Design and Analysis of Privacy-Preserving Medical Cloud Computing Systems

by

Övünç Kocabaş

Submitted in Partial Fulfillment of the

Requirements for the Degree

Doctor of Philosophy

Supervised by

Professor Tolga Soyata

Department of Electrical and Computer Engineering
Arts, Sciences and Engineering
Edmund A. Hajim School of Engineering and Applied Sciences

University of Rochester
Rochester, New York

2016

Dedication

This thesis is dedicated to my parents, Gülsüm and Mehmet Kocabaş, and my sister Özge Kocabaş.

Biographical Sketch

Övünç Kocabaş was born in Bodrum, Turkey, in 1983. He received a Bachelor of Science in Microelectronics Engineering in 2006 and a Master of Science in Computer Science and Engineering in 2008, both from Sabanci University, Istanbul, Turkey. In August 2008, he joined the Electrical and Computer Engineering department at Rice University and received a Master of Science degree in 2010. He started his PhD program at the University of Rochester in 2010 in Electrical and Computer Engineering under the supervision of Professor Tolga Soyata and Professor Wendi Heinzelman. He worked on medical cloud computing using homomorphic encryption and collaborated with faculty from the University of Rochester Medical Center and the University of Rochester Computer Science Department. Between February and December 2012, he worked at Intel Corporation as an intern.

The following publications are a result of the work conducted during doctoral research:

O. Kocabas, T. Soyata, and M. Aktas, "Emerging Security Mechanisms for Medical Cyber Physical Systems," *IEEE/ACM Transactions on Computational Biology and Bioinformatics (TCBB)*, Feb. 2016.

P. Glasser, **O. Kocabas**, B. Kantarci, T. Soyata, and J. Matthews, "Energy efficient VM migration revisited: SLA assurance and minimum service dis-

ruption with available hosts," in *Proceedings of the 12th International Conference on High-capacity Optical Networks and Emerging/Enabling Technologies (HONET)*, Dec. 2015.

O. Kocabas and T. Soyata, "Utilizing Homomorphic Encryption to Implement Secure and Private Medical Cloud Computing," in *IEEE International Conference of Cloud Computing (CLOUD)*, pp. 540-547, Jun. 2015.

O. Kocabas and T. Soyata, "Towards Privacy-Preserving Medical Cloud Computing Using Homomorphic Encryption," *Enabling Real-Time Mobile Cloud Computing through Emerging Technologies*, ed. T. Soyata, ch. 7, pp. 213-246, IGI Global Publishers, 2015.

O. Kocabas, R. Gyampoh-Vidogah, and T. Soyata, "Operational Cost of Running Real-time Mobile Cloud Applications," *Enabling Real-Time Mobile Cloud Computing through Emerging Technologies*, ed. T. Soyata, ch. 10, pp. 294-321, IGI Global Publishers, 2015.

S. Ames, M. Venkitasubramaniam, A. Page, **O. Kocabas**, and T. Soyata, "Secure Health Monitoring in the Cloud Using Homomorphic Encryption, A Branching-Program Formulation," *Enabling Real-Time Mobile Cloud Computing through Emerging Technologies*, ed. T. Soyata, ch. 4, pp. 116-152, IGI Global Publishers, 2015.

A. Page, **O. Kocabas**, S. Ames, M. Venkitasubramaniam, and T. Soyata,"Cloud-based Secure Health Monitoring: Optimizing Fully-Homomorphic Encryption for Streaming Algorithms," in *IEEE Globecom Workshops (GC Wkshps)*, pp. 48-52, Dec. 2014.

A. Page, **O. Kocabas**, T. Soyata,M. Aktas, and J.P. Couderc, "Cloud-Based Privacy-Preserving Remote ECG Monitoring and Surveillance," *Annals of Noninvasive Electrocardiology*, Vol. 20, No. 4, pp. 328-337, Dec 2014.

O. Kocabas and T. Soyata, "Medical Data Analytics in the cloud using Homomorphic Encryption," *Handbook of Research on Cloud Infrastructures for Big Data Analytics*, pp. 471-488, IGI Global Publishers, 2014.

O. Kocabas, T. Soyata, J.P. Couderc, M. Aktas, J. Xia, and M. Huang, "Assessment of Cloud-based Health Monitoring using Homomorphic Encryption," in *IEEE International Conference on Computer Design (ICCD)*, 2013.

A. Carpenter, A. Hu, **O. Kocabas**, M. Huang, and H. Wu, "Enhancing effective throughput for transmission line-based bus," in *IEEE International Symposium in Computer Architecture (ISCA)*, 2012.

Acknowledgments

I would like to thank my supervisor Professor Tolga Soyata for his support and help during the long journey of my PhD. Without his support, this work could not be finished. Also, I would like express my gratitude to Professor Muthu Venkitasubramaniam for his collaboration and great ideas.

I would like to thank Professors Wendi Heinzelman, Joel I. Seiferas, and Muthu Venkitasubramaniam for acting as members of my thesis committee.

I would like to thank Professors Jean-Philippe Couderc, Michael Huang and Doctor Mehmet Aktas for their valuable feedback and ideas.

I would also like to thank my peers, Alex Page, Moeen Hassanalieragh and Scott Ames, for their friendship and help during my PhD studies.

Last but not the least, I would like to thank my family, for always being there and supporting me unconditionally. Without their support, this work could have never finished.

Abstract

Current financial and regulatory pressure has provided strong incentives to institute better disease prevention, improved patient monitoring, and push U.S. healthcare into the digital era. Outsourcing medical applications to a cloud operator helps healthcare organizations (HCO) to provide better patient care without increasing the associated costs. Despite these advantages, the adoption of medical cloud computing by HCO's has been slow due to the strict regulations on the privacy of Personal Health Information (PHI) dictated by The Health Insurance Portability and Accountability Act (HIPAA).

In this dissertation, we propose a novel privacy-preserving medical cloud computing system with an emphasis on "secure computation." The proposed system enables monitoring patients remotely outside the HCO using ECG signals. To eliminate privacy concerns associated with the public cloud providers, we utilize Fully Homomorphic Encryption (FHE) to enable computations on encrypted PHI data. Despite well-known performance penalties associated with FHE, we propose two methods for an efficient implementation. Specifically, we model our applications using two computational models: circuit and branching program, and propose optimizations to improve run-time performance. We compare our FHE-based solution with conventional and Attribute Based Encryption schemes for secure a) storage, b) computation, and c) sharing of the medical data. We show that despite the overhead compared to existing encryption schemes, our system can be implemented with a reasonable budget with major public cloud service

providers. With the recent advances on FHE coupled with the decreasing costs of cloud services, we argue that our study is a novel step towards privacy-preserving cloud-based health monitoring that can improve the diagnosis of cardiac diseases, which are responsible for the highest percentage of deaths in the United States.

Contributors and Funding Sources

This work was supervised by a dissertation committee consisting of Professors Tolga Soyata (advisor) and Wendi Heinzelman of the Department of Electrical and Computer Engineering and Professors Joel I. Seiferas and Muthu Venkitasubramaniam of the Department of Computer Science.

The cloud-based medical application design proposed in Chapter 2 was developed in collaboration with Professor Jean-Philippe Couderc and Doctor Mehmet Aktas of the University of Rochester Medical Center.

The branching program based FHE application described in Chapter 6 and its simulation results presented in Section 7.4.2 were developed in collaboration with Scott Ames and Professor Muthu Venkitasubramaniam of the Department of Computer Science.

All other work conducted for the dissertation was completed by the student independently.

This research was supported in part by the National Science Foundation grant CNS-1239423 and a gift from Nvidia Corp.

Table of Contents

List of Tables

List of Figures

List of Algorithms

1 Introduction

Utilizing cloud computing resources such as Amazon EC2 [2], Google Compute Cloud Platform [3], or Microsoft Azure [4] is commonplace for many corporations, due to its ability to avoid vast infrastructure investment. This concept dates back to the beginning of the internet boom more than a decade ago with the emergence of the Application Service Provider (ASP) model: Rather than making an investment in costly server hardware, software licensing fees, and the personnel to manage this infrastructure, corporations can rent computation time, storage space, and licensing fees by running such applications as Salesforce.com [5] over the internet. The ASP model avoids upfront costs: a monthly subscription fee and a flexible licensing scheme allow smaller corporations to immediately start using such programs and expand with virtually no boundaries, since the computational and storage resources are provided by the ASP and the ASP can pool resources for many other clients. Additionally, this eliminates the need for corporations to have any expertise in setting up such sophisticated server infrastructure and the training of the IT personnel. Therefore, it is natural to shift the responsibility of computing (and storage) infrastructure investments to operators that can deliver their services by using the internet as the delivery channel (i.e., Cloud Operators). By virtualizing their computational and storage resources, these cloud operators can provide these resources to their customers at a fraction of the cost

the customers can build them for [6].

While endless examples exist for such generic cloud computing offerings, one area that can benefit significantly from cloud computing deserves specific attention: Medical cloud computing. When the data storage is outsourced to a cloud operator over the internet, an important issue arises: data privacy. Although different applications have different sensitivity levels to this issue, the highest level of sensitivity is clearly in the medical arena [7, 8]. Personal Health Information (PHI) is one of the most scrutinized concepts, protected by laws and regulations of the USA. The Health Insurance Portability and Accountability Act (HIPAA) [9] dictates a strict set of rules and regulations to prevent the PHI from being misused. Therefore, to expand cloud computing into the medical arena, one must clearly formulate the entire concept around these restrictions.

Cloud computing is an active research area for medical applications, partly due to the push by the US government to modernize the US Health system [10]. The motivations behind this move are: 1) improving the quality of healthcare by using additional cloud-based long-term patient monitoring data that are otherwise unavailable to the healthcare professionals, and 2) reducing the operational costs at healthcare organizations (HCO) by eliminating the datacenters operated by HCOs. Long-term patient monitoring data (e.g., patient vitals such as ECG and blood pressure), obtained by sensors that transmit their patient information over the cloud can be used as an auxiliary diagnostic tool to improve diagnostic accuracy. This expands the boundaries of an HCO to outside the HCO by allowing the patients to use long-term monitoring devices, such as ECG patches.

Long-term patient monitoring requires ensuring data privacy at three distinct phases: Phase I. Acquisition, is where the medical data is acquired from a patient, whether it is within the HCO, or outside the HCO via disposable devices such as ECG patches [11], Phase II. Storage, where the data is stored in the cloud for future access and, Phase III. Computation, is where the data is processed,

whether during a real-time application execution by a doctor, or by the long-term patient monitoring software. Phases II (storage) and III (computation) imply multi-million dollar IT infrastructure investments for the HCP to properly handle the digital medical data while staying compliant with HIPAA privacy regulations. Currently, not only it is challenging for the US government to regulate these three phases for every HCP, but also it forces HCPs to hire IT staff that is capable of managing a full-fledged data center to ensure the integrity and the privacy of the stored and processed data.

With the existing technologies, data privacy in Phases I and II can be assured satisfactorily by encrypting the medical data with conventional encryption schemes such as AES [12]. However, ensuring data privacy during the computation (Phase III) is only possible by transferring the data back and forth between the cloud and the mobile device [13]. During this transfer, data must be in encrypted format, and can be decrypted only when it reaches the mobile device. Therefore, all existing solutions are based on *encrypted data storage*; however, there is currently no service that offers secure long-term patient *monitoring*, which would require *computation* on encrypted data.

The objective of this dissertation is to develop a technological framework for expanding the technology of cloud-based health records to secured cloud-based patient monitoring, i.e., enabling cloud-based computing resources to monitor patients physiological status continuously, ubiquitously, and securely. The proposed system will provide secure long-term health monitoring by strictly utilizing existing cloud computing resources (e.g., Amazon EC2 [2], Microsoft Azure [4], or Google [3]), accessed by *thin* devices such as tablets or mobile phones, while assuring complete patient data privacy. In contrast to the conventional encryption schemes, we utilize an emerging new technique called Fully Homomorphic Encryption (FHE) [14,15] to allow secure computations over encrypted medical data with public cloud service providers. This solution allows for data analysis in the

cloud, without making patient information available to the cloud provider. Our entire application runs in the cloud, and only the data acquisition and the visualization of analytics are achieved by thin devices (i.e., devices with significantly lower computational and storage capability as compared to the cloud resources). Therefore, these end nodes are disposable and the entire functionality of the application execution is outsourced to the cloud. With the proposed system, HCO has no computational resources allocated to the storage or computation (i.e., no datacenter to host the medical application), except the *thin* mobile devices (i.e., devices with limited computational and storage capability), which strictly act as the graphical user interface (GUI) devices.

The end goal of this dissertation is to develop methods for efficient computations over encrypted medical data. Our proposed system uses FHE, which allows the cloud to perform computations on encrypted data, without actually observing the data (i.e., patient private health information). While using FHE holds the promise to completely eliminate the cloud-based privacy concerns, it comes at a steep price: FHE-based operations are orders of magnitude slower than regular operations, rendering FHE impractical for generic applications [7, 16–19]. We propose two methods for computing our applications efficiently with a state-of-the-art FHE scheme called BGV [15]. Our first method models applications as binary-circuits and evaluates them homomorphically. We propose optimizations to reduce the multiplication depth and costly homomorphic operations to improve performance. The second method uses an alternative approach and represents applications as a branching program. Using branching program opens the door to borrowing from a rich body of research that exists for this computational model [20–22]. While the branching program approach restricts the applicability of FHE, we show that for certain applications, significant speed-ups can be obtained compared to the circuit-based method. Finally, we compare our FHE-based solution with conventional and Attribute-Based Encryption schemes and present

the overheads associated with secure a) storage and b) sharing of the medical data. We show that despite the associated overheads, our FHE-based solution can be implemented with a reasonable budget in the public cloud. Therefore, we argue that our study is a novel step towards privacy-preserving cloud-based health monitoring that can improve the diagnosis of cardiac diseases, which are responsible for the highest percentage of deaths in the United States.

Our contributions in this work are as follows:

- A privacy-preserving medical cloud computing system is proposed. The proposed system enables long-term monitoring of patients based on ECG signals [23]. Security and resource requirements of every layer of the system are analyzed to select appropriate encryption schemes to protect the privacy of the medical data.

- Secure computation methods over the encrypted medical data by using Fully Homomorphic Encryption (FHE) are proposed. A state-of-the-art FHE scheme called BGV is utilized to implement the proposed applications. Specifically, the *leveled* variant of the BGV scheme is used to enable efficient computations over encrypted data. Performance characteristics of the leveled BGV scheme are analyzed to optimize operations on encrypted data.

- A circuit-based FHE implementation of the medical applications is proposed. Essential building blocks that can be used in generic algorithms are designed. Methods for reducing multiplication-depth of the applications are proposed, which improves the performance of the FHE-based computations. Parameters for efficient computations with the leveled BGV scheme are derived.

- A branching program based FHE implementation for efficient comparison operation is proposed [24]. This implementation reduces multiplication-

depth compared to the circuit-based approach and improves the performance up to 20x with the availability of parallel processors.

- A comparison of the proposed FHE-based solution with conventional and Attribute Base Encryption (ABE) is provided. Performance and storage overheads are analyzed for secure a) storage, b) computation, and c) sharing of the medical data.

- Performance metrics for the circuit-based implementation is investigated for their associated cost when public cloud services are utilized. Implementation cost for the circuit-based solution with major cloud service providers is provided.

The rest of this dissertation is organized as follows. Details of the proposed privacy-preserving cloud-based system are discussed in Chapter 2. Background of conventional and emerging cryptography schemes is provided in Chapter 3. Fully Homomorphic Encryption schemes are reviewed in Chapter 4. Circuit-based FHE implementation of the medical applications are detailed in Chapter 5. Branching program based FHE implementation is discussed in Chapter 6. The performance of the FHE implementation of medical applications is analyzed and compared to conventional and emerging encryption schemes in Chapter 7. A cost analysis of running medical applications with major public cloud service providers is presented in Chapter 8. Finally, conclusions and future research directions are provided in Chapter 9.

2 Proposed System Design

The Patient Protection and Affordable Care Act is one of the most significant government efforts to generalize the use of electronic medical records (EMRs) and to incentivize the development of innovative technologies that can help curb rising US healthcare costs. In 2011, a study by Encinosa et al. [25] showed the impact of adoption of EMRs in healthcare organizations (HCO): a 34% reduction of death, a 39% reduction on readmission, and a 16% reduction on hospital spending costs. These observations support the urgency in transitioning US healthcare into the digital era. However, this transition is a challenging and costly task. Approximately 25% of the executives surveyed at 1,852 hospitals by the US News & World Report and Fidelity Investment identified costs associated with transitioning to EMRs as the most important concern in their organization [26]. The technology costs are currently around $200M and can rise to $1B for implementation, integration, and training for EMR-related products in the US alone [27].

Cloud computing could be a viable option to reduce costs associated with EMRs by outsourcing the storage of medical data to cloud operators [28], such as Amazon Web Services [2] and GoogleCloud [3]. The additional computational augmentation achieved from the cloud computing platforms can also provide capabilities that are unattainable through desktop- or traditional server-based computation; for eample, visualization of remote patient monitoring data in a highly

summarized format to reduce the "data burden" on the doctor [29, 30] could allow the doctor to benefit from this capability to improve diagnostic quality [31]. However, even if hospitals are willing to embrace cloud computing, cloud operators are reluctant to sign a Business Associate Agreement (BAA) and accept to store Personal Health Information (PHI) due to the high risks associated with a potential breach of data. Yet, smaller cloud storage operators have accepted such risks and offer *HIPAA-compliant storage* of static data: Examples include CareCloud [32] that uses Terremark, and DrChrono [33] based on Box technologies. Using these existing components, there are essentially three ways to design a remote monitoring system:

1. The hospital itself can maintain its own datacenter, but this is costly in terms of hardware, utilities, personnel, and space [28, 34]. Locally-maintained datacenters also limit scalability and interoperability [35].

2. Servers can be rented from a cloud provider that is willing to sign a business associate agreement (BAA) [35–37]. However, this restricts the hospital's options. Furthermore, while such an agreement ensures HIPAA compliance, it cannot guarantee privacy; breaches may happen even when best practices are followed.

3. The cloud can be used in a "storage only" mode, i.e., without the ability to analyze data. Data can be encrypted to prevent any third parties from reading it, adding a layer of security to option (2) and also enabling the use of "untrusted" cloud providers. (In the context of this work, "untrusted" providers are providers who should not have the ability to read the data that they are hosting.)

Thus, while it is *technically* possible to design a remote-monitoring system using existing tools, it is typically *not* possible to do so within the guidelines of

HIPAA without sacrificing computational capabilities or incurring large additional costs. While we can *store* encrypted PHI in the cloud, we have no method to securely *process* it — e.g., to generate reports or alerts — on an untrusted cloud platform. To take full advantage of secure cloud computing for medical applications, it is necessary to enable long-term patient monitoring outside the HCO by outsourcing both the storage and the computation of monitoring information.

The objective of this dissertation is to formulate a system-level framework for secure cloud-based patient health monitoring. The key distinction of our proposal is the ability to perform **HIPAA-compliant computation**, in addition to the already-existing **HIPAA-compliant storage**. To this end, our proposal includes: a) the adaptation of existing medical data acquisition technologies (e.g., ECG patches [11]), b) a thorough analysis of system-level design tradeoffs, such as computation scheduling among the acquisition and cloud resources, and c) the adaptation and improvement of emerging cryptographic techniques, such as Fully Homomorphic Encryption (FHE).

Our conceptualized system is shown in Figure 2.1, where phase I (Acquisition) of the long-term health monitoring is achieved via the use of remote sensors that are capable of regular AES-based encryption and transmission to the cloud via existing wireless access points. Currently, due to the computational complexity of AES encryption [12], which translates to high sensor power consumption in the mW range [38, 39], passive RFID sensors are not capable of AES encryption because they are only capable of μW operational ranges [40]. Without loss of generality, we specifically focus on ECG-based applications and the resulting improved diagnosis possibilities for heart diseases based on remotely-acquired long-term data in this fashion. Our system can be applied to any sensor that has similar capabilities with a backend application that has similar characteristics. In this system, phase II (storage) and III (computation) are strictly in the cloud, which leaves only the GUI responsibility to the mobile end-device during execution [41].

The rest of this chapter is organized as follows. In Section 2.1 an overview of the Electrocardiogram (ECG) is provided. In Section 2.2 medical applications based on the ECG are introduced, which will be implemented for the cloud-based patient monitoring. The details of the proposed system are provided in Section 2.3. Finally, in Section 2.4 security requirements to protect the privacy of the medical data in every component of our system are investigated.

2.1 Electrocardiogram (ECG)

A cloud-based system for health monitoring can be designed to support a variety of medical data that are routinely acquired by healthcare organizations (HCO) from their patients. Amongst this list, ECG, echo/MRI imaging data, subject drug treatment, and physiological monitoring signals are relevant to physicians for assessing an individual's health state. To gain insight into the challenges in applying FHE into medical applications, we will use Electrocardiogram (ECG) as our pilot application. We choose ECG, because currently in the US, heart diseases account for the highest percentage of cause of death, and providing a long-term patient monitoring solution based on ECG could provide better healthcare for cardiac patients. Also, ECG signals have relatively low sampling rate,

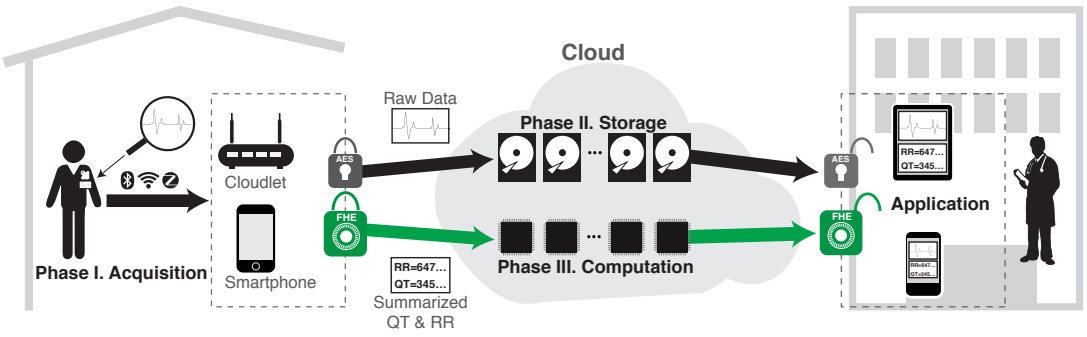


Figure 2.1: Proposed Cloud-based secure long-term patient monitoring system.

which is beneficial for the high-performance requirements of FHE. For the rest of this section, we will provide background information on ECG by using sample data acquired from the THEW worldwide ECG database [42] and identify operations that are necessary to provide insight for a doctor during the diagnosis of cardiavascular diseases.

2.1.1 ECG Dataset

In this work, we have opted to limit our feasibility assessment to a simple, yet real, set of data acquired from a subject coming to the Emergency Department (ED) of the University of California San Francisco Hospital for chest pain [43] and shared by the THEW initiative [42]. This data contains information about the electrical activity of the heart of the patient recorded using a 12-lead Holter system that records the patient ECG continuously for 24-hours. The device was hooked up to the patient when (s)he arrived at the ED. In order to demonstrate the feasibility of our concept, we used information about the patient heart rate (HR). There are standard ECG measurements that a cardiologist needs to access from this information that require computational tasks. Among these, we selected five measurements to be extracted from ECG tracing as examples, these are: 1) the minimum HR, 2) the maximum HR, 3) the average HR, 4) the presence of abnormal cardiac beats, and 5) the frequency of the ectopic beats. These five quantifiers can be extracted from the annotation file of the ECG, i.e., the file containing the information about each cardiac beat type and duration as shown in Figure 2.2. These five ECG measurements provide essential information to the cardiologist about the patients' heart state. First, the cardiologist will evaluate if the average heart rate is in normal ranges, then if the heart rate variation during the recordings are appropriate based on the patient physical activity, and finally the frequency of abnormal cardiac beats. These abnormal beats can be discriminated based on their morphology. They are often present in

healthy individuals, but they may be associated with some risk if their frequency of occurrence is too high.

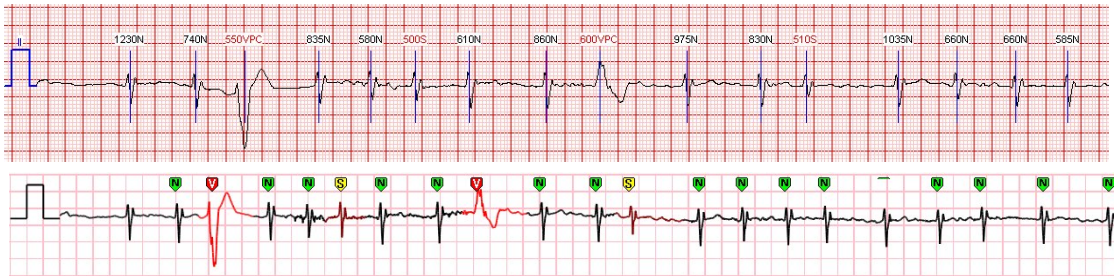


Figure 2.2: One-lead ECG tracing in which the number on top of each cardiac beat signal represents the time distance in milliseconds between the current beat (R peak) and the beat before. The character letter **N** provides the type of cardiac contraction which is normal in all displayed beats, while the other characters, such as V and S, denote irregular beats corresponding to potential heart conditions.

2.1.2 Structure of the Captured ECG Data

In general, the electrocardiogram (ECG) annotation file provides information related to cardiac contraction for each beat and the temporal distance between consecutive beats. The temporal distance is usually measured between two consecutive *R peaks*, which is the peak of positive deflection in the QRS complex.

We will assess the feasibility of implementing secure cloud-based monitoring using the ECG annotation file. This is a binary file containing two parts: 1) the header information and 2) the beat annotation. The header provides the information related to the original ECG, such as number of leads, sampling frequency, recording time, and other technical specifications of the digital ECG signal. The overall file structure is depicted in Figure 2.3.

Following the header is a 4-byte binary data which gives the first annotation position in the number of samples. Each 4-bit annotation segment is structured as follows:

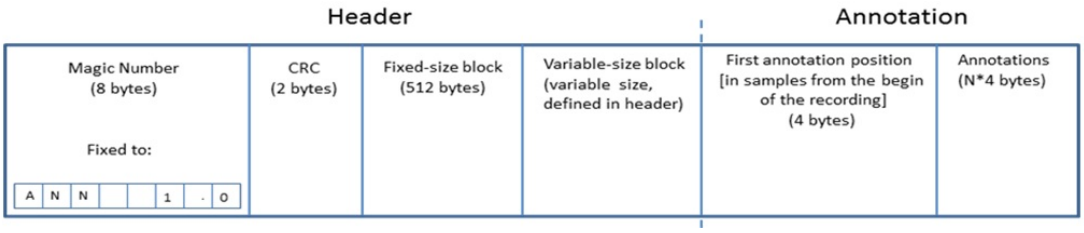


Figure 2.3: Overall file structure of the ECG annotation file used in our project.

1. **Label1 [char]:** beat annotation
2. **Label2 [char]:** internal use (e.g., further beat descrip.)
3. **toc [unsigned int]:** digital samples from last beat.

A typical structure of an annotation segment is shown in Figure 2.4.

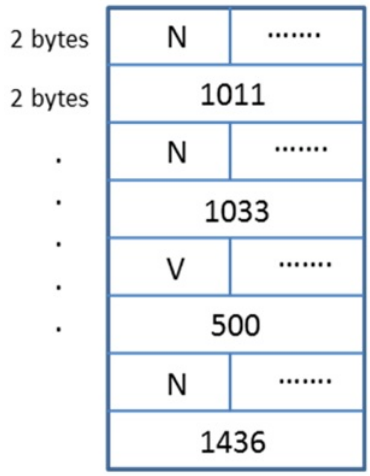


Figure 2.4: Annotation segment: each segment consists of beat type [1 byte], internal use [1 byte] and distance in sample from last beat [2 bytes].

The binary annotation we used in this work supports eight generic beat labels. These labels are:

1. **N (Normal Beat):** Normal beat is a heart impulse generated at the sinus node followed by a normal electrical pathway.

2. **V (Premature ventricular contraction):** This is a heart impulse initiated by Purkinje fibers in the ventricle rather than by the sinus node.

3. **S (Supraventricular premature beat):** This is a heart impulse originated from the atria or the atrioventricular node.

4. **C (Calibration Pulse):** A calibration signal may be included in a record for calibration. It usually occurs before the real ECG signal starts.

5. **B (Bundle branch block beat):** This impulse is generated by a heart with a defect in the electrical conduction system.

6. **P (Pace):** This impulse is initiated by a pacemaker device.

7. **X (Artifact):** An ECG artifact is used to indicate something that is not *heart-made*. These include (but are not limited to) electrical interference by outside sources, electrical noise from elsewhere in the body, poor contact, and machine malfunction.

8. **! (Timeout):** Usually when an arrhythmia alarm is detected, Timeout periods are automatically started. Beats that are within the time out period are labeled timeout.

The length of the annotation file depends on the length of the acquired ECG tracings. Continuous ECG monitoring usually contains hundreds of thousands of beats information that need to be analyzed and from which a report must be extracted.

2.2 Target Medical Applications

The proposed cloud-based application will enable the physicians to monitor patients and implement automatic alarms providing feedback on the patient's long-

term health status. We choose QT prolongation detection as our primary application. Our proposed system will also measure heart rate related statics such as Average Heart Rate, Minimum Heart Rate and Maximum Heart Rate to provide essential cardiac information to the doctors. The monitoring can be continuous in patients with high risk for life-threatening events, or periodic with a recording frequency depending on disease severity. Such technology will be disruptive, because it has the potential to shift the paradigm of patient management in the overall US health care system. Although

We have chosen to measure QT prolongation as our primary application, because measuring cardiac safety remains one of the most challenging hurdles in the development of new drugs and biotechnological products. The propensity of the drugs to cause potentially fatal arrhythmia, called torsades des pointes (TdP), is a significant public health issue [44]. An estimated 86% of all of the new drugs that are tested in pharmaceutical development show hERG inhibitory activity leading to TdP [45]. hERG is a gene that codes a protein subunit of potassium ion channels, and its contribution to the electrical activity of the heart is well known [46]. Many drugs potentially prolong the heart's ventricular repolarization process (VR), which in some cases trigger TdP, degenerate in ventricular fibrillation, and can cause sudden cardiac death (SCD) [47]. An example case of drug induced QT prolongation is presented in Figure 2.5, where a patient's condition becomes critical right after 30 minutes of drug injection.

