$, random numbers r_2

Output: Chameleon hash as encryption key

1. Store UUID of server KMS_{id} as m_2

m_2=KMS_{id}

2. Compute Chameleon hash using [5]

$key''= CHAM(m_2, hk, r_2)$

3.Store key' as encryption key

4. Run Algorithm 1

Algorithm 6. Dynamic key generation at KMS

After receiving r_2 from gateway, KMS runs Algorithm 6 and generates a key (let's call it key'')

$$key''= CHAM(m_2, hk, r_2) = g^{m2} y^{r2}\ mod\ p\ .$$

By the property of chameleon collision[92] using trapdoor, it can be inferred that the key generated by the IoT device(key') and the KMS (key'') are the same.

$$CHAM(m_1, hk, r_1) = CHAM(m_2, hk, r_2)$$

After generating the new device key, the KMS runs Algorithm 1 once again, to the encrypt the new device key under homomorphic scheme and distribute the key shares to the proxy servers. This is one of the salient features of the current work as it enables the system to continue Phase III-data flow phase, even post compromise. Chapter 5 discusses the security properties of this phase in more detail.

CHAPTER 4. CASE STUDY: SMART HEALTHCARE

This Chapter focuses on the need for secure and privacy preserving cloud computations for a specific real-world IoT application: smart healthcare. First, the chapter discusses the importance of IoT and cloud computing for quality medical assistance. Finally, the chapter discusses how *Proxy re-ciphering as a service* can help healthcare organizations benefit from cloud computing without compromising their patient privacy for IoT device data, with minimal overhead.

4.1 Internet of Things in Healthcare

IoT market has been witnessing a steady increase in demand and investment for healthcare services that prioritize improving the quality of living, by shifting focus from diagnosis and treatment to monitoring and prevention. Smart healthcare applications, which are an integral part of IoT-healthcare, combine embedded medical devices that record vital signs and communicate them to the cloud servers, allowing remote medical assistance and care. Continuous glucose monitoring [94], insulin pens [95], smart inhalers [96] and ingestible sensors [97] are some examples of smart healthcare products that promise to improve people all over the world. Thus, it comes as no surprise that IoT-healthcare market has a projected value of 348 Billion USD by 2025 [98].

In June 2018, a report of a clinical trial consisting of 357 patients undergoing head and neck cancer treatment was presented at the American Society of Clinical Oncology (ASCO) Annual meeting. The report compared the effects of treatment and the severity of symptoms witnessed by two groups of patients. The first group consisted of patients (169 people) that used a Bluetooth powered weight scale and a blood pressure cuffs with a symptom tracking app, called CYCORE, with daily updates to their physicians. The second group followed the

traditional way of treatment with regular weekly visits to their physician, without any additional monitoring [99]. At the end of the trial, it was identified that the first group showed better improvement to their treatment and experienced less severe symptoms along the way, compared to the latter.

In 2013, it was estimated that incomplete medical histories and insufficient/lack of patient data contributed to 400,000 deaths a year [100]. Furthermore, American Cancer Society estimated that around 51540 new head and neck cancers will be diagnosed in 2018, with over 10000 fatalities [101]. These alarming numbers, along with the report presented at the ASCO annual meeting compel the necessity to shift towards the digital healthcare era where medical IoT devices can help improve the quality of assistance, even for chronic conditions. There is also a push from Government to digitize health record to improve interoperability of patient's health record [102].

In a digital healthcare world, an electronic medical record (EMR) is the equivalent of a patient's medical record. It is maintained by a heath care provider and is updated constantly to include all relevant and important information like name, age, address, health concerns, administered medications and their dosages, vital signs and any laboratory reports. With Smart healthcare, an effort towards comprehensive medical care thus includes continuous monitoring of patients' vital signs from medical IoT devices and recording them as a part of EMR. Such an overarching medical data collection and processing can potentially provide new dimensions to a patient's overall health profile, by the virtue of the enormous and continuous data recorded by the medical IoT devices.

However, the adoption and transition towards digital healthcare world can be challenging and expensive for healthcare providers. Close to 25% executives from over 1800

hospitals associated transitioning cost as their primary concern while adapting to the digital era [103]. Cloud computing is a promising solution that can ease this transition for the healthcare organizations by allowing them to outsource the storage and processing of medical data to cloud vendors like Google Cloud Healthcare [104].

4.2 Smart Healthcare and Cloud

Cloud service providers usually operate on pay-per-use model. Thus, offloading the storage and processing of medical records to the cloud servers can help healthcare organizations eliminate the need to maintain expensive data centers, hardware equipment and IT personnel. Cloud computing for healthcare is an active research topic, that aims to modernize US healthcare system [102].

Cloud-based smart heath care solutions typically involves three phases: 1) *Data generation*, where the medical data is recorded either manually by a physician/nurse or by a medical IoT device 2) *Data storage*, where the data collected from the patient is uploaded to a cloud storage service like Amazon S3, for future access 3) *Data computation*, where cloud applications process the data to extract intelligence from the stored data. To provide a holistic service, during phase 3, cloud service providers can potentially make use of data driven services like analytics and machine learning [104] to process the medical data coming from multiple sources (heart rate monitors, fitness bands, EMR, insulin pumps etc.) and correlate the data points, for a patient. Such a technique can provide a comprehensive view about the patient's overall health. The availability of such multi-faceted medical reports can help doctors deliver high quality medical assistance to their patients. To this end, in 2018, Fitbit acquired a company named *Twine Health* and announced a collaboration with Google cloud to combine Fitbit data with EMR aiming to accelerate innovation in digital health [105]. Twine health has

proven to help people manage chronic conditions like diabetes and hypertension with their remote health coaching platform[106].

4.3 Security for Healthcare

In 2017, the healthcare industry witnessed twice as many attacks as any other industry [107] , forcing healthcare providers to improve their cybersecurity hygiene. The Health Insurance Portability and Accountability Act (HIPAA) has also laid down rules and regulations to prevent Personal Health Information (PHI) from misuse. PHI almost same as EMR but are primarily designed to be used by patients, while the latter is primarily designed to be used by healthcare providers. PHI and EMR are valuable in the underground black market since an identity theft from medical data gives a perpetrator the chance to receive health care through channels like Medicare, buy pharmaceuticals and commit insurance fraud. In 2016, FBI estimated that health information is valued close to $60-$70 on the black market while a Social security number is valued close to $1 in the same underground market [108]. Figure 6 shows an overview of the number of complaints due to medical identity theft issues from 2013-2017 across different states in the US [109]. It becomes imperative to make sure that PHI/EMR is secure and protected against data breaches and leakage.

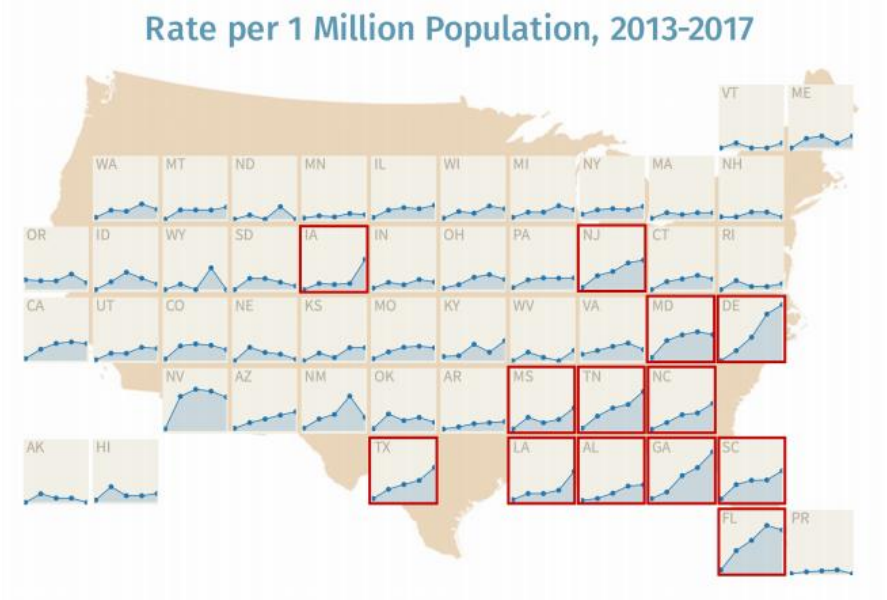

Figure 6. Overview of complaints regarding medical identity thefts from 2013-2017 [110]

A natural concern for healthcare organizations when outsourcing medical data to cloud service providers is protecting patients' privacy. In addition, consumers are concerned that small privacy invasions by cloud service providers may eventually result in loss of civil rights[73]. As discussed in Chapter 1, encrypting the medical data before outsourcing them to cloud can assuage these concerns. However, as explained in Chapter 2, this precludes cloud service providers from performing any meaningful computations on the medical data. FHE schemes can address this concern, but as explained in Chapter 2 and Chapter 3, there are limitations in the works so far proposed in the literature which can inhibit real-world adoption of FHE schemes for healthcare industry.

4.4 Proxy re-ciphering as a service for Healthcare IoT

In this section, we will consider a realistic scenario to explain the deployment of the proposed framework in a health care service provider company. Here we will focus on a healthcare company that provides healthcare assistance to hundreds of thousands of patients

across US. Consider a situation where they have been witnessing an exponential growth in the number of remote health monitoring devices like ECG monitors, smart inhalers, defibrillators etc., among their patients. This healthcare provider realizes that delivering a quality assistance to patients will require them to evaluate the overall healthcare profile of each patient and this includes in-house data about vital signs created manually by nurses and those received from the medical IoT devices. The executive management team along with IT department personnel at the company will have to evaluate the options they have in order to address this situation:

1. Solution A: Invest in a bigger in-house datacenter. This requires them to invest in additional hardware equipment's, procure land space, and hire more IT personnel to tackle the installation and maintenance.
2. Solution B: Utilize cloud service providers to store encrypted patient data.

While solution A guarantees data security and privacy, it can quickly reach bottlenecks, since it is not a scalable solution. While solution 2 guarantees data security and privacy, the solution cannot provide privacy-preserving data insights that healthcare providers require, for quality assistance.

Contoso Inc., a (fictional) third-party company, can use the proposed framework to provide *Proxy re*-ciphering *as a service* as shown in Figure 7 . Contoso Inc. promises to act as a secure intermediary between hospitals across US like the healthcare service provider company considered in this scenario and their cloud service providers. Contoso guarantees that it can securely enable health care providers to use the cutting-edge cloud technologies from the hospital's choice of cloud service provider with a modest overhead to the hospital and to its patients. Under these circumstances, solution B is certainly a viable option for the healthcare organizations to benefit from cloud computing while preserving the privacy of its patients.

Thus, the workflow can effectively have multifold gains for the end-users and businesses like the health care organization considered in this scenario :

i. First, cloud service providers do not have access to raw device data, eliminating potential sharing and secondary usage of the device data.

ii. Second, a device key compromise or revocation does not render data in the cloud vulnerable and does not require rekeying [53] of the entire past data. This is because, the secondary and processed homomorphic data in the cloud servers are transparent to the encryption key used by the devices. In addition, a device key refresh operation to protect future data does not affect the functionality of the cloud servers or the information stored by them.

iii. Finally, with cloud servers storing the processed homomorphic context derived from the raw device data, the storage blow-up can be minimized to an acceptable overhead. This can in turn foster real-world adoption of FHE based solutions by the cloud service provider thereby enabling more users and businesses.

In a nutshell, when cloud providers only store a minimally required homomorphic data derived based on application logic from the raw device data, end-users can be comfortable about not losing data control to the cloud service providers and can be confident that their device data is secure throughout its lifecycle. A past raw device data is not rendered vulnerable with a key compromise since the service providers store only the derived data encrypted under the homomorphic scheme. In addition, by the virtue of key refresh, a newly generated device data cannot be decrypted by an attacker with the stolen past keys. Chapter 5 discusses the security properties of the framework in greater detail.