In addition, when existing chemical entities are tested in a new patient population, the torsadogenic potential of the drug often needs to be re-evaluated. Because TdP occurs infrequently but is always associated to a delayed ventricular repolarization duration, a prolongation of the QT interval on electrocardiogram (ECG) has become an accepted surrogate/risk factor for torsadogenic drug cardiotoxicity. A study published in the Journal of American Cardiology, determined the risk relationship between QT duration and the probability of SCD in 6,134

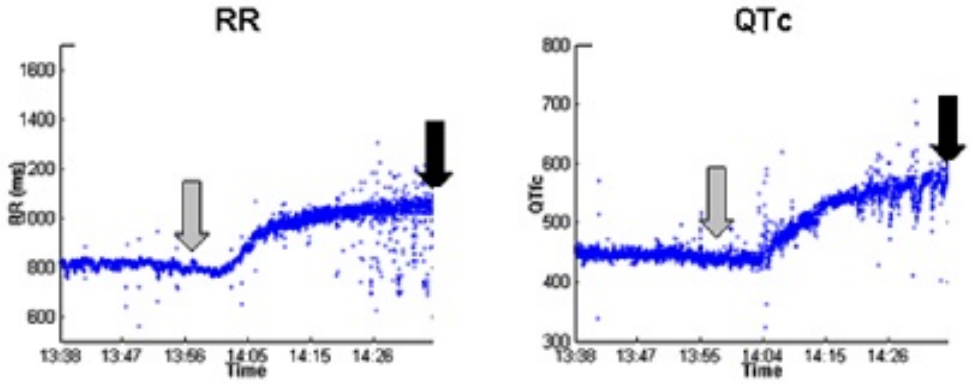


Figure 2.5: Monitoring of the repolarization parameters (RR and QT_c intervals) on long term ECG recordings with drug induced QT prolongation (right panel). The grey arrows mark the starting time of sotalol injection while the black arrows locate the onset of the first TdPs. Increased RR duration (slowing heart rate) and heart-rate corrected QT (QT_c) prolongation induced by the drug are presented between the two arrows in both panels. Figure from [1].

subjects who were enrolled in a prospective study in which the average follow up was 6.7 years [48]. Subjects with an abnormally prolonged QT interval had a more than 3 fold increased risk for SCD after adjustment for relevant covariates.

2.2.1 Long QT Syndrome (LQTS)

Prolongation of the QT/QTc interval is an accepted surrogate marker of an increased risk for life-threatening events [49]. As of today, there are hundreds of drugs available on the US market that can slightly prolong the QT interval. While these drugs are generally safe, the accumulation of their small QT effects when patients are prescribed with multiple drugs can become a health concern. The acquired LQTS is modulated by more than just drug interaction; the variability in individuals' response, the diet, the circadian variation of heart regulations are amongst a set of factors that can play a crucial role in the patient response to a drug with potential QT effect. Therefore, a solution is to continuously monitor patients and enable QT surveillance. This concept implies that QT intervals are

continuously assessed from an ambulatory ECG signal and corrected for heart rate. The proposed cloud-based medical application will enable physicians to monitor patients' cardiac rhythms securely and automatically.

2.2.2 QT and RR Intervals

The inputs to the LQTS algorithm are the QT and RR intervals, computed from raw ECG data by using a set of computer algorithms already validated on a large cohort of subjects [50]. An example of QT and RR intervals is presented in Figure 2.6. The QT interval on the ECG represents the time interval of the ventricular recovery phase of the heart and its prolongation is a marker for a potential TdP. The RR interval is the temporal distance between two consecutive *R peaks* which is the peak of positive deflection in the QRS complex.

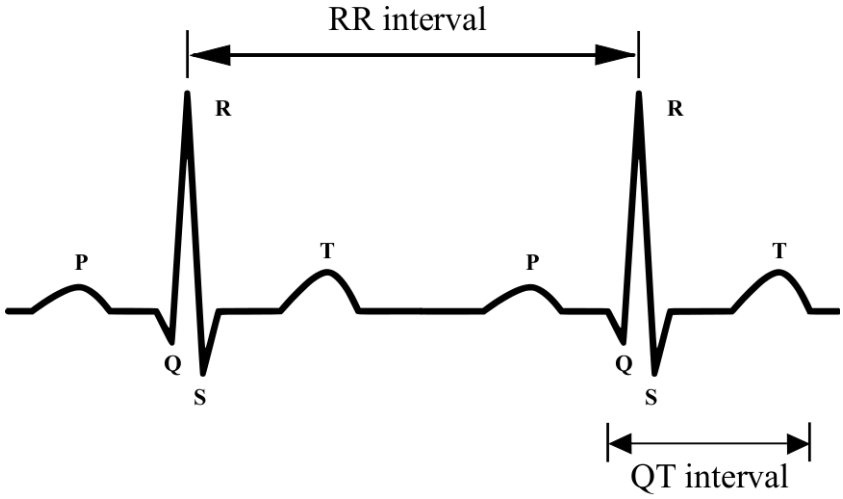


Figure 2.6: QT and RR intervals in an ECG signal.

2.2.3 LQTS Detection Algorithms

Detecting QT prolongations involves calculating the corrected QT (QT_c) values from the existing QT and RR values. QT_c can be calculated by using Fridericia's

formula [51] as

$$QT_c = \frac{QT}{\sqrt[3]{RR}} \tag{2.1}$$

and will be used as the basis of our case study in this section. The calculated QT_c values can be used to monitor abnormal ECG trends. One possible application scenario is to monitor QT_c intervals and send an alarm to the cardiologist if the values are higher than a clinical threshold (e.g., 500 ms). Therefore, our application will calculate the QT_c in an automated fashion and send an alert to the doctor when QT_c exceeds the clinical threshold.

Figure 2.7 provides a description of the association between the heart-rate corrected QT (QT_c) duration (Figure 2.7 left panel) and the relative risk of LQTS. The average QT_c trends with end-stage renal disease during and after hemodialysis is described in Figure 2.7 (right panel). The patients who died during the 13-month follow up period show a trend toward QT_c prolongation just after the hemodialysis (≈250 min after the start of the session), while the other patients who survived did not.

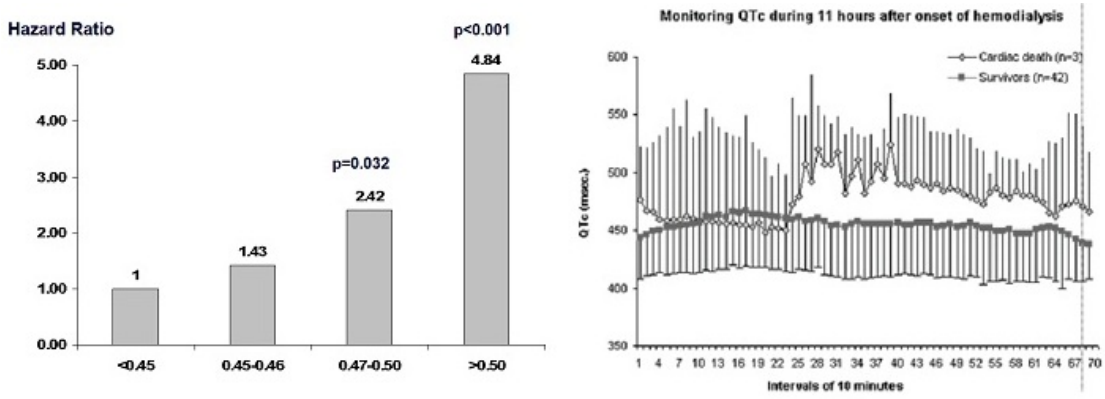


Figure 2.7: LEFT: Evolution of hazard ratios for cardiac events in relation to the QT_c. RIGHT: Continuous monitoring of the QT_c in patients going through hemodialysis. Patients who died from cardiac events in the following year showed significant QT_c prolongation just after hemodialysis while the other patients who survived did not.

2.3 Proposed System Architecture

The goal of our system is to push the entire workload into the cloud, and allow thin devices with minimal storage and computational capabilities to be used as acquisition sensors and GUI. This will enable true cloud computing by turning the *end nodes* of this system (i.e., acquisition and GUI) into simple (disposable) devices, while leaving the core of the system in the cloud. Based on our survey of the medical doctors who use such medical applications daily, the *portability* of the medical application is a very important issue. The proposed system achieves improved portability by permitting thin end nodes to be used for GUI. Furthermore, since nearly no information is stored on the end-devices, and the entire core of the application is in the cloud along with all of the data, data privacy concerns due to the loss of the ECG patches and/or mobile devices by the patient or the doctors (or nurses) are minimized, if not eliminated.

We propose a system that allows ECG data to be: aggregated on a smartphone (or PC) in the vicinity of a patient, uploaded to a cloud services provider (such as Amazon or Google), analyzed in the cloud (with results forwarded to the doctor), and protected along the entire path using a combination of standard encryption techniques and FHE. The doctor can review the results (e.g., an annotated ECG waveform), release the patient's diagnosis, and decide on a course of action.

There are three main components to the proposed system, which will be described below:

1. **Acquisition Devices:** The sensor(s) attached to patient and embedded system(s) in the vicinity of the patient.

2. **Public Cloud Services:** The servers and storage located in the remote datacenters provided by public cloud service providers.

3. **GUI Devices:** The doctor's mobile device(s) (e.g., tablet or smartphone).

The patient wears sensors that acquire the medical data and transmit it wirelessly and securely (e.g., using AES [52]) to a nearby computationally powerful device such as a cloudlet. The cloudlet decrypts the incoming sensor data, re-encrypts the data using two different techniques — one conventional, the other using FHE — then uploads it to the cloud. The cloud will store both the conventionally and the homomorphically encrypted data, and perform computations on the FHE-encrypted data to generate FHE-encrypted results. The cloud will transmit the encrypted results to GUI devices. The GUI devices will decrypt the FHE-encrypted results and notify the doctor in case of an emergency. We now detail each component of the proposed system.

2.3.1 Acquisition Devices

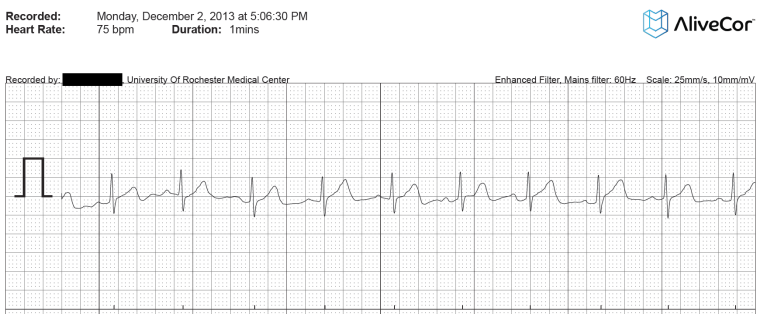
Acquisition devices are the front-end of our cloud based medical application, enabling the monitoring of a large variety of physiological signals. These devices are capable of acquiring real-time medical data. Most of today's acquisition devices acquire punctual measures, but do not provide long-term trend analysis, which can play a significant role in disease prevention. In the field of clinical cardiology, long term monitoring has been implemented in certain implantable devices such as implantable cardioverter-defibrillators or ECG loop recorders [53]. However, since these are invasive devices, their application space is confined to sick individuals with a strong risk of progressive deterioration of their health state.

Alternatively, with decades of research and development, current ECG recording technologies have matured enough to allow a patient to self-monitor at home. Figure 2.8(a) shows a sample device from Alivecor [54], which can be attached to a Smartphone. The software that is included with the device is capable of recording ECG samples, compressing and emailing them to any email address. A sample ECG recording is shown in Figure 2.8(b), which has sufficient accuracy to

be useful in clinical diagnostics since it obtained FDA approval for communication and patient prescription in May 2013.

To protect patient privacy, we assume the acquisition devices have the minimal capability of performing AES encryption of patient data and transmitting the encrypted data wirelessly [55–58]. Considering the significant computational difference of encrypting data between AES [52] and FHE, it is unrealistic for an acquisition node to execute real-time FHE encryption, while AES encryption has trivial computational demands and is available even in the least expensive devices. Therefore, we formulate the acquisition node as being oblivious to FHE encryption and only responsible for encrypting the patient data with AES encryption, while the conversion of AES encrypted data to FHE encrypted data is performed in the cloud.

(a) Alivecor ECG acquisition device.



(b) Sample ECG data produced by Alivecor device.

Figure 2.8: Alivecor device and sample ECG data

2.3.2 Public Cloud Services

In the proposed system, the cloud services will be utilized for two essential functionalities: secure storage, and secure computation of the medical data. We will also consider secure sharing of the medical data, since in many real-world healthcare scenarios, more than one party need to access the encrypted medical data stored in the cloud. Examples of the data sharing can include parties such as i) the patient being monitored, ii) doctor(s) from another healthcare provider, and iii) in an emergency, some other health care personnel.

Secure storage of medical data can be achieved by using conventional encryption schemes. However, as mentioned before, these encryption schemes have a major limitation: they preclude data analysis in the cloud (unless the decryption key(s) are also available to the cloud provider). That is, most basic arithmetic operations will not yield correct results — or even meaningful results — when applied to data that has been encrypted by a standard encryption algorithm such as the Advanced Encryption Standard (AES) [59]. To overcome this severe limitation and enable secure computation in the cloud, we look to *fully homomorphic encryption (FHE)* [60]. FHE is a novel encryption method that allows analysis (i.e., arithmetic operations) to take place on encrypted data, yielding correct, encrypted results. Using this technique, a server can analyze data and provide results to a doctor, *without the server ever knowing what the data or results were*; the doctor is only able to view the results because (s)he possesses the decryption key. This capability does not come without a cost, though: the set of mathematical operations that can be performed on FHE-encrypted data is still somewhat limited, and calculations can be very time and memory consuming. Our main contribution in this work is to propose methods that enable efficient computation of our medical applications with FHE.

Secure sharing of medical data requires either sharing a secret or a private key

of the encryption schemes or creating duplicates of the medical data encrypted with the designated party's public key. These solutions are not efficient, because they either require key-sharing protocols to share secret/private keys or extra storage space to store additional ciphertexts. Unfortunately, FHE schemes need to go through one of the two steps above. Recently, an emerging public-key encryption scheme called Attribute-Based Encryption (ABE) [61–64] has been proposed to solve the aforementioned problems related to secure data sharing. However, like conventional encryption schemes, ABE does not permit evaluating arithmetic operations on the encrypted data. We will use two ABE schemes (Ciphertext-Policy ABE [63] and Key-Policy ABE [64]) to compare the performance of our FHE-based solution for secure data sharing.

We will compare the performance of our FHE-based solution with conventional and ABE encryption schemes and present the overheads associated with secure data storage and secure data sharing.

2.3.3 GUI Device

The backend of our application is the GUI device, which runs the GUI portion of the medical application and displays the results to the doctor. Specifically, an application on the healthcare provider's tablet/smartphone will actively decrypt the encrypted results from the cloud, and notify the doctor of any changes in a patient's status for which they had requested an alert. In response to these notifications — or at any other time — the doctor can request more information. For example, the doctor may be notified that QT_c exceeded a given threshold for a 30 second interval. The doctor would then request to see the patient's ECG waveform for that 30 second interval. This waveform would be retrieved from the server, and decrypted for display.

Since the cloud is responsible for performing the entire set of computations

with FHE, the end result will be in the FHE-encrypted format when it is transferred to the GUI device. This necessitates that the GUI device has to perform decryption of FHE-encrypted ciphertexts. Furthermore, to avoid exposing the medical data at any point, decryption needs to be performed only on the smartphone/tablet of the authorized personnel. Considering that, within the FHE framework, the decryption has a fairly low compute-intensity as compared to the intermediate homomorphic computations on ciphertexts, this is feasible for the GUI device. Since most current smartphones have multiple processor cores and are expected to amass an ever increasing computational power, it is reasonable to expect the decryption process to take close to real-time and be acceptable to the user. Additionally, as the waveforms requested by the doctor are encrypted with conventional encryption schemes such as AES, they have to be decrypted for display. Decryption operation for conventional encryption schemes does not require extra processing power and can be performed efficiently.

Therefore, we conclude that the GUI end-node has to have minimal capability in 1) running an OS such as Android or iOS to provide a user interface to the doctor, and 2) perform homomorphic decryption. One way to reduce the computation burden on the end nodes is to outsource some of the heavy computations through Acceleration as a Service (AXaaS) mechanism [65–67], which utilizes cloud resources to provide acceleration for common computations such as Generalized Matrix-Matrix Multiplications (GEMM). However, the security implications of this must also be carefully considered. In such a methodology, the computation burden on the weak nodes such as the sensors is shifted to either the cloudlet [68] or the cloud, which acts as a *rented cloudlet* [67]. It is also possible consider crowdsourcing as a means to achieve computational acceleration [69], which is subjected to similar privacy considerations.

2.4 Privacy of Medical Data

The privacy of medical data must be protected from its acquisition to displaying it to the doctor due to strict rules and regulations imposed by the Health Insurance Portability and Accountability Act (HIPAA) [9]. Violating the privacy of the medical data can lead to severe penalties and associated costs can go up to $1.5M. Privacy of the medical data can be protected via encryption, which allows access only to authorized parties.

While individual encryption schemes, such as AES, are designed to encrypt isolated data blocks, ensuring system-level security requires conceptualizing a crypto-architecture for the system as a whole. An encryption/decryption scheme for each component of the system must be determined. In this section, an overall view of such a crypto-architecture is provided. The details of individual encryption mechanisms will be covered in Chapters 3 and 4.

2.4.1 Key Management Techniques

Regardless of the type of encryption scheme, communicating parties must agree on key(s) to encrypt/decrypt messages. In the public-key cryptography, *sender* uses the public key of *receiver* to encrypt messages and the *receiver* uses his/her private key to decrypt encrypted messages. Every user in the system has a dedicated public and private key pair generated by a Public-Key Infrastructure (PKI). PKI is a trusted third party such as certificate authority that authenticates the key pairs by binding them to the identity of user.

For the symmetric-key cryptography, both *sender* and *receiver* must share the same secret key to encrypt/decrypt messages. Both parties perform a key-exchange protocol such as Diffie-Hellman key exchange to generate the secret key. Once both parties share the same key, they can use symmetric-key cryptography to securely transfer the data.

2.4.2 Data Acquisition Privacy

The acquisition layer in Fig. 2.1 has limited computational capability due to the battery restrictions of the acquisition devices, which typically consist of body area network (BAN) sensors. Communication of the devices within the BAN-to-BAN and BAN-to-Cloudlet networks must be secured with lightweight encryption schemes.

One possible option is to use the ZigBee protocol, which requires low-cost micro-controller based devices to secure communication with AES. Communicating devices need to agree on a secret-key before using AES encryption. The key agreement can be achieved by using generic key exchange algorithm such as Diffie-Hellman (DH) [70] or its elliptic curve counterpart Elliptic Curve Diffie-Hellman (ECDH).

Communication of devices can be also secured by using biomedical signals. In [71], the authors propose a low-power bio-identification mechanism using the interpulse interval (IPI) to secure the communication between BAN sensors. IPI is the distance between two R peaks and is available to all sensors. In [72], the authors use physiological signals to agree on a secret key of the symmetric key cryptosystem for pairwise BAN sensor communication. They showed that compared to generic key exchange algorithms such as Elliptic Curve Diffie-Hellman (ECDH), their method requires fewer clock cycles to execute, albeit with a larger memory footprint. Their method features authentication capability and offers a viable option for key agreement in BANs.

2.4.3 Data Computation Privacy

While the data acquired from medical sensors can be secured with encryption, no computation can be done on this data using conventional cryptosystems without first decrypting it. Decryption necessitates a trusted storage such as a health-

care organization's datacenter or a private cloud. This eliminates the option to run analytics, monitoring algorithms (e.g., ECG cardiac monitoring [30]) or other algorithms in a public cloud to reduce health care costs.

A novel encryption technique called Fully Homomorphic Encryption (FHE) [14] allows computation on encrypted data without decrypting it first. By using FHE, the data can be stored in untrusted storage environments such as public clouds and computations on the encrypted data can be performed without violating the privacy of the data. In [13], we proposed a privacy-preserving medical cloud computing method based on FHE. We show that this method can be implemented at a reasonable cost despite the complexity of FHE.

2.4.4 Data Sharing Privacy

Using conventional cryptography schemes such as AES and RSA, the privacy of medical data can be assured. However, in many real-world healthcare scenarios, more than one party may need to access the data such as i) the patient being monitored, ii) his/her doctor, and iii) in an emergency, some other health care personnel. In such cases, conventional cryptography schemes cannot handle the sharing of the secret key among multiple parties. Encrypting the data using each party's public key is not a solution either since it creates duplicates of the data, which must be managed separately.

Recently a novel public-key cryptographic scheme called attribute based encryption (ABE) [61–63] has been proposed to solve the problem of secure sharing of the data between multiple parties. ABE is a public-key cryptosystem that provides fine-grained access control similar to Role Based Access Control [73]. Only the users whose credentials/attributes satisfy the rules determined by the access policy can retrieve the data. In [74], the authors propose methods to secure data storage in BANs and distribute data access control. They use the Attribute Based

Encryption (ABE) scheme [62] to control who accesses the patient data. ABE encryption is applied to data on a nearby computationally powerful local server, and the communication between BANs and the local server is secured with symmetric key encryption.

3 Cryptography Background

An essential part of designing a secure Medical Cloud Computing System (MCCS) is the determination of the system security requirements based on the capabilities of potential attackers [12]. In this chapter, first, we study the available adversary models and potential side channel attacks that are related to the security vulnerabilities of an MCCS. Then, we will review the conventional and emerging encryption schemes that will be used during the implementation of the proposed medical application. Specifically, we will cover the encryption schemes for secure storage and sharing of the medical data. Encryption schemes for secure computation will be covered in Chapter 4. The rest of this chapter is organized as follows. Section 3.1 introduces the adversary models that will used for designing MCCS. In Section 3.2, details of the AES encryption are provided. Elliptic Curve Cryptography based schemes are reviewed in Section 3.3. Finally, an emerging encryption scheme called Attribute-Based Encryption is detailed in Section 3.4.

3.1 MCCS Adversary Models

An MCCS must be resilient to attacks on all of its components. An adversary model captures the capabilities of an attacker. We consider two adversary models [75]: active (i.e., *malicious*) and passive (i.e., *honest but curious*). An *active*

adversary takes control of the host and can arbitrarily deviate from a specified protocol in order to steal secret information. Alternatively, a *passive adversary* follows the protocols correctly (*honest*), but can look at the encrypted data during the execution of protocols (*but curious*) to obtain information.

Data Privacy is one of the features that an MCCS must provide at every level. All of the encryption schemes that are considered in this paper protect data privacy against an active adversary. The only exceptions are the case where there is an attack directly at the crypto-level that "breaks" the encryption through a brute-forte attack. This could happen if the security parameters of an encryption scheme are chosen to be weak. Alternatively, a side channel attack could attempt to steal the secret/private key, as will be detailed in Section 3.1.1.

Correctness of the computed results (verification) is another feature that must be provided for an MCCS that aims to perform secure (encrypted) computations. As will be detailed in Chapter 4, secure computation over medical data in a public cloud can only be achieved using homomorphic encryption schemes. However, homomorphic encryption schemes are *malleable* by design; an active adversary can modify the computation result without knowing the private key. Therefore the correctness of the computations cannot be guaranteed when an active adversary model is considered.

To summarize; an MCCS provides *only* data privacy against an active adversary, while it can guarantee *both* data privacy and correctness against a passive adversary. The passive adversary model has been widely used for determining the security requirements of many cloud-based secure computation systems [76–78]. We also assume that an adversary cannot collude with the parties that hold the secret/private key of the symmetric/public key encryption schemes, since this type of an attack cannot be protected against by using any encryption scheme. We further note that the correctness of the secure computation can be achieved by using techniques from verifiable computing [79] or homomorphic signatures [80]. How-

ever, these techniques introduce additional performance penalties to encryption schemes that are already too slow to be practical.

3.1.1 Side Channel Attacks

Although encryption schemes go through rigorous mathematical and theoretical cryptanalysis to provide security and privacy, the system can still leak information due to the vulnerabilities in its software and hardware implementations. Attacks based on such leaked information are called *side channel attacks*. These attacks can be prevented by using leakage resistant cryptography [81], albeit at the expense of severe performance penalties that make an MCCS impractical.

Side channel attacks concentrate on obtaining the secret/private key by using every layer of the *system*, rather than just the *data* that is being processed by the system. While many types of side channel attacks exist for nearly every encryption scheme [82], we restrict our focus on attacks on AES and Elliptic Curve Cryptography (ECC), which are the most common encryption schemes for building an MCCS.

Timing Attacks are based on observing the execution time of the operations performed during encryption/decryption to reveal the secret key. Depending on the implementation, execution time of the operations can vary based on the bits of the secret key [83]. Timing attacks on AES usually observe cache memory access patterns during the execution of AES operations. Timing attacks on ECC target the scalar multiplication operation, and they can be prevented by using Montgomery's multiplication method [84], which performs the multiplication independent from the bits of the private key [85].

Power Analysis Attacks are based on observing the power consumption during the execution of cryptographic operations [86]. Power consumption can vary based on the bit values of the secret/private key, allowing an attack by either

observing the power usage of devices (simple power analysis) or using statistical methods to capture information in the presence of measurement errors and noise (differential power analysis). Differential power analysis attacks are more powerful due to their noise tolerance in power measurements. Power analysis attacks on AES can be prevented by using randomized masks for AES operations [87] that scramble the relationship between the AES secret key and the intermediate values generated during each AES round. Power analysis based attacks on ECC-based encryption schemes can be mitigated by methods proposed in [88] that randomize intermediate computations to avoid information leakage about the private key from power consumption patterns.

Fault-Based Attacks are based on introducing faults to bits during the execution of cryptographic operations [89], by applying a power glitch, magnetic field, light source, etc. This would cause errors in operations that can reveal the secret/private key to the attacker. In [90], the authors propose a method to thwart fault based attacks against AES by verifying the correctness of the encryption. The message is first encrypted and compared against the decrypted ciphertext to determine whether a fault was introduced during the encryption. Correctness of the decryption can be verified in a similar fashion by reversing the operations. Their method introduces significant hardware overhead. In [91], the authors propose a novel technique to detect faults based on Error Detecting Codes (EDC), which reduce the hardware overhead and latency. For ECC-based encryption schemes, fault-based attacks are focused on introducing error during the decryption to produce a point that is not on the elliptic curve [92]. These attacks can be mitigated by checking if the calculated point is on the elliptic curve and discarding the output of incorrect computations.

Cache Attacks are based on measuring the cache access latency of the cryptographic instructions to recover the cache lines that store the secret key [93, 94]. The information about memory access patterns can be measured by running a

malicious program in parallel with other processes. Cache attacks on AES implementations generally target the lookup tables that store S-Boxes [52]. Intel AES-NI instructions [95] can thwart cache attacks by making the cache access latency independent of the data and performing operations on the hardware without using lookup tables. Cache attacks on ECC exploit the precomputed values that are used during point addition in OpenSSL implementations [96]. ECC-based cache attacks can be prevented by i) using blinding scalar for point multiplication, ii) randomizing addition and multiplication chains, and iii) balancing number of additions and multiplications [96].

3.2 Advanced Encryption Standard (AES)

Currently AES is one of the most widely used symmetric-key encryption algorithms and is accepted as the standard in both industry and government applications. By design, AES is optimized for speed, low memory footprint and energy efficiency. This allows AES to become ubiquitous across a wide range of hardware, including 8-bit processors to high-end CPU/GPU's. Recently, instruction set extensions have been included in x86 architectures [95] to improve the performance of AES encryption and decryption.

AES is a block-cipher and operates on 128-bit blocks of data in multiple rounds (n_r). AES has three different key sizes: 128-bit (AES-128), 192-bit (AES-192) and 256-bit (AES-256). Longer key sizes increase the security but require higher number of rounds (n_r). AES-128, AES-192, and AES-256 use n_r=10, n_r=12, and n_r=14 rounds, respectively.

AES uses 128-bit blocks to represent both ciphertext and plaintext. Blocks are arranged as a 4×4 matrix, which is called AES *state*. Each entry in the matrix is 1B = 8-bit and represents an element in the finite field $\mathbb{F}_{2^8}$ with the reduction polynomial $G(x) = x^8 + x^4 + x^3 + x + 1$. The structure of an AES state matrix

is shown below

$$\begin{pmatrix} s_{0,0} & s_{0,1} & s_{0,2} & s_{0,3} \\ s_{1,0} & s_{1,1} & s_{1,2} & s_{1,3} \\ s_{2,0} & s_{2,1} & s_{2,2} & s_{2,3} \\ s_{3,0} & s_{3,1} & s_{3,2} & s_{3,3} \end{pmatrix}$$

AES is based on *Confusion* and *Diffusion* operations, which provides strong encryption. Confusion is performed via obscuring the relationship between ciphertext and the secret key. The diffusion property ensures that a change in plaintext is uniformly spread over the ciphertext so that plaintext is statistically hidden. All the operations in AES are either XOR or shuffling the data via shifting or replacing by a lookup table. Therefore, AES operations are very fast and have low-power consumption. Algorithm 1 shows the steps involved with AES Encryption, which has basically the repetition of four basic operations.