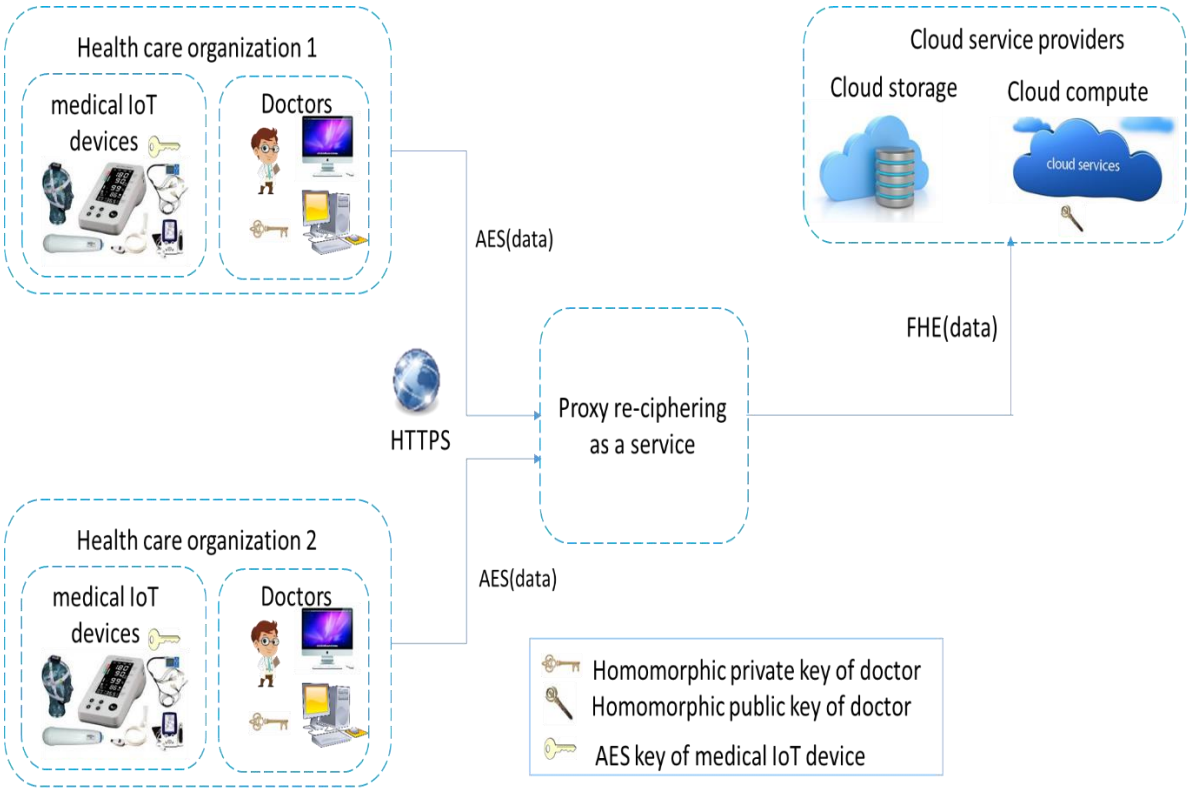

Figure 7. An example of proxy re-ciphering as a service framework for smart healthcare

4.5 Deployment Scenarios

Studies like [73] have shown that end-users are concerned about the usage of their data, with 89% of the surveyed consumers expressing discomfort with third party service providers accessing their information without consent and with 72% feeling that sharing their personal information may have more demerits than benefits. The sample narrative mentioned in the previous section considers a deployment scenario where Proxy re-ciphering as a service is entirely managed by a third-party company named Contoso Inc. However, the service can be hosted by different entities and this section discusses their merits and demerits. The current study identifies four different deployment scenarios:

1. The KMS, the distributed proxy servers and the cloud servers are hosted by three different companies as discussed in previous section. For example, KMS could be hosted by Fitbit Inc., distributed proxy servers could be hosted by Contoso Inc. (mentioned earlier), and cloud servers could be hosted by Microsoft Azure.
2. The KMS and the proxy servers are hosted by the same company while cloud servers are hosted by a different company.
3. The distributed proxy servers and the cloud servers are hosted by the same company while the KMS is hosted by a different company.
4. The KMS, the distributed proxy servers and the cloud servers are all hosted by the same company.

An ideal workflow will be one where a customer does not have to trust any service providers (KMS/ proxy server/ cloud server) with their data and yet receive privacy preserving services. In the current work KMS manages the device encryption keys, the proxy servers store the homomorphic device encryption keys and the public key of the recipient and the cloud servers have access to the public key of the recipient. Thus, the following section views each entity as a potential adversary from an end-user's perspective and lays out the merits and demerits of each scenario.

Scenario 1

If the KMS, proxy servers and the cloud servers do not collude, no entity has enough information to decrypt the device data. The bandwidth required to share device encryption key shares from KMS to the proxy servers can be high, but this is a one-time cost. The bandwidth required to share homomorphic data from proxy servers to cloud servers can be high depending on the location of the servers.

This can be a recommended scenario since it is privacy preserving, however the communication latency during data flow phase can be high if the proxy servers and cloud servers are in different regions.

Scenario 2

The KMS and proxy servers can together decrypt the device data. This is because, proxy servers receive the encrypted device data and the KMS have the access to the device keys. The bandwidth required to share the device encryption keys can be reduced with this configuration. The bandwidth required to share the homomorphic ciphertext from the proxy servers to the cloud servers can be high depending on the location of the servers.

This is not a recommended scenario since this requires the end users to fully trust the company hosting KMS and the proxy servers.

Scenario 3

If the KMS does not collude the company hosting proxy servers and cloud servers, no entity has enough information to decrypt the device data. The bandwidth required to share device encryption key shares from KMS to the proxy servers can be high, but this is a one-time cost. The bandwidth required to share homomorphic data from proxy servers to cloud servers can be reduced with this configuration. If proxy servers remain semi-honest and do not store the device encrypted data, a device key compromise do not render device data vulnerable. This is a recommended scenario since it protects users' privacy with reduced latency during data flow phase.

Scenario 4

The company hosting all the services (the KMS, the distributed proxy servers and the cloud servers) has a complete control over the device data and thus can decrypt or store the

raw data, as required. The bandwidth required to share device encryption key shares from KMS to the proxy servers can be reduced with this configuration. The bandwidth required to share homomorphic ciphertext from the proxy servers to the cloud servers can be reduced with this configuration.

However, this requires an end-user to completely trust the service provider and hence is not recommended.

CHAPTER 5. PERFORMANCE EVALUATION

This chapter focuses on evaluating the feasibility and utility of the conceptualized system model in the real-world by applying it to the case study performed in Chapter 4. First, the chapter discusses the performance observed using an experimental testbed. Later, the chapter presents the security properties of the framework theoretically.

5.1 Experimental Evaluation

5.1.1 Experimental Setup

A basic IoT testbed was developed to simulate the problem scenario explained in the previous chapter. The testbed architecture consists of a gateway device, the distributed proxy servers, the cloud server, the end-user device and the KMS. The following platforms were used for the implementation: i) Android Google pixel 2 emulator with 2 GB RAM as a gateway device ii) Dual core 8 GB Ubuntu 14.04 LTS virtual machines (VM) hosted on Microsoft Azure as the proxy servers and the cloud compute server iii) Microsoft Azure blob storage as a backend storage for the cloud compute server and a iv) Single core 1GB Ubuntu 14.0 LTS VM hosted on Microsoft Azure as an end-user device and the KMS.

The current study uses ECG records from TELE ECG Database [110] and activity records from FITBIT [111] servers as the medical IoT data source. An android app running on the google pixel emulator acts both as a medical IoT device and as a gateway. The app fetches the data from the databases and encrypts them using AES encryption in the counter mode before sending it to the proxy servers. The former database consists of ECG data recorded using TeleMedCare Health Monitor (TeleMedCare Pty. Ltd. Sydney, Australia). It consists of 300 ECG single lead-I signal recordings where 250 of them were selected randomly from 120

patients and 50 of them were manually selected from 168 patients to include poor quality data in the sample population [110]. The sampling frequency for TELE database was 500 Hz with voltages ranging from 5.556912223578890 to -5.554198887532222 mV. Each record in the database is available as a comma separated value (CSV) file. Each line in the CSV file contains the ECG sample value in mV, among other values. The latter data source was generated using a Fitbit Charge 2 tracker. Fitbit APIs are available for developers and researchers to access data collected by Fitbit devices. However, access to intraday time series values are directly available only for personal use [111]. Among the several data points collected by the Fitbit device, the current study uses the endpoints for heart rate time series and daily activity time series. For example, the API request to get step values for a specific date with a 15-minute interval value will be: $makeApiRequest(\text{“}user/-activities/steps/date/\text{”} + search\_date + \text{”}/1d/15min.json\text{”})$;

The current study uses the following libraries: HElib [57] , Archistar [88] , CloudCrypto [93] and default Java crypto package to execute the algorithms discussed in chapter 3. The current study uses the default parameter values available in the libraries. However, when implementing a cloud-based application, depending on the application logic, the values of the parameters in the libraries should be chosen appropriately.

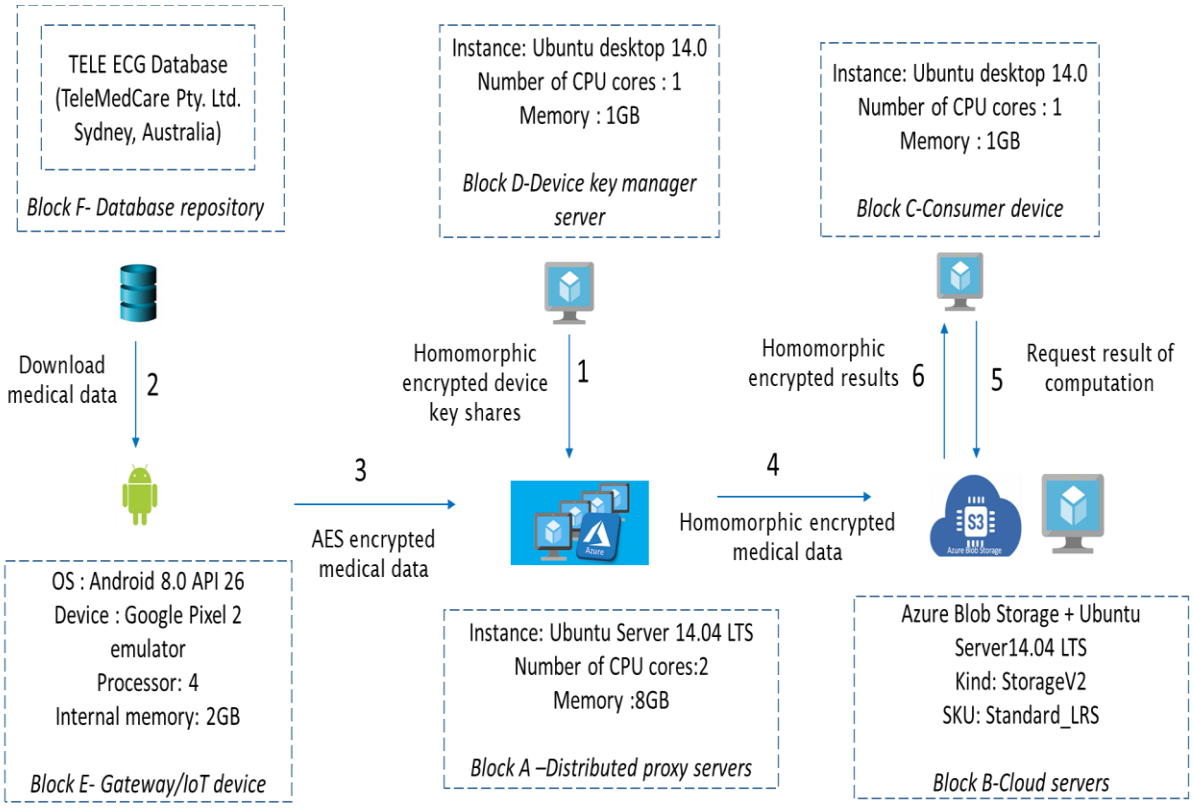

Figure 8. Experiment setup with high-level workflow

Figure 8 aims to explain the interactions between various devices/platforms (mentioned above) within the proposed model.

Phase I- Phase III

The following steps correspond to the workflow shown in Figure 8.

1. -VM in Block-D (KMS) encrypts the AES encryption key used by the Android device under BGV homomorphic encryption scheme using HElib library and distributes the Krawczyk's shares generated using Archistar library to all the VMs running in Block-A (proxy servers)

   -VMs in Block-A (proxy servers) communicate with each other to reconstruct homomorphic encrypted Android device key from the Krawczyk's shares using Archistar library

2. Android emulator (acting as medical device) fetches data from one of the databases and encrypts it using AES encryption in CTR mode using its device key
3. Android emulator (acting as gateway) sends the AES encrypted data to a VM in Block-A
4. -A VM in Block-A (proxy server) receiving the ciphertext (from step 3) transforms the ciphertext from AES encryption scheme to FHE scheme by evaluating the homomorphic decryption function with off-line pre-processing using HElib

   -The VM uploads the homomorphic encrypted ciphertext to Azure blob storage (Block-B).