Algorithm 1: AES Encryption

input : Plaintext Block $ptxt_b$, Secret Key sk
output: AES state $state$
$state = InitState(ptxt_b, sk)$
$AddKey(state, sk_0)$
for $i = 1$ **to** $n_r - 1$ **do**
    $SubBytes(state)$
    $ShiftRows(state)$
    $MixColumns(state)$
    $AddKey(state, key_i)$
$SubBytes(state)$
$ShiftRows(state)$
$AddKey(state, key_{n_r-1})$

We explain each of the four operations applied during AES encryption as follows:

KeyExpansion: generates the round keys from the AES secret key sk. The number of rounds n_r depends on the key size, which is equal to $n_r + 1$.

Round keys are generated in an iterative fashion and stored as 4B = 32-bits.

AddKey: applies XOR to AES *state* with the round keys that are generated in the KeyExpansion step. The AES secret key is used only during this step.

SubBytes: applies a non-linear transformation to AES *states* by transforming each byte b of AES *state* as follows

$$b = Ab^{-1} \oplus c$$

First, the inverse of the byte is computed (b^{-1}) in $\mathbb{F}_{2^8}$, and then an affine transform operation is performed with the following matrix A and vector c.

$$A = \begin{pmatrix} 1 & 0 & 0 & 0 & 1 & 1 & 1 & 1 \\ 1 & 1 & 0 & 0 & 0 & 1 & 1 & 1 \\ 1 & 1 & 1 & 0 & 0 & 0 & 1 & 1 \\ 1 & 1 & 1 & 1 & 0 & 0 & 0 & 1 \\ 1 & 1 & 1 & 1 & 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 & 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 & 1 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 & 1 & 1 & 1 & 1 \end{pmatrix}, \quad c = \begin{pmatrix} 1 \\ 1 \\ 0 \\ 0 \\ 0 \\ 1 \\ 1 \\ 0 \end{pmatrix}$$

The SubBytes step is the only non-linear operation in AES. This step can be further optimized by computing all possible outcomes and storing them as a table called S-Box.

ShiftRows: applies cyclic left shifts to the rows of state matrix as shown below:

$$\begin{pmatrix} s_{0,0} & s_{0,1} & s_{0,2} & s_{0,3} \\ s_{1,0} & s_{1,1} & s_{1,2} & s_{1,3} \\ s_{2,0} & s_{2,1} & s_{2,2} & s_{2,3} \\ s_{3,0} & s_{3,1} & s_{3,2} & s_{3,3} \end{pmatrix} \rightarrow \begin{pmatrix} s_{0,0} & s_{0,1} & s_{0,2} & s_{0,3} \\ s_{1,1} & s_{1,2} & s_{1,3} & s_{1,0} \\ s_{2,2} & s_{2,3} & s_{2,0} & s_{2,1} \\ s_{3,3} & s_{3,0} & s_{3,1} & s_{3,2} \end{pmatrix}$$

The ShiftRows step is one of the steps that provide the *diffusion* property of the AES. This step can be reversed by applying cyclic right shifts.

MixColumns: applies transformation on the columns of the AES state based on operations in $\mathbb{F}_{2^8}$ and can be represented as a matrix multiplication as follows

$$\begin{pmatrix} s_{0,0} & s_{0,1} & s_{0,2} & s_{0,3} \\ s_{1,0} & s_{1,1} & s_{1,2} & s_{1,3} \\ s_{2,0} & s_{2,1} & s_{2,2} & s_{2,3} \\ s_{3,0} & s_{3,1} & s_{3,2} & s_{3,3} \end{pmatrix} = \begin{pmatrix} 0\text{x}02 & 0\text{x}03 & 0\text{x}01 & 0\text{x}01 \\ 0\text{x}01 & 0\text{x}02 & 0\text{x}03 & 0\text{x}01 \\ 0\text{x}01 & 0\text{x}01 & 0\text{x}02 & 0\text{x}03 \\ 0\text{x}03 & 0\text{x}01 & 0\text{x}01 & 0\text{x}02 \end{pmatrix} \times \begin{pmatrix} s_{0,0} & s_{0,1} & s_{0,2} & s_{0,3} \\ s_{1,0} & s_{1,1} & s_{1,2} & s_{1,3} \\ s_{2,0} & s_{2,1} & s_{2,2} & s_{2,3} \\ s_{3,0} & s_{3,1} & s_{3,2} & s_{3,3} \end{pmatrix}$$

The multiplication is performed as polynomial multiplication, where each column of the AES *state* is represented as an polynomial in $\mathbb{F}_{2^8}$

$$s_{0j} + s_{1j}x + s_{2j}x^2 + s_{3j}x^3$$

and the hexadecimal values converted to polynomials as

$$0\text{x}01 = 0000\ 0001 \rightarrow 1$$
$$0\text{x}02 = 0000\ 0010 \rightarrow x$$
$$0\text{x}03 = 0000\ 0011 \rightarrow x + 1$$

The polynomial multiplication is then followed by reduction with the poly-

nomial $x^4 + 1$.

AES Decryption uses the same operations as encryption and applies them in reverse order as shown in Algorithm 2.

Algorithm 2: AES Decryption

input : Ciphertext Block $ctxt_b$, Secret Key sk
output: AES state $state$
$state = InitState(ctxt_b, sk)$
$AddKey(state, sk_0)$
for $i = 1$ **to** $n_r - 1$ **do**
    $ShiftRows(state)$
    $SubBytes(state)$
    $AddKey(state, key_i)$
    $MixColumns(state)$
$ShiftRows(state)$
$SubBytes(state)$
$AddKey(state, key_{n_r-1})$

When a plaintext is longer than the AES block size, AES encryption/decryption can be used by choosing one of these modes of operation: Electronic Code Book (ECB), Ciphertext Chain Blocking (CBC), and Counter (CTR). A recent proposal is Galois Counter Mode (GCM) [97], which provides authentication as well as confidentiality. GCM combines the speed of CTR mode with hashing to provide an authenticated encryption mechanism. Confidentiality of the messages is protected using AES and integrity of the communication is provided using a universal hash function.

While AES is very energy efficient and provides strong security, it has the disadvantage of not allowing operations on encrypted data. Once data is encrypted with AES, it cannot be operated on and it has to be decrypted first for any possible operation.

3.2.1 AES Implementations

CPU Instruction Set implementations of AES, such as the Intel AES-NI [95] and ARM v8 Cryptography extensions [98], accelerate AES encryption/decryption and generally provide countermeasures against side channel attacks such as timing and cache-based attacks.

Embedded hardware implementations of AES encryption/decryption utilize restricted resources available in hardware platforms such as ASIC and FPGA. Efficient hardware implementations focus on the SubBytes step, which is the only non-linear step in AES. This step involves computing inverse of an element in $\mathbb{F}_{2^8}$, which is the most compute-intensive operation, followed by an affine transformation. Usually SubBytes can be computed by storing all possible combinations in an Substitution Box (S-Box) and use the S-Box as a lookup table. However, this requires additional hardware resources.

Several proposed optimizations [99–101] improve S-Box computation functionality by representing the AES finite field $\mathbb{F}_{2^8}$ as a composite field such as $\mathbb{F}_{(2^4)^2}$ or $\mathbb{F}_{((2^2)^2)^2}$ (i.e., *tower field*). While representing operations in the composite field requires additional back-and-forth conversions to $\mathbb{F}_{2^8}$, overall computation time is reduced due to the simplified intermediate operations.

Choosing a basis for the tower field is also crucial for the implementation, and three different choices exist for selecting a basis: polynomial [99], normal [100], and mixed [101]. While normal basis provides efficient inversion operation, polynomial basis provides better multiplication performance. In [101], the authors propose using both polynomial and normals basis as a mixture, and show that the critical path delay can be improved compared to using polynomial-only or normal-only basis. Finite fields can have many irreducible polynomials; 432 possible options are considered in [100] up to 20% reduction in terms of gates is reported by picking the optimum choice. Efficiency of AES implementation in the tower field

also depends on choosing the coefficients of irreducible polynomials. In [102], 16 possible choices are studied for choosing these coefficients and a reduction in gate size and critical path delay has been reported.

Implementations of AES-GCM are provided using dedicated hardware [103] or by using the instruction set support within Intel CPUs [104].

3.3 Elliptic Curve Cryptography

Elliptic Curve Cryptography (ECC) [105, 106] emerged as an alternative public key cryptosystem, which achieves the same security level of other public-key cryptosystems such as RSA with a shorter key size. For example, to achieve the same security strength as AES-128, RSA requires a 3072-bit key, while ECC only needs a 256-bit key. Reduced storage and bandwidth requirements combined with efficient arithmetic operations makes ECC suitable for resource limited devices such as BANs.

Security of ECC is based on the hardness of the elliptic curve discrete logarithm problem (ECDLP). The ECDLP problem is to find integer k, given two points on the curve G and $k \cdot G$. ECDLP is shown to be computationally hard, and the fastest algorithm [107] to solve the ECDLP requires approximately $\sqrt{p}$ steps for an elliptic curve on prime field $\mathbb{F}_p$. Therefore choosing 160-bit prime p provides a security level of 1024-bit RSA. Existing cryptography schemes based on the regular discrete log problem [70, 108] can be implemented with elliptic curves with a smaller key size.

ECC schemes are based on the arithmetic operations performed on the elliptic curves. Elliptic curves are proven to be a very versatile tool for designing sophisticated crypto-operations such as key sharing, encryption with data integrity and digital signatures. However, ECC schemes do not provide a mechanism for computations to be performed on the encrypted data. For the rest of this section,

we will first look at the arithmetic operations on elliptic curves that allow constructing ECC based schemes. Then we will provide details of important ECC based schemes.

3.3.1 Elliptic Curve Arithmetic

Elliptic curves over real numbers are defined as the set of points (x, y) that satisfy

$$y^2 = x^3 + a \cdot x + b$$

where a and b are chosen such that $4 \cdot a^3 + 27 \cdot b^2 \neq 0$. The points on the elliptic curve are symmetric with respect to the x-axis and form a group when a special point O is included that is not on the curve. The addition operation + for the group can be defined by adding two points P and Q (i.e., *point at infinity*), which results in another point R on the elliptic curve. Two types of addition exist with elliptic curves: point addition and point doubling.

Point Addition: adds two points $P(x_p, y_p)$ and $Q(x_q, y_q)$ on the elliptic curve to find point $R(x_r, y_R)$, which is also on the elliptic curve. Addition can be performed geometrically by drawing a line that connects P and Q and finding the intersection with the elliptic curve. The mirror of the intersection point with respect to the x-axis, which is on the curve, will be the point $R = P + Q$ (see Figure 3.1).

Point Doubling: computes the double of point $P(x_p, y_p)$ and requires a different type of operation than point addition. To find $2P$ geometrically, first a tangent line to point P is drawn and an intersection point is found between the line and the elliptic curve. The mirror of the intersection point with respect to the x-axis is the point $R = 2P$ (see Figure 3.1).

The point addition and point doubling operations can be expressed also alge-

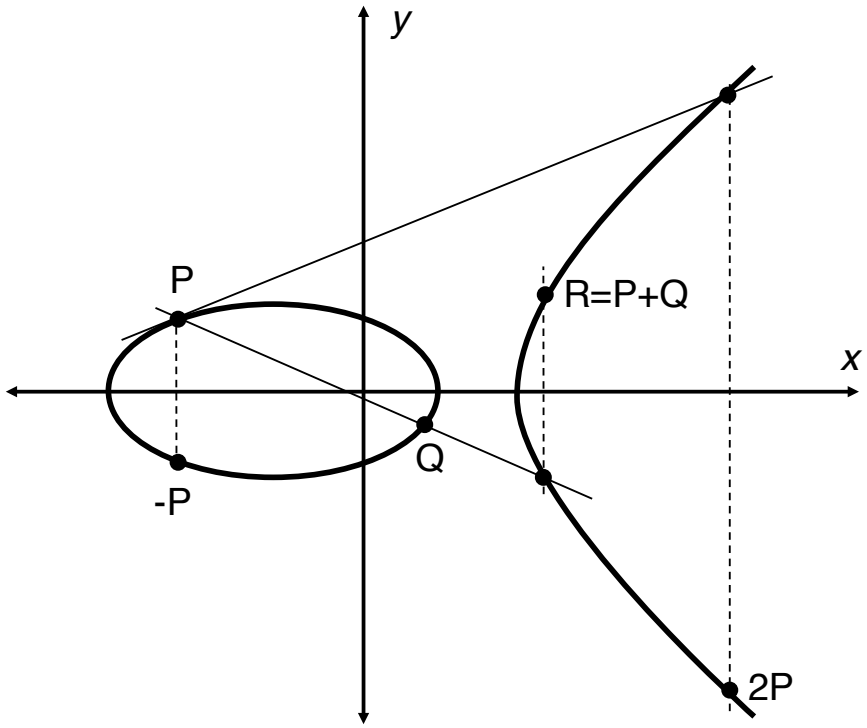


Figure 3.1: An Elliptic Curve and the *point addition* and *point doubling* operations on this curve.

braically as follows:

$$x_r = m^2 - x_p - x_q$$
$$y_r = m(x_p - x_r) - y_p$$

where m denotes the slope of the line (intersecting line for point addition and tangential line for point doubling as explained above) which can be expressed as

$$m = \begin{cases} \frac{y_q - y_p}{x_q - x_p} & P \neq Q \\ \frac{3 \cdot x_p^2 + a}{2 \cdot y_p} & P = Q \end{cases} \tag{3.1}$$

Additive inverse of a point $P(x_p, y_p)$ is calculated based on identity element O (i.e., *point at infinity*), such that $P + (-P) = O$. The inverse of the point P has the coordinates $-P(x_p, -y_p)$. While there is no point multiplication on

elliptic curves, multiplying a point with a scalar can be performed with repeated additions as $k \cdot G = \underbrace{G + G + G + \cdots + G}_{\text{k}}$. The repeated addition operation is very similar to modular exponentiation in RSA.

The elliptic curves used in cryptosystems are chosen from either prime fields ($\mathbb{F}_p$) or binary fields ($\mathbb{F}_{2^k}$). While choosing a prime field yields better implementation results in software, binary fields are used for hardware implementations due to efficient bit-level operations.

3.3.2 Bilinear Maps over Elliptic Curves

Bilinear maps are used for pairing-based cryptography protocols such as Identity Based Encryption (IBE) [109] and Attribute Based Encryption [61–63]. Given cyclic groups $\mathbb{G}_0, \mathbb{G}_1, \mathbb{G}_T$ of prime order p, Bilinear maps over Elliptic Curves can be defined as map $e : \mathbb{G}_0 \times \mathbb{G}_1 \rightarrow \mathbb{G}_T$ with the following properties:

- $\forall P, Q \in \mathbb{G}_0, \forall a, b \in \mathbb{Z}_q : e(aP, bQ) = e(P, Q)^{ab}$ (Bilinearity)
- $P \neq 0 \Rightarrow e(P, P) \neq 1$ (Non-degeneracy)
- $\forall P, Q \in \mathbb{G}_0$ there is an efficient algorithm that computes $e(P, Q)$ (Computable).

3.3.3 ECC Diffie-Hellman Key Exchange

ECC is widely used for key exchange in a similar fashion as Diffie-Hellman (DH) key-exchange protocol [70]. Regular DH can be converted to its ECC counterpart by replacing modular multiplications to point additions and modular exponentiations to repeated point additions.

A shared session key between two parties (i.e., sender and receiver) is established with ECCDH as shown in Figure 3.2. First, two parties agree on an elliptic

curve on prime field $\mathbb{F}_p$ and a point P on the curve. Then, the parties select integers k_A and k_B as their private keys. Based on their private keys, they compute a point Q_A, Q_B on the curve by performing repeated additions. They exchange their computations and due to the ECDLP neither party or any other third-party can find out private keys. Finally, each party performs another point multiplication with his/her private key to find a common point Q_{AB} on the elliptic curve, which can be used as the shared secret key for a symmetric cipher.

ECC Diffie-Hellman Key Exchange

Sender		**Receiver**
$2 \leq k_A \leq p-1$		$2 \leq k_B \leq p-1$
$Q_A = k_A \cdot P$		$Q_B = k_B \cdot P$
	$\xrightarrow{Q_A}$	
	$\xleftarrow{Q_B}$	
$k_{AB} = k_A \cdot Q_B$		$k_{AB} = k_B \cdot Q_A$
$k_{AB} = k_A \cdot k_B \cdot P$		$k_{AB} = k_B \cdot k_A \cdot P$

Figure 3.2: ECCDH Key Exchange Protocol

3.3.4 EC Integrated Encryption Scheme (ECIES)

One of the standard ways to use ECC for public-key cryptography is the ECIES method [110, 111]. ECIES provides data confidentiality by using a symmetric-key encryption such as AES. Integrity of the data is protected by message authentication code (MAC). Elliptic curves are employed to generate keys for both symmetric-key encryption (k_{ENC}) and MAC (k_{MAC}). A new key pair (i.e., k_{ENC} and k_{MAC}) is generated for each ECIES encryption.

Encryption with ECIES is performed as shown in Algorithm 3. First, sender generates a session key pair that will be used only for the current encryption.

The session key is generated by choosing an element $u \in \mathbb{Z}_p^*$ and computing elliptic curve point $U = u \cdot G$. Based on the session key, a shared secret value is generated by using receiver's public key as $S = u \cdot Q_B = u \cdot k_b \cdot G$. A standard Key Derivation Functions (KDF) [112] inputs the shared secret value to generate two keys: k_{ENC} and k_{MAC}. Message m is encrypted as $C = ENC(m, k_{ENC})$ using a symmetric key encryption and the key k_{ENC}. The tag of the ciphertext C is $tag = HMAC(C, k_{MAC})$, which is calculated using a keyed-hash message authentication code (HMAC). Finally, the sender transfers C, tag and U (session key) to the receiver.

Algorithm 3: ECIES Encryption

input : Message m, receiver's public key Q_B
output: U, C, tag
Set random $u \in \mathbb{Z}_p$
Compute $U = u \cdot G$
Compute $S(x_s, y_s) = u \cdot Q_B$
Generate $(k_{ENC}, k_{MAC}) = \text{KDF}(x_s)$
Encrypt $C = \text{ENC}(m, k_{ENC})$
Generate $tag = \text{HMAC}(C, k_{MAC})$

Decryption with ECIES is performed as shown in Algorithm 4. The receiver generates the shared secret S by computing $S = U \cdot k_b = u \cdot k_b \cdot G$. k_{ENC} and k_{MAC} are regenerated with the KDF and S. First, authenticity of the ciphertext C is verified by computing $tag_B = HMAC(C, k_{MAC})$ and comparing with the tag computed by sender. If both tags match, then message is retrieved as $m = DEC(C, k_{ENC})$, otherwise ciphertext C is discarded.

3.3.5 Elliptic Curve Digital Signature Algorithm (ECDSA)

Digital signature schemes provide data integrity, authentication and non-repudiation just like their analog counterpart. Elliptic curves are preferred for digital signatures due to their smaller parameters compared to other existing Digital Signature

Algorithm 4: ECIES Decryption

input : Ctxt C, tag, U, receiver's private key k_b
output: m
Compute $S(x_s, y_s) = U \cdot k_B$
Generate $(k_{ENC}, k_{MAC}) = \text{KDF}(x_s)$
Compute $\text{tag}_B = \text{HMAC}(C, k_{MAC})$
Check $tag_B == tag$
Decrypt $m = \text{DEC}(C, k_{ENC})$

Algorithms (DSA), which results in less usage of storage, bandwidth and power consumption.

In a digital signature scheme, a message is signed by the sender's private key and the receiver verifies the signature with the sender's public key. Assume the sender wants to use ECDSA to sign message M before sending it to receiver. The sender generates the signature by following the protocol shown in Algorithm 5. The receiver verifies the signature by using the sender's public-key as shown in Algorithm 6.

Algorithm 5: ECDSA Signature Generation

input : Message M, Alice's private key k_A
output: Signature (S_1, S_2)
Choose $r \in \mathbb{Z}_p^*$
Compute $P = r \cdot G = (x_p, y_p)$
Set $S_1 = x_p \mod p$
Compute $e = HASH(M)$
Set $S_2 = r^{-1} \cdot (e + r \cdot k_A) \mod p$

3.4 Attribute Based Encryption

In conventional public-key cryptography [105,113], a user has two keys: The *public key* is shared with anyone that wants to send encrypted data to the user, while the *private key* is used to decrypt the received messages and is not shared with

Algorithm 6: ECDSA Signature Verification

input : Signature (s_1, s_2), Message m, Alice's public key Q_A
output: Accept or Reject
Compute $e = HASH(m)$
Compute $\omega = S_2^{-1} \mod p$
Compute $u_1 = e \cdot \omega \mod p$
Compute $u_2 = S_1 \cdot \omega \mod p$
Compute $T = u_1 \cdot G + u_2 \cdot Q_A = (x_t, y_t)$
Compute $v = x_t \mod p$
if $v == S_1$ **then**
  Accept
else
  Reject

anyone. In many real-world healthcare scenarios, more than one party may need to access the data. This requires creating duplicates of the data by encrypting it using each party's public key.

Attribute-based encryption (ABE) [61, 62] is a novel public-key encryption that enables secure data sharing to multiple users when the data is stored in untrusted public clouds. ABE provides flexible access control of the encrypted data compared to public-key cryptography. The data is encrypted with an access policy based on credentials (i.e., *attributes*) and only users whose credentials satisfy the access policy can access the data. The attributes are defined as credentials such profession (e.g., Doctor, Nurse), and department (e.g., Cardiology, Emergency). An access policy P can be defined as conjunctions, disjunctions and (k, n)-threshold gates of attributes such as

$$(\text{Doctor} \wedge \text{Cardiology}) \vee \text{Nurse} \vee \text{Emergency} \vee \textit{Patient}$$

which grants access to a Doctor from Cardiology OR a nurse OR an Emergency personnel OR the patient. The aforementioned access policy can be represented as a tree shown in Figure 3.3.

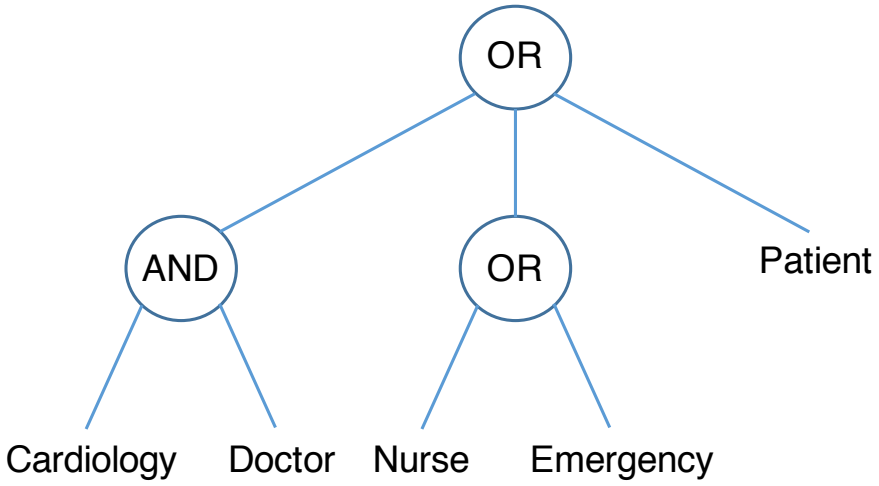


Figure 3.3: An example of access policy P.

Two types of ABE exist based on the placement of the access policy: Ciphertext-Policy ABE and Key-Policy ABE. We provide details of each scheme as follows.

3.4.1 Ciphertext-Policy ABE (CP-ABE)

CP-ABE scheme provides a fine-grained access control to encrypted data similar to Role-Based Access control schemes [73]. Private key of a user is associated with his/her credentials. Ciphertexts specify an access policy and only users whose credentials satisfy the requirements of the access policy can decrypt them. The data can be encrypted without the knowledge of the actual users beforehand and the policy can be specified afterwards. This enables the addition of future users without re-assigning the keys. CP-ABE scheme consists of four algorithms [63] are detailed below:

Setup: generates a master key (k_M) and public parameters (Params). A bilinear group $\mathbb{G}_0$ of order prime p and a generator g is chosen. Two random exponents $\alpha, \beta \in \mathbb{Z}_p$ are selected to compute the parameters:

$$h = g^{\beta}, f = g^{1/\beta}, e(g,g)^{\alpha}$$

where $e(g,g)^{\alpha}$ is the bilinear mapping $\mathbb{G}_0 \rightarrow \mathbb{G}_T$. Public parameters are then published as $\mathsf{Params} = (\mathbb{G}_0, g, h, f, e(g,g)^{\alpha})$ and k_M is selected as $k_M = (\beta, g^{\alpha})$.

Key Generation: takes k_M as input and a set of attributes S specific to a user and generates a private key (k_{PRIV}) by choosing a random $r \in \mathbb{Z}_p$ and computing $D = g^{(\alpha+r)/\beta}$. For each attribute $s_j \in S$, a random $r_j \in \mathbb{Z}_p$ is selected to compute following:

$$D_j = g^r \cdot H(s_j)^{r_j}, \tilde{D}_j = g^{r_j}$$

where $H(s_j)$ is the hash of s_j that maps string s_j to a group element in $\mathbb{G}_0$. The private key k_{PRIV} is published as

$$k_{PRIV} = (D = g^{(\alpha+r)}, \forall s_j \in S : D_j, \tilde{D}_j)$$

Encryption: takes Params, an access policy represented as a tree T defined over all possible attributes and message M to generate ciphertext C.

Decryption: inputs Params, k_{PRIV}, and ciphertext C to generate message M. Decryption will be successful if user's private key satisfies the access structure embedded in ciphertext C.

3.4.2 Key-Policy ABE (KP-ABE)

In KP-ABE [61, 62] the access policy is encoded into the users' private key and a ciphertext is labeled with a set of attributes. KP-ABE schemes place the access policy on the private key of the users and the attributes are associated with the ciphertexts.

3.4.2.1 Lightweight KP-ABE Scheme

A recently proposed ABE scheme [64], which is based on KP-ABE, is proposed as a *lightweight* ABE solution to provide security for resource constrained de-

vices such as Internet-of-Things (IoTs). This scheme is based on ECC instead of bilinear pairings. Bilinear pairings are very expensive for resource constrained devices and lightweight ABE scheme improves both communication and computation overhead by using ECC. Specifically, [64] uses ECIES [111] to provide both data confidentiality and data integrity. This scheme is composed of the following four steps:

Setup: In this step, a central attribute authority who is responsible for key generation, generates public parameters (Params) and master key (k_M). The setup is based on the the universal set of attributes U. For each attribute i in U, a point on elliptic curve P_i is generated by choosing a random $r_i \in \mathbb{Z}_q^*$ and then computing $P_i = r_i \cdot G$. Then a random $r \in \mathbb{Z}_q^*$ is chosen as k_M and master public key is set to $PK = r \cdot G$. Finally Params is published as the set Params= $\{PK, P_1, P_2, \cdots, P_{|U|}\}$.

Key Generation: takes k_M and access policy P and generates decryption key (k_{DEC}).

Encryption: takes input attribute set S, message M and public key parameters Params to generate the corresponding ciphertext. For each attribute i in S, $C_i = r_i \cdot P_i$ is computed by choosing random $r_i \in \mathbb{Z}_q^*$. Encryption of the M is done by using secret key for the symmetric-key cryptography generated by ECIES to compute C. Finally the MAC of the message is computed as $MAC_M = HMAC(M, k_{MAC})$, where k_{MAC} is the y-coordinate of the elliptic curve $Q = r \cdot PK$. Ciphertext is published as the set $\{S, C, MAC_M, C_1, C_2, \cdots C_{|S|}\}$

Decryption: takes ciphertext set $\{S, C, MAC_M, C_1, C_2, \cdots C_{|S|}\}$ encrypted using the attribute set S and uses decryption key k_{DEC} for the policy P to decrypt message M.

4 Fully Homomorphic Encryption

The goal of secure medical cloud computing is to allow computations to be performed on the medical data stored in the cloud while maintaining privacy. Maintaining privacy in cloud storage, can be achieved by encrypting data before storing it. However, doing so will not allow computations to be performed on data as traditional encryption mechanisms reviewed in Chapter 3 do not support such operations. In this chapter, we will introduce a powerful encryption mechanism called *Fully Homomorphic Encryption* (FHE) that enables computations on the encrypted data without decrypting it.

Until very recently, constructing an FHE scheme remained one of the biggest open problems in the field of cryptography. In 2009, a breakthrough was achieved by Craig Gentry who proposed the first "provably-secure" FHE scheme [14]. Schemes proposed before were *partially* homomorphic encryptions [108, 114–118] that allow single type of computation over the ciphertexts. While Gentry's scheme is the first plausible mechanism for an FHE scheme, it is inefficient both in terms of storage and computation, making the scheme impractical. However, developing a practical FHE scheme remains an active research area, and several FHE schemes' implementations have been proposed recently [15, 17, 119–128] to make FHE more practical.

The rest of this chapter is organized as follows. In Section 4.1, a formal definition of FHE is provided. Partially homomorphic encryption schemes are reviewed in Section 4.2. Gentry's FHE scheme is explained in Section 4.3, followed by introduction to one of the most efficient FHE schemes called BGV in Section 4.4. Finally, computation models that will be used for implementing our medical application with FHE are introduced in Section 4.5.

4.1 Fully Homomorphic Encryption Definition

Conventional symmetric-key and public-key cryptosystems encrypt the data such that only authorized parties can access the data. In order to perform operations on the data, one needs to decrypt the encrypted data first and then perform the operations. These conventional schemes become very inefficient, where computation is outsourced to a third party and privacy of the data must be preserved at all times [129].

As early as 1978, Rivest, Adleman and Dertouzos [60] posed the following question

> *Can we do arbitrary computations on data while it remains encrypted, without ever decrypting it?*

The solution to this question is a *fully homomorphic encryption* scheme that allows a third-party (such as the cloud), given any function f, to compute $Enc(f(\text{DATA}))$ from $Enc(\text{DATA})$.