   -A VMs in Block-B (cloud servers) performs privacy-preserving computation on the data in blob storage <not implemented in the current study>
5. A VM in Block-C (end user) requests result of computation (performed in step 4). from the VM in Block-B
6. -VM in Block-B shares homomorphic encrypted result to VM in Block-C

   -VM in Block-C decrypts the homomorphic results using its private key and views the plaintext result

   Phase IV

   The following steps are not shown in Figure 8 for simplicity.
7. Android emulator (acting as gateway) shares a random number with VM in Block-D (KMS)
8. Android emulator (acting as medical device) and the VM in Block-D execute chameleon hash function using CloudCrypto library to generate a dynamic encryption key.

5.1.2 Performance Evaluation

The current system uses a distributed proxy framework primarily to overcome the loss of availability of service and loss/corruption of device keys in the proxy servers. Since the

system stores homomorphic encrypted shares of keys instead of the actual keys, confidentiality of the encryption keys is guaranteed. The parameters used in the key sharing model are m, k, n , where, m is the number of proxy servers providing service to a healthcare organization, k is the threshold or the minimum number of proxy servers required to reconstruct the homomorphic encryption key for every device and n is the total number of proxy servers available in a given geographical region. The relation between these parameters are $m < k < n$ in the system and more specifically we require that $n - m \geq k$. This guarantees system resiliency which is explained in Chapter 5.2. In the following experiments, all the traffic generated by medical devices belonging to a specific healthcare organization are handled by a single proxy server ($m = 1$). $< k, n >$ are the tunable system parameters. Hence, the important aspects of the problem: values for $< k, n >$ and its impact on the performance are first studied.

The threshold sharing scheme says that, the difficulty in collapsing a system is directly proportional to the value of n, for a given value of k. It is implied from Krawczyk's scheme that the size of the shares distributed to each proxy server is inversely proportional to the value of k. Thus, it is desirable to have larger values for n, k as they provide better security and storage complexity. However, it is essential to analyze the impact of k, n on the computational and communication latency, before finalizing the values. Both these latencies are calculated at the KMS as well as at the proxy server. The computational latency is calculated at the KMS based on the time taken to construct the key shares and at the proxy server it is based on the time taken to re-construct the key after receiving the shares. The communication latency at the KMS is defined as the time taken to distribute the shares to n proxy servers, and at a proxy server, it is defined as the time taken to receive $k - 1$ shares for reconstruction. Since the exact impact of k, n on the above-mentioned latencies is not clearly evident, it is analyzed using the

testbed. The evaluation consists of a series of experiments conducted with 10 trials. Each trial consists of generating a 128-bit AES key and encrypting it under homomorphic scheme. The performance of computation and communication latency is observed for values of $n = 5,7,9,11,13,15$. The reason for choosing odd values of n is explained in Chapter 5.2. For every value of n , the performance is tested for different values of k, to understand the behavior of the system. For the purpose of this experiment, the clock in all the servers were synchronized to Central Standard Time. Any clock skews or differences in the clock synchronization is negligible and hence is ignored for the calculation purposes.

Effect of threshold secret sharing parameters <k,n> on the total latency during key distribution at KMS

The first evaluation comprises of analyzing the time taken at the KMS to distribute the IoT device key encrypted under the homomorphic scheme. The total time taken at the KMS is the sum of the computation latency and communication latency, as explained earlier. Figure 9 shows the behavior of the total time for various values of $< k, n >$. It is observed that for a given value of n, as the value of k is increased, the computational latency increases. However, it is also observed that as the value of k increases, the size of each share decreases. Thus, the communication latency decreases. Since the communication latency is much higher than the computation latency, it is observed that as the value of k increases, for a given value of n, the overall time taken reduces. Along the same lines, it is observed that, for a given value of k, as the value of n increases, there is an overall increase in the size of the data $\left(n.\left(\frac{|Data|}{k} + |Key|\right)\right)$ that is distributed. Thus, increasing the value of n results in an overall increase in the time taken.

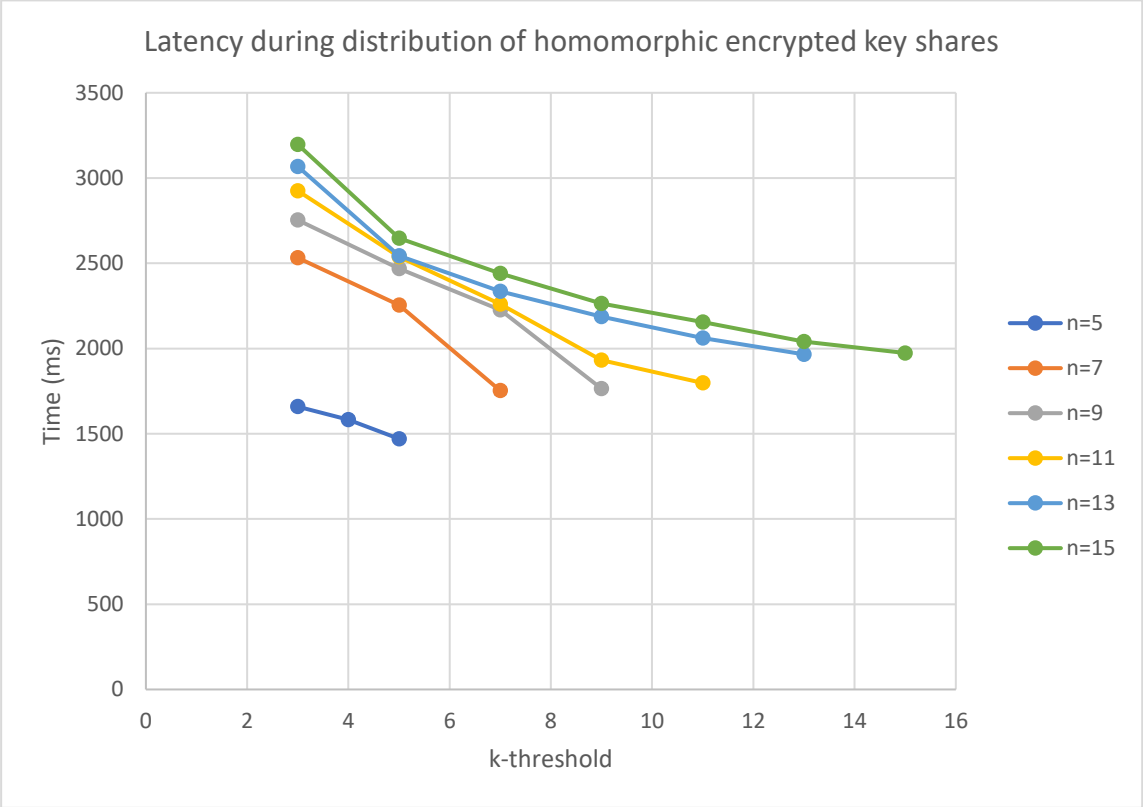

Figure 9. Evaluation of latency to distribute Krawczyk's shares of homomorphic encrypted device key

Effect of threshold secret sharing parameters <k,n> on the total latency during reconstruction at Proxy server

Next, the total latency to reconstruct the homomorphic key at a proxy server is studied. The total latency is the sum of communication latency and computational latency for the proxy server. From Figure 10 it can be observed that when the value of k increases, the communication latency increases. This is because, a proxy server needs $k-1$ server shares to reconstruct a device key, and thus when the value of k increases, total amount of shares/data required for reconstruction increases, thus increasing the communication latency. Also, from the experiments it was observed that increasing the value of k and n increases the computation

time to reconstruct the homomorphic key. Thus, the total latency, which is a sum of the two latencies, increases as the values of k and n are increased.

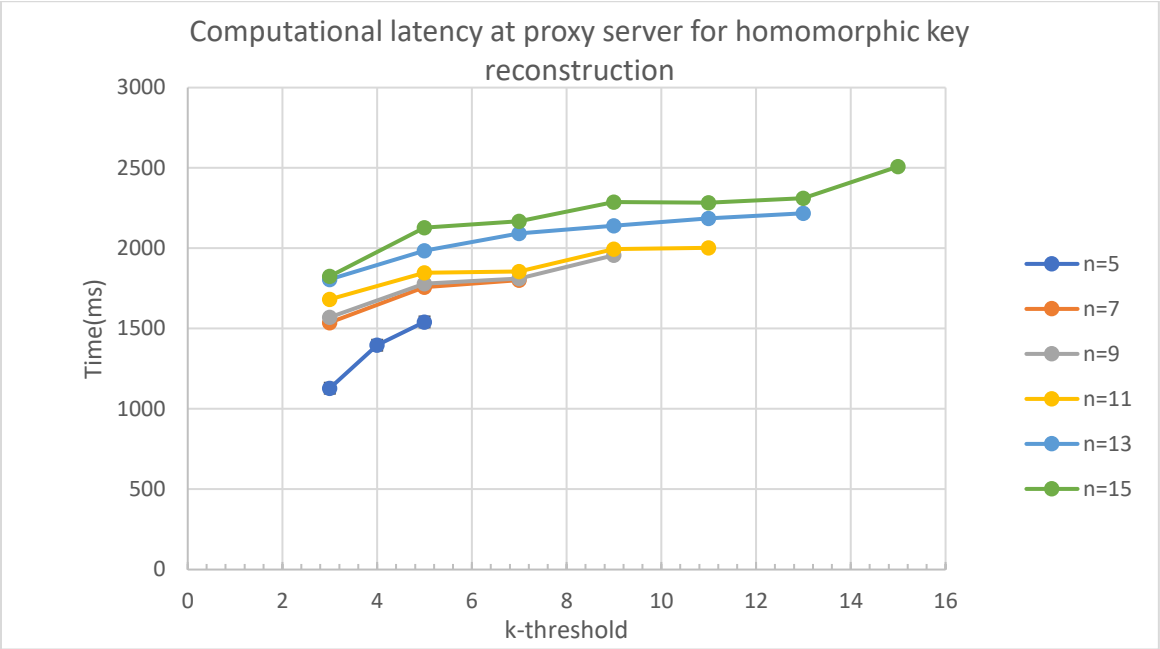

Figure 10. Evaluation of latency at each proxy server for homomorphic key reconstruction

Comparison of Krawczyk's computational secret sharing vs Shamir's perfect secret sharing during key distribution at KMS

Next, the study evaluates the merits and limitations of Krawczyk's, and Shamir's secret sharing scheme and explains the rationale behind the choice made in the current study. Figure 11 shows the latency at the KMS when using Shamir's scheme instead of Krawczyk's, during key distribution. It is observed that as the value $of\ k$ increases, the computation latency increases. It is also observed that as the value of n increases, the communication latency increases. This may be because, when k increases, the size of the secret polynomial increases and when n increases the number of shares (and thus the total size of data transmitted)

increases. Hence, the overall latency which is a sum of the two latencies increases as the value of k, n are increased. Given that the frequency of key distribution operation is not high in the current framework, the latency overhead of Shamir's scheme compared to Krawczyk's may not be significant. However, in the current study, the secret that is shared is the AES device key encrypted under the homomorphic scheme. The default parameter values in HElib library yielded a homomorphic encrypted AES key of size close to 3MB, for a 128-bit AES device key. Thus, when storing a homomorphic encrypted key share compared to the actual key, Shamir's scheme experiences a memory expansion of close to $\frac{n.|Hom-key|}{|key|} = \frac{n.|Hom-key|}{|key|} \cong n.\frac{3.2\text{x}10^6 bytes}{16\ bytes} \cong n.10^5$. Compared to this, Krawczyk's scheme witnesses a memory expansion of close to $\frac{n.(\frac{|Hom-key|}{k}+|sskey|)}{|key|} = n.\frac{\frac{|Hom-key|}{k}+|sskey|}{|key|} \cong \frac{\frac{3.2\text{x}10^6}{k}+16\ bytes}{16\ bytes} \cong \frac{n.10^5}{k}$. Thus, for higher values of k, Krawczyk's scheme provides better storage efficiency. This is the reason why despite its merits regarding *perfect security*, Krawczyk's scheme was chosen over Shamir's scheme for the threshold secret sharing scheme.