FHE schemes enable computing meaningful operations on the encrypted data without observing the actual data. In other words, an example computation, c = a + b, becomes possible using FHE without actually knowing a and b. With an FHE scheme, one can encrypt the data and store it in a database/cloud, and later ask a third party to perform some operations on the encrypted data. The third

party never sees the original data but performs operations on the ciphertexts only, returning the result in encrypted form, which can only be decrypted by the secret key owner.

A public-key FHE scheme comprises four procedures defined as follows:

Key Generation: takes the security parameter k as input and outputs $(pk, sk, evalk)$ where pk is the public key, sk is the secret key and $evalk$ is the evaluation key that allows computations over encrypted data.

Encryption: takes as input a single bit b and a public key pk and outputs the ciphertext c.

Decryption: takes as input the ciphertext c and secret key sk and outputs b^* (the decrypted data).

Evaluation: takes as input the evaluation key $evalk$, a function $f : \{0,1\}^\ell \rightarrow \{0,1\}$ and ciphertexts $c_1, \ldots, c_\ell$ and outputs a ciphertext c_f.

To compute arbitrary functions on encrypted data, an FHE scheme should be capable of performing homomorphic additions and homomorphic multiplications over the ciphertexts, which corresponds to addition and multiplication operations of the unencrypted message, respectively. Since addition and multiplication provide a complete Turing set of operations, any function can be represented as a combination of additions and multiplications, thereby achieves computing "arbitrary" functions on encrypted data.

Until 2009, the state-of-the-art homomorphic encryption schemes were "partially homomorphic" that permit a single type of operation to be computed over ciphertexts. Specifically, partially homomorphic encryptions allow only homomorphic additions or homomorphic multiplications over the ciphertexts. Additively homomorphic encryption allows an arbitrary number of homomorphic addition

operations to be performed on ciphertexts. Analogously, multiplicatively homomorphic encryption allows an arbitrary number of homomorphic multiplication operations to be performed on ciphertexts. A number of contemporary encryption schemes are in fact either additively or multiplicatively homomorphic. Examples include the Paillier encryption scheme [116] and the ElGamal encryption scheme [108], which are additively and multiplicatively homomorphic, respectively.

The first (practical) scheme that went beyond a single operation was developed by Boneh, Goh and Nissim [118] based on bilinear pairings on elliptic curves that could perform a single multiplication operation in between arbitrary many additions (i.e., before and after).

With his breakthrough work in 2009, Gentry [14] proposed the first mechanism for an FHE scheme that could perform an arbitrary number of additions and multiplications homomorphically. The groundbreaking work of Gentry not only constructed the first FHE scheme, but also provided a *blueprint* to construct such schemes. The starting point of his blueprint is a "Somewhat" Homomorphic Encryption (SWHE) scheme that allows computation of polynomials of some maximum degree d over encrypted data. SWHE had limited capability of executing homomorphic operations due to the increase of "noise" inside the ciphertext with each homomorphic operation, which causes a decryption error. Gentry proposed a remarkable "bootstrapping" method to convert SWHE to FHE. The bootstapping method allows evaluating the decryption operation homomorphically with SWHE. This refreshes the noise inside to ciphertext, thereby allows executing arbitrary number of homomorphic operations over ciphertexts. Unfortunately, this step makes the FHE scheme inefficient and relies on complex cryptographic assumptions. Since Gentry's work, there have been several works following the same blueprint yielding simpler and more efficient schemes [14, 119–125, 130].

4.2 Partially Homomorphic Encryption Schemes

The homomorphic schemes proposed before Gentry's first FHE scheme [14] were *partially* homomorphic that could perform only homomorphic additions or homomorphic multiplications. Table 4.1 presents important partially homomorphic encryption schemes and their homomorphic properties. The only exception among these schemes is the BGN scheme [118], which could perform an arbitrary number of homomorphic additions and a single homomorphic multiplication.

Table 4.1: Partially Homomorphic Encryption Schemes

	Homomorphic Property	
Encryption Scheme	Additive	Multiplicative
RSA [113]		✓
Goldwasser-Micali [114]	✓	
ElGamal [108]		✓
Paillier [116]	✓	
BGN [118]	✓	1

4.2.1 RSA

RSA is one of the most widely used public-key encryption schemes, and its properties gave birth to the idea of homomorphic encryption.

RSA is based on the hardness of factoring $N = p \cdot q$ into its prime number multiplicands p and q. Just given N, it is computationally infeasible to find p or q.

RSA consists of three algorithms defined as follows:

Key Generation: generates public key pk and secret key sk as follows: First, two distinct prime numbers p and q are selected randomly to compute $N = p \cdot q$. Then, the Euler toitent function ($\phi(N)$) is computed as

$$\phi(N) = \phi(p) \cdot \phi(q) = (p-1) \cdot (q-1)$$

An integer e with the property $gcd(e, \phi(N)) = 1$ is selected. Then integer d, which is the multiplicative inverse of e is calculated. Finally, the public key pk is selected as

$$pk = (N, e)$$

and the secret key sk is selected as

$$sk = (p, q, d)$$

Encryption: encrypts message m to ciphertext c by using the public key $pk = (e, N)$ as follows:

$$c = m^e \pmod{N}$$

Decryption: decrypts the ciphertext c by using the secret key $sk = (p, q, d)$ as follows:

$$\begin{aligned} m &= c^d \pmod{N} \\ &= (m^e)^d \pmod{N} \\ &= m^{e \cdot d = 1} \pmod{N} \end{aligned}$$

RSA is *multiplicatively* homomorphic, such that the multiplication of ciphertexts c_1, c_2 which encrypt messages m_1, m_2 results in the ciphertext that encrypts

message $m_1 \cdot m_2 \pmod N$. The multiplicative homomorphic property of RSA can be shown as

$$\begin{aligned} Enc(m_1, pk) &= m_1^e \pmod N \\ Enc(m_1, pk) &= m_2^e \pmod N \\ Enc(m_1, pk) \cdot Enc(m_2, pk) &= m_1^e \cdot m_2^e \pmod N \\ &= (m_1 \cdot m_2)^e \pmod N \\ &= Enc(m_1 \cdot m_2, pk) \end{aligned}$$

Unfortunately, RSA in practice requires padding that removes its homomorphic property.

4.2.2 Goldwasser-Micali Encryption Scheme

The Goldwasser-Micali (GM) encryption scheme [114] is a public-key encryption algorithm that achieves first semantically secure homomorphic encryption. The GM scheme is also the first probabilistic public-key encryption scheme that introduced the widely used definition of semantic security.

The GM scheme relies on the hardness of the quadratic residuosity problem. Specifically, given an integer x in modulo N, it is computationally intractable to find whether x is a quadratic residue. A quadratic residue check can be efficiently performed if the factorization of $N = p \cdot q$ is known by first computing

$$\begin{aligned} x_p &= x \pmod p \\ x_q &= x \pmod q \end{aligned}$$

and then checking

$$x_p^{(p-1)/2} = 1 \pmod{p}$$
$$x_q^{(q-1)/2} = 1 \pmod{q}$$

If both equalities are satisfied, then x is a quadratic-residue, otherwise it is a non-residue.

The GM scheme consists of three algorithms defined as follows:

Key Generation: choses two random prime numbers p and q such that $p = q = 3 \pmod 4$ and $N = p \cdot q$. A non-residue integer a is selected that satisfies the following properties

$$a_p^{(p-1)/2} = -1 \pmod{p}, \quad a_q^{(q-1)/2} = -1 \pmod{q}$$

Finally, the public key pk is selected as

$$pk = (a, N)$$

and the secret key sk is selected as

$$sk = (p, q)$$

Encryption: encrypts bits of message $m = (m_1, m_2, \ldots, m_n)$ with the public key $pk = (a, N)$ by choosing a random number $r_i \pmod N$ for each bit such that $gcd(r_i, N) = 1$. For each bit m_i of the message a ciphertext c_i is generated as

$$c_i = r_i^2 \cdot a^{m_i} \pmod N$$

Decryption: decrypts ciphertext c_i with secret key $sk = (p, q)$ by checking

whether c_i is a quadratic residue. If c_i is a quadratic residue, then $m_i = 0$ otherwise $m_i = 1$. Checking whether c_i is a quadratic residue can be easily computed, since secret key sk is the factorization of N. The original message m is reconstructed by combining all m_i as $m = (m_1, m_2, \ldots, m_n)$.

The GM scheme is *additively* homomorphic such that the multiplications of ciphertexts c_1 and c_2 encrypting messages m_1 and m_2, results in the ciphertext that encrypts $m_1 + m_2 \pmod 2$. Since m_i's are the bits of message m, the addition operation corresponds to an XOR operation. The additive homomorphic property can be shown as

$$\begin{aligned} Enc(m_1, pk) &= r_1^2 \cdot a^{m_1} \pmod N \\ Enc(m_2, pk) &= r_2^2 \cdot a^{m_2} \pmod N \\ Enc(m_1, pk) \cdot Enc(m_2, pk) &= (r_1^2 \cdot a^{m_1}) \cdot (r_2^2 \cdot a^{m_2}) \pmod N \\ &= (r_1 \cdot r_2)^2 \cdot a^{m_1 + m_2} \pmod N \\ &= Enc(m_1 + m_2, pk) \end{aligned}$$

4.2.3 ElGamal Encryption Scheme

The ElGamal encryption scheme [108] is a probabilistic public-key encryption based on the DiffieHellman key exchange [70]. The ElGamal encryption scheme is defined over the cyclic group G, and its security is based on computing discrete logarithms in cyclic groups.

The ElGamal encryption scheme consists of three algorithms defined as follows:

Key Generation: generates a cyclic group G of order q with generator g. Secret key sk is selected as a random element $x \in 1, \ldots, q-1$. Public key pk is selected as

$$pk = (G, q, g, h = g^x)$$

Encryption: encrypts message m with public key $pk = (G, q, g, h)$, by first choosing random $r \in 1, \ldots, q-1$, and then computing ciphertext $c = (v, y)$ as follows:

$$v = g^r$$
$$y = m \cdot h^r$$

Decryption: decrypts ciphertext $c = (v, y)$ with secret key $sk = x$, by first computing

$$a = v^x$$

and then using a to generate message m as follows:

$$\begin{aligned} m &= y \cdot a^{-1} \\ &= (m \cdot h^r) \cdot (v^x)^{-1} \\ &= (m \cdot (g^x)^r) \cdot ((g^r)^x)^{-1} \\ &= m \cdot g^{xr} \cdot g^{-rx} \\ &= m \end{aligned}$$

The ElGamal scheme is *multiplicatively* homomorphic such that the multiplication of ciphertexts c_1 and c_2 encrypting messages m_1 and m_2, results in the ciphertext that encrypts $m_1 \cdot m_2$. The multiplicative homomorphic property can be shown as

$$\begin{aligned} Enc(m_1, pk) &= (g^{r_1}, m_1 \cdot h^{r_1}) \\ Enc(m_2, pk) &= (g^{r_2}, m_2 \cdot h^{r_2}) \\ Enc(m_1, pk) \cdot Enc(m_2, pk) &= ((g^{r_1} \cdot g^{r_2}), (m_1 \cdot h^{r_1} \cdot m_2 \cdot h^{r_2})) \\ &= (g^{r_1+r_2}, (m_1 \cdot m_2) \cdot h^{r_1+r_2}) \end{aligned}$$

4.2.4 Paillier Encryption Scheme

The Paillier Encryption scheme [116] is a public-key cryptosytem, which is based on difficulty of finding n^{th} residue of the composite numbers. Specifically, given z and N^2, where $N = p \cdot q$ is a composite number, it's hard to find y that observe the following relationship

$$z = y^N \mod N^2$$

The Paillier encryption scheme consists of three algorithms defined as follows:

Key Generation: selects two random primes p and q to generate composite number $N = p \cdot q$. Then $\lambda = lcm(p-1, q-1)$, which is the least common multiplier of $p-1$ and $q-1$, is calculated. Random $g \in \mathbb{Z}^*_{N^2}$, which is a generator for the $\mathbb{Z}^*_{N^2}$ is selected and its multiplicative inverse with respect to module N is calculated as

$$\mu = L(g^\lambda \mod N^2)^{-1} \pmod N$$

where L is the function that computes $L(k) = (k-1)/N$. Finally, the public key is selected as

$$pk = (N, g)$$

and the secret key is selected as

$$sk = (\lambda, \mu)$$

Encryption: encrypts the message m with random $r \in \mathbb{Z}^*_{N^2}$ to the ciphertext c using public key $pk = (N, g)$ as follows

$$c = g^m \cdot r^n \pmod{N^2}$$

Decryption: decrypts the ciphertext c to the message m using secret key $sk = (\lambda, \mu)$ as follows

$$m = L(c^{\lambda} \pmod{N^2}) \cdot \mu \pmod{N}$$

The Paillier encryption scheme is *additively* homomorphic such that the multiplication of ciphertexts c_1 and c_2 encrypting messages m_1 and m_2, results in the ciphertext that encrypts $m_1 + m_2 \pmod{N}$. The additive homomorphic property can be shown as

$$\begin{aligned}
Enc(m_1, pk) &= g^{m_1} \cdot r_1^N \pmod{N^2} \\
Enc(m_2, pk) &= g^{m_2} \cdot r_2^N \pmod{N^2} \\
Enc(m_1, pk) \cdot Enc(m_2, pk) &= (g^{m_1} \cdot r_1^N \pmod{N^2}) \cdot (g^{m_2} \cdot r_2^N \pmod{N^2}) \\
&= g^{m_1+m_2} \cdot (r_1 \cdot r_2)^N \pmod{N^2} \\
&= Enc(m_1 + m_2, pk)
\end{aligned}$$

4.2.5 Boneh-Goh-Nissim (BGN) Encryption Scheme

The BGN encryption scheme [118] was the first encryption scheme to allow multiple homomorphic additions and a single homomorphic multiplication. The BGN scheme is based on pairings on elliptic curves, where a pairing on an elliptic curve is a map defined as follows:

$$e : G_1 \times G_2 \rightarrow G_T$$

where G_1, G_2 are additive groups and G_T is multiplicative group, all with prime order p. If $G_1 = G_2 = G$, then the pairing is called as symmetric pairing. Furthermore if G is cyclic, then for any generator P of G_1 and Q of G_2 , the pairing map e is commutative

$$e(P, Q) = e(Q, P)$$

The BGN scheme consists of three algorithms defined as follows:

Key Generation: starts with generating the pairing map $e : G \times G \to G_1$. G is a cyclic group of order $N = p_1 \cdot p_2$, where p_1 and p_2 are two large prime numbers. g, u are selected as two generator of G and a generator for the subgroup of G with order p_1 is computed as $h = u^{p_2}$. Finally, the public key is selected as

$$pk = (N, G, G_1, e, g, h)$$

and the secret key is selected as

$$sk = p_1$$

Encryption: encrypts message m with public key $pk = (N, G, G_1, e, g, h)$ by choosing a random $r \in 1, 2, \ldots, N$ and computing the ciphertext c as follows:

$$c = g^m \cdot h^r$$

Decryption: decrypts ciphertext c to the message m using the secret key $sk = p_1$, by first computing

$$t = c^{p_1} = (g^m h^r)^{p_1} = (g^{p_1})^m$$

and then finding the discrete logarithm of t with respect to the base g^{p_1}.

The BGN scheme is *additively* homomorphic such that the multiplication of ciphertexts c_1 and c_2 encrypting messages m_1 and m_2, results in the ciphertext that encrypts $m_1 + m_2 \pmod N$. The additive homomorphic property can be

shown as

$$\begin{aligned} Enc(m_1, pk) &= g^{m_1} \cdot h^{r_1} \\ Enc(m_2, pk) &= g^{m_2} \cdot h^{r_2} \\ Enc(m_1, pk) \cdot Enc(m_2, pk) &= (g^{m_1} \cdot h^{r_1}) \cdot (g^{m_2} \cdot h^{r_2}) \\ &= g^{m_1+m_2} \cdot h^{r_1+r_2} \\ &= Enc(m_1 + m_2, pk) \end{aligned}$$

The BGN scheme also allows a *single* homomorphic multiplication by using the bilinear map properties. Assume that, $g_1 = e(g, g)$ and $h1 = e(g, h)$ are two pairings, where $g1$ and h_1 are of order N and p_1, respectively. For an $\alpha \in Z$, compute $h = g^{\alpha \cdot p_2}$ and generate ciphertexts c_1 and c_2 as

$$c_1 = g^{m_1} \cdot h^{r_1}, \qquad c_2 = g^{m_2} \cdot h^{r_2}$$

Homomorphic multiplication that generates ciphertext c encrypting $m_1 \cdot m_2 \pmod N$ is computed by setting $c = e(c_1, c_2)$. The multiplicative homomorphic property can be shown as

$$\begin{aligned} c &= e(c_1, c_2) \\ &= e(g^{m_1} h^{r_1}, g^{m_2} h^{r_2}) \\ &= e(g^{m_1+\alpha \cdot p_2 \cdot r_1}, g^{m_2+\alpha \cdot p_2 \cdot r_2}) \\ &= e(g, g)^{(m_1+\alpha \cdot p_2 \cdot r_1)(m_2+\alpha p_2 \cdot r_2)} \\ &= e(g, g)^{m_1 \cdot m_2+\alpha \cdot p_2 \cdot (m_1 \cdot r_2+m_2 \cdot r_1+\alpha \cdot p_2 \cdot r_1 \cdot r_2)} \\ &= e(g, g)^{m_1 \cdot m_2} h^{r+m_1 \cdot r_2+m_2 \cdot r_1+\alpha \cdot p_2 \cdot r_1 \cdot r_2} \end{aligned}$$

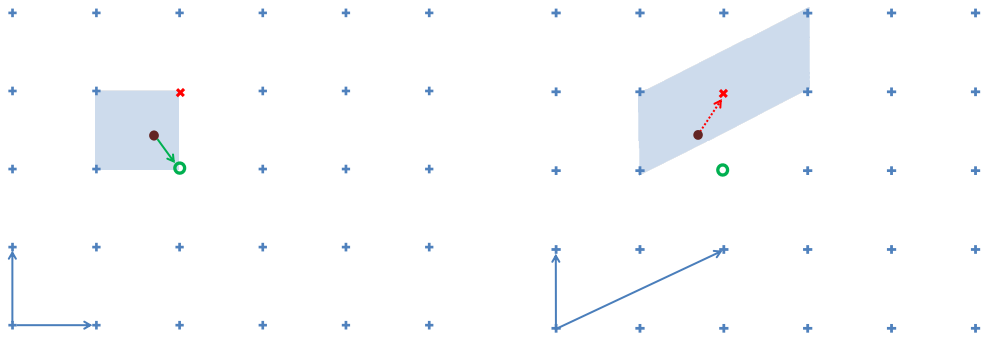
where c is an encryption of $m_1 \cdot m_2 \pmod N$, but in G_1 rather than G. Homomorphic additions can be still performed in G_1.

4.3 Gentry's FHE Scheme

In 2009, Gentry [14] proposed the first plausible mechanism for an FHE scheme that could perform an arbitrary number of additions and multiplications homomorphically. His proposal was based on ideal lattices, which was a variant of GGH cryptosystem [131]. GGH scheme was the first public-key cryptosystem based on hard problems in lattices.

A lattice is a discrete additive subgroup in n-dimensional space, which can be represented by its basis vector. The fact that a lattice can have an infinite number of bases plays a key role for creating a public-key cryptosystem. Similar to other public key cryptosytems [113] [70], security of the lattice based cryptosystems is based on an intractable problem that is very hard to solve unless the secret key is known. The hard problem in Gentry's FHE scheme is the Closest Vector Problem (CVP), which states that given a point in n-dimensional space, it is hard to find the closest lattice point. If a *good* basis is known for the lattice, then one can use Babai's nearest-vector approximation algorithm [132] to solve the CVP problem efficiently. The *good* basis of a lattice consists of almost orthogonal base vectors having a large decryption radius, and it is used as the secret key. The public key is chosen as Hermite-Normal-Form of the good basis [133], such that its base vectors are skewed and having a small decryption radius. Figure 4.1 demonstrates the difference of decrypting a ciphertext with a good (Figure 4.1(a)) and a bad (Figure 4.1(b)) basis vector, where the result is mapped to an incorrect point on the lattice when a bad basis vector is used.

In Gentry's FHE scheme, encryption is performed by first mapping a message to a lattice point and then adding a small random noise to create the final ciphertext. The decryption can be done only by using a good basis, which is only known by the secret-key holder. Homomorphic addition and homomorphic multiplication operations are performed by adding and multiplying lattice points,



(a) HE with good basis (b) HE with bad basis

Figure 4.1: Homomorphic decryption with good (a) and bad (b) basis vectors mapping to a correct and incorrect result, respectively.

respectively. During the homomorphic operations, the noise inside the ciphertext grows with each operation. Specifically, homomorphic addition roughly doubles the noise, while homomorphic multiplication squares the noise. After several operations, the magnitude of the noise in the ciphertext exceeds the threshold at which a successful decryption is no longer possible even with the knowledge of a good basis. This limits the number of operations that can be performed with this scheme, and hence this scheme is also referred to as SomeWhat Homomorphic Encryption (SWHE). Gentry proposed a remarkable bootstrapping method (i.e., recryption) to transform the SWHE scheme into an FHE scheme by evaluating the decryption function homomorphically. The recryption operation resets the noise inside the ciphertext and enables computation of arbitrary functions indefinitely.

Although Gentry's scheme is the first plausible mechanism for an FHE scheme, it has several inefficiencies both in terms of storage and computation. Messages are encrypted bitwise, and in order to increase the noise threshold, the ciphertext size must be large, which results in expansion in storage space. For example, the size of a ciphertext encrypting a 1-bit message could be multi-million bits, which presents an unacceptable data expansion ratio for most practical implementations. The homomorphic operations over very large ciphertexts are also compute-intensive,

and the recryption operation that needs to be performed before each homomorphic multiplication makes Gentry's FHE scheme impractical.

Following Gentry's FHE scheme [14], several new FHE schemes [17, 119, 122–125] have been proposed. These schemes followed similar steps as Gentry's scheme. Specifically, they start off with a somewhat homomorphic scheme (SWHE) that is capable of evaluating low-degree polynomials. To achieve FHE from SWHE, they apply a bootstrapping method and evaluate the decryption circuit to reduce the noise inside the ciphertexts. The bootstrapping method usually requires squashing the decryption circuit to lower its depth. Gentry's scheme and the schemes that followed his *blueprint* steps faced several problems. First, the bootstrapping method, which enables FHE, is a very compute intensive operation. Implementation of Gentry's FHE [134] reports the execution time of bootstrapping as 30 seconds for "small" setting and 30 minutes for "large" setting. Furthermore, the ciphertext sizes are usually large to allow enough room for noise growth during homomorphic additions and multiplications. The same implementation in [134] has ciphertext size of 70MB for "small" setting and 2.3GB for "large" setting. Both the bootstrapping method and the large ciphertext size affect the performance of these schemes poorly and make them impractical.

Recently proposed FHE schemes [15,120,121,126,127] deviate from the "blueprint" and introduced different techniques and methods to make FHE more practical. These schemes [135–137] rely on polynomial rings that makes operations faster. A common theme for these schemes is to base their security assumptions on hard problems in lattices. Most of the schemes rely on the Ring Learning with Errors (RLWE) problem to build an FHE scheme.

4.4 BGV Scheme

Our proposed system is based on the Brakerski-Gentry-Vaikuntanathan (BGV) scheme [15]. We choose the BGV scheme to implement our medical applications because at present the BGV scheme [15] is currently one of the theoretically most efficient FHE schemes. Also, the BGV scheme has an open-source implementation called HElib [128], which has been actively developed and improved.

The BGV scheme is based on Ring Learning with Errors (RLWE) primitives [138] and is defined over polynomial rings of the form $\mathbb{A} = \mathbb{Z}[X]/\Phi_m(X)$. $\Phi_m(X)$ is the m'th cyclotomic polynomial and m is the input parameter for the scheme. Both plaintexts and ciphertexts are represented as elements in polynomial rings. The plaintext space for the scheme is the ring $\mathbb{A}_p = \mathbb{A}/p\mathbb{A}$, where p is a prime number. The ciphertext space of the scheme is also defined as $\mathbb{A}_q = \mathbb{A}/q\mathbb{A}$, where q is a product of prime numbers.

Several methods are introduced by the BGV scheme to improve the performance of earlier FHE schemes. The plaintext can be packed with multi-bit messages by using the techniques proposed in [130]. The packing is achieved by factoring polynomial $\Phi_m(X)$ modulo p into ℓ irreducible polynomials (i.e., $\Phi_m(X) = F_1(X) \cdot F_p(X) \cdots F_\ell(X) \pmod{p}$). Each factor is called a "slot" with a degree $d = \phi(m)/\ell$ polynomial. This allows ciphertexts to encrypt multiple messages at once, thereby permits execution of homomorphic operations in parallel as Single Instruction Multiple Data (SIMD) fashion, where operations on a ciphertext translate into slot-wise operations over plaintext. The expensive recrypt operation can be avoided by using the *leveled* version of the BGV scheme, where homomorphic operations are performed up to L levels. Since each homomorphic addition and multiplication increases the noise in the ciphertext, only a limited number of homomorphic operations can be performed. While homomorphic addition does not increase the noise level significantly, homomorphic multiplication

roughly squares the noise amount. Thus the level L is determined by the depth of multiplication operations for the function to be evaluated. The level of the function to be computed can be defined beforehand, and then the parameters of the scheme can be adjusted during the key generation.

4.4.1 Overview of the techniques used in BGV

The BGV scheme proposes several techniques to improve the performance. One of the biggest improvements is a better noise management technique called Modulus Switching [121]. This method reduces the noise inside the ciphertext by switching to a smaller modulus q. This reduces the noise inside the ciphertext and therefore allows more homomorphic operations to be performed without a decryption error. Specifically, modulus switching increases the number of homomorphic multiplications exponentially. With this method, BGV proposes a variant called *leveled* FHE scheme that avoids performing costly bootstrapping operation. We will use the *leveled* version of BGV to implement the proposed medical application.

Another key technique is called key-switching or *relinearization*, which reduces the size of the ciphertexts after homomorphic operations are performed. While homomorphic addition does not affect the ciphertext size, homomorphic multiplication increases the ciphertext size by adding another ring element to the ciphertext. The key-switching method reduces ciphertext size back to two ring elements. This is done using multiple keys, where each key is encrypted with another key. While key-switching reduces the ciphertext size, it increases the public key size due to the inclusion of multiple keys.

Finally another improvement is called *batching*, which enables packing multiple messages inside a plaintext. This allows homomorphic operations to be executed in parallel similar to SIMD fashion.

4.4.2 *Leveled* FHE scheme

Following Gentry [14], FHE schemes up to date [119–125, 130] rely on introducing a small *noise* into a ciphertext during encryption. This noise grows with each operation and can cause decryption errors if it is allowed to exceed a certain threshold. This requires performing the computationally expensive *recrypt* operation to reset the noise after every homomorphic multiplication, which would otherwise increase the noise exponentially.

BGV introduces a *leveled* FHE scheme that avoids the expensive *recrypt* operation. The *leveled* FHE scheme uses a better noise management technique called *modulus-switching* [121], which allows performing cascaded homomorphic multiplications ($\times_h$) without causing decryption errors. A parameter L (the *Level*) is introduced, which must be determined before starting any computation. The level L is predominantly determined by the depth of the $\times_h$ operations for the function to be evaluated, and for the rest of this dissertation we will use *multiplicative depth* and level L interchangeably. Right after encryption, each ciphertext is set to a level L and L is reduced by one after each $\times_h$ until $L=1$, at which point further $\times_h$ operations can cause decryption errors.

Leveled FHE improves traditional FHE performance, but introduces an implementation burden: L must be determined *a priori* (before performing any homomorphic operation). During the implementation of our medical application, we will calculate the multiplicative depth of each computation and set the level L accordingly. For the rest of this section, we will use the lower case $x_7 \cdots x_0$ notation to denote the plaintext slots, and the upper case bold notation $\boldsymbol{X}$ to denote the ciphertext, i.e., the encrypted version of plaintext $X=(x_7 \cdots x_0)$. Therefore, $\boldsymbol{X} = Enc(X) = Enc(x_7 \cdots x_0)$, where $Enc()$ is the homomorphic encryption operation.

4.4.3 Plaintext Space

In the BGV scheme, plaintexts are represented as polynomial rings in the $GF(p^d)$, where p is a prime number that defines the range of polynomial coefficients and d is the degree of the polynomials. Homomorphic addition and multiplication of ciphertexts correspond to addition and multiplication of plaintexts in the specified polynomial ring, respectively. We choose the polynomial ring in $GF(2)$ (i.e., p= 2, d=1), where homomorphic addition and multiplication of ciphertexts translate to XOR and AND operations on the plaintexts, respectively. This "functionally complete" set (i.e., XOR and AND) will allow the fundamental operations of our medical application to be represented as a binary circuit using a combination of XOR and AND gates.

4.4.4 Message Packing

Representing plaintexts as polynomial rings in $GF(p^d)$ allows the "packing" of multiple messages into a plaintext by partitioning it into independent "slots" [130], thereby permitting the execution of the same homomorphic operation on multiple slots in a Single Instruction Multiple Data (SIMD) fashion. Figure 4.2 exemplifies a ciphertext $\boldsymbol{X}$, encrypting a plaintext that packs two 4-bit messages ($X[0]$ and $X[1]$), into 8 plaintext slots.

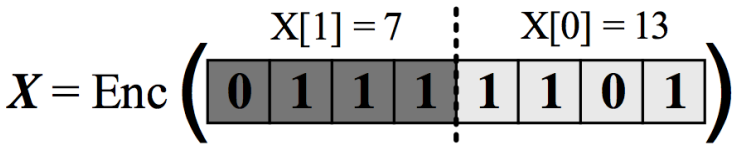


Figure 4.2: Two 4-bit messages ($X[0]$, $X[1]$) packed into 8 plaintext slots.

4.4.5 Primitive Operations in BGV

Packing allows `nSlots` parallel operations on plaintext slots, but with restricted applicability, as will be detailed in Chapter 5. From a set of existing operations

in BGV, we use an orthogonal set of four as shown in Figure 4.3.

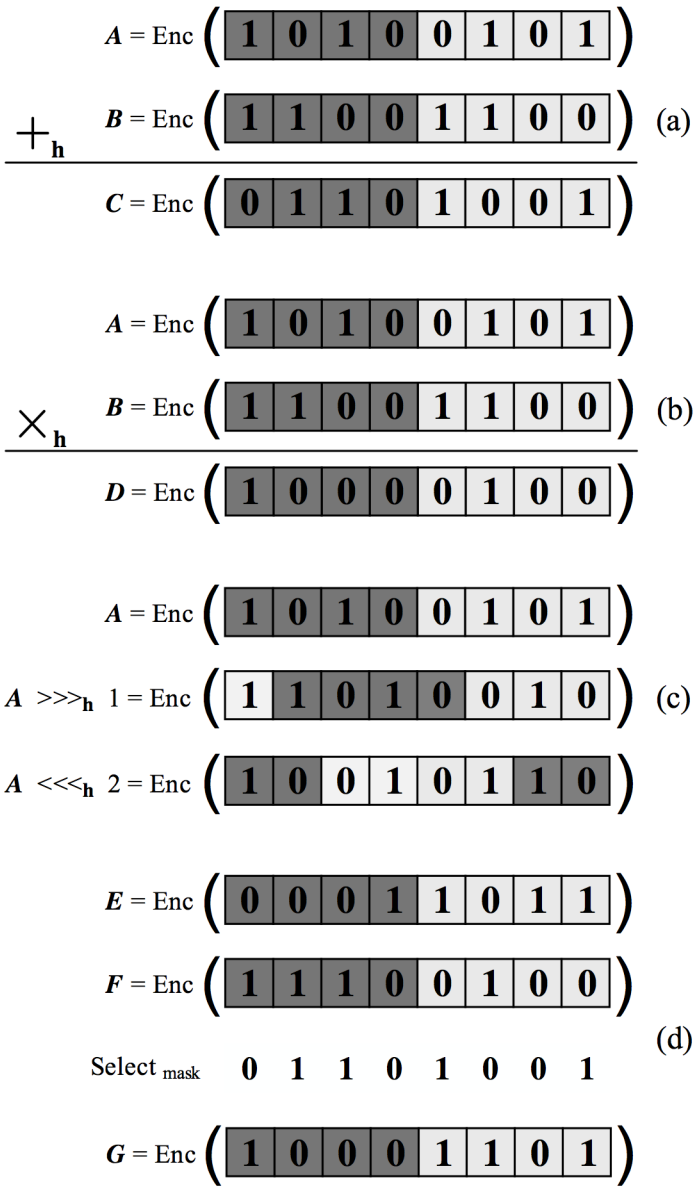


Figure 4.3: Computational Primitives in BGV.

Homomorphic Addition ($+_h$): of two ciphertexts corresponds to a slot-wise XOR of the corresponding plaintext in $GF(2)$, as shown in Figure 4.3a. $+_h$ does not affect the level L of the BGV scheme.

Homomorphic Multiplication ($\times_h$): of two ciphertexts corresponds to a slot-wise AND operation of the corresponding plaintexts, as shown in Figure 4.3b.

$\times_h$ operation adds one to the level L of the ciphertext. Therefore, the depth of multiplications will determine the required level of the BGV scheme.

Rotate ($>>>_h$, $<<<_h$): provides rotation of slots similar to a barrel shifter as shown in Figure 4.3c. Slots will wrap around based on the rotation direction, thereby potentially garbling the data contained in neighboring slots. This will be corrected using Select operations.

Select (sel_{mask}): chooses between the slots of two plaintexts based on the selection mask as shown in Figure 4.3d, through an unencrypted binary vector. We will use Select to mask out the bits that are diffused from other messages after a Rotate.

4.4.6 HElib Library

The HElib library that will be used in our analysis relies on the BGV encryption scheme [15] and is an open source implementation [128] by Halevi and Shoup. To gain insight into the internal operation of this library, certain implementation concepts must be understood. For example, since HElib uses a leveled FHE scheme, this concept of the computation level will be explained in detail shortly.

The "Level" Concept: One of the major improvements in the FHE schemes that were introduced after Gentry's original scheme is the concept of *computation level*. All FHE schemes introduced up to date rely on a small noise that is incorporated into the ciphertext during encryption. When the decryption key is not known, this noise makes decryption intractable, since it makes the decryption problem substantially harder than the case where there is no noise. While this intentional noise helps the security of the FHE scheme, it comes at a steep price: Each FHE operation performed on this *noisy* ciphertext makes this ciphertext more noisy after each operation. While the effect of this growing noise is much smaller for addition operations, multiplication makes the noise grow expo-

nentially. While evaluating a function homomorphically, a chain of addition and multiplication operations are performed, each contributing to the noise partially.

At some point during these chains of operations, the noise reaches a threshold, where decryption of the ciphertext no longer yields the correct plaintext. This threshold of the noise must never be exceeded to ensure correct decryption. *Noise Management*, i.e., guessing and controlling the amount of noise is, therefore, one of the most important aspects of any lattice-based FHE scheme. For example, for a threshold parameter of 40, a maximum of 40 multiplications can be performed, which can be intermixed with a much higher number of additions. Once this point has been reached (i.e., 40 multiplications), a decryption must be performed to reset the noise. Clearly, this implies switching to the unencrypted domain. Therefore, only 40 multiplications can be done in encrypted domain, after which the results must be transferred to the "friendly" source and decrypted. An important contribution by Gentry was to provide a special bootstrapping procedure that allows re-encrypting and resetting the noise without explicitly decrypting. Thereafter, 40 more multiplications can be performed before invoking the bootstrapping procedure again.

More recent FHE schemes are leveled-FHE schemes where there is a control parameter known as the *level*. In most constructions, this level is the maximum number of cascading multiplication operations that can be performed in a sequence of computations. If for a particular application this level can be estimated *a priori*, then the leveled-FHE scheme can be instantiated at the right level.

Plaintext and Ciphertext Spaces: A "message" is defined as a string of bits to communicate between two parties. In the case of conventional cryptography, a message is encrypted with a public key and can only be decrypted when a private key is known. Therefore, in a communication system, the *transmitter* encrypts a message of M-bit length, and the *receiver* decrypts this message to obtain the original M-bits consisting of the message. In the case of the BGV

scheme, the goal of the receiver is not to observe the message, but, rather, to perform computations on it. Therefore, the encryption operation should simply convert the message into a form which can be later used for computation. In the case of BGV, messages are encoded as polynomial rings in the $GF(p^d)$, where p is a prime number, and d is the degree of the polynomial that is representing the message. With this definition, homomorphic addition of a plaintext corresponds to the addition of the ciphertext. Furthermore, homomorphic multiplication of the plaintext polynomial ring corresponds to the multiplication of its ciphertext.

In the simplest case, where $p = 2$ and $d = 1$, each message is in $GF(2)$ and the multiplication operation reduces to logical AND. Alternatively, addition operation reduces to XOR. These two operations are a functionally complete set, i.e., any operation can be represented as a combination of these two operations. An extension of the BGV scheme pursued by Smart and Vercauteren [130] allows "packing" of multiple messages into a plaintext. Each packed message occupies a "slot" of the plaintext and corresponding ciphertext using their terminology. Any homomorphic operation performed on a packed ciphertext has the effect of applying the same operation on every slot. In essence, packed ciphertexts allow SIMD-like operations to be performed on an encrypted vector of data. The HElib implementation, in fact, considers this extension. Each packed plaintext will be encrypted into a single ciphertext, on which the aforementioned XOR and AND (i.e., homomorphic addition and multiplication, respectively) operations can be performed.

Available Operations in HElib: As mentioned above the extended BGV scheme allows operations to be applied to a set of messages packed in a ciphertext. While this results in significant performance improvements, it requires careful formulation of the functions that are being evaluated. Furthermore, it makes certain operations, such as, rotation and selection necessary to cope with the complexities arising from the "packing" concept. Details of the operations are as

follows:

Homomorphic Addition: While the $GF(p^d)$ implementation of homomorphic addition is capable of adding packed integers in the corresponding ring, we prefer $GF(2)$ due to the nature of the problems we are applying FHE to. In $GF(2)$, homomorphic addition operation simply turns into the bitwise XOR operation. Assume that, two plaintexts A and B contain 682 slots each in $GF(2)$ and we are interested in performing a homomorphic addition on $Enc_h(A)$ and $Enc_h(B)$. The result is $Enc_h(C) = Enc_h(A) + Enc_h(B)$. This assumes that, the result is being stored in a ciphertext whose decrypted version is C which also contains 682 slots, just like A and B. Since we performed a $GF(2)$ addition (i.e., XOR) operation on A and B, what we did corresponds to performing $C = A \oplus B$ in the un-encrypted domain, which is bitwise XOR or every plaintext slot (bit) individually. More specifically, we performed $C[n] = A[n] \oplus B[n]$, where $A[n], B[n]$, and $C[n]$ are the n^{th} individual slot (i.e., n^{th} bit in $GF(2)$) of A, B, and C plaintexts.

Homomorphic Multiplication: Homomorphic multiplication performs bitwise AND operations on every plaintext slot. Following the same example as before, assume that, A and B are plaintexts with 682 slots each, and $Enc_h(A)$ and $Enc_h(B)$ are their corresponding ciphertexts. Therefore, $Enc_h(C) = Enc_h(A) \times_h Enc_h(B)$ operation on ciphertexts corresponds to the $C = A \wedge B$ in the un-encrypted domain, which is bitwise AND or every plaintext slot (bit) individually. More specifically, we performed $C[n] = A[n] \wedge B[n]$, where $A[n], B[n]$, and $C[n]$ are the n^{th} individual slot (i.e., n^{th} bit in $GF(2)$) of A, B, and C plaintexts.

Rotation: As can be observed from the description of the homomorphic addition and multiplication operations, it is very difficult to define the evaluation function in terms of just these two operations. This is an artifact due to SIMD nature of these operations, where same operation is applied to in parallel to the bits stored in the same slot index. However, many complex function implementations require performing operations over bits that have different bit-index. HElib

uses Rotation or Shift operations to shuffle the plaintext slots for proper alignment. Out of these two operations, we found the Rotation to be slightly less computationally intensive and will be using it in our implementation.

The Rotation operation, when applied to a ciphertext, rotates all of the messages in the plaintext slots. More specifically, assume that A is a plaintext and $Enc_h(A)$ is its encrypted version. $<<<_h$ operation on the ciphertext $Enc_h(A)$ is equivalent to $<<<$ of plaintext A in the unencrypted domain. This has the effect of converting the original 682 bit plaintext A[681 680 679 ... 2 1 0] to A[680 679 678 ... 2 1 0 681], where 681, 680, ... 0 indicate the slot number of the plaintext. A few important notes to make about this operation are : 1) An "undesired" data element from slot 681 "diffuses" into slot 0 in the example above, which must be eliminated later, after the Rotation operation, 2) Since the value of the diffused undesired bit is not known, the only way to eliminate it involves setting it to a known value, 3) while a Left Rotation is described above, a Right rotation also available.

Bit Selection: Let A, and B be two plaintexts and S be a selection mask. This operation chooses specific slots from A and B according to the selection masks. For example, assume that, S = [0 1 0 0 0 1 ...]. In the selection operation $C = Select(A, B, S)$, where A and B are 682 slot plaintexts as described before, C would end up including a C = [A[681] B[680] A[679] A[678] A[677] B[676] ...], which is a masked selection of bits from either plaintext A or B. While applicable to many useful functions, one immediate use of this is in eliminating the undesired diffused message bits after a rotation. For this, if a fixed value (i.e., all ones) is stored in plaintext B, selecting bits from B by using $C = Select(A, B, S)$ will guarantee a known value in C in every slot that is specific as "1" in the selection mask.

Performance Characteristics of the Available HElib Operations: Figure X shows the runtime of the homomorphic multiplication, addition, and ro-

tation operations, denoted as HMul, HAdd, and HRotate. As can be clearly observed from this plot, homomorphic addition operations require a negligible runtime as compared to multiplication and rotation operations. Therefore, the goal of an evaluation function is to avoid the homomorphic multiplication operations as much as possible. Additionally, since rotation is performed in terms of computationally-expensive operations, they should be avoided too. Also note that, only multiplications are considered when determining the level, whereas rotation and addition operations do not affect the level. This is the reason behind the "multiplicative depth" terminology, which implies that, any number of select, add, and rotate operations can be performed on ciphertexts without worrying about the BGV level, however, multiplications take away from the maximum-available-level which was determined at the very beginning of a sequence of homomorphic operations.

4.4.7 Performance Characteristics of BGV primitives

Leveled FHE implies an untraditional trade-off scheme: choosing L too high slows down the entire chain of homomorphic operations, while small L values prohibit evaluating complex functions [13]. Figure 4.4 shows impact of level L on ciphertext and public key sizes which grow substantially with the increased L, even to represent the same data. For example, one bit of plaintext might correspond to a 100KB of storage at $L=10$, but it might grow to 1MB when the level is $L=20$, requiring 10x more storage space to perform 20 cascaded homomorphic multiplications instead of 10. The increase in size of ciphertexts and public-keys are related to the methods: modulus-switching and relinearization, respectively. Recall that modulus-switching is performed after each multiplication by switching to a smaller modulus to reduce the noise of the ciphertext. For higher values of L, starting modulus of the ciphertext is multiplication of L prime numbers, thus requires more storage space. Similarly, after each multiplication a relinearization

operation is performed to reduce size of the ciphertext. This requires storing encryptions of the multiple keys, therefore, for higher values of L more keys are stored in public-key which increases its size.

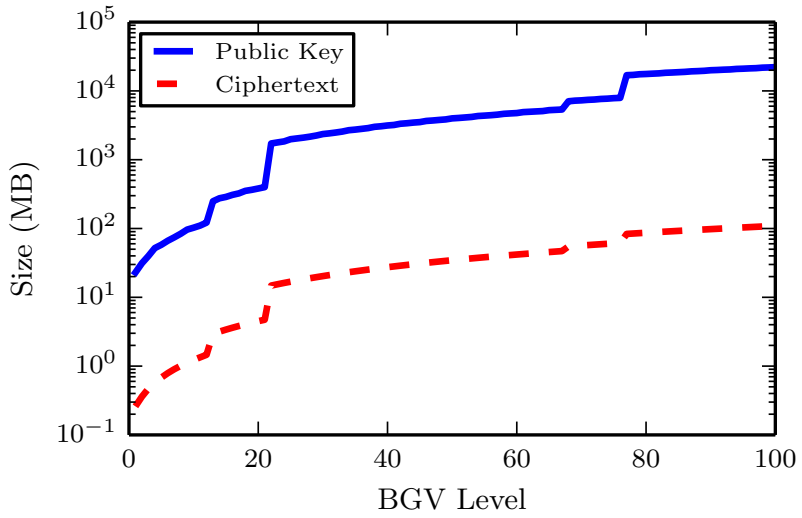


Figure 4.4: Public Key and Ciphertext Sizes for different BGV level.

Figure 4.5 shows a second disadvantage of increased L: Homomorphic operations execute slower with increased L. For example, while a $\times_h$ operation might take one second at L=20 , it takes 10 seconds at L=40. With higher L, homomorphic operations are applied to larger ciphertexts, which slows-down their execution time. Figure 4.5 also presents the performance of individual FHE operations. While addition operation is almost *free*, rotation and multiplication operations are expensive and will dominate the execution time. Therefore, the goal of an evaluation function is to avoid homomorphic multiplication and rotation operations as much as possible. Also note that, only multiplications are considered when determining the level, whereas the rest of the operations do not affect the level. This is the reason behind "multiplicative-depth" terminology, which implies that, any number of select, add, and rotate operations can be performed on ciphertexts without worrying about BGV level, however, multiplications take away from the maximum-available-level which was determined at the very beginning of evalua-

tion. This emphasizes the importance of L in formulating our application. When designing our medical application, optimizing the chain of computations to reduce L, multiplications, and rotations will be our priority.

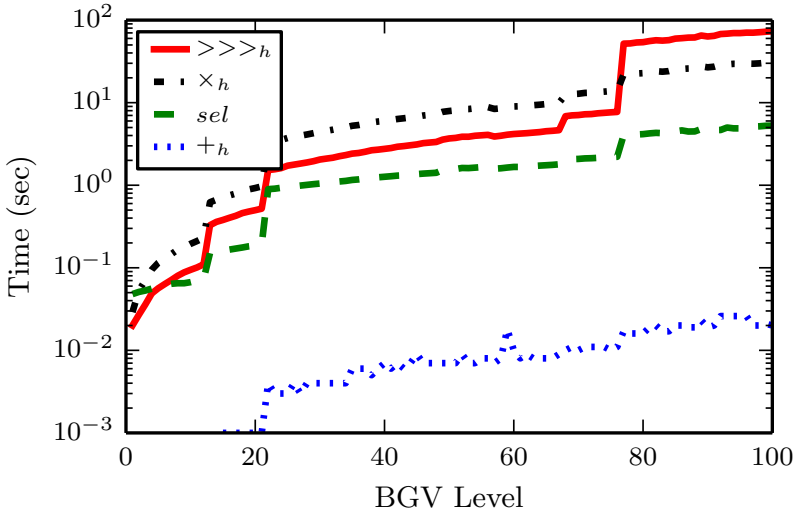


Figure 4.5: Level-dependent execution times of BGV primitives.

4.5 Computational Models for FHE Evaluation

To evaluate a function f with FHE, we need to represent f as an efficient computation. Oded Goldreich [139] defines computation as "a process that modifies an environment via repeated applications of a predetermined rule." In the context of computers, this refers to defining artificial rules in an artificial environment towards achieving a precise and specific side effect. In order to formally model computation, we need to mathematically model the environment and the "transition" rules. Such a model will additionally provide a platform to understand the intrinsic complexity of computing any task in that environment. Turing machine is the simplest and most powerful model of computation that allows us to study this complexity, as it can simulate most physical environments. While the goal of complexity theory is to understand the limits of computation, our focus

is to identify the best computational model that can represent function f as a computational task and the best environment to securely evaluate it.

Almost all of the FHE schemes proposed to date represent the function f as a circuit. In fact, we will use circuit model to implement our medical application. We will also employ branching programs as an alternative computation for LQTS detection. Towards this, we first discuss the computational models relevant to our medical application implementation.

4.5.1 Circuit Model

The circuit model is the most popular model of computation to represent processes in electronic circuits and is more commonly referred to as "digital logic". It is a generalization of Boolean formulas and can be defined via directed acyclic graphs where each vertex has the effect of applying a certain Boolean operator on the values from incoming vertices and delivering the result of the computation to an output vertex. Now, we turn our focus to providing a formal framework for our circuit model.

Definition 1. A circuit C is a 6-tuple $(n, G, S, \varphi, \theta, v^\star)$ where $n \in N$, $G = (V, E)$ is a directed acyclic graph where each vertex has in-degree zero or two, $S \subseteq V$ is the set of vertices with in degree zero, $\varphi : S \to \{i\}_{1 \leq i \leq n}$ labels vertices in S with an input bit, $\theta : V \setminus S \to \{\vee, \oplus\}$ labels the rest of the vertices with a Boolean operation, and $v^\star \in V$ is the output vertex. In the context of circuits, we use the terms "vertex" and "gate" interchangeably. The gates in S are input gates, and the rest of the gates are work gates. For any string $x \in \{0,1\}^n$, the output of any input gate $s \in S$ is equal to $x_{\theta_{(s)}}$, and the output of any work gate $v \in V \setminus S$ is the result of the boolean operation $\theta(v)$ when applied to the output of the two parent gates of v. The output of the circuit on input $x \in \{0,1\}^n$ is the output of $v^\star$ on input x. The input size of the circuit is n.

An important parameter of the circuit that will be of particular interest in this work will be the depth of the circuit. The depth of any input gate $s \in S$ is zero, and the depth of any work gate $v \in V \setminus S$ is one plus the larger of the depths of its two parent gates. The depth of a circuit is the depth of its output gate $v^\star$. We denote this number by $Depth(C)$. We define $Depth_\vee(C)$ similarly, except that $\oplus$ gates do not increase $\vee$-depth: the $\vee$-depth of a $\oplus$ work gate is defined as the maximum of the $\vee$-depth of its two parent gates.

Definition 2. A family of circuits $\{C\}_{i \in N}$ is a sequence of circuits such that for all $i \in N$, circuit C_i has input size i. The family is uniform if and only if there is a polynomial time Turing Machine such that for any $n \in N, M(1^n)$ outputs the description of C_i. The family is polynomial-sized if there exists a polynomial p such that for all $n \in N$, the number of gates in C_n is at most $p(n)$. We note that uniform families are always polynomial-sized because polynomial time Turing machines can only output polynomially many bits.

NC or Nick's Class is a family of circuits that are polynomial-size and have poly-logarithmic depth. The motivation of considering this particular subclass is that these circuits can be evaluated in poly-logarithmic time on parallel computer with polynomially many processors.

Definition 3. NC^i is the set of languages accepted by a uniform, polynomial-sized family of circuits $\{C_j\}_{j \in N}$ such that $Depth(C_n) \leq O(\log^i n)$. We call any such family an NC^i circuit family.

The class of NC^i circuits is already a rich class which can compute basic arithmetic operations such as addition, multiplication and division on n-bit integers. In particular, this model will allow computation of our medical applications as we will show later in Chapter 5.

4.5.2 Branching Programs Model

Another natural model of computation that can represent Boolean formulas are Boolean branching programs. These are represented via directed acyclic graphs but have a different mode of operation compared to circuits.

Definition 4. A branching program is a tuple (V, E, φ, s, t, n), where (V, E) is a directed acyclic graph, $\varphi : E \to \{T\} \cup \{x_i, \overline{x_i}\}_{i \in V}$ maps edges to labels, $s \in V$ is the start vertex, $t \in V$ is the end vertex, and n is the input length. The size of the program is the number of vertices in V. For a string $x \in \{0,1\}^n$, we define the graph G_x as (V, E_0) where $E_0 \subseteq E$ is the set of edges which are labeled with T, or x_i such that $x_i = 1$ or $\overline{x_i}$ such that $x_i = 0$. In other words, E_0 is the set of edges labeled as always present or are labeled with a corresponding input bit. The branching program accepts input x if and only if there is a path from s to t in G_x, and we say it outputs 1 if it accepts, and outputs 0 otherwise. A branching program has width $k \in N$ if and only if for all $i \in N$, the set of vertices reachable from s in G in exactly i steps has cardinality at most k. The labels of the edges in E do not affect the width: any outgoing edge of a vertex can be traversed, regardless of label.

A branching program can compute any function with one bit of output by completely branching on all n input bits in sequence so that each of the $2n$ inputs x results in a unique path from s in G_x. The paths associated with strings such that $f(x) = 1$ are attached to the terminal vertex t. This sort of method is impractical, since for large n it would require too much space to store the branching program itself. We are primarily interested in what branching programs with a small number of vertices can do. By Barrington's theorem, any circuit can be converted into an equivalent branching program.

Theorem 1 (Barrington's Theorem)**.** *For any circuit of depth d, there is an equivalent branching program with width 5 and at most 5×4^d vertices.*

This means that branching program families of polynomial size and constant width can compute anything that can be computed by circuit families of logarithmic depth. We state without proof that NC^1 circuits can simulate branching programs with constant width. However, with polynomial width, branching programs can also do any computation that a log-space Turing machine can do.

Definition 5. A logs-pace Turing machine is a deterministic Turing machine that has a read only input tape and a small work tape. For all $n \in N$, and for any string of length n, the machine must use at most $O(\log n)$ cells of its work tape.

Lemma 2. *Any log-space Turing machine can be uniformly converted into an equivalent branching program family of polynomial size.*

Proof. We present a polynomial time algorithm to produce the branching program with input size n. First, we form the graph of the branching program as rows of vertices representing possible configurations of the machine. There are polynomially-many configurations of the log-space machine, and it can only take a polynomial maximum number of steps. Except in the last row, we give each vertex the appropriately-labeled outgoing edge to two others in the next row, one for each possible value of the current cell of the simulated input tape. Vertices in the final row connect to a terminal vertex t if and only if the vertex represents an accepting configuration. This program can be outputted in polynomial time, it has polynomial size, and it is equivalent to the logspace machine on inputs of the desired length. □

5 Circuit Based FHE Implementation

In this chapter, we provide details on the circuit based FHE implementation of medical applications presented in Chapter 2.2. Efficient implementation of the medical applications require utilizing the properties of the BGV scheme, and the entire cloud-based medical application must be centered around the road-map shown in Figure 5.1. The top of this figure (i.e., the *data transformation path*) shows the transformations that the incoming data stream $d[i]$ must go through to produce a properly-formatted ciphertext. This top part is assumed to be performed during data acquisition by a computationally-capable device, such as the patient's smartphone or a cloudlet [56] as shown in Figure 2.1. The bottom of Figure 5.1 (i.e., the *functional transformation path*) will be the focus of this chapter, which details the steps that must be taken to implement medical applications with circuit based FHE.

The medical applications operate on integer data, which is a higher level compared to circuits that operate on individual bits. Therefore we need to convert the medical applications to be compatible with circuit based FHE implementation. We design a process for the conversion as follows: first we represent the medical applications in terms of functions. The medical applications will provide a summarized information to the doctor such as "What is the average heart rate

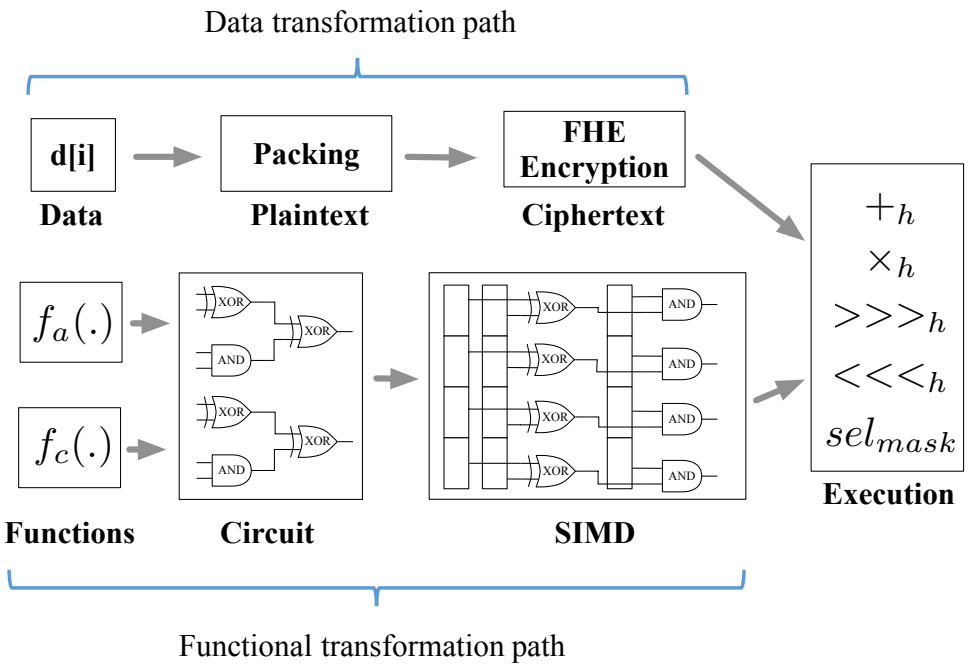


Figure 5.1: Road-map for secure cloud computing with FHE

between 10AM-1PM?", "Did LQTS occur during last 4 hours?". Therefore a medical application can be represented as a combination of two families of functions: computation and aggregation. While computation functions (f_c) perform desired operations on medical data, accumulation functions (f_a) generate a summary from the results produced by the f_c. Once we represent the medical applications in terms of f_c and f_a, we convert these functions into binary circuits. To utilize the parallelism of the BGV scheme, we arrange binary circuits into a SIMD format. Finally we replace a SIMD formatted circuit with the corresponding BGV primitives introduced in Section 4.4.5.

The rest of this chapter is organized as follows. In Section 5.1, details of the road-map for secure cloud computing are provided. FHE based building blocks are introduced in Section 5.2. Finally, Section 5.3 details the implementation of medical applications.

5.1 Function to FHE Conversion Process

In this section, we present the details of implementing any generic function f with the circuit based FHE. The circuit based FHE allows XOR or AND operations that forms a Turing-complete set of operations. With this set, any function can be represented as a combination of XOR and AND gates and can be evaluated as a binary circuit with FHE. Performance of the circuit based FHE implementation depends on two factors as shown in Section 4.4.7: 1) the level L of the FHE scheme (i.e., multiplication-depth of the application), and 2) the number of compute-intensive multiplication and rotation operations. Therefore, during conversion of a function to circuit based FHE, we will propose several optimizations to reduce both the level L and the number of expensive FHE operations.

5.1.1 Function to Circuit Conversion

The first step of function to FHE conversion requires converting function f to a binary circuit. Without loss of generality, we will demonstrate steps of conversion on comparison function ($X > Y$). The comparison function for two 4-bit integers X and Y can be expressed as a binary circuit as follows:

$$X{>}Y = (x_3\overline{y_3} \oplus x_2\overline{y_2}e_3 \oplus x_1\overline{y_1}e_3e_2 \oplus x_0\overline{y_0}e_3e_2e_1) \tag{5.1}$$

where x_i is the value of bit at index i of X, $\overline{y_i}$ is the inverse of bit at index i of Y and e_i is their bitwise equality ($x_i{==}y_i$).

Figure 5.2 presents the implementation of Equation 5.1 with only XOR and AND gates corresponding to homomorphic addition ($+_h$) and multiplication ($\times_h$) in $\mathbb{GF}(2)$, respectively. For clarity we depict inverters which can be implemented by XOR gates. The minimum multiplication depth for comparison function for 4-bit integers is three, which is also represented by different shades of gray in

Figure 5.2.