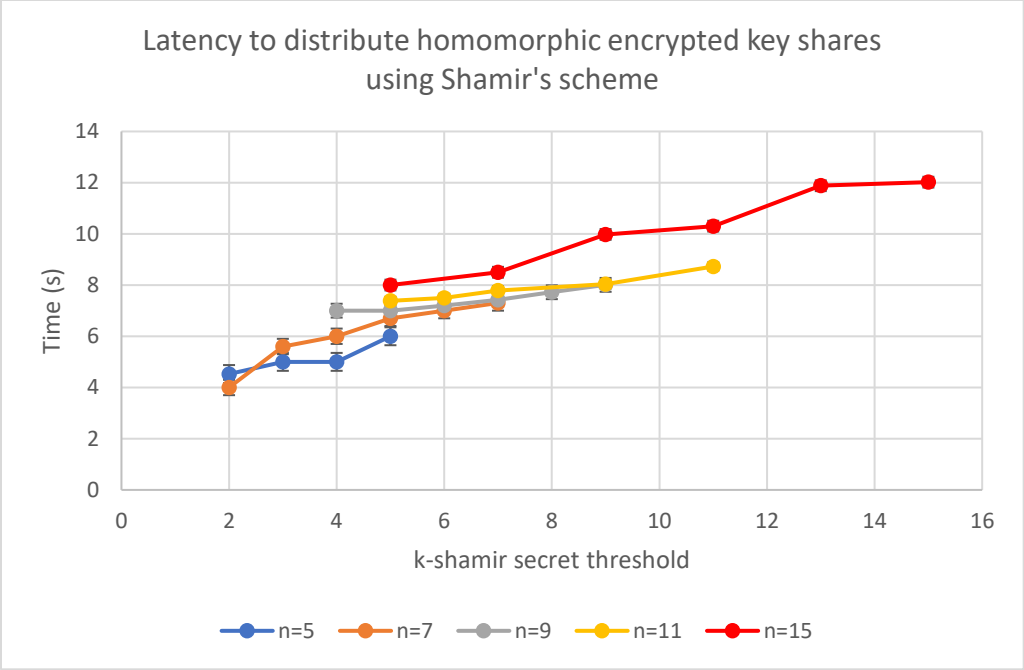

Figure 11. Evaluation of latency to distribute Shamir's shares of homomorphic encrypted device key

Trade-off between privacy and latency in encryption schemes

Next, the study evaluates the computation cost of using fully homomorphic encryption schemes compared to the traditional schemes. To do this, the current study chooses 4 approaches that allow cloud servers to compute on the IoT data: 1) share device data without encryption , 2)share device data encrypted under AES scheme in block cipher mode such that it can be decrypted using AES and re-encrypted using a fully homomorphic encryption scheme , 3) share device data encrypted under AES scheme in block cipher mode and perform homomorphic evaluation the AES decryption 4) share data encrypted under AES scheme in stream cipher mode and perform homomorphic evaluation the AES decryption. The total latency measured is the sum of time taken to send a 32-byte data from the Android emulator to the proxy server VM and the time taken at the proxy server VM to prepare the data for cloud

computation. Figure 12 displays the total time taken during each approach. The current study explains the observed behavior below.

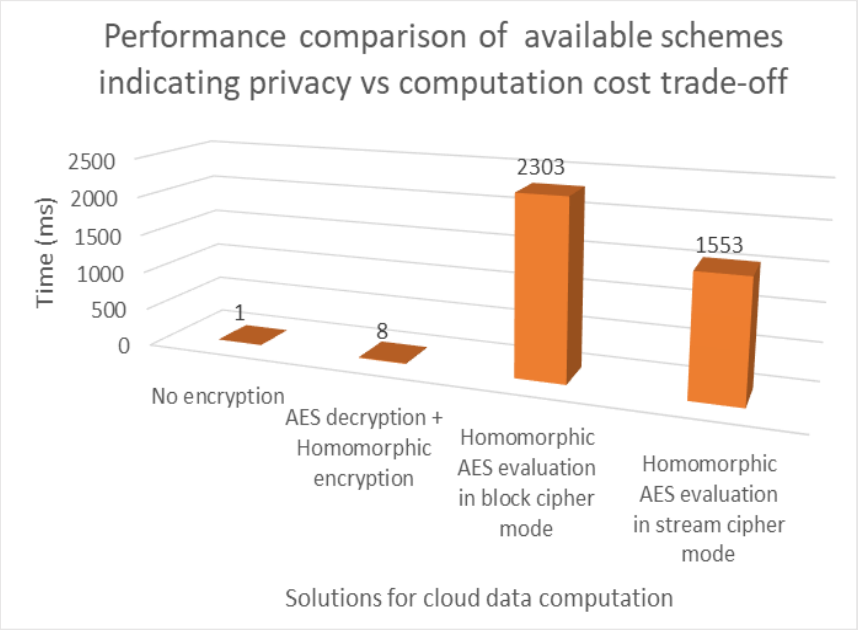

Figure 12. Evaluation of privacy vs latency trade-off for encryption schemes

The first approach is to share the data in the plaintext form, i.e. without any encryption. This is the most naïve (although not recommended) solution for constrained IoT devices. A communication cost of 1ms is observed to send the data. Since the data is immediately available for computation, time taken to transform the data at the proxy server is 0ms. Thus, the total time taken is 1ms. The second approach is to share device data encrypted under AES scheme. A communication cost of 1ms is observed to send the data. A computation cost of 2ms is observed for AES decryption function and a cost of 5ms is observed for homomorphic encryption of the plaintext device data. Thus, a total cost of 8ms is observed to make the data computation-ready. The third approach is to encrypt the device data under AES scheme in block cipher mode. A communication cost of 1ms is observed to send the data. A computation

cost of 2302ms is observed to perform the homomorphic evaluation of the AES decryption function in block cipher mode. After this, the data is computation-ready. Thus, a total time of 2303ms is observed. The last approach encrypts the device data under AES scheme using counter mode. A total communication cost of 1ms is observed to send the data. A computation cost of 1552ms is observed for the homomorphic evaluation of AES decryption function in the counter mode. After this, the data is computation-ready. Thus, a total time of 1553ms is observed. HElib library currently provides implementation for AES evaluation in block cipher mode. The current study utilizes HElib and extends it to evaluate AES evaluation in stream cipher mode.

The current study now analyzes the trade-offs in these schemes. Although the first approach is the fastest, it can be noted that it is not privacy preserving and is not a good fit for this study. The second approach is the next best approach in terms of latency. However, it requires a fully trusted VM that stores the device key to decrypt the AES encrypted data, and thus, is not a suitable approach for the current study. The third and fourth approaches meet the requirement of this study. Thus, the fourth approach is chosen due to its lower latency requirement.

Next, since the fourth approach utilizes stream cipher mode, it thus possible to separate the homomorphic evaluation of decryption function into two stages: offline and online as shown in Algorithm 3. This is from the inherent construction of AES encryption scheme in stream cipher mode.

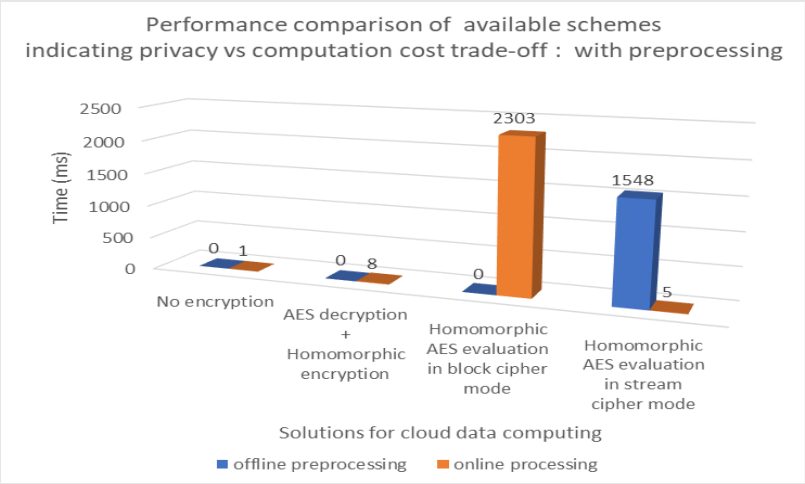

Figure 13. Evaluation of privacy vs latency trade-off for encryption schemes – with offline pre-processing

Figure 13 shows the performance of the proxy server with offline pre-processing capability. A computation cost of 1548ms is observed during the offline stage for the homomorphic AES encryption of the counter values. During the online stage, a communication cost of 1ms is observed for the encrypted device data to reach the proxy server and a computation cost of 4ms is observed for the homomorphic decryption of the AES data. Thus, a total online (after data is sent out) cost of 5ms is observed. This is a huge reduction in the total cost, which increases the throughput from 0.66 operations/ms to 250 operations/ms for the proxy server. Here, an operation is defined as the process of homomorphic evaluation of AES decryption function to transform device data from AES encryption scheme to homomorphic encryption scheme.

Effect of data size on throughput at proxy server

Next, the current study evaluates the time taken (with offline pre-processing) at the proxy server, during a data-flow phase with the real-world data. The current study assumes

that the offline homomorphic AES encryption of the counter values is always pre-computed and available. To perform this test, data from TELE ECG database is downloaded to the Android device. The performance is evaluated for sampling frequencies $f = 100Hz, 200\ Hz, 300\ Hz, 400Hz, 500Hz$, since these frequencies were identified to be most widely used by ECG devices. Figure 14 shows the performance of the above-mentioned test. *Client-side encryption* refers to the time taken at the android device to encrypt the data using AES encryption in counter mode. The current study utilizes the standard java crypto package for the evaluation. *Transformation at proxy* indicates the total time taken at a proxy server VM to evaluate the homomorphic decryption function of the encrypted data, with offline pre-processing. It is observed that for a sampling frequency of 500Hz (sampling frequency used in TELE ECG database), it takes 712ms to transform the data at the proxy server. This means that after receiving the data from the Android device, the proxy server VM can prepare the data for computation close to 700ms. The overhead of 700ms for a long term patient monitoring services like [99] can be acceptable, since they do not require real-time updates.

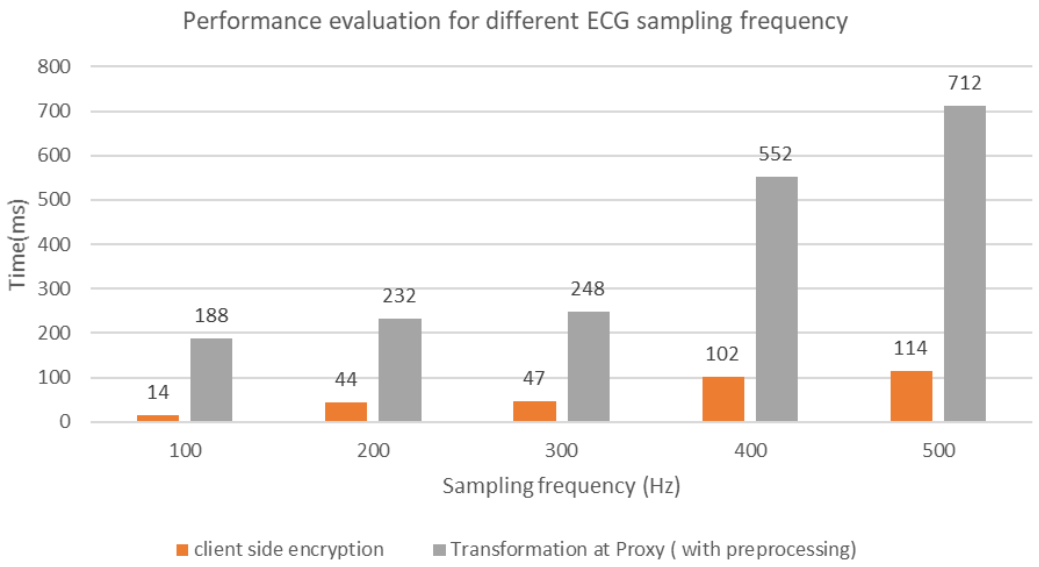

Figure 14. Evaluation of latency for different ECG sampling frequencies

Effect of CPU utilization level on throughput at proxy server

The latency observed in Figure 14 can be considered ideal, since the CPU load was under 5% throughout the experiment. However, the current study realizes that in a practical environment, a server typically runs multiple applications and hence it is vital to study the latencies under various CPU utilization levels. To test this, the current study uses Stress-ng [112] tool to emulate the desired CPU loads the proxy server VM and evaluates the time taken to transform the data from AES scheme to FHE scheme. For example, to simulate a CPU utilization of 40% a command as shown in Figure 15 was used.