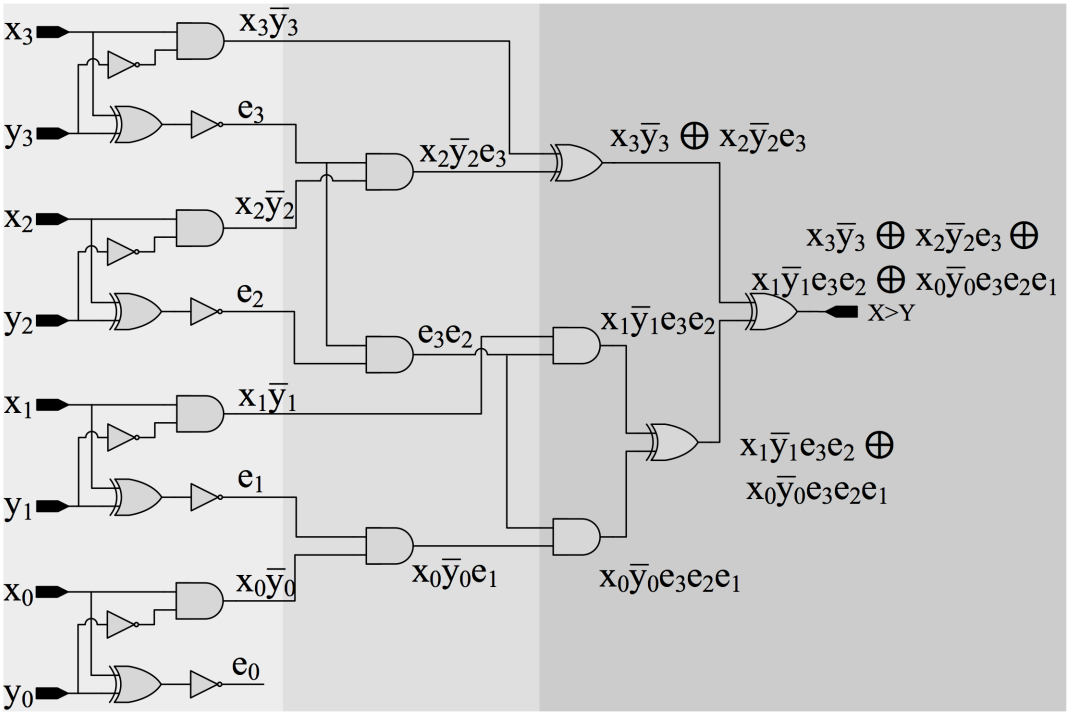


Figure 5.2: Depth-3 Binary Circuit for implementing
$X > Y = (x_3\overline{y_3} \oplus x_2\overline{y_2}e_3 \oplus x_1\overline{y_1}e_3e_2 \oplus x_0\overline{y_0}e_3e_2e_1)$

5.1.2 Circuit to SIMD Mapping

This step is necessary to execute homomorphic operations in a SIMD fashion. We recall from Section 4.4.4 that each ciphertext encrypts the bits of a message by mapping them to plaintext slots in $\mathbb{GF}(2)$. Let $\boldsymbol{X}$ and $\boldsymbol{Y}$ be such ciphertexts that encrypt plaintexts packing k-bit messages X and Y. Further, assume that bit i of message X is mapped to plaintext slot index i (e.g., x_0 is mapped to slot 0) and the plaintext is represented as $(x_{k-1}\ x_{k-2}\ \cdots\ x_1\ x_0)$. The homomorphic addition operation $\boldsymbol{X} +_h \boldsymbol{Y}$ adds (i.e., XOR's) each plaintext bit having the *same slot index*. To state alternatively, a single $+_h$ operation on the ciphertext performs bitwise-XOR operations on all plaintext slots in parallel. However,

packing bits of a message to plaintext slots has a drawback: No operation can be performed on messages with a *different slot index*, unless proper rotation and selection operations are performed, as detailed in Section 4.4.5.

To exemplify these trade-offs, let us focus on the slot index assignments of messages in Equation 5.1. Computing terms like $x_0\overline{y_0}$, $x_1\overline{y_1}$, $x_2\overline{y_2}$, $x_3\overline{y_3}$ is equal to performing a single $\times_h$ operation on ciphertexts as

$$\boldsymbol{X} \times_h \overline{\boldsymbol{Y}} \iff (x_3\ x_2\ x_1\ x_0) \wedge (\overline{y_3}\ \overline{y_2}\ \overline{y_1}\ \overline{y_0})$$

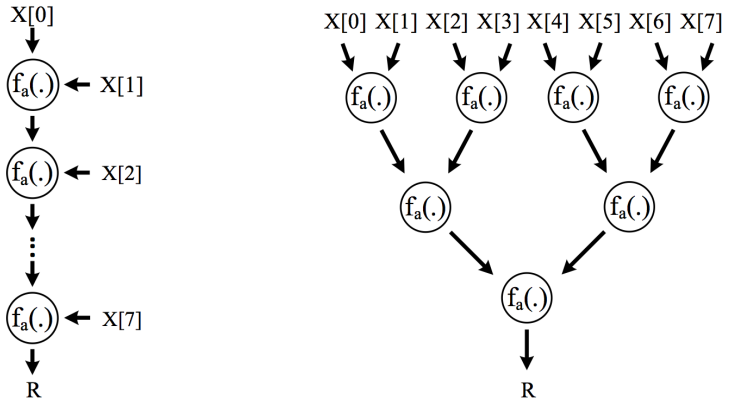
where $\iff$ denotes the relationship between the ciphertext and plaintext, and $\wedge$ is slotwise AND. Alternatively, computing terms like $x_2\overline{y_2}e_3$ poses a problem since e_3 is in a different slot index than x_2 and $\overline{y_2}$. This means that we need to rotate $\boldsymbol{E}$ right to align it with $\boldsymbol{X} \times_h \overline{\boldsymbol{Y}}$. After rotating $\boldsymbol{E}$, a selection operation is needed to mask out the bits diffusing from neighboring plaintext slots.

In general mapping a circuit to SIMD has following issues. First, not every circuit is parallel in bit-level and this will increase the number of SIMD operations to achieve the same result. Second, to fully utilize parallelism of SIMD, the bits must be aligned properly. This introduces using rotate and select operations to align bits, thus incurs additional computation time. We will further explain details of circuit to SIMD mapping in Section 5.2 and propose techniques to minimize the cost of circuit to SIMD mapping. Once the circuit is converted to its SIMD counterpart, conversion to BGV can be done by replacing SIMD circuit with the BGV primitives presented in Section 4.4.5.

5.1.3 Accumulation Functions

The proposed medical applications will provide summarized statistics/results by aggregating the results of computation over the medical data. The order of applying aggregation to results affects the depth of the circuit implementation. Since

our aggregations are associative, they can be performed in two ways [140]: sequential (Figure 5.3(a)) or as a binary tree (Figure 5.3(b)). While both methods require applying the same number of f_a's for aggregation, binary tree method results in a lower depth circuit. Specifically, if f_c has a multiplication depth of d, then aggregating N results requires $O(N \cdot d)$-depth using the sequential method, while $O(\lceil \log_2 N \rceil \cdot d)$-depth is sufficient for the binary tree method. Therefore, we will implement aggregation by applying f_a to the results generated by f_c in a binary tree fashion to achieve a low-depth circuit.



(a) Sequential (b) Binary Tree

Figure 5.3: An example of aggregating results by applying aggregation function f_a in sequential and binary tree fashion.

5.2 FHE Building Blocks

In this section, we will introduce the building blocks that will be used during the implementation of target medical applications. While these blocks will be sufficient for our target medical applications, they also provide a great coverage for implementing any signal processing algorithm.

5.2.1 Adder

Addition operation is central to the many general-purpose algorithms. Therefore, an efficient addition operation is crucial for the performance of the applications implemented with FHE. For an efficient implementation with BGV, we need to consider two important design aspects: parallelism and multiplication-depth. The classic adder implementation such as ripple-carry adder is not suitable for FHE implementation. The ripple-carry adder requires $O(k)$ multiplication-depth for adding two k-bit numbers and does not take advantage of parallelism. Towards an efficient adder implementation, we consider two adders: Kogge-Stone Adder and Carry Save Adder. We will show that these adders are highly parallelizable and have low multiplication-depth, therefore can be utilized for fast addition with FHE implementation.

5.2.1.1 Kogge-Stone Adder

Kogge-Stone adder [141] is a parallel-prefix adder, which has a logarithmic multiplication-depth and its implementation is amenable to SIMD as exemplified in in Figure 5.4.

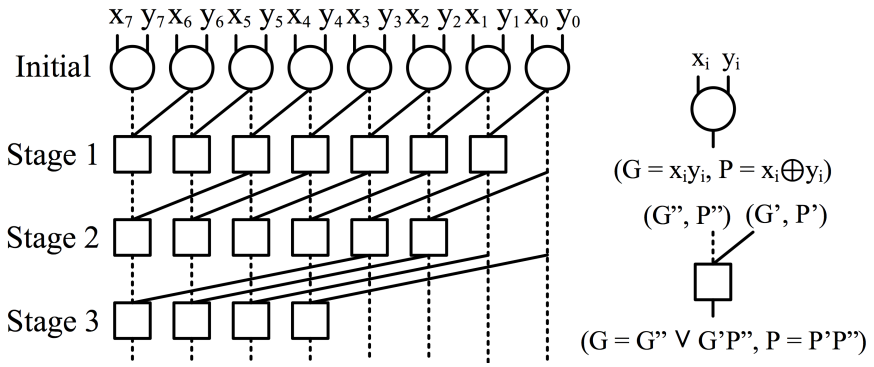


Figure 5.4: Kogge-Stone Parallel Prefix Adder

To add two k-bit numbers, Kogge-Stone adder first computes the initial Gen-

erate (G) and Propagate (P) as

$$G = X \wedge Y$$
$$P = X \oplus Y$$

, where $\wedge$ (AND) and $\oplus$ (XOR) operations are applied to the bits of X and Y in parallel. This step has a single multiplication-depth due to the computing $G = X \wedge Y$.

Then G and P are updated in $\log_2 k$ stages based on the generate and propagate values of the lower bits (i.e., G' and P') as

$$G = G'' \vee G'P'$$
$$P = P' \wedge P''$$

where P''_i and G''_i denote the propagate and generate values of the current bit i from the previous stage respectively. During the update, generate (G') and the propagate (P') values of the lower bits is also involved to achieve logarithmic-depth addition. In each stage different bit indices are used for G' and P' as shown in Figure 5.4. The final sum $S = X + Y$ is computed at the end as $S = P \oplus (G << 1)$.

Overall multiplication-depth of Kogge-Stone adder is the sum of the depth required for the initial generate and propagate computation and then updating them in $\log_2 k$ stages. The multiplication-depth of each stage depends on updating generate G as $G = G'' \vee G'P'$. This step has a multiplication-depth of two, single multiplication-depth from computing $G'P'$ and another for the $\vee$ (OR) operation, which can be expressed as $OR(a, b) = a \oplus b \oplus (a \wedge b)$ that requires single multiplication depth. Since we have $\log_2 k$ stages, multiplication-depth of all stages is $2 \log_2 k$. Therefore, overall multiplication-depth of Kogge-Stone Adder is $2 \log_2 k + 1$.

Algorithm 7: FHE Implementation of Kogge-Stone Adder

input : Ciphertexts $\boldsymbol{X}$ and $\boldsymbol{Y}$
output: Ciphertext $\boldsymbol{S}$
$\boldsymbol{G} = \boldsymbol{X} \times_h \boldsymbol{Y}$
$\boldsymbol{P} = \boldsymbol{X} +_h \boldsymbol{Y}$
for $i = 1$ **to** $num\_stages + 1$ **do**
  $\boldsymbol{G''} = \boldsymbol{G}$
  $\boldsymbol{P''} = \boldsymbol{P}$
  $\boldsymbol{G'} = (\boldsymbol{G} <<<_h i)\ sel_{mask}\ \boldsymbol{0}$
  $\boldsymbol{P'} = (\boldsymbol{P} <<<_h i)\ sel_{mask}\ \boldsymbol{1}$
  $\boldsymbol{P} = \boldsymbol{P'} \times_h \boldsymbol{P''}$
  $\boldsymbol{G'} = \boldsymbol{G'} \times_h \boldsymbol{P''}$
  $\boldsymbol{G} = \boldsymbol{G'} \vee_h \boldsymbol{G''}$
  $i = i \cdot 2$
$\boldsymbol{S} = \boldsymbol{P} +_h ((\boldsymbol{G} <<<_h 1)\ sel_{mask}\ \boldsymbol{0})$

We present the steps for FHE Implementation of Kogge-Stone Adder in Algorithm 7. Initial generate and propagate computation, which requires bit-wise $\wedge$ and $\oplus$, can be calculated with single homomorphic multiplication and addition respectively. For each stage, encryptions of previous generate and propagate values are saved into ciphertexts $\boldsymbol{G''}$ and $\boldsymbol{P''}$. The ciphertexts $\boldsymbol{G'}$ and $\boldsymbol{P'}$ are the encryptions of the generate and propagate values of the lower bits. We use rotation followed by selection to correctly align corresponding bits. Note that ciphertexts $\boldsymbol{0}$ and $\boldsymbol{1}$ are encrypting 0's and 1's in each plaintext slot and they are used during selection with proper selection mask to replace diffusing bits at the of rotation.

5.2.1.2 Carry Save Adder (CSA)

Carry Save Adder is different than regular adders such that it outputs two values instead of one. Specifically, Carry Save Adder compresses 3 k-bit inputs (X, Y,

Z) to 2 outputs, k-bit Sum S and $k+1$-bit Carry, as follows:

$$S = X \oplus Y \oplus Z$$
$$C = \Big((X \wedge Y) \vee (X \wedge Z) \vee (Y \wedge Z)\Big) << 1$$

where $\oplus$, $\vee$ and $\wedge$ are SIMD operations performed on all k bits of the input in parallel. Actual sum is then computed by adding S and C with a regular adder such as Kogge-Stone Adder.

Multiplication depth of the CSA adder is determined by the computation of C which requires a depth-3 multiplication circuit. Single multiplication depth is required for computing each of $X \wedge Y$, $X \wedge Z$ and $Y \wedge Z$, and then combining them with OR ($\vee$) further requires depth-2 multiplication.

Table 5.1: Replacing OR with XOR in CSA.

X	Y	Z	XY	XZ	YZ	XY$\perp$XZ$\perp$YZ	
						$\perp = \vee$	$\perp = \oplus$
0	0	0	0	0	0	0	0
0	0	1	0	0	0	0	0
0	1	0	0	0	0	0	0
0	1	1	0	0	1	1	1
1	0	0	0	0	0	0	0
1	0	1	0	1	0	1	1
1	1	0	1	0	0	1	1
1	1	1	1	1	1	1	1

Optimizing depth of CSA: The depth of CSA depends on computing C which requires OR operations over products of inputs. We show in Table 5.1 that OR operations can be replaced with XORs for computing C, yielding an equivalent result. With this optimization, computing C is now expressed as

$$C = \Big((X \wedge Y) \oplus (X \wedge Z) \oplus (Y \wedge Z)\Big) << 1$$

This would not be a meaningful substitution in a VLSI implementation, since XOR

gates typically have a higher delay than OR gates [142]. However, it reduces the multiplication-depth of CSA from 3 to 1, which results in a $\approx 3\times$ performance improvement in an FHE implementation.

5.2.2 Accumulator

While Kogge-Stone Adder provides a low-depth circuit for adding two messages, adding more than two messages will significantly increase the required depth. For example, adding N k-bit numbers with Kogge-Stone Adder requires $\log_2 N \times (2\log_2 k + 1)$ multiplication-depth by arranging adders in a binary tree fashion. For an efficient implementation of FHE-based accumulation, we use conventional VLSI design techniques similar to Wallace [143] and Dadda [144] multipliers that perform high-speed multi-operand additions by reducing both the depth and the number of carry operations. This design approach benefits our FHE-based accumulation in two ways: 1) reducing the number of carry operations avoids compute-intensive $\times_h$ operations, and 2) reducing the depth of computations translates to a reduced L in FHE.

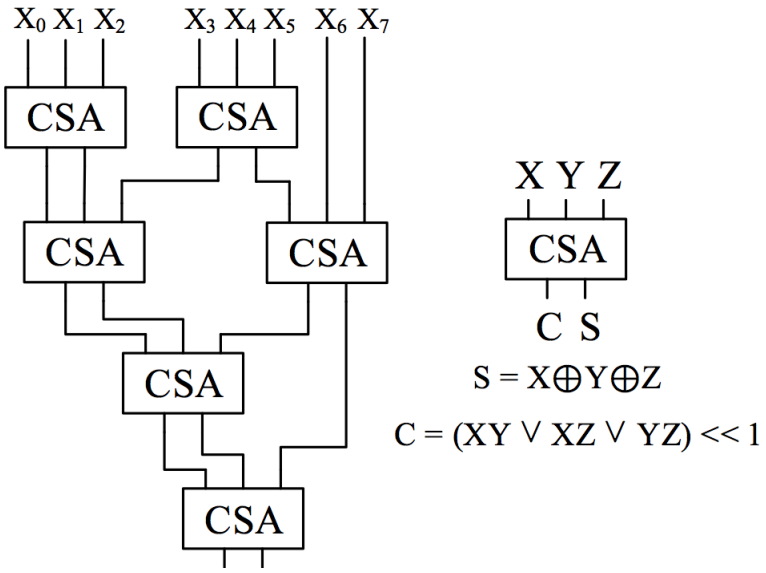


Figure 5.5: 8:2 compression of the operands using tree of CSAs.

To lower the depth of accumulating N k-bit numbers, we will use combination of Kogge-Stone Adder and Carry Save Adder (CSA). We use CSA to compress N operands to two numbers, and then add the remaining two numbers with Kogge-Stone Adder. To compress N k-bit numbers to two, we will use Carry Save Adders by arranging them in multiple stages similar to a tree and connect S and C as inputs to other CSAs. Figure 5.5 exemplifies the reduction of 8 operands down to 2 by using four levels of CSAs. The number of CSA stages (`nCSAStages`) for reducing N operands is lower-bounded by Equation 5.2 [140]. At the end of compressing N k-bit numbers, CSA tree will output two variables Sum S and Carry C, which are now (k + `nCSAStages`)-bit numbers. We use Kogge-Stone Adder for adding S and C to generate final result.

$$\left\lceil \frac{\log_2(N/2)}{\log_2(3/2)} \right\rceil + 1 \leq \texttt{nCSAStages} \tag{5.2}$$

Multiplication-depth of the accumulator is equal to sum of multiplication-depth of CSA tree and Kogge-Stone Adder. The depth of CSA tree is equal to number of CSA stages (`nCSAStages`), since each CSA adder requires depth-1 multiplication as shown in Section 5.2.1.2. The depth of Kogge-Stone adder depends on bit-length of the inputs (i.e., S and C) and is equal to $2\log_2(k +$ `nCSAStages`$) + 1$. Therefore the overall multiplication depth d required by the accumulator is lower-bounded as

$$d \geq \left(\left\lceil \frac{\log_2(N/2)}{\log_2(3/2)} \right\rceil + 1 \right) + \left(2\log_2 \left(k + \left(\left\lceil \frac{\log_2(N/2)}{\log_2(3/2)} \right\rceil + 1 \right) \right) + 1 \right) \tag{5.3}$$

5.2.3 Multiplier

Another central block for implementing general-purpose algorithms is integer multiplication. The standard integer multiplication of two 4-bit numbers A and B is presented in Table 5.2. Multiplication is carried out by generating partial products

and then adding them to generate the result $C = A \times B$. While steps involved in standard multiplication is trivial for regular multiplication, FHE implementation requires special care.

Table 5.2: Multiplication of two 4-bit numbers

	A			a_3	a_2	a_1	a_0
$\times$	B			b_3	b_2	b_1	b_0
				$a_3.b_0$	$a_2.b_0$	$a_1.b_0$	$a_0.b_0$
			$a_3.b_1$	$a_2.b_1$	$a_1.b_1$	$a_0.b_1$	
		$a_3.b_2$	$a_2.b_2$	$a_1.b_2$	$a_0.b_2$		
$+$	$a_3.b_3$	$a_2.b_3$	$a_1.b_3$	$a_0.b_3$			

An observation of partial products reveal that each partial product can be computed in parallel by multiplying A with replicated bits of B. For example, first partial product from Table 5.2, $[a_3.b_0\ a_2.b_0\ a_1.b_0\ a_0.b_0]$ can be computed as $\boldsymbol{A} \times_h \boldsymbol{B}[0]$ where $\boldsymbol{A}$ is FHE encrypted version of number A and $\boldsymbol{B}[0]$ denotes encryption of bit 0 of number B (i.e., $\boldsymbol{B}[0] = [b_0\ b_0\ b_0\ b_0]$). In general, encryption of partial product i can be computed by multiplying $\boldsymbol{A}$ with $\boldsymbol{B}[i]$ and then rotating the result to right by i followed by selection.

To implement integer multiplication with FHE, we break-down multiplication into 3 steps: bit replication, partial product computation and accumulation of partial products. We discuss each steps in the following sections.

5.2.3.1 Bit Replication

Bit replication is required for partial product computation step to implement integer multiplication in SIMD fashion. This step inputs ciphertext $\boldsymbol{B}$ that encrypts k-bit integers and outputs a vector of k ciphertexts (i.e, $\boldsymbol{R}[0:k]$). Ciphertext i of the vector contains encryption of bit i replicated into k plaintext slots. For example, ciphertext $\boldsymbol{R}[1]$ is encryption of the plaintext $[b_1\ b_1\ b_1\ b_1]$. Algorithm 8

presents steps involved in bit replication procedure. We start by shifting bit i to least significant bit position and replacing other bits with 0. Then we copy bit i to all plaintext slots by using Total Sums method. The bit replication can be computed without any multiplication, therefore multiplication-depth of this step is 0.

Algorithm 8: FHE based Bit Replication

input : Ciphertext $\boldsymbol{B}$
output: Ciphertext vector $\boldsymbol{R}[0:k]$
for $i = 0$ **to** k **do**
  $\boldsymbol{M} \leftarrow \boldsymbol{B}$
  $\boldsymbol{M} \leftarrow (\boldsymbol{M} >>>_h i)\ sel_{mask}\ \boldsymbol{0}$
  $e \leftarrow 1$
  `// Total sum method`
  for $j = k - 2$ **to** 0 **do**
    $\boldsymbol{T} \leftarrow \boldsymbol{M}$
    $\boldsymbol{T} \leftarrow (\boldsymbol{T} <<<_h e)\ sel_{mask}\ \boldsymbol{0}$
    $\boldsymbol{M} \leftarrow \boldsymbol{M} +_h \boldsymbol{T}$
    $e \leftarrow 2.e$
    if $k[j] == 1$ **then**
      $\boldsymbol{M} \leftarrow (\boldsymbol{M} <<<_h 1)\ sel_{mask}\ \boldsymbol{0}$
      $\boldsymbol{M} \leftarrow \boldsymbol{M} +_h \boldsymbol{B}$
      $e \leftarrow e + 1$
  $\boldsymbol{R}[i] \leftarrow \boldsymbol{M}$

5.2.3.2 Partial Product Computation

Once all k-bits of the ciphertext $\boldsymbol{B}$ is replicated into a vector of k ciphertexts $\boldsymbol{R}[k]$, we proceed with partial product computation step. In the partial product computation, we perform $\times_h$ operation of first operand $\boldsymbol{A}$ with each ciphertext in $\boldsymbol{R}[0:k]$ to generate partial products, which is a vector of encrypted ciphertexts $\boldsymbol{P}[0:k]$. After each multiplication, a rotation and select with $\boldsymbol{0}$ is required to transform encrypted bits into the format presented in Table 5.2. Algorithm 9 presents steps involved in Partial Product Computation. The partial product

computation has a multiplication depth of one, since a single $\times_h$ is performed to generate each ciphertext in $\boldsymbol{P}[0:k]$.

Algorithm 9: FHE Partial Product Computation

input : Ciphertext $\boldsymbol{A}$ and Ciphertext $\boldsymbol{B}$
output: Ciphertext vector $\boldsymbol{P}[0:k]$
$\boldsymbol{R}[0:k] \leftarrow$ `FHEBitReplicate`$(\boldsymbol{B})$
for $i = 0$ **to** k **do**
  $\boldsymbol{T} \leftarrow \boldsymbol{A} \times_h \boldsymbol{R}[i]$
  $\boldsymbol{T} \leftarrow (\boldsymbol{T} <<<_h i)\ sel_{mask}\ \boldsymbol{0}$
  $\boldsymbol{P}[i] \leftarrow \boldsymbol{T}$

5.2.3.3 Aggregating Partial Products

Once partial products are generated, we aggregate the ciphertexts in vector $\boldsymbol{P}[0:k]$ to generate final result $\boldsymbol{C}$. For aggregation, we use the accumulator presented in Section 5.2.2. Multiplying two k-bit integers will generate k partial products and based on the Equation 5.3 the multiplication depth for accumulating them is equal to

$$d \geq \left(\left\lceil \frac{\log_2(k/2)}{\log_2(3/2)} \right\rceil + 1 \right) + \left(2\log_2 \left(k + \left(\left\lceil \frac{\log_2(k/2)}{\log_2(3/2)} \right\rceil + 1 \right) \right) + 1 \right)$$

The overall multiplication-depth is equal to sum of multiplication-depth of partial product computation and aggregating the partial products. Equation 5.4 presents the minimum multiplication-depth required to implement integer multiplication.

$$d \geq 1 + \left(\left\lceil \frac{\log_2(k/2)}{\log_2(3/2)} \right\rceil + 1 \right) + \left(2\log_2 \left(k + \left(\left\lceil \frac{\log_2(k/2)}{\log_2(3/2)} \right\rceil + 1 \right) \right) + 1 \right) \quad (5.4)$$

5.2.4 Constant Integer Multiplier

While the FHE multiplier presented in the previous section is capable of computing multiplication of two ciphertexts, there are cases where only a multiplication with a constant integer is needed. One option is to encrypt the constant with FHE and then use the multiplier from the previous section. However, multiplication is already very expensive operation and multiplication with a constant integer can be implemented in a more efficient way that requires fewer number of operations and depth. With constant multiplication only addition of shifted ciphertexts is required. Number of shifted ciphertexts to be added depends on the constant integer. If the bit i of constant integer is one, then ciphertext is rotated by i and proper selection operation is performed. Then all the shifted versions of the ciphertexts are added with the accumulator. Algorithm 10 presents the steps for the constant multiplier.

Algorithm 10: FHE Multiplication with a Constant Integer

input : Ciphertext $\boldsymbol{A}$ and k-bit constant integer C
output: Ciphertext $\boldsymbol{R}$
$j \leftarrow 0$
for $i = 0$ **to** k **do**
  if $C[i] == 1$ **then**
    $\boldsymbol{T} \leftarrow (\boldsymbol{A} <<<_h i)\ sel_{mask}\ \boldsymbol{0}$
    $\boldsymbol{P}[j] \leftarrow \boldsymbol{T}$
    $j \leftarrow j + 1$
$\boldsymbol{R} \leftarrow$ `FHEAccumulate`$(\boldsymbol{P}[0 : j])$

Constant multiplier avoids costly bit replication and partial production operations of the multiplier presented in previous section. The multiplication-depth of the constant multiplier only depends on the depth of the accumulator. In the worst case scenario (i.e., all the bits of the constant is one), k shifted versions of the ciphertexts needs to be added. Therefore the maximum depth of the constant

integer multiplier is equal to

$$d \geq \left(\left\lceil \frac{\log_2(k/2)}{\log_2(3/2)} \right\rceil + 1 \right) + \left(2\log_2 \left(k + \left(\left\lceil \frac{\log_2(k/2)}{\log_2(3/2)} \right\rceil + 1 \right) \right) + 1 \right) \tag{5.5}$$

5.2.5 Comparator

We propose two implementation of comparator to use with different settings. Our first comparator is based on digital comparator logic with shallow depth. The second approach is based on addition and checking the sign bit of the results.

5.2.5.1 Digital Comparator

The digital comparator compares two numbers based on their bit-values as presented in Equation 5.1. We will use the same example of comparing two 4-bit numbers X and Y presented in Section 5.1.1 and provide the details of implementing digital comparator circuit with BGV.

To evaluate the circuit in Equation 5.1 using BGV primitives, we decouple the computation into two separate homomorphic multiplications ($\times_h$) as follows:

$$\begin{aligned} (\boldsymbol{X} >_h \boldsymbol{Y}) \;\; &= \;\; (\boldsymbol{X} \times_h \overline{\boldsymbol{Y}}) \times_h \boldsymbol{M} \;\; \Longleftrightarrow \\ &\Big((x_3\ x_2\ x_1\ x_0) \;\wedge\; (\overline{y_3}\ \overline{y_2}\ \overline{y_1}\ \overline{y_0})\Big) \;\wedge\; (1\ e_3\ e_3e_2\ e_3e_2e_1) \end{aligned} \tag{5.6}$$

$\boldsymbol{M}$ and $\boldsymbol{E}$ ciphertexts are the encrypted versions of $(1\ e_3\ e_3e_2\ e_3e_2e_1)$ and $(e_3\ e_2\ e_1\ e_0)$. Table 5.3 lists the steps for evaluating Equation 5.6 using BGV primitives, by detailing the intermediate ciphertext levels L and the status of the encrypted plaintext slots. First, we compute $\boldsymbol{E}$ in Steps 1-3 which requires an XNOR operation to check if the bits of X and Y are equal as follows:

$$e_i \;=\; XNOR(x_i, y_i) = \overline{x_i \oplus y_i} = x_i \oplus y_i \oplus 1 \tag{5.7}$$

Table 5.3: Sequence of FHE operations for 4-bit comparison. X, Y are 4-bit messages, x_i, y_i's are the bits of the messages at index i.