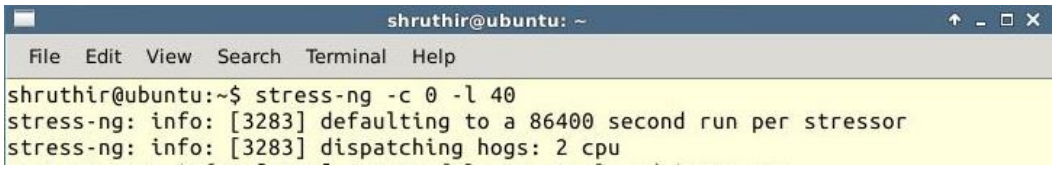

Figure 15. An example of stress-ng command line to emulate a CPU load of 40%

To test the performance, random records from TELE ECG database with a sampling frequency of 500Hz were chosen. The first parameter used for evaluation is the summation of the communication latency (defined as the time taken for the proxy server VM to receive the encrypted data as a JSON from the android device) and computation latency (defined as the time taken at the proxy server to parse the JSON data and prepare it for homomorphic decryption). The second parameter used for evaluation is the computation latency at the proxy server (defined as the time taken to perform homomorphic evaluation of the decryption function). The performance of the above parameters was evaluated for $CPU\ loads = 8\ \%, 15\%, 30\%, 40\%, 50\%, 70\%\ and\ 80\%$. Figure 16 shows the performance of the evaluation parameters for difference values of CPU loads. It is observed that with higher loads on CPU, the proxy server needed longer time to receive the JSON packet and parse it.

However, the time taken for the server to perform homomorphic decryption increases modestly. This is because as mentioned earlier, with off-line preprocessing the operations required during online processing are data encoding and XORing the encoded data with the homomorphic encrypted key streams, which are lightweight in nature. Since the overall throughput of a proxy server for *proxy re-ciphering as a service* depends on both the parameters, it can be inferred that the throughput of the proxy server for *proxy re-ciphering as a service* decreases with increased CPU loads. However, the homomorphic evaluation of the decryption function in stream cipher mode does not require a high computational cost even during high CPU utilization levels.

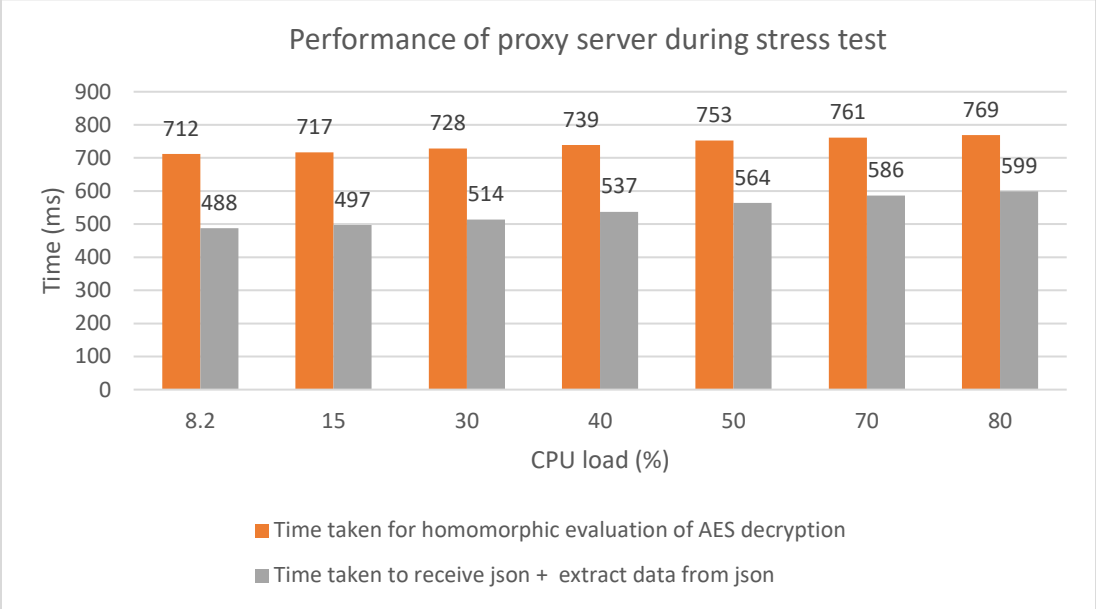

Figure 16. Evaluation of latency at a proxy server under various CPU loads

Analysis of end-to-end latency for an end-user

Next, to understand the impact of the system on the end-user, the current study evaluates the following parameters: i) time taken to share data from proxy server to cloud server (upload) ii) time taken for an end-user to receive the data from cloud server (download)

iii) time taken for an end-user to decrypt the received data (decryption) iv) time taken to delete an existing entry in the cloud server upon an end-user request (deletion). Each evaluation parameter was tested in two chosen networks during two fixed times of the day (10 am and 4 pm) to get a holistic view of the overall latency. The values observed in Figure 17 is an average of the 10 trials for each experiment. It is observed that deletion operation takes the least amount of time. This operation corresponds to deleting the user requested entry from the Azure blob storage. Next, it can be observed that upload time is lesser compared to the download time. This is because, as discussed in Chapter 3, for every healthcare organization, the current study chooses proxy servers that are closest (VMs are placed in same regions) to the cloud service provider. Hence, the time taken for data to reach blob storage from VM in Block A is relatively smaller compared to time taken to download data from the blob storage for an end-user (VM in a different region). Finally, it can be observed that the homomorphic decryption of cloud data requires around 730ms in a VM in Block C with 1GB RAM.

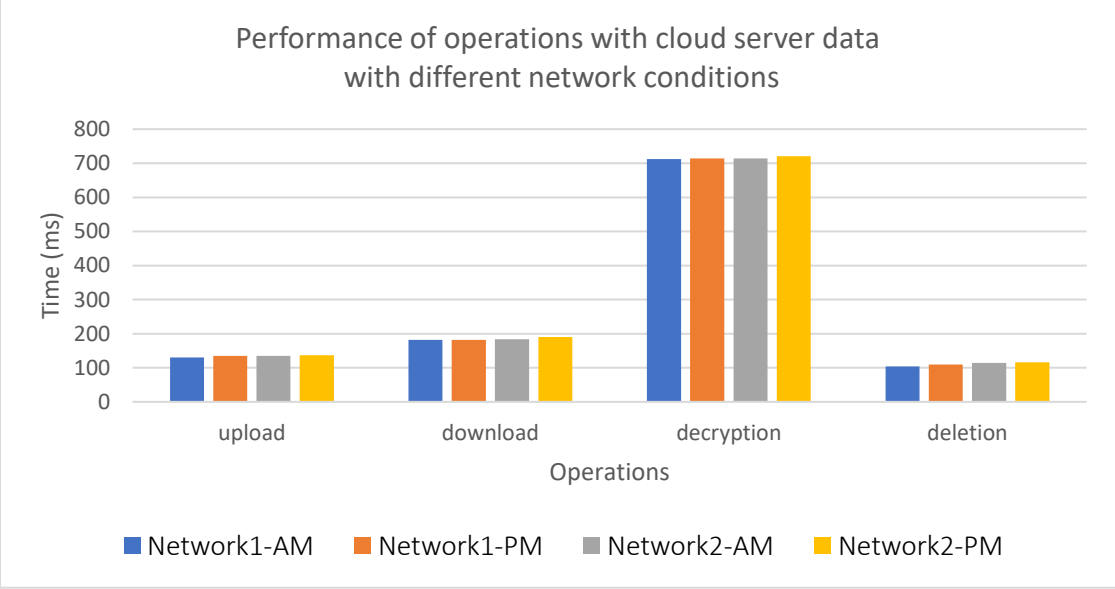

Figure 17. Evaluation of latency for upload, download, decrypt and delete operations

To evaluate the overall usability of the framework, it is critical to evaluate the end-to-end latency of this workflow. End-to-end latency in the current context can be expressed as a summation of the following latencies : a) computation latency at the Android emulator to encrypt device data b) communication latency to send the AES encrypted data to a proxy server VM c) computation latency at the proxy server VM to transform AES encrypted device data to homomorphic encrypted device data d) communication latency to send the homomorphic ciphertext to the cloud compute server e) computation latency at the cloud server to perform privacy-preserving computations on the homomorphic data as required by the application f)communication latency to send homomorphic result of the cloud-computation to the end-user g)computation latency at the end-user VM to decrypt the homomorphic result to view the plaintext result. The current study does not implement the application logic at the cloud compute server. Hence it is assumed that cloud servers take $x\ ms$ to execute an FHE based computation, based on the application. For a 32-byte of device data, the end-to-end latency from the above-mentioned experiments is observed to be: $(a)\ 1ms + (b)\ 1ms + (c)\ 4ms + (d)\ 130ms + (e)\ x\ + (f)\ 170ms + (g)\ 730ms = 1036 + x\ ms.$ If there is no cloud-processing required and if an end-user is interested in decrypting the device data using their private key, then $x = 0$ and hence the total latency can be around $1036ms$. Along the same lines, for a 500 Hz device data, assuming a cloud-computation time of $x\ ms$ to produce a 32-byte computation result, the end-to-end latency from the earlier experiments is observed to be: $(a)\ 114ms + (b)\ 488ms + (c)\ 712ms + (d)\ 13039\ ms\ + (e)\ x\ + (f)\ 170ms + (g)\ 730ms = 15253 + x\ ms$. It should be noted that the end-to-end latency value of close to $15.2 + x\ s$ is observed without any optimization in the above-mentioned latencies. However, with solutions like, homomorphic ciphertext packing to allow SIMD operations, homomorphic

dimension reduction before transmission to an end-user, a reduced latency can be achieved in practice. Also, it can be observed that the major contributor to the high latency is the upload time of homomorphic encrypted ciphertext to the cloud server from the proxy server. However, as explained in chapter 4, this value can be reduced based on the deployment of the service and the location of the servers. For example, if proxy servers and the cloud servers are hosted by the same provider and the servers are in the same region, the communication latency can be greatly reduced.

Analysis of upper bound on the post-compromise attack window

Finally, the study evaluates the feasibility of a dynamic key refresh operation for an IoT device. To assess this, the study calculates the time taken at the android emulator (acting as medical device) to generate new key, the time taken at the android emulator (acting as gateway) to generate collisions using trapdoor key and the total time taken to send random numbers between gateway-medical device and gateway-key manager server. Figure 18 shows that the total computation time taken at the android emulator to generate a dynamic key is around $157ms$. The current study did not record the time taken at the key manager server to generate the key since it is assumed that KMS is computationally more powerful compared to an android emulator and hence the overhead will be negligible, due to the infrequent nature of this operation. It can be observed that finding the collision r_2 at the gateway (Android device) takes only around 5s. This is due to the lightweight nature of collision calculation, as shown in Algorithm 6. Finally, to share $rand_{id}, r_1, r_2$ over the channel, the communication cost is around $110ms$. Thus, it can be observed that the overall time taken to dynamically renew encryption keys is modest and not heavy weight on the resource constrained IoT device. Thus,

the current scheme limits the time window between attack-detection and key-refresh to around $(157\ ms + 5\ ms + 110\ \ ms = 272\ ms\ )\ 280ms$, limiting the secure data exposure to an adversary.

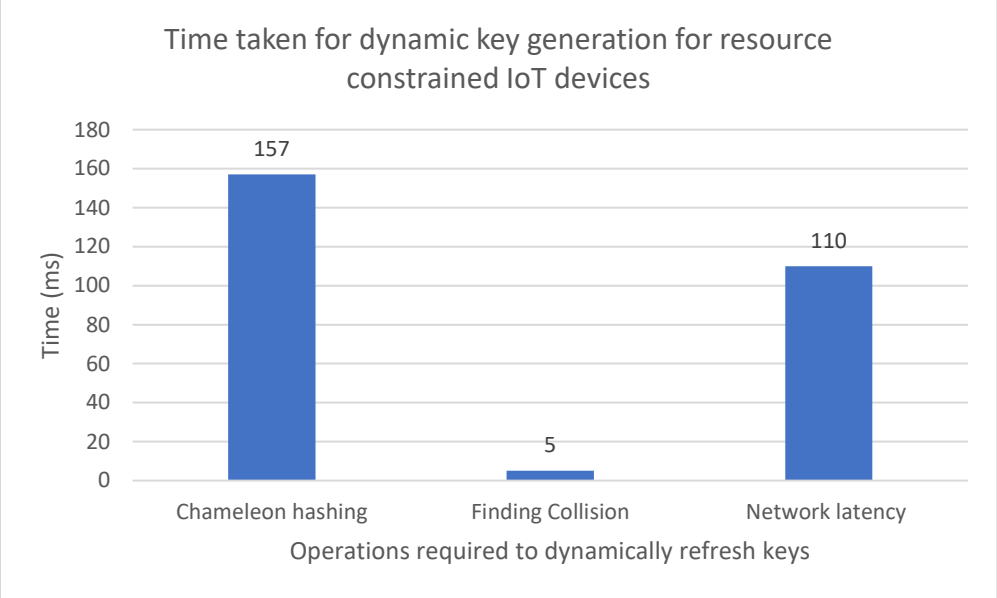

Figure 18. Evaluation of total latency to dynamically refresh an encryption key

5.2 Theoretical Evaluation

In this section the current study will briefly evaluate the security properties of the system and discuss how the current study resists the threats discussed in chapter 3.

The main security goals of the current study are to ensure confidentiality of the data, privacy of the users and availability of the service from passive and active. This study currently does not focus on the integrity of the data, but the focus is primarily given to the confidentiality and availability aspects of the CIA triad. Focusing on integrity of the data is deferred for the future works.

This section analyzes the security properties of the proposed work against commonly faced threats discussed in chapter 3.

Threat 1: Cloud database/server compromise:

The current study does not store any device keys in the cloud servers. Instead, it stores the homomorphic encryption public keys of the users to perform privacy-preserving computations. The cloud servers primarily store the homomorphic encrypted device data received from the IoT devices. The security of homomorphic encrypted data stored in the cloud servers directly follows the security of the BGV scheme. The security of BGV homomorphic scheme is based on Ring-Learning with Errors (RLWE) problem, which is a hard problem related to high-dimensional lattices. RLWE guarantee a 2^{λ} security against known attacks, where λ is a security parameter [61]. The default value of the security parameter used in HElib library (and thus in this work as well) is 80 bits of security.

The current system assumes the cloud servers to be resilient against loss of availability attacks, such as denial of service.

Threat 2: Proxy server compromise

The current study does not store any device keys in the proxy servers. Instead, it stores the shares of the device keys encrypted under homomorphic encryption scheme in n proxy servers and the reconstructed homomorphic device keys in m proxy servers. The security of the homomorphic encrypted device keys is directly derived from the security of BGV scheme, that are based on RLWE and have a 2^{λ} security [61] against known attacks, as mentioned above. The homomorphic encrypted device key shares are further encrypted using ChaCha20 encryption scheme as per Krawczyk's distribution scheme. ChaCha20 is practically found to be secure so far [115]-[116]. Upon reconstruction at the proxy servers, the security of homomorphic device keys is once again guaranteed by the security of BGV encryption scheme.

The following section evaluates the performance of proxy servers against loss of availability attacks.

In the current study, a total of m proxy servers provide service to every health care organization. As discussed in Chapter 3, the system parameters m, k, n are chosen such that the following requirement is satisfied $n - m \geq k$. The requirement for this condition is directly derived from the resiliency requirements of the system. In the current world of healthcare industries, adversaries targeting a particular healthcare organization is a very relevant and a critical issue [115]–[117] especially with a 13% increase in the denial of service based attacks from the past year [118] which is also expected to grow even further. When an adversary targets a specific healthcare organization, attacking the m proxy servers by either crashing the servers or corrupting the data can cause sufficient damages to the service. However, in the current system, $n - m \geq k$ and the homomorphic key shares of the IoT devices are shared across n servers. Hence by the property of threshold secret sharing, it is possible to reconstruct the data required to provide service to the healthcare organization. If an attacker targets the proxy servers arbitrarily, the system is resilient to loss of servers up to $n - k$, by the property of threshold secret sharing. That is, if an attacker attacks p servers, as long as $p \leq n - k$, the system is resilient to the denial of service attack or corruption of data attack.

Threat 3: Network-based attack

In the current study, medical devices encrypt data using AES-128-bit encryption in counter mode before sending them out. Although researchers have studied attacks [119]–[122] against AES, when implemented correctly AES is currently secure against any practical attack

as long as an adversary does not have access to the encryption key. This guarantees data security during acquisition in the device.

The communication between the medical device and the gateway is done over a local Bluetooth low energy (BTLE) link. One of the best features of BTLE, according to its specifications is its *privacy awareness* [123] that allows a participating device to change its private address to avoid tracking. However literature [68] shows that this is not used in practice. In addition, BTLE communication channel can be protected by means of BTLE encryption, However works like [65] have shown that the link is typically unprotected relying on device level encryption to secure the data during transit. Thus, as long as the device encryption key is secure, the communication between an IoT device and the gateway is protected against an eavesdropper. The communication channels between gateway-proxy server, proxy server-cloud server and cloud server-end user are secured with HTTPS.

Threat 4: Byzantine proxy servers

The above discussions so far assume that all the proxy servers are honest but non-trusted. This is indeed one of the assumptions of the current study. However, if an attacker is successful in controlling the behavior of a few servers in the system during the key reconstruction phase, this can result in a loss of service due to incorrect key reconstruction at the honest servers. The security requirement of the system in the presence of such byzantine servers are discussed now.

Let f denote the number of byzantine proxy servers. The f proxy servers can corrupt their shares or can arbitrarily distribute corrupt shares to the m proxy servers. This scenario is different from typical PBFT considerations since each server holds a share that is unique to the

server. During share reconstruction phase, the m proxy servers wait to receive shares from at least $k - m$ servers. So, it is necessary that a) $f < m$ (a coalition of malicious only servers should not be chosen to provide service to a healthcare organization), b) $n - f \geq k$ (there should always be enough honest servers in the system to reconstruct homomorphic keys, if required) and c) $m < k$ (a pool of proxy servers providing service to an organization should not be able to reconstruct keys for devices from other organizations, as per the principle of least privilege). With these relations it can be inferred that $\boldsymbol{2f < n}$. That is, n should at least be $2f + 1$. Under these circumstances, to reconstruct the homomorphic keys, there should be at least k honest server shares in any pool of shares, since some of them can be byzantine shares. To reconstruct the secret shares in the presence of malicious server shares, Kwarczyk extended his original secret sharing scheme from a computational secret sharing scheme to a robust secret sharing [85] by adding digital fingerprints to the shares at the time of distribution. Thus utilizing the work described in [124] can provide fault tolerance up to f server corruptions during key reconstruction. Krawczyk's robust secret sharing scheme introduces a total blow-up in the share sizes of $\cong \frac{n}{n-f}$ [124].

Threat 4: IoT device compromise

This section first evaluates the security properties of the dynamic key refresh scheme against the requirements described in Chapter 3. Then the performance of the system towards the threat is discussed.

The expected properties of a key refresh scheme introduced in Chapter 3 are discussed now.

Uniqueness of encryption keys

From the collision resistance property of the Chameleon hash function [92] the following can be inferred:

Given two pairs $m_1, r_1 \; and \; m_2, r_2,$ where $m_1 \neq m_2 \; and$ for $m_1, m_2 \in Z_q^*$ and $r_1, r_2 \in Z_q^*$, probability of $CHAM(m_1, hk, r_1) = CHAM(m_2, hk, r_2)$ is negligible without the knowledge of a trapdoor key, where hk is the hashing key .

In the current study $m = \text{XOR}(\; device_{id}, \; rand_{id})$.Every medical device has a unique identifier $device_{id}$. Using a CSPRNG will ensure that $rand_{id}$ generated for the devices are unique. Thus, for two devices with $device_{id1}$ and $device_{id2}$, $m_1 \neq m_2$ and hence the key generated from chameleon hash functions $CHAM(m_1, hk, r_1)$ and $CHAM(m_2, hk, r_2)$ are unique.

Randomness of encryption keys

The *uniformity* property of Chameleon hash function states that the probability distribution functions of the $CHAM(m, hk, r)$ for all messages m and for a random number r chosen at random is indistinguishable [92]. The *Semantic security* of chameleon hash functions which is derived from the u*niformity* property guarantees that the probability distributions of the random variables $CHAM(m_1, hk, r)$ and $CHAM(m_2, hk, r)$ for messages m_1 and m_2 are computationally indistinguishable [125]. This also means that the conditional entropy $H(^{m}/_{hash})$ of a message m given its hash $hash$ equals the total entropy of the message m , $H(m)$ [126]. Since an attacker may learn about $device_{id}$ during a compromise, this can reduce the entropy of m, $H(m)$. The probability of an attacker knowing the bits of $device_{id}$ can be expressed as : $P(id_i = 1) = w$, where $0 \leq w \leq 1$ and $0 \leq i \leq |device_{id}|$. Since $rand_{id}$ used in $m_1 = \text{XOR}(\; device_{id}, \; rand_{id})$ is generated using a CSPRNG, the

probability of an attacker knowing the bits can be expressed as P($rand_i$=1)=0.5 , $0 \leq i \leq |device_{id}|$. Thus, the probability of an attacker guessing the bits of dynamic device id generated from $m = \text{XOR}(\ device_{id},\ rand_{id})$ can be expressed as follows:

$$\Rightarrow P(m_i = 1) = P(id_i = 1) * P(rand_i = 0) + P(id_i = 0) * P(rand_i = 1)$$

$$\Rightarrow w * 0.5 + (1 - w) * 0.5$$

$$\Rightarrow 0.5$$

Thus, the probability is same as guessing the bits. Thus, the XOR operation overcomes a potential loss of entropy if $m = device_{id}$ during a device-key compromise. This is the reason the study generates a dynamic-id for the IoT device during dynamic key refresh using $m = \text{XOR}(\ device_{id},\ rand_{id})$.

Correctness of encryption keys

One of the important conditions of a key refresh operation is that the new keys generated at the IoT device end and the KMS end exactly match. This is defined as the correctness as the key refresh scheme. The correctness of the scheme used in this study directly follows the correctness of Chameleon hash functions.

Latency of key refresh algorithm

The computational and communicational latency associated with the proposed scheme is discussed under the experimental evaluation section. The communication complexity is $2|r|$ where r is the random number (r_1, r_2) shared by the gateway device with the medical device and the KMS.

Post-compromise Secrecy of the system

A scheme that guarantees two participating nodes to communicate securely even when the keys protecting the communication are compromised is informally termed to provide post-compromise secrecy [77]. Authors in [127] define post-compromise security as *"Full compromise of a node at a point in time does not reveal future messages sent within the group"*. IoT devices that use static encryption keys and no session keys suffer from post-compromise security where the advantage of an adversary possessing the knowledge of device encryption keys is non-zero. The current study addresses a weaker notion of post-compromise security, which it now defines with the help of Figure 19 .

Definition 2*:* A session under attack at time t_{attack} is *post-compromise secure* if there exists an intermediate session between t_{attack} and $t_{compromise}$ (time of compromise) such that

(i) $id_{attack} \neq id_{compromise}$ and

(ii) $id_{attack} + x.\ r_1 = m_{server} + x.\ r_2 \ mod\ q$

where,

id_{attack} , $id_{compromise}$ are dynamic ids used by an IoT device to generate device encryption keys used during time t_{attack} and $t_{compromise}$ respectively (using $CHAM(id, hk, r_1)$)

m_{server} is the id used by KMS to generate the synchronized device key using $CHAM(m_{server}, hk, r_2)$

x is the trapdoor for Chameleon hash function

r_1 is a random number used to generate chameleon collision r_2 .

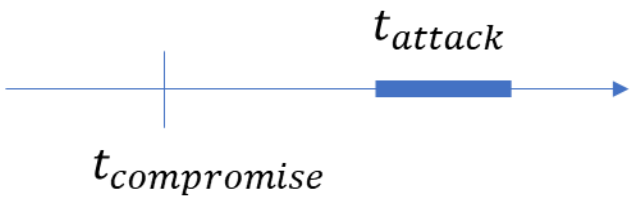

Figure 19. An example of attack scenario to define post-compromise security

The above definition is explained in greater detail now. If the dynamic-id used by an IoT device to generate device encryption keys before $t_{compromise}$ is $id_{compromise}$, then the device key available to an attacker at the time of compromise can be expressed as $key1 = CHAM(id_{compromise}, hk, r_1)$. As per the definition, after $t_{compromise}$ let us assume that there existed a small time window (~280 ms for the current testbed) before an attack such that a) the dynamic-id of an IoT device is updated to id_{attack} where $id_{attack} \neq id_{compromise}$ and b) a new collision r_2' is calculated using random variable r_1' and the trapdoor key x. Then the new encryption keys refreshed by the IoT device after this time window can be expressed as $key2 = CHAM(id_{attack}, hk, r_1')$, while the keys refreshed at the KMS can be expressed as $key2' = CHAM(m_{server}, hk, r_2')$. By the property of chameleon collision, it can be proven that $key2 = key2'$.