Step	BGV Operation on ciphertext	Plaintext Slots				Level of Ctxt
		slot 3	slot 2	slot 1	slot 0	
1	$\boldsymbol{E} = \boldsymbol{X}$	x_3	x_2	x_1	x_0	L
2	$\boldsymbol{E} = \boldsymbol{E} +_h \boldsymbol{Y}$	$x_3 \oplus y_3$	$x_2 \oplus y_2$	$x_1 \oplus y_1$	$x_0 \oplus y_0$	L
3	$\boldsymbol{E} = \boldsymbol{E} +_h \boldsymbol{1}$	$e_3 \Leftarrow \overline{x_3 \oplus y_3}$	$e_2 \Leftarrow \overline{x_2 \oplus y_2}$	$e_1 \Leftarrow \overline{x_1 \oplus y_1}$	$e_0 \Leftarrow \overline{x_0 \oplus y_0}$	L
4	$\boldsymbol{A} = \boldsymbol{E} >>>_h 1$	?	e_3	e_2	e_1	L
5	$\boldsymbol{A} = \boldsymbol{A}\ sel_{0111}\ \boldsymbol{1}$	1	e_3	e_2	e_1	L
6	$\boldsymbol{B} = \boldsymbol{E} >>>_h 2$	?	?	e_3	e_2	L
7	$\boldsymbol{B} = \boldsymbol{B}\ sel_{0011}\ \boldsymbol{1}$	1	1	e_3	e_2	L
8	$\boldsymbol{C} = \boldsymbol{E} >>>_h 3$	?	?	?	e_3	L
9	$\boldsymbol{C} = \boldsymbol{C}\ sel_{0001}\ \boldsymbol{1}$	1	1	1	e_3	L
10	$\boldsymbol{A} = \boldsymbol{A} \times_h \boldsymbol{B}$	1	e_3	e_3e_2	e_2e_1	$L-1$
11	$\boldsymbol{M} = \boldsymbol{A} \times_h \boldsymbol{C}$	1	e_3	e_3e_2	$e_3e_2e_1$	$L-2$
12	$\boldsymbol{Q} = \boldsymbol{Y}$	y_3	y_2	y_1	y_0	L
13	$\boldsymbol{Q} = \boldsymbol{Q} +_h \boldsymbol{1}$	$\overline{y_3}$	$\overline{y_2}$	$\overline{y_1}$	$\overline{y_0}$	L
14	$\boldsymbol{Q} = \boldsymbol{Q} \times_h \boldsymbol{X}$	$x_3\overline{y_3}$	$x_2\overline{y_2}$	$x_1\overline{y_1}$	$x_0\overline{y_0}$	$L-1$
15	$\mathbf{M} = \mathbf{M} \times_h \mathbf{Q}$	$x_3\overline{y_3}$	$x_2\overline{y_2}e_3$	$x_1\overline{y_1}e_3e_2$	$x_0\overline{y_0}e_3e_2e_1$	$L-3$

Naïve Computation of $\boldsymbol{M}$: Calculating $\boldsymbol{M}$ from $\boldsymbol{E}$ requires storing rotated versions of $\boldsymbol{E}$ within temporary ciphertexts $\boldsymbol{A}$, $\boldsymbol{B}$, $\boldsymbol{C}$ which store encrypted values of (1 e_3 e_2 e_1), (1 1 e_3 e_2) and (1 1 1 e_3) in Steps 4-9. Rotation diffuses unwanted bits into $\boldsymbol{E}$ (represented as "?"), which must be replaced with "1"s via a proper selection mask. $\boldsymbol{M}$ (encrypted (1 e_3 e_3e_2 $e_3e_2e_1$)) is computed by multiplying these temporary ciphertexts as $\boldsymbol{M}$=$\boldsymbol{A} \times_h \boldsymbol{B} \times_h \boldsymbol{C}$ in Steps 10-11. Note that level L of the ciphertexts is reduced by one after each $\times_h$ (as described in Section 4.4.2). In general, computing $\boldsymbol{M}$ for k-bit messages requires first generating $k-1$ rotated versions of $\boldsymbol{E}$ and then multiplying them by a $\log_2 k$ depth binary-tree circuit.

Running Products Method: The naïve method for computing $\boldsymbol{M}$ requires $O(k)$ expensive $\times_h$ and $>>>_h$ operations which dominate the run-time of $>_h$. A close observation of $\boldsymbol{M}$ reveals that plaintext slots store running products of the e_i bits. Therefore, calculation of $\boldsymbol{M}$ can be optimized by computing the running

Algorithm 11: Optimized FHE Implementation of Comparator

input : Ciphertexts $\boldsymbol{X}$ and $\boldsymbol{Y}$
output: Ciphertext $\boldsymbol{R} = \boldsymbol{X} >_h \boldsymbol{Y}$
$\boldsymbol{E} \leftarrow \boldsymbol{X} +_h \boldsymbol{Y} +_h \boldsymbol{1}$
$\boldsymbol{M} \leftarrow \boldsymbol{E}$
for $i = 1$ **to** k **do**
  $\boldsymbol{T} \leftarrow (\boldsymbol{E} >>>_h i)\ sel_{mask}\ \boldsymbol{1}$
  $\boldsymbol{M} \leftarrow \boldsymbol{M} \times_h \boldsymbol{T}$
  $i \leftarrow i \cdot 2$
$\boldsymbol{Q} = (\boldsymbol{Y} +_h \boldsymbol{1}) \times_h \boldsymbol{X}$
$\boldsymbol{R} = \boldsymbol{M} \times_h \boldsymbol{Q}$

products, as presented in Algorithm 11.

With the running product optimization, the multiplication-depth stays same as $\log_2 k$, however the number of $\times_h$ and $>>>_h$ operations are reduced to $O(\log_2 k)$. This results in a speedup of $\approx \frac{O(k)}{O(\log_2 k)}$ compared to the naïve method. Figure 5.6 presents the speedups achieved using the optimized method with different BGV levels for comparing 16-bit numbers.

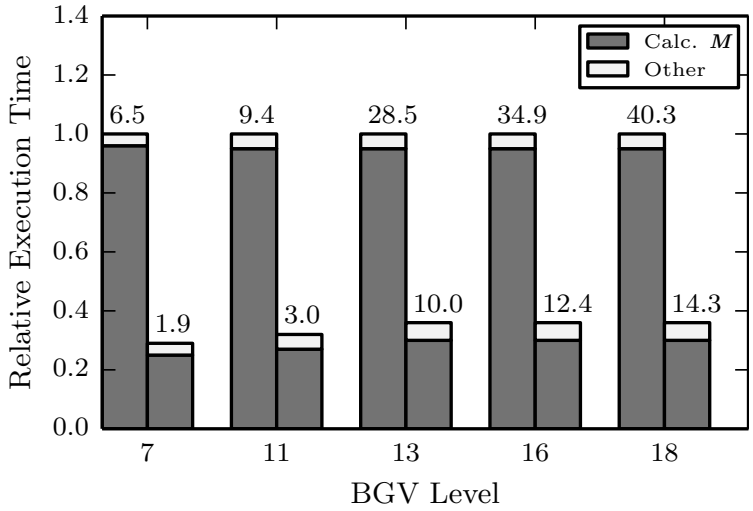


Figure 5.6: Normalized run-times (sec) for $>_h$ using naïve(left bars) and running-products (right bars) methods for k=16 (16-bit messages).

Once $\boldsymbol{M}$ is computed, the final result of $>_h$ is determined by first computing $\overline{\boldsymbol{Y}}$ in Steps 12-13 and then calculating $(\boldsymbol{X} \times_h \overline{\boldsymbol{Y}} \times_h \boldsymbol{M})$ in Steps 14-15. Note

that the resulting ciphertext is at level $L-3$ indicating the cost of $>_h$ as 3 levels, which is same as the multiplicative depth of the comparison circuit in Figure 5.2. In general, comparison of k-bit messages requires $\log_2 k+1$ levels ($\log_2 k$ levels for computing $\boldsymbol{M}$ and 1 level for $\times_h$ at the end).

Figure 5.7 shows $>_h$ applied to BGV ciphertexts $\boldsymbol{X}$ and $\boldsymbol{Y}$ encrypting two 4-bit messages. Since $X[0] < Y[0]$, digital comparator generates 4-bit message with all bits are 0. Conversely, $X[1] > Y[1]$ and digital comparator will output 4-bit message where *only* one the bits is 1. Therefore, to decode the result of $\boldsymbol{X} >_h \boldsymbol{Y}$, we first decrypt the resulting ciphertext and then for each message packed in the plaintext we check the bits of the message. If all the bits of message i is 0, then $X[i] > Y[i]$ is FALSE (i.e., $X[i] < Y[i]$), otherwise (i.e., if there is a bit 1) $X[i] > Y[i] =$ is TRUE.

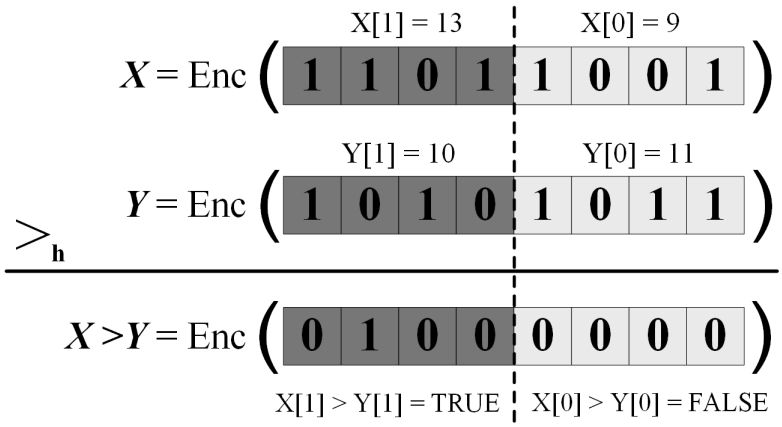


Figure 5.7: The result of the comparison in Equation 5.1 is a k-bit integer denoting ZERO (FALSE) or non-zero (TRUE).

5.2.5.2 Comparison based on Subtraction

Another method for computing comparison is to use subtraction with signed arithmetic. With this method, $X > Y$ operation can be performed by $X-Y$ and then checking the result is positive or negative. This is equivalent to performing sub-

traction and then checking whether sign bit of the result (i.e., most significant bit). The sign bit can be used to determine the $X > Y$ as follows

$$\text{sign bit} \quad = \quad \begin{cases} 0 & \text{if } X > Y \\ 1 & \text{if } Y > X \end{cases} \tag{5.8}$$

Subtraction operation can be performed with the adders introduced in Section 5.2.1. First we need to compute $-Y$, which is the 2's complement of Y. 2's complement of Y can be calculated by first inverting the bits of Y (i.e., 1's complement) by XORing them with 1's and then adding integer one as shown below

$$X - Y \Longrightarrow X + \overline{Y} + 1$$

Additions of k-bit $X, \overline{Y}$ and integer 1 can be computed by first using CSA to reduce them into two numbers and then adding final two numbers with Kogge-Stone Adder. The overall multiplication-depth required for the subtraction based comparison is therefore equal to $1 + (2\log_2(k+1) + 1)$ (depth-1 for CSA adder and $2\log_2(k+1) + 1$ depth for Kogge-Stone Adder).

Although the subtraction based comparison has a higher depth than binary comparator (i.e., $2\log_2(k+1)+2$ vs. $\log_2 k+1$), it has an advantage for the cases that involves multiplications. For example, assume we want to compute $X^2 > Y$. With the binary comparator, first multiplication of X^2 is computed and then the result is compared with Y. Overall depth required is equal to sum of the depth of X^2 computation and the digital comparator depth. On the other hand, with the subtraction based comparison, we can add $\overline{Y} + 1$ together with the aggregation of partial products (Section 5.2.3.3) of the X^2 computation. Aggregation of partial products already performs compression of k partial products and adding two more operands ($\overline{Y}$ and 1) will incur one additional depth in the worst case scenario. Therefore subtraction based comparison will require depth of computing X^2 plus

one, which is less than the depth required with digital comparator. We will provide more details for the subtraction based comparison in Section 5.3.2 during the implementation of the LQTS detection.

5.3 FHE Implementation of Medical Applications

In this section, we will detail the implementation of our medical applications by using the FHE building blocks introduced in the previous section. Specifically, we will implement Average Heart Rate, Minimum/Maximum Heart Rate and LQTS detection with circuit based FHE. We assume the applications operates over N ciphertexts encrypting multiple k-bit medical data. Finally, we will calculate the minimum required BGV level L for each application.

5.3.1 Average Heart Rate

Finding the Average Heart Rate (HR) requires accumulating encrypted values in N ciphertexts and then dividing the result by N. To avoid costly division operation with FHE, we will only compute the accumulation part. We note that, division by N can computed by rotating ciphertexts to right if N is a power of two. If N is not power of two, then division by N can be performed with the back-end device after decryption of the resulting ciphertext.

To accumulate N ciphertexts, we use the accumulator introduced in Section 5.2.2. We recall that the accumulator first reduces N ciphertexts down to two ciphertexts with tree of Carry Save Adders, and then adds the remaining ciphertexts with Kogge-Stone Adder. The multiplication-depth of the accumulator is stated in Equation 5.3. Therefore, minimum required BGV level L for calculating

Average HR is equal to

$$L > \left(\left\lceil \frac{\log_2(N/2)}{\log_2(3/2)} \right\rceil + 1 \right) + \left(2\log_2 \left(k + \left(\left\lceil \frac{\log_2(N/2)}{\log_2(3/2)} \right\rceil + 1 \right) \right) + 1 \right) \quad (5.9)$$

5.3.2 LQTS Detection

LQTS detection requires evaluating Fredericia's formula [51] homomorphically as described in Section 2.2.1. For our LQTS Detection application, we propose two different implementation method. The first method evaluates the Fridericia's Formula and the second method uses pre-computed values to speed up the calculations.

5.3.2.1 LQTS Detection with Fridericia's Formula

We reformulate Fridericia's Formula to detect LQTS by using subtraction based comparator introduced in Section 5.2.5.2 as follows

$$\begin{aligned} \frac{QT}{\sqrt[3]{RR}} > t \quad &\Longrightarrow \quad QT^3 > RR \cdot t^3 \\ &\Longrightarrow \quad QT^3 - RR \cdot t^3 > 0 \end{aligned} \quad (5.10)$$

As we showed in Section 5.2.1.1, Kogge-Stone Adder requires $2\log_2 k + 1$ multiplication-depth for adding two k-bit numbers. We need Kogge-Stone adder *only* during the generation of the final result, and we avoid using Kogge-Stone Adder during the accumulations until the very last step. Algorithm 12 presents the steps involved in detecting LQTS with Fridericia's formula. First, we replicate the bits of the QT into the ciphertext vector $\boldsymbol{QTBR}$, which will have k-ciphertexts. Second, we calculate k partial products for QT^2, and store them in vector $\boldsymbol{PP}$. Then, we use CSA compression to reduce the partial products to two ciphertexts: $\boldsymbol{QTS}[0]$ and $\boldsymbol{QTS}[1]$. Third, we again perform partial product

computation to the output of the CSA compression. This will generate total of $2k$ partial products for aggregation to compute QT^3. Fourth, we start computing $RR \cdot t^3$ with the constant integer multiplier introduced in Section 5.2.4. Again, we only aggregate partial products for constant multiplication with CSA compression, returning two ciphertexts $\boldsymbol{RRK}[0]$ and $\boldsymbol{RRK}[1]$. Finally, we accumulate all the ciphertexts produced during the previous steps. The number of ciphertexts involve in the final accumulation is $2k + 4$: $2k$ ciphertexts from the partial products QT^3 computations and four ciphertexts are for computing $-RR \cdot t^3$.

Algorithm 12: FHE-based LQTS Detection with Fridericia's Formula

input : Ciphertexts $\boldsymbol{QT}$ and $\boldsymbol{RR}$, k-bit threshold t
output: Ciphertext $\boldsymbol{R}$

$\boldsymbol{QTBR}[0:k] \leftarrow$ `FHEBitReplicate`$(\boldsymbol{QT})$
$\boldsymbol{PP}[0:k] \leftarrow$ `FHECalcPartialProducts`$(\boldsymbol{QT}, \boldsymbol{QTBR}[0:k])$
$\boldsymbol{QTS}[0:2] \leftarrow$ `FHECSATree`$(\boldsymbol{PP}[0:k])$
$\boldsymbol{QTC}[0:k] \leftarrow$ `FHECalcPartialProducts`$(\boldsymbol{QTS}[0], \boldsymbol{QTBR}[0:k])$
$\boldsymbol{QTC}[k+1:2k] \leftarrow$ `FHECalcPartialProducts`$(\boldsymbol{QTS}[1], \boldsymbol{QTBR}[0:k])$
$K \leftarrow t \cdot t \cdot t$
$\boldsymbol{RRK}[0:2] \leftarrow$ `FHEConstantMult`$(\boldsymbol{RR}, K)$
`// Ciphertexts required for 2's complent of` $RR \cdot t^3$
$\boldsymbol{QTC}[2k+1] \leftarrow \boldsymbol{RRK}[0] +_h \boldsymbol{1}$ `// 1's Complement of` $\boldsymbol{RRK}[0]$
$\boldsymbol{QTC}[2k+2] \leftarrow \boldsymbol{RRK}[1] +_h \boldsymbol{1}$ `// 1's Complement of` $\boldsymbol{RRK}[1]$
$\boldsymbol{QTC}[2k+3] \leftarrow \boldsymbol{one}$ `// Encryptions of integer 1`
$\boldsymbol{QTC}[2k+4] \leftarrow \boldsymbol{one}$ `// Encryptions of integer 1`
$\boldsymbol{R} \leftarrow$ `FHEAccumulate`$(\boldsymbol{QTC}[2k+4])$

The overall multiplication-depth required for computing Fridericia's Formula is the sum of depth of: partial products, CSA compression applied to $\boldsymbol{PP}[0:k]$ and accumulator applied to $\boldsymbol{QTC}[0:2k+4]$. Only two partial product computation is required during the process, thus they incur depth-2 multiplication. We use Equation 5.2 and 5.3 to calculate the depth of CSA compression and accumulator respectively. The CSA compression operates on k ciphertexts, therefore it requires multiplication-depth of

$$\left\lceil \frac{\log_2(k/2)}{\log_2(3/2)} \right\rceil + 1$$

The accumulator operates on $2k+4$ ciphertexts, therefore required multiplicative-depth is equal to

$$\left(\left\lceil \frac{\log_2(k+2)}{\log_2(3/2)} \right\rceil + 1 \right) + \left(2\log_2 \left(k + \left(\left\lceil \frac{\log_2(k+2)}{\log_2(3/2)} \right\rceil + 1 \right) \right) + 1 \right)$$

We can upper-bound the Kogge-Stone adder depth as $2\log_2 3k + 1$ to simplify the equation above. This simplification assumes that taking the cube of a k-bit number will result in at most $3k$-bits. Therefore, overall multiplication depth of computing LQTS with Fredericia's Formula is equal to

$$2 + \left(\left\lceil \frac{\log_2(k/2)}{\log_2(3/2)} \right\rceil + 1 \right) + \left(\left\lceil \frac{\log_2(k+2)}{\log_2(3/2)} \right\rceil + 1 \right) + (2\log_2 3k + 1) \qquad (5.11)$$

The process above computes LQTS detection for ciphertexts encrypting QT and RR. To check if an LQTS incident happened within QT and RR samples encrypted in N ciphertexts, first we repeat the process of computing LQTS detecting N times. Then, we aggregate the results encrypted in N ciphertexts. We recall from Equation 5.8 that we will check the sign-bit of the message to determine if $QT^3 - RR \cdot t^3 > 0$. If the sign bit of the message is 0, then an LQTS incident happened, otherwise patient's condition is NORMAL. We want to warn the doctor if *any* LQTS incident occurred within the duration of medical data captured and encrypted in N ciphertexts. Towards this, we will use reduction-AND operation and arrange AND gates in binary tree fashion to reduce N ciphertexts to one. Since AND operation will carry out any 0's of the sign-bit to the final result, we decrypt the resulting ciphertext and check if any of the sign bit of the messages packed in plaintext is 0. If this is the case, then we warn the doctor that an LQTS incident happened. The multiplication-depth of aggregation $\log_2 N$, since each AND operation requires depth-1 multiplication.

Overall multiplication-depth of detecting LQTS is the sum of the depths for:

computing Fridericia's formula and aggregation. Therefore, minimum required BGV level L for detecting LQTS with Fridericia's formula is equal to:

$$L > \left(\left\lceil \frac{\log_2(k/2)}{\log_2(3/2)} \right\rceil + \left\lceil \frac{\log_2(k+2)}{\log_2(3/2)} \right\rceil + 2\log_2 3k + 5 \right) + \log_2 N \tag{5.12}$$

5.3.2.2 LQTS Detection with Pre-computation

FHE implementation of LQTS detection presented in the previous section requires integer multiplication, which is compute intensive. An alternative way to implement LQTS detection is to avoid the expensive integer multiplication operation by reformulating Fredericia's Formula as follows:

$$\begin{aligned} \frac{QT}{\sqrt[3]{RR}} > t \quad &\Longrightarrow \quad QT^3 > RR \times t^3 \\ &\Longrightarrow \quad QT_H > RR_H \end{aligned} \tag{5.13}$$

where $QT_H = QT^3$ and $RR_H = RR \times t^3$ are pre-computed using front-end devices (left side of Figure 2.1), which transmit the FHE-encrypted versions of QT_H and RR_H into the cloud for LQTS detection. The cloud can perform LQTS detection by evaluating comparison operation as digital comparator presented in Section 5.2.5.1. The result of this comparison is TRUE or FALSE (i.e., LQTS Detected / Not Detected) in the format shown in Figure 5.7.

To check if an LQTS incident happened within QT and RR samples encrypted in N ciphertexts, first we evaluate $>_h$ to each of the N ciphertexts. Then, we aggregate the results encrypted in N ciphertexts. Towards this, we will use reduction-OR operation for aggregation and arrange OR gates in a binary tree fashion to reduce N comparison results to one. Since OR operation will carry out any bit-1 of the comparison results, we decrypt check presence of

a "1" for each message packed in the decrypted plaintext. If even a single "1" is present, this indicates that during that interval, QT_H was greater than RR_H at least once, i.e., *LQTS condition detected*. OR operation can be expressed as $OR(x_i, y_i) = x_i \oplus y_i \oplus (x_i \wedge y_i)$, which requires single multiplication-depth. Therefore, the multiplication-depth of the aggregation is $\log_2 N$.

Overall multiplication-depth of detecting LQTS with pre-computation is the sum of the depths for: digital comparator and aggregation. Therefore, minimum required BGV level L for detecting LQTS with pre-computation is equal to:

$$L > (\log_2 k + 1) + \log_2 N \tag{5.14}$$

5.3.2.3 Comparison of LQTS Detection Methods

In the previous subsections, we presented two methods to compute LQTS detection. The first method, evaluates the Friedericia's Formula over encrypted QT and RR values. The second method only evaluates homomorphic comparison operation over encryptions of pre-computed QT^3 and $RR \cdot t^3$. We now compare performance of the two methods. We assume the computations are performed over 16-bit medical samples and set the required level L for both methods based on Equations 5.12 and 5.14. To prevent overflow due to calculating QT^3, 16-bit medical sample is allocated in 48 plaintext slots. The performance of the methods over different number of medical samples is presented in Figure 5.8. LQTS detection with pre-computed QT^3 is ≈20x faster than evaluating Fridericia's Formula. The main reason for the performance difference is due to the nature of computations performed with two methods. Pre-computed LQTS detection only requires performing $>_h$ operation which has low multiplication-depth. Alternatively, computing Fridericia's formula involves expensive FHE-integer multiplication and requires higher level L, therefore it is significantly slower than pre-computation method.

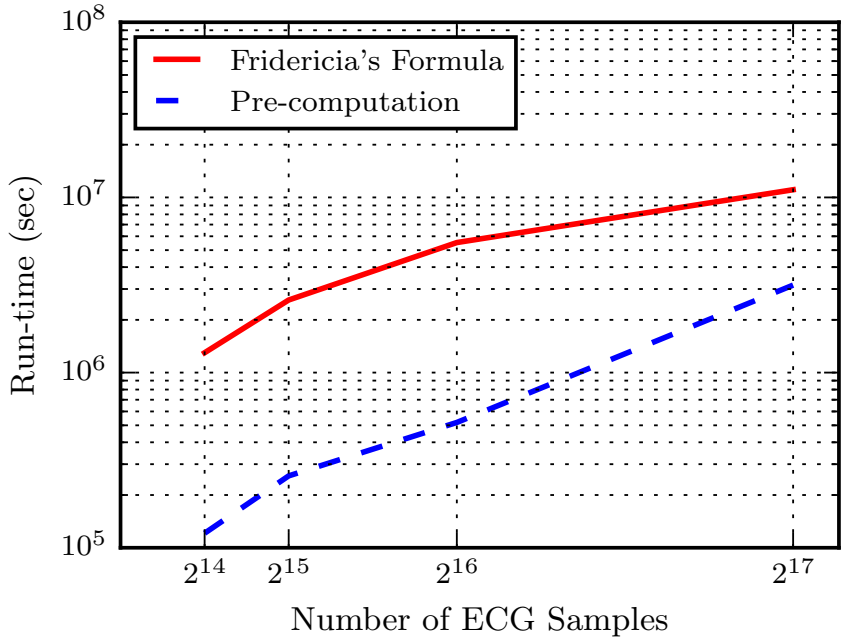


Figure 5.8: Comparison of detecting LQTS with two different methods: Fridericia's Formula and Pre-computed QT^3.

Required level and run-time performance of the two methods is detailed in Table 5.4. As presented in previous subsections, computing Friedericia's Formula requires higher level L and more compute-intensive homomorphic multiplication and rotation operations compared to Pre-computation method. This leads to longer execution time as presented in Table 5.4. We note that for the pre-computed method, the run-time of computation increases by ≈ 3.5 for the 131,072 samples. The performance difference is due to change of parameters of BGV between level 21 and 22. The number of plaintext slots and ciphertext sizes are different than level 21, which reduces the speed of the operations.

Since there is an order of magnitude performance difference between two methods, we will use pre-computation method when we evaluate LQTS detection in Chapter 7.

Table 5.4: Comparison of LQTS detection methods. Run-times are reported in seconds.

# of ECG	Fridericia's Formula		Pre-computed QT^3	
Samples	Run-time	Level (L)	Run-time	Level (L)
16,384	2053.7	40	102.5	19
32,768	2055.1	41	108.9	20
65,536	2187.1	42	109.9	21
131,072	2189.9	43	358.4	22

5.3.3 Minimum & Maximum Heart Rate

Minimum and Maximum Heart Rate (HR) computations are based on selecting between the same indexed messages packed inside two ciphertexts. Figure 5.9 presents an example of finding maximum of 4-bit messages packed into two ciphertexts.

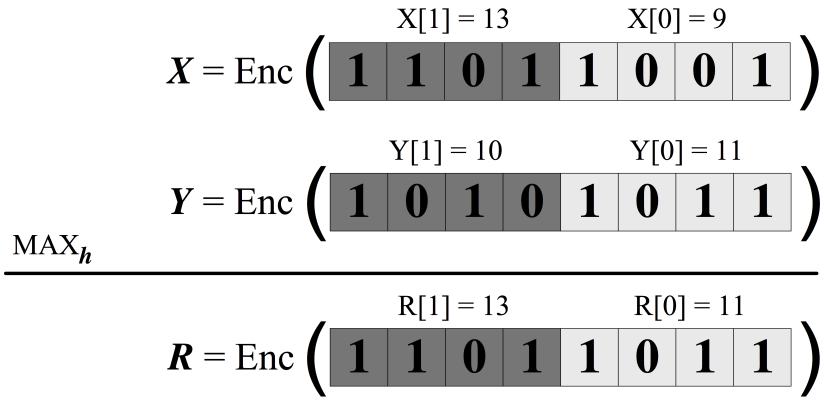


Figure 5.9: FHE based Maximum Computation

We model maximum operation as a multiplexer circuit as follows:

$$\boldsymbol{R} = (\boldsymbol{X} \times_h \boldsymbol{S}) +_h (\boldsymbol{Y} \times_h \overline{\boldsymbol{S}}) \tag{5.15}$$

where $\boldsymbol{S}$ acts as the selector of the multiplexer. The depth of computing Maximum will depend on generating $\boldsymbol{S}$. We use the result of $>_h$ to compute $\boldsymbol{S}$ as demonstrated in Figure 5.10. Note that, computing Minimum can be performed

in a similar fashion as Maximum with an additional step: inverting $\boldsymbol{S}$, which can be formulated as $\overline{\boldsymbol{S}} = \boldsymbol{S} +_h \boldsymbol{1}$. Using $\overline{\boldsymbol{S}}$ (i.e., inverted $\boldsymbol{S}$) as the selector in Equation 5.15 will yield the intended Minimum result.

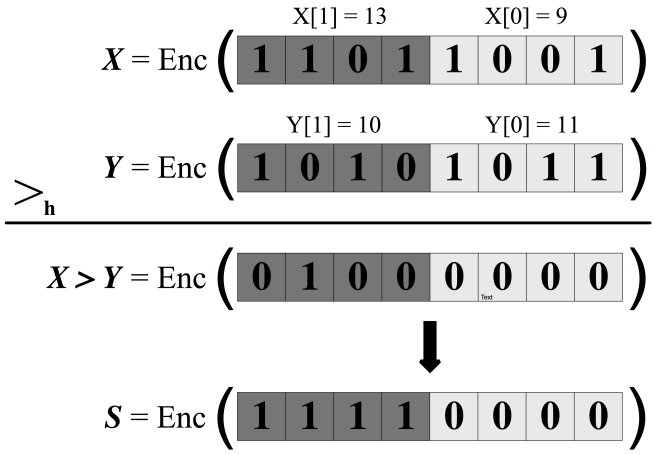


Figure 5.10: Generating selector of the multiplexer from comparison.