Now, to prove the post-compromise security of the framework, let us assume the contrary. Let us assume that the system is not post-compromise secure at t_{attack} . This means that an adversary can use $key1$ stolen at $t_{compromise}$ to learn about encrypted IoT device data during t_{attack} .However, if $t_{attack} > t_{compromise}$ and if the system calculated $r_1' \; and \; r_2'$ using $id_{attack} + x.\, r_1' = m_{server} + x.\, r_2' \; mod \; q$, then during t_{attack} the device data is encrypted using $key2$. By the collision resistance property of chameleon hash function, $P(key1 = key2) \cong \varepsilon$, where $\varepsilon \; is \; a \; negilible \; value$ when $id_{attack} \neq id_{compromise}$. Thus, this

invalidates the assumption that an attacker can use $key1$ stolen at $t_{compromise}$ to learn about encrypted IoT device data during t_{attack} , proving that the system is post-compromise secure, under the given conditions.

CHAPTER 6. CONCLUSION AND FUTURE WORK

6.1 Summary

This thesis investigates a practical and scalable solution to enable long-term secure and privacy preserving cloud computation for IoT devices. To this end, this thesis proposes utilizing distributed semi-trusted proxy servers, threshold secret sharing schemes and chameleon hash functions to provide *proxy re-ciphering as a service*. The proposed framework securely transforms data encrypted by IoT devices to a fully homomorphic encrypted data, thereby enabling secure cloud computation on the data. To evaluate the framework, testbed-based experiments were performed, and the proposed algorithms were individually analyzed. It was observed that although a distributed proxy server-based solution introduces additional delays, the security guarantees gained by such a distributed system outweighs the computational and communicational overhead. Also, it was inferred from the experiments that homomorphic evaluation of AES in stream cipher mode provides better throughput compared to the block cipher mode, and this can be further improved with an offline pre-processing. The proposed framework was also tested during diverse CPU loads to emulate a practical busy server. It was inferred that with an offline pre-processing of the counter values, the computations required during the online processing phase are lightweight and thus the latency increases only by a modest amount, even during high CPU loads. Furthermore, the study also evaluates the total time taken to dynamically refresh keys between an IoT device and the key manager server as it provides an upper bound on attack window available to an adversary after a key compromise. Finally, the study analyzes the key security properties of the proposed scheme and evaluates it against the critical threats discussed in Chapter 3. The salient features of the proposed framework are:

i) it does not require IoT device users to use additional (fully-trusted) hardware

ii) it accounts for real-world implementations where device vendors do not store device encryption keys in end-user (smartphone like) devices

iii) it uses libraries that are publicly available and tested for their performance by the authors, guaranteeing reliability and accessibility for application developers

iv) it introduces a modest computational overhead for key manager servers maintained by device vendors and a negligible computational overhead at the IoT devices, making the framework readily adoptable and back-portable with existing solutions

v) it encourages cloud service providers to store minimal yet adequate information to provide service to end-users, avoiding secondary data usage and privacy violations

vi) it guarantees security for past and future data for IoT devices by discouraging unnecessary storage of raw device data and by allowing devices to dynamical refresh keys after a compromise

vii) it studies the expansion in size with homomorphic encryption schemes and provides an efficient storage solution using threshold secret sharing schemes that also guarantees resilience against loss of availability of service and loss/ corruption of data

These salient features are the result of tailoring the framework to meet the requirements of a practical solution described in Chapter 3. Thus, it can be inferred that **Proxy re-ciphering as a service** is a practical, scalable and an easy-to-adopt secure framework that guarantees long-term privacy-preserving cloud computations for encrypted IoT data.

6.2 Limitations and Future work

Limitations

- *Knowledge about device compromise*: The system assumes the knowledge of a device compromise to initiate key refresh request. Although this can be true for active attacks, in many real-world IoT applications, such an information may not be readily available.
- *Storage:* Fully homomorphic encryption scheme is a relatively new and active research area allowing privacy preserving computations on encrypted data. It was observed from the experiments that the cipher text size expansion compared to the underlying plaintext is high. Thus, the storage overhead when adopting an FHE based solution can still be high.
- *End-user device:* The current study considers laptops like devices to receive encrypted insights and reports from the cloud servers. When an optimized version of HElib library which is compatible with smartphones is available, current work can be extended to include smart phones as end-user devices.
- *Default parameters:* The current study uses the default parameter values for the libraries used from the literature. For example, the system parameters used for homomorphic evaluation of AES are the default values available in the HElib library. Depending on the actual application logic, parameter values other than the ones used in the current study might be required.

Future work

- Developing a fully homomorphic encryption-based solution in the cloud would be the interesting next step. This is currently an on-going area of research where multiple real-world applications are still unexplored.

- Evaluating and proposing a technique for IoT devices to securely store the refreshed encryption keys within the device-hardware could be an interesting future work.
- During dynamic key refresh, the current study assumes a gateway device to be a fully trusted intermediary. Another interesting problem for the future work would be to develop a robust key refreshing scheme that addresses the assumption of device compromise-knowledge and uses no intermediary, like a gateway device.

REFERENCES

[1] S. R. I. C. B. Intelligence, "Disruptive civil technologies," *Six Technol. with potential impacts US Interes. out to*, vol. 2025, 2008.

[2] A. K. C. Bormann, M. Ersue, "Terminology for Constrained-Node Networks," 2014.

[3] C. B. Z. Shelby, K. Hartke, "The Constrained Application Protocol (CoAP)," 2014.

[4] C. Perrin, "The CIA triad," *Dostopno na http//www. techrepublic. com/blog/security/the-cia-triad/488*, 2008.

[5] P. Mell and T. Grance, "The NIST definition of cloud computing," 2011.

[6] "Amazon Web Services (AWS)- Cloud Computing Services," 2016. [Online]. Available: https://aws.amazon.com/.

[7] "Microsoft Azure Cloud Computing Platform & Services," 2010. [Online]. Available: https://azure.microsoft.com/.

[8] "Red HAT OpenShift," 2011. [Online]. Available: https://www.openshift.com/.

[9] M. Benioff and C. Adler, *Behind the cloud: the untold story of how Salesforce. com went from idea to billion-dollar company-and revolutionized an industry*. John Wiley & Sons, 2009.

[10] A. C. F. Chan and J. Zhou, "A Secure, Intelligent Electric Vehicle Ecosystem for Safe Integration with the Smart Grid," *IEEE Trans. Intell. Transp. Syst.*, vol. 16, no. 6, pp. 3367–3376, 2015.

[11] R. Smith, "Computing in the cloud," *Res. Manag.*, vol. 52, no. 5, pp. 65–68, 2009.

[12] "Google Drive: Free Cloud Storage for Personal Use." [Online]. Available: https://www.google.com/drive/.

[13] "Dropbox." [Online]. Available: https://www.dropbox.com/.

[14] "Siri - Apple." [Online]. Available: https://www.apple.com/siri/.

[15] "Alexa: Keyword Research, Competitive Analysis, & Website Ranking." [Online]. Available: https://www.alexa.com/.

[16] "Office 365." [Online]. Available: https://www.office.com/.

[17] "Apache Hadoop." [Online]. Available: https://hadoop.apache.org/.

[18] S. Brief, "Fast, Low-Overhead Encryption for Apache Hadoop."

[19] J. Gubbi, R. Buyya, S. Marusic, and M. Palaniswami, "Internet of Things (IoT): A vision, architectural elements, and future directions," *Futur. Gener. Comput. Syst.*, vol. 29, no. 7, pp. 1645–1660, 2013.

[20] "Internet of Things (IoT) connected devices installed base worldwide from 2015 to 2025 (in billions)."

[21] C. Dobre and F. Xhafa, "Intelligent services for big data science," *Futur. Gener. Comput. Syst.*, vol. 37, pp. 267–281, 2014.

[22] "Fitbit Inc," 2007. [Online]. Available: https://www.fitbit.com/.

[23] "Nest | Create a Connected Home," 2010. [Online]. Available: https://nest.com/.

[24] "Ring: Home Security Systems | Smart Home Automation," 2012. [Online]. Available: https://ring.com/.

[25] "Tesla: Electric Cars," 2003. [Online]. Available: https://www.tesla.com/.

[26] "Announcing the ADVANCED ENCRYPTION STANDARD (AES)," 2001.

[27] R. L. Rivest, A. Shamir, and L. Adleman, "A method for obtaining digital signatures and public-key cryptosystems," *Commun. ACM*, vol. 21, no. 2, pp. 120–126, 1978.

[28] E. Barker, W. Barker, W. Burr, W. Polk, and M. Smid, "Recommendation for key management part 1: General (revision 3)," *NIST Spec. Publ.*, vol. 800, no. 57, pp. 1–147, 2012.

[29] J. Edney and W. A. Arbaugh, *Real 802.11 security: Wi-Fi protected access and 802.11 i*. Addison-Wesley Professional, 2004.

[30] T. Hardjono and L. R. Dondeti, "Security in Wireless LANS and MANS (Artech House Computer Security)," *Artech House Inc*, 2005.

[31] N. Koblitz, "Elliptic curve cryptosystems," *Math. Comput.*, vol. 48, no. 177, pp. 203–209, 1987.

[32] V. S. Miller, "Use of elliptic curves in cryptography," in *Conference on the theory and application of cryptographic techniques*, 1985, pp. 417–426.

[33] S. Gueron, "Intel's new AES instructions for enhanced performance and security," in *International Workshop on Fast Software Encryption*, 2009, pp. 51–66.

[34] J. J. Stephen *et al.*, "Entitled Securing Cloud-Based Data Analytics: A Practical Approach Head of the Departmental Graduate Program Date," 2015.

[35] S. K. Shiho Moriai, Miyako Ohkubo, "Cryptographic Technology Guideline (Lightweight Cryptography) Title," 2017.

[36] T. Shirai, K. Shibutani, T. Akishita, S. Moriai, and T. Iwata, "The 128-bit blockcipher CLEFIA," in *International workshop on fast software encryption*, 2007, pp. 181–195.

[37] J. Borghoff *et al.*, "Prince–a low-latency block cipher for pervasive computing applications," in *International Conference on the Theory and Application of Cryptology and Information Security*, 2012, pp. 208–225.

[38] R. Beaulieu, S. Treatman-Clark, D. Shors, B. Weeks, J. Smith, and L. Wingers, "The SIMON and SPECK lightweight block ciphers," in *2015 52nd ACM/EDAC/IEEE Design Automation Conference (DAC)*, 2015, pp. 1–6.

[39] M. Matsui, J. Nakajima, and S. Moriai, "A description of the Camellia encryption algorithm," 2004.

[40] S. Banik *et al.*, "Midori: a block cipher for low energy," in *International Conference on the Theory and Application of Cryptology and Information Security*, 2015, pp. 411–436.

[41] V. Kolesnikov and R. Kumaresan, "Improved secure two-party computation via information-theoretic garbled circuits," in *International Conference on Security and Cryptography for Networks*, 2012, pp. 205–221.

[42] A. C.-C. Yao, "Protocols for secure computations," in *FOCS*, 1982, vol. 82, pp. 160–164.

[43] P. Mohassel and B. Riva, "Garbled circuits checking garbled circuits: More efficient and secure two-party computation," in *Advances in Cryptology–CRYPTO 2013*, Springer, 2013, pp. 36–53.

[44] R. L. Rivest, L. Adleman, and M. L. Dertouzos, "On data banks and privacy homomorphisms," *Found. Secur. Comput.*, vol. 4, no. 11, pp. 169–180, 1978.

[45] T. ElGamal, "A public key cryptosystem and a signature scheme based on discrete logarithms," *IEEE Trans. Inf. theory*, vol. 31, no. 4, pp. 469–472, 1985.

[46] P. Paillier, "Public-key cryptosystems based on composite degree residuosity classes," in *International Conference on the Theory and Applications of Cryptographic Techniques*, 1999, pp. 223–238.

[47] S. Goldwasser and S. Micali, "Probabilistic encryption & how to play mental poker keeping secret all partial information," in *Proceedings of the fourteenth annual ACM symposium on Theory of computing*, 1982, pp. 365–377.

[48] D. Boneh, E.-J. Goh, and K. Nissim, "Evaluating 2-DNF formulas on ciphertexts," in *Theory of Cryptography Conference*, 2005, pp. 325–341.

[49] J. Benaloh, "Verifiable secret-ballot elections [Ph. D. Thesis]," *Yale Univ.*, 1987.

[50] H. Shafagh, L. Burkhalter, and A. Hithnawi, *Talos a Platform for Processing Encrypted IoT Data: Demo Abstract*. 2016.

[51] R. A. Popa, C. Redfield, N. Zeldovich, and H. Balakrishnan, "CryptDB: protecting confidentiality with encrypted query processing," in *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, 2011, pp. 85–100.

[52] M. I. Sarfraz, M. Nabeel, J. Cao, and E. Bertino, "Dbmask: Fine-grained access control on encrypted relational databases," in *Proceedings of the 5th ACM Conference on Data and Application Security and Privacy*, 2015, pp. 1–11.

[53] H. Shafagh, A. Hithnawi, L. Burkhalter, P. Fischli, and S. Duquennoy, "Secure sharing of partially homomorphic encrypted iot data," in *Proceedings of the 15th ACM Conference on Embedded Network Sensor Systems*, 2017, p. 29.

[54] "Ava- Fertility Tracking Bracelet," 2014. [Online]. Available: https://www.avawomen.com/.

[55] C. Gentry and D. Boneh, *A fully homomorphic encryption scheme*, vol. 20, no. 09. Stanford University Stanford, 2009.

[56] Z. Brakerski, C. Gentry, and V. Vaikuntanathan, "(Leveled) fully homomorphic encryption without bootstrapping," *ACM Trans. Comput. Theory*, vol. 6, no. 3, p. 13, 2014.

[57] V. S. Halevi, Shai, "An Implementation of homomorphic encryption," 2013. [Online]. Available: https://github.com/shaih/HElib.

[58] S. Halevi and V. Shoup, "Algorithms in helib," in *Annual Cryptology Conference*, 2014, pp. 554–571.

[59] H. Au and V. Beach, "CallForFire: A Mission-Critical Cloud-Based Application Built Using the Nomad Framework," in *Financial Cryptography and Data Security: FC 2016 International Workshops, BITCOIN, VOTING, and WAHC, Christ Church, Barbados, February 26, 2016, Revised Selected Papers*, 2016, vol. 9604, p. 319.

[60] S. Ji and K. Wan, "k-Anonymously Private Search over Encrypted Data," *arXiv Prepr. arXiv1703.08269*, 2017.

[61] M. Naehrig, K. Lauter, and V. Vaikuntanathan, "Can homomorphic encryption be practical?," in *Proceedings of the 3rd ACM workshop on Cloud computing security workshop*, 2011, pp. 113–124.

[62] C. Gentry, S. Halevi, and N. P. Smart, "Homomorphic evaluation of the AES Circuit (Updated Implementation), 2015." .

[63] V. Shoup, "NTL: A Library for doing Number Theory." [Online]. Available: https://www.shoup.net/ntl/.

[64] "The GNU Multiple Precision Arithmetic Library." [Online]. Available: http://www.gmplib.org.

[65] J. Classen, D. Wegemer, P. Patras, T. Spink, and M. Hollick, "Anatomy of a vulnerable fitness tracking system: dissecting the fitbit cloud, App, and firmware," *Proc. ACM Interactive, Mobile, Wearable Ubiquitous Technol.*, vol. 2, no. 1, p. 5, 2018.

[66] M. Schellevis, "Getting access to your own Fitbit data," Radboud University, 2016.

[67] "Fitbit App." [Online]. Available: https://play.google.com/store/apps/details?id=com.fitbit.FitbitMobile&hl=en_US.

[68] B. Cyr, W. Horn, D. Miao, and M. Specter, "Security analysis of wearable fitness devices (fitbit)," *Massachusets Inst. Technol.*, vol. 1, 2014.

[69] A. Page, O. Kocabas, T. Soyata, M. Aktas, and J. Couderc, "Cloud-Based Privacy-Preserving Remote ECG Monitoring and Surveillance," *Ann. Noninvasive Electrocardiol.*, vol. 20, no. 4, pp. 328–337, 2015.

[70] O. Kocabas and T. Soyata, "Towards privacy-preserving medical cloud computing using homomorphic encryption," in *Enabling Real-Time Mobile Cloud Computing through Emerging Technologies*, IGI Global, 2015, pp. 213–246.

[71] O. Kocabas and T. Soyata, "Utilizing homomorphic encryption to implement secure and private medical cloud computing," in *2015 IEEE 8th International Conference on Cloud Computing*, 2015, pp. 540–547.

[72] A. Botta, W. De Donato, V. Persico, and A. Pescapé, "Integration of cloud computing and internet of things: a survey," *Futur. Gener. Comput. Syst.*, vol. 56, pp. 684–700, 2016.

[73] V. Lara, "What the Internet of Things means for consumer privacy," 2018.

[74] H. Fereidooni *et al.*, "Breaking fitness records without moving: Reverse engineering and spoofing fitbit," in *International Symposium on Research in Attacks, Intrusions, and Defenses*, 2017, pp. 48–69.

[75] "TP-Link." [Online]. Available: https://www.tp-link.com/us/.

[76] S. Khandelwal, "Millions of IoT Devices Using Same Hard-Coded CRYPTO Keys," 2015.

[77] K. Cohn-Gordon, C. Cremers, and L. Garratt, "On post-compromise security," in *2016 IEEE 29th Computer Security Foundations Symposium (CSF)*, 2016, pp. 164–178.

[78] J. Bottomley, "Converting Fault Resilience to Fault Tolerance." [Online]. Available: https://www.usenix.org/legacy/publications/library/proceedings/usenix04/tech/sigs/full_papers/bottomley/bottomley_html/node22.html.

[79] H. Chen, K. Laine, and R. Player, "Simple encrypted arithmetic library-SEAL v2. 1," in *International Conference on Financial Cryptography and Data Security*, 2017, pp. 3–18.

[80] S. S. Sathya, P. Vepakomma, R. Raskar, R. Ramachandra, and S. Bhattacharya, "A Review of Homomorphic Encryption Libraries for Secure Computation," *arXiv Prepr. arXiv1812.02428*, 2018.

[81] A. Shamir, "How to share a secret," *Commun. ACM*, vol. 22, no. 11, pp. 612–613, 1979.

[82] E. Waring, "VII. Problems concerning interpolations," *Philos. Trans. R. Soc. London*, no. 69, pp. 59–67, 1779.

[83] G. R. Blakley, "Safeguarding cryptographic keys," in *Proceedings of the national computer conference*, 1979, vol. 48, no. 313.

[84] A. Parakh and S. Kak, "Space efficient secret sharing for implicit data security," *Inf. Sci. (Ny).*, vol. 181, no. 2, pp. 335–341, 2011.

[85] H. Krawczyk, "Secret sharing made short," in *Annual International Cryptology Conference*, 1993, pp. 136–146.

[86] M. O. Rabin, "Efficient dispersal of information for security, load balancing, and fault tolerance," *J. ACM*, vol. 36, no. 2, pp. 335–348, 1989.

[87] A. L. Nir, Y., "ChaCha20 and Poly1305 for IETF Protocols," 2015.

[88] T. Loruenser, A. Happe, and D. Slamanig, "ARCHISTAR: towards secure and robust cloud based data sharing," in *2015 IEEE 7th International Conference on Cloud Computing Technology and Science (CloudCom)*, 2015, pp. 371–378.

[89] L. Carlitz, "The arithmetic of polynomials in a Galois field," *Am. J. Math.*, vol. 54, no. 1, pp. 39–50, 1932.

[90] P. E. Naeini *et al.*, "Privacy expectations and preferences in an IoT world," in *Thirteenth Symposium on Usable Privacy and Security ({SOUPS} 2017)*, 2017, pp. 399–412.

[91] P. Klasnja, S. Consolvo, T. Choudhury, R. Beckwith, and J. Hightower, "Exploring privacy concerns about personal sensing," in *International Conference on Pervasive Computing*, 2009, pp. 176–183.

[92] H. M. Krawczyk and T. D. Rabin, "Chameleon hashing and signatures." Google Patents, Aug-2000.

[93] W. Liu, "CloudCrypto." [Online]. Available: https://github.com/liuweiran900217/CloudCrypto.

[94] "Continuous Glucose Monitoring."

[95] "Different Types of Insulin Pens."

[96] "3M Intelligent Control Inhaler." [Online]. Available: https://www.3m.com/3M/en_US/drug-delivery-systems-us/technologies/inhalation/intelligentcontrol/.

[97] "Proteus Digital Health." [Online]. Available: https://www.proteus.com/.

[98] "Revealing the Billion-dollar Growth Opportunities in the Global Healthcare Industry by 2025." .

[99] K. Baldwin, "Use of Mobile and Sensor Technology Lowers Symptom Severity for People With Head and Neck Cancer."

[100] J. T. James, "A New, Evidence-based Estimate of Patient Harms Associated with Hospital Care," *J. Patient Saf.*, vol. 9, no. 3, pp. 122–128, 2013.

[101] "American Cancer Society. Cancer Facts & Figures, 2018." .

[102] A. Boyd, "Kushner Announces 'Whole of Government' Plan To Improve Health Tech."

[103] "Hospital Executives Survey," *The Lancet*, 2003. .

[104] "Google cloud Healthcare API." .

[105] "Fitbit and Google Announce Collaboration to Accelerate Innovation in Digital Health and Wearables," 30-Apr-2018.

[106] "Fitbit, Inc. to Acquire Twine Health," 2018.

[107] R. Ackerman, "The healthcare industry is in a world of cybersecurity hurt." .

[108] B. Rossi, "Healthcare fraud: a five step plan for diagnosis and treatment," *Information Age*, 2016.

[109] J. E. Pam Dixon, "The Geography of Medical Identity Theft."

[110] H. Khamis, R. Weiss, Y. Xie, C.-W. Chang, N. H. Lovell, and S. J. A.-A. R. C. Redmond, "TELE ECG Database: 250 telehealth ECG records (collected using dry metal electrodes) with annotated QRS and artifact masks, and MATLAB code for the UNSW artifact detection and UNSW QRS detection algorithms." Harvard Dataverse.

[111] "Fitbit Web API Basics." .

[112] "stress-ng - a tool to load and stress a computer system." [Online]. Available: http://manpages.ubuntu.com/manpages/bionic/man1/stress-ng.1.html.

[113] G. Procter, "A Security Analysis of the Composition of ChaCha20 and Poly1305.," *IACR Cryptol. ePrint Arch.*, vol. 2014, p. 613, 2014.

[114] "Security Analysis of ChaCha20-Poly1305 AEAD."

[115] J. Spitzer, "266k LA medical center patients' PHI compromised in ransomware attack." .

[116] J. K. Cohen, "Home medical equipment supplier Airway Oxygen hit with ransomware, affects 500k." .

[117] G. Johansson, "Cyber-attack shuts down US Regional Hospital's online system." .

[118] J. Davis, "Denial-of-service attacks on healthcare poised to explode." .

[119] A. Biryukov and D. Khovratovich, "Related-key cryptanalysis of the full AES-192 and AES-256," in *International Conference on the Theory and Application of Cryptology and Information Security*, 2009, pp. 1–18.

[120] A. Bogdanov, D. Khovratovich, and C. Rechberger, "Biclique cryptanalysis of the full AES," in *International Conference on the Theory and Application of Cryptology and Information Security*, 2011, pp. 344–371.

[121] H. Gilbert and T. Peyrin, "Super-Sbox cryptanalysis: improved attacks for AES-like permutations," in *International Workshop on Fast Software Encryption*, 2010, pp. 365–383.

[122] A. Biryukov, O. Dunkelman, N. Keller, D. Khovratovich, and A. Shamir, "Key recovery attacks of practical complexity on AES-256 variants with up to 10 rounds," in *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, 2010, pp. 299–319.

[123] S. Ashok and R. V Krishnaiah, "Overview and evaluation of bluetooth low energy: An emerging low-power wireless technology," *Int. J.*, vol. 3, no. 9, 2013.

[124] H. Krawczyk, "Distributed fingerprints and secure information dispersal," in *Proceedings of the twelfth annual ACM symposium on Principles of distributed computing*, 1993, pp. 207–218.

[125] G. Ateniese and B. de Medeiros, "Identity-based chameleon hash and applications," in *International Conference on Financial Cryptography*, 2004, pp. 164–180.

[126] X. Chen, F. Zhang, H. Tian, B. Wei, and K. Kim, "Discrete logarithm based chameleon hashing and signatures without key exposure," *Comput. Electr. Eng.*, vol. 37, no. 4, pp. 614–623, 2011.

[127] N. Messaging, L. Security, and S. Active, "Messaging Layer Security (mls) Charter for Working Group," 2018.