Naïve Computation of $\boldsymbol{S}$: In Section 5.2.5.1, we showed that comparing two k-bit numbers will produce a k-bit result. If the first number is greater, result will have a single "1" and $(k-1)$ "0"s. Otherwise the result will contain k "0"s. Generating $\boldsymbol{S}$ requires diffusing the single "1" of the greater than case to all plaintext slots for the corresponding message. We use a combination of rotate and select operations to route the "1" and add the rotated result to generate k "1"s. Pseudo-code for generating $\boldsymbol{S}$ from the comparison result is shown in Figure 5.11 (left).

Total Sums Method: Computing $\boldsymbol{S}$ with the naive method requires $2k$ rotations, selections and additions. Since we would like to perform a sum operation $(+_h)$ on each slot entry, we can use the *Total Sums Algorithm* for vector summation to reduce the number of required rotations to $2\log_2 k$ as shown in Figure 5.11 (right). Figure 5.12 presents the speedups achieved using the optimized method with different BGV levels for generating $\boldsymbol{S}$ from 16-bit integers. Average speed-up is $\approx$3.37×, which is close to the theoretical best-case speed-up of $\approx \frac{O(k)}{O(\log_2 k)}$

Naïve Method

1. $\boldsymbol{C} \leftarrow \boldsymbol{X} >_h \boldsymbol{Y}$
2. $\boldsymbol{S} \leftarrow \boldsymbol{C}$
3. **for** $i = 1$ to k **do**
4. $\quad \boldsymbol{T} \leftarrow \boldsymbol{C} >>>_h i$
5. $\quad \boldsymbol{T} \leftarrow \boldsymbol{T}\ sel_{mask}\ \boldsymbol{0}$
6. $\quad \boldsymbol{S} \leftarrow \boldsymbol{S} +_h \boldsymbol{T}$
7. **end for**
8. **for** $i = 1$ to k **do**
9. $\quad \boldsymbol{T} \leftarrow \boldsymbol{C} <<<_h i$
10. $\quad \boldsymbol{T} \leftarrow \boldsymbol{T}\ sel_{mask}\ \boldsymbol{0}$
11. $\quad \boldsymbol{S} \leftarrow \boldsymbol{S} +_h \boldsymbol{T}$
12. **end for**

Total Sums Method

1. $\boldsymbol{C} \leftarrow \boldsymbol{X} >_h \boldsymbol{Y}$
2. $\boldsymbol{R} \leftarrow \boldsymbol{C}, i \leftarrow 1$
3. **while** $i < k$ **do**
4. $\quad \boldsymbol{T} \leftarrow \boldsymbol{R}$
5. $\quad \boldsymbol{T} \leftarrow \boldsymbol{T} >>>_h i$
6. $\quad \boldsymbol{T} \leftarrow \boldsymbol{T}\ sel_{mask}\ \boldsymbol{0}$
7. $\quad \boldsymbol{R} \leftarrow \boldsymbol{R} +_h \boldsymbol{T}$
8. $\quad i \leftarrow i \cdot 2$
9. **end while**
10. $\boldsymbol{L} \leftarrow \boldsymbol{C}, i \leftarrow 1$
11. **while** $i < k$ **do**
12. $\quad \boldsymbol{T} \leftarrow \boldsymbol{L} <<<_h i$
13. $\quad \boldsymbol{T} \leftarrow \boldsymbol{T}\ sel_{mask}\ \boldsymbol{0}$
14. $\quad \boldsymbol{L} \leftarrow \boldsymbol{L} +_h \boldsymbol{T}$
15. $\quad i \leftarrow i \cdot 2$
16. **end while**
17. $\boldsymbol{S} \leftarrow \boldsymbol{L}$
18. $\boldsymbol{S} \leftarrow \boldsymbol{S} +_h \boldsymbol{R}$

Figure 5.11: Pseudo-code for computing $\boldsymbol{S}$ using Naïve (left) and Total Sums (right) methods

compared to the naïve method.

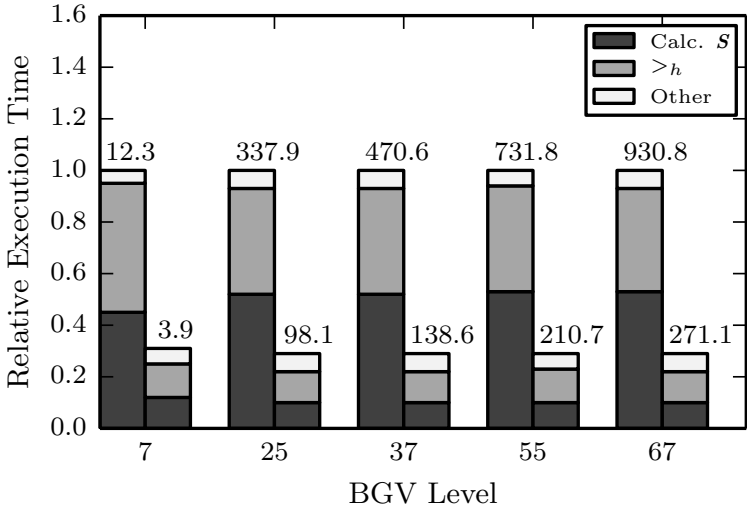


Figure 5.12: Normalized run-times (sec) for Maximum using Naïve (left bars) and Total-Sums (right bars) methods for k=16 (16-bit messages).

Generating $\boldsymbol{S}$ does not involve multiplication. Therefore, the multiplication-depth is same as digital comparator, which is $\log_2 k+1$. Once $\boldsymbol{S}$ is computed, Maximum and Minimum operations require multiplications in Equation 5.15, which adds one more multiplication-depth, totaling overall depth to $\log_2 k + 2$.

To find minimum/maximum of N ciphertexts encrypting a vector of k-bit messages, we keep applying minimum/maximum operations to ciphertexts in $\log_2 N$ stage binary tree shown in Figure 5.3(b). The overall multiplication-depth of the process is $(\log_2 k + 2) \cdot \log_2 N$. Therefore, the minimum required level L of BGV for Minimum and Maximum HR computation should be chosen as

$$L > (\log_2 k + 2) \cdot \log_2 N \tag{5.16}$$

Note that all of the functions described in this section can be accelerated by using memory-based accelerators [145–147], however, the conventional logic-based accelerators, such as the AES accelerators [98] will not work due to the orders-of-magnitude higher memory required during processing.

6 Branching Program Based FHE Implementation

In Chapter 5, we provided the implementation of our medical applications with the circuit model and showed that overall depth of the applications has two components: computation of function f over medical data and aggregation of results generated by f; we consider only multiplication-depth of our applications, which determines the effective BGV Level. In this chapter, we will look at an alternative implementation of the medical applications by using branching programs. Specifically, we will show how to improve computational efficiency by reducing the depth of overall computation to just aggregation of results by using branching program. In addition, branching program method will be easily parallelizable and have small input locality, where our sequence of homomorphic operations can be broken down to smaller sets each only dependent on a few ciphertexts. Our solution is somehow inspired from combinatorial optimization-based approaches [148–150].

Most FHE implementations show how to compute any function f as a circuit C over encrypted data. Our starting point deviates from this by first representing the function f as a branching program instead of a circuit. A branching program is a directed acyclic graph with a special start node s and final node t where each edge is labeled with either an input bit or its negation. The result of the computation is TRUE if there is a path from the start to the final node, otherwise result is FALSE.

Since a branching program can compute any function with one bit of output, we will use branching program to implement only LQTS detection application. We will show how to use BGV scheme to evaluate a branching program and aggregate the results of the computation for LQTS Detection. Using elementary linear algebra we first show that evaluating a branching program for function f is equivalent to computing the determinant of a particular matrix. More precisely, the determinant will be the result of computing $f(x)$ for the matrix corresponding to input x. Given the matrix representation of two inputs x_1 and x_2, computing the aggregation of $f(x_1)$ and $f(x_2)$ now reduces to simply multiplying the matrices corresponding to the inputs, since

$$det(AB) = det(A) \cdot det(B)$$

On a high level, our idea is to obtain encryption of the elements in the matrix from encrypted inputs via homomorphic evaluation and then multiply the matrices corresponding to all medical data during a time period. We can already see the benefit of our approach from observing that matrix multiplication is easily parallelizable. The main benefit, however, will result from the low multiplication-depth of our computation. BGV scheme or for that matter most known FHE schemes have a different cost model where performing a multiplication operation homomorphically is typically far more expensive than an addition operation and the cost of multiplication grows significantly [139] with the multiplication depth (i.e., a cascaded set of multiplications). By using branching programs, we will effectively reduce the multiplication depth of LQTS detection to $\log_2 N$, where N is the number of QT and RR pairs that will be used for the computation.

The rest of this chapter is organized as follows. In Section 6.1, details of implementing LQTS detection with branching program method is introduced. Design methodology for implementing LQTS detection application with branching

program is provided in Section 6.2. Finally, further optimizations and scalability of the proposed method is discussed in Section 6.3.

6.1 LQTS Detection with Branching Program

We recall from Chapter 2.2.1 that LQTS Detection can be computed by first finding the corrected QT value ($\mathrm{QT_c}$) using Fridericia's formula as $QT_c = \frac{QT}{\sqrt[3]{RR}}$, and then comparing ($\mathrm{QT_c}$) with a clinical threshold t which is a constant integer.

The LQTS detection processes described above can be rewritten to avoid expensive division and cube-root operations with FHE as follows

$$QT^3 > t^3 \cdot RR \tag{6.1}$$

This reduces LQTS detection to compute comparison function over the QT and RR values. We will represent the LQTS computation briefly as

$$f(QT_i, RR_i) = \alpha_i \tag{6.2}$$

where α_i will be the output 1 or 0 denoting whether $QT^3 > t^3 \cdot RR$ or otherwise.

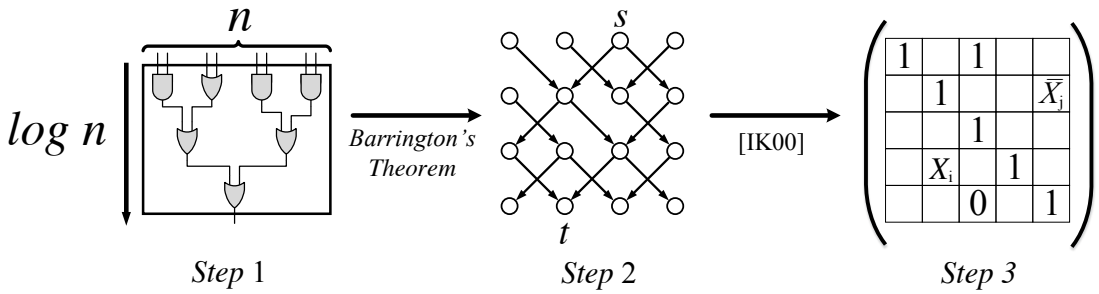


Figure 6.1: Converting function f to a branching program.

We will now focus on implementing LQTS detection with branching program model by considering the problem of comparing two numbers X and Y. We follow

the steps shown in Figure 6.1 to compute the reformatted Fridericia's Formula shown in Equation 6.1:

Step 1: Represent Equation 6.1 as a NC circuit. As presented in Section 5.2.5.1, comparison operation can be implemented via an NC circuit with a multiplication depth of $d = \log_2 k + 1$, where k is the number of bits to compare.

Step 2: Use Barrington's Theorem to transform the NC circuit to a branching program of size $T = O(4^d)$. We convert our NC circuit into a branching program to allow a matrix representation with the required properties for computing comparisons.

Step 3: Compute matrix B from the adjacency matrix A of the branching program and identity matrix I as $B = (I - A)^{-1}$. We will show that computing comparison is equal to finding determinant of matrix B', where B' is the matrix obtained by replacing i^{th} column of the matrix B with j^{th} column of I. The column replacement is performed based on the checking whether there is a path between vertex i and vertex j of the branching program. We remark that this idea is inspired by the work of Ishai and Kushilevitz [151].

We will now detail first two steps in Section 6.1.1 and step 3 in Section 6.1.2.

6.1.1 Representing Comparisons as Branching Program

In order to compare two numbers, we look at the most significant digit then work down to the least significant digit. If the most significant digit of X is larger than the most significant digit of X, then $X > Y$. For example, $200 > 193$ because $2 > 1$. Once we find a digit where there is a difference, we don't have to look at any more digits. Such a process works even when the numbers are expressed in binary format.

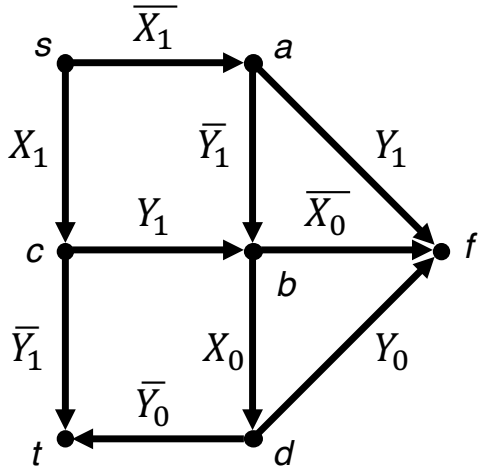


Figure 6.2: Branching Program for 2-bit Comparison

A branching program for comparing two numbers can be represented as a directed acyclic graph (DAG) based on the Definition 4. A directed acyclic graph is a graph whose edges are directed and contain no cycles. More precisely, consider the graph shown in Figure 6.2. For any given input assignments to X and Y, we construct the incident graph as follows: for every i, if $X_i = 1$ then remove all edges labeled with $\overline{X}_i$ and if $X_i = 0$ remove all edges labeled with X_i.

The problem of determining whether $X > Y$ is equivalent to determining whether there exists a path from the vertex s to the vertex t in the incident graph [24, 129]. In Figure 6.3, the example on the left considers $X > Y$ and the incident graph has a path from s and t. The other example considers $X < Y$ which results in no path from s to t.

6.1.2 Evaluating Branching Programs

In Section 6.1.1, we show that the comparison operation can be converted to a branching program by representing comparison as a DAG based on the bits of X and Y. Evaluating the corresponding branching program for $X > Y$ requires checking whether there exists a path from vertex s to t. We will show that this is

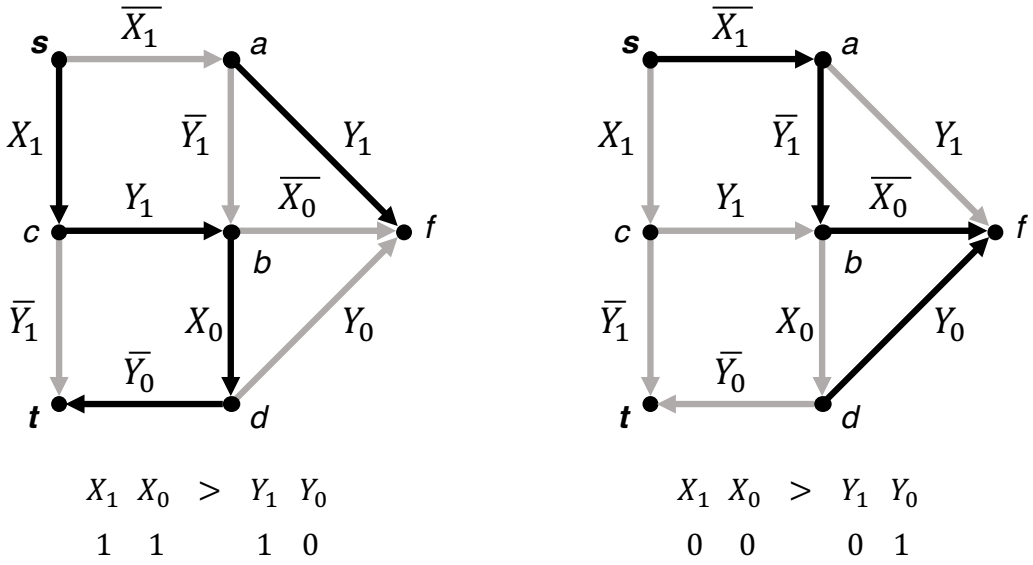
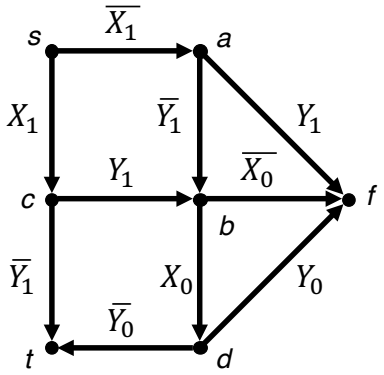


Figure 6.3: Evaluation of comparison with Branching Program

equivalent to computing *determinant* of the matrix generated during the process, which is also same as checking the *rank* of the matrix. Recall that rank of a matrix is defined to be the number of linearly independent column or row vectors and is equivalent to a non-zero determinant.

We start our process by generating the adjacency matrix of the DAG generated for the branching program to compute comparison as shown in Section 6.1.1. The adjacency matrix of the DAG is the matrix where the element at row i and column j is 1 if vertex i has an outgoing edge to vertex j, and 0 if there is no such edge. Figure 6.4 presents the corresponding adjacency matrix for the branching program shown in Figure 6.2.

For a branching program, let G be the corresponding directed acyclic graph and $A(G)$ denote the adjacency matrix corresponding to G. By the definition of a branching program, each entry in $A(G)$ is either $0, 1, x_i$ or $\overline{x_i}$ for some input bit x_i. We explain below how the adjacency matrix $A(G)$ will help encoding the computation of comparison. Assume that we have a directed acyclic graph $G = (V, E)$ for our branching program with N vertices. First we show that the



$$A = \begin{array}{c} \\ s \\ a \\ b \\ c \\ d \\ f \\ t \end{array} \begin{array}{c} \begin{array}{ccccccc} s & a & b & c & d & f & t \end{array} \\ \begin{pmatrix} 0 & \overline{X}_1 & 0 & X_1 & 0 & 0 & 0 \\ 0 & 0 & \overline{Y}_1 & 0 & 0 & Y_1 & 0 \\ 0 & 0 & 0 & 0 & X_0 & \overline{X}_0 & 0 \\ 0 & 0 & Y_1 & 0 & 0 & 0 & \overline{Y}_1 \\ 0 & 0 & 0 & 0 & 0 & 0 & \overline{Y}_0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{pmatrix} \end{array}$$

Figure 6.4: Matrix representation of the Comparison Function

number of paths of length t from vertex i to j is equal to $A(G)^t_{i,j}$.

Lemma 3. *Let $A(G)$ be the adjacency matrix of a direct acyclic graph $G = (V, E)$. Then for any $t \in N$, and for any vertices $i, j \in V$, the number of paths of length t from i to j is equal to $A(G)^t_{i,j}$.*

Proof. We will prove this by induction. As a base case, we note that $A^0 = I$ is the matrix of length zero paths, where I is the identity matrix of the appropriate size. There is a path from i to j of length 0 if and only if $i = j$. This is equivalent to saying that there is a path from i to j if and only if $I_{i,j} = 1$. As another base case, we note that by definition, $A^1_{i,j} = 1$ if and only if there is a path of length 1 from i to j. We now prove the inductive case. By inductive hypothesis, $A^{t-1}_{i,j}$ is the number of paths from i to j of length $t-1$. Every path of length t from i to j can be decomposed into a path from i to k of length $t-1$ and a path from k to j of length 1. Therefore the number of paths of length t from i to j is the sum over all k of $A^{t-1}_{i,k} A^1_{k,j}$. We observe that this is equal to $(A^{t-1}A^1)_{i,j} = A^t_{i,j}$. Therefore $A^t_{i,j}$ is the number of paths of length t from i to j. □

Now we define the path counting matrix P, where $P_{i,j}$ is the number of paths from vertex i to vertex j. Therefore, if there is a path from i to j in G, then the

$(i,j)^{th}$ entry of path counting matrix P must be non-zero. We prove that path counting matrix P is equivalent to $(I-A)^{-1}$.

Corollary 4. *For any directed acyclic graph $G=(V,E)$, and for any vertices $i,j \subseteq V$, the total number of paths from i to j in G is $(I-A)^{-1}_{i,j}$.*

Proof. Applying Lemma 3, we can see that the total number of paths from i to j is

$$\sum_{t=0}^{n-1} A^t_{i,j}$$

We claim that $I-A(G)$ is full rank. The matrix $A(G)$ is the adjacency matrix of a directed acyclic graph. Without loss of generality, a DAG never has an edge from i to j if $j<i$. Therefore, $A(G)$ is a strict upper-triangular matrix with zeroes on its diagonal. This means that $I-A(G)$ has ones all the way along its diagonal, so its determinant is one, proving that $I-A(G)$ is full rank. By observation,

$$I = (I-A(G))\sum_{t=0}^{n-1} A(G)^t$$

which is equivalent to

$$(I-A(G))^{-1} = \sum_{t=0}^{n-1} A(G)^t$$

□

Let $P(G)=(I-A(G))^{-1}$ be the path counting matrix of G. To determine $P(G)_{i,j}$, which is the number of paths between vertex i and vertex j, we have to find a particular element of the inverse of $I-A$. By applying some elementary linear algebra, we know that solving the linear system $(I-A)x = I_{,j}$ gives the column j of P, where $I_{,j}$ is column j of the identity matrix. Cramer's rule states

that if $Bx = b$ and $det(B) \neq 0$ then $x_i = \frac{det(B)'}{det(B)}$, where B' is B, except its i^{th} column is replaced with b. Therefore if we replace the i^{th} column of $I - A$ with the j^{th} column of the identity matrix and call the result A', then $P_{i,j} = \frac{det(A')}{det(I-A)}$. By inspection, $det(I - A) = 1$ so this simplifies to $P_{i,j} = det(A')$. We observe that this reasoning applies to finite fields as well. If we only want to know whether the number of paths from i to j is a multiple of p, then this expression simplifies to just $P_{i,j} = det(A') \pmod{p}$. If we know for a fact that there are either zero paths or exactly one path, then it is sufficient to compute this determinant modulo two.

6.2 Methodology

Our proposed system for computing LQTS Detection with branching program has three phases: pre-computation, cloud computation, and post computation. In the pre-computation phase, input data elements (i.e., QT and RR values) are encrypted under BGV and streamed to the cloud. In the cloud-computation phase, for each data element, the cloud generates a matrix representing the computation and aggregates the computation results via matrix multiplication. The final matrix that is the product of all matrices is downloaded and decrypted at the client. To compute the result of the computation in the post-computation phase, the client evaluates the determinant of the decrypted matrix.

6.2.1 Precomputation Phase

Recall from Section 2.3.1 that the QT and RR values from the ECG signals are computed in the ECG patch. Since encrypting with FHE and transmitting resulting ciphertext is computationally expensive, the data is transferred to a nearby computationally capable device. The privacy of the medical data during the transfer can be protected by encrypting it with AES. The values received at

the edge device is first decrypted using AES secret-key and then re-encrypted with FHE and then transmitted to the cloud. Since we will be employing the BGV encryption scheme for FHE to manipulate the data in the cloud, these values will be encrypted with BGV at the device. Towards this a public-key/private-key is generated for the appropriate BGV level L, which in turn depends on the multiplicative depth of the computation performed at the cloud. Only the public-key is stored at the edge device, since it is sufficient to encrypt the data. The private-key is maintained by the doctor and is used only to decrypt the final result of the computation. Using the public-key, the device encrypts the QT and RR values and transmits to the cloud. Since the BGV allows encrypting multiple plaintexts at once, we encrypt as many of the values in a single ciphertext.

6.2.2 Cloud Computation Phase

As the encrypted inputs arrive at the cloud, they will be first encoded into a matrix via the branching program. This matrix will have the property that its determinant will be exactly the output of computing function f on that sample. The benefit of this approach will be that the matrices corresponding to different inputs can be aggregated easily in encrypted form. We first explain the matrix encoding and then the aggregation process.

Input/Computation Encoding: From the definition of branching programs, we know that it is represented via a directed acyclic graph $G = (V, E)$ with an edge label function $\varphi : E \rightarrow \{True/False\} \cup \{x_i, \overline{x_i}\}_{i \in V}$. We will associate True with the value 1 and False with 0. Given such a graph G, we will consider the adjacency matrix $A(G)$ where the (i, j) entry of $A(G)$ is $\varphi(i, j)$. We generate the adjacency matrix $A(G)$ based on the branching program for evaluating Equation 6.2 by replacing the variables x_i with corresponding bits of QT_i and RR_i. The adjacency matrix $A(G)$ will be used to generate the matrix $B = (I - A)^{-1}$ and

B' as explained in Section 6.1.2. The determinant of the matrix B' will produce the result of the Equation 6.2.

We compute the matrix B' "symbolically" for all the bits of QT and RR values, and then plug in the values of the corresponding bits to generate the "literal" matrix. The cloud only receives the encryption of each bit of the inputs and then constructs the matrix B' by replacing literal zero and one elements with encryptions of zero and one. This can be performed by replacing elements labeled with input bit indices with the appropriate input ciphertext, negating the ciphertext if necessary. Negation of a ciphertext is an efficient homomorphic operation, which requires single homomorphic addition of the encrypted bits with encryption of ones. At the end of the encoding process, we have the bit by bit encryption of the matrix B' and now it is possible to homomorphically evaluate the regular matrix multiplication algorithm by using homomorphic multiplication and addition operations.

6.2.3 Aggregation

To detect if an LQTS occurred during a time period t, we need to aggregate the result of computing Equation 6.2 for each QT and RR values acquired during t. More formally, given a stream of QT_i and RR_i, we want to detect if there exists an element $x_i = (QT_i, RR_i)$ such that $f(x_i) = 1$ for function f in Equation 6.1. Concisely, we wish to compute

$$\bigvee_i f(QT_i, RR_i) \tag{6.3}$$

Given the matrix representation of two inputs x_1 and x_2, computing the "AND" of $f(x_1)$ and $f(x_2)$ reduces to multiplying the matrices corresponding to the inputs, since

$$det(AB) = det(A) \cdot det(B)$$

Recall that multiple data elements can be encrypted in a single BGV ciphertext and SIMD-like operations can be performed homomorphically on the ciphertext. It would be highly desirable to pack multiple elements of the matrix in such a manner that would facilitate efficient matrix multiplication.

A first approach would be to pack the elements of the matrix row-wise or column-wise. Consider the following two 3×3 matrices.

$$A = \begin{pmatrix} a_{1,1} & a_{1,2} & a_{1,3} \\ a_{2,1} & a_{2,2} & a_{2,3} \\ a_{3,1} & a_{3,2} & a_{3,3} \end{pmatrix}, \quad B = \begin{pmatrix} b_{1,1} & b_{1,2} & b_{1,3} \\ b_{2,1} & b_{2,2} & b_{2,3} \\ b_{3,1} & b_{3,2} & b_{3,3} \end{pmatrix}$$

The product of these two matrices will be

$$C = \begin{pmatrix} a_{1,1}b_{1,1} + a_{1,2}b_{2,1} + a_{1,3}b_{3,1} & a_{1,1}b_{1,2} + a_{1,2}b_{2,2} + a_{1,3}b_{3,2} & a_{1,1}b_{1,3} + a_{1,2}b_{2,3} + a_{1,3}b_{3,3} \\ a_{2,1}b_{1,1} + a_{2,2}b_{2,1} + a_{2,3}b_{3,1} & a_{2,1}b_{1,2} + a_{2,2}b_{2,2} + a_{2,3}b_{3,2} & a_{2,1}b_{1,3} + a_{2,2}b_{2,3} + a_{2,3}b_{3,3} \\ a_{3,1}b_{1,1} + a_{3,2}b_{2,1} + a_{3,3}b_{3,1} & a_{3,1}b_{1,2} + a_{3,2}b_{2,2} + a_{3,3}b_{3,2} & a_{3,1}b_{1,3} + a_{3,2}b_{2,3} + a_{3,3}b_{3,3} \end{pmatrix}$$

Suppose we packed the elements of A row-wise and B column-wise in the following manner.

$$\begin{aligned} Row[1] &= (a_{1,1}\ a_{1,2}\ a_{1,3}) & Col[1] &= (b_{1,1}\ b_{2,1}\ b_{3,1}) \\ Row[2] &= (a_{2,1}\ a_{2,2}\ a_{2,3}) & Col[2] &= (b_{1,2}\ b_{2,2}\ b_{3,2}) \\ Row[3] &= (a_{3,1}\ a_{3,2}\ a_{3,3}) & Col[3] &= (b_{1,3}\ b_{2,3}\ b_{3,3}) \end{aligned}$$

Then using a single operation of $Row[1] \times Col[1] = (a_{1,1}b_{1,1} \quad a_{1,2}b_{2,1} \quad a_{1,3}b_{3,1})$ is required to compute $C[1,1]$, which looks promising. However HElib does not provide operations to directly sum the elements of a packed ciphertext, i.e., summation of $a_{1,1}b_{1,1}$, $a_{1,2}b_{2,1}$ and $a_{1,3}b_{3,1}$. It is possible to use rotate and select operations to compute the first entry of the product matrix. However, this still does not give the product matrix in the same form directly as A or B so that we can

continue multiplying with C to aggregate the next matrix. More homomorphic operations needs to be performed to bring C to that form. We instead propose a new representation that will allow easy multiplication of matrices and yields the product matrix in the same form.

We represent matrices as a list of vectors representing "diagonals" in the original matrix. More precisely, given a $n \times n$ matrix, we pack the elements into n ciphertexts, where the i^{th} diagonal contains the elements $(1, 1+(i \ mod \ n)), (2, 1+(i+1 \ mod \ n)), \ldots, (n, 1+(i+n-1 \ mod \ n))$. For any matrix A, let $A\{i\}$ denote the i^{th} diagonal. We only consider square matrices, so an $n \times n$ matrix has n diagonals.

In our 3×3 matrix A, the diagonals will be

$$A\{1\} = (a_{1,1} \ a_{2,2} \ a_{3,3})$$
$$A\{2\} = (a_{1,2} \ a_{2,3} \ a_{3,1})$$
$$A\{3\} = (a_{1,3} \ a_{2,1} \ a_{3,2})$$

Given this representation, the matrix product C can be computed as follows

$$C\{1\} = A\{1\}B\{1\}^0 + A\{2\}B\{3\}^1 + A\{3\}B\{2\}^2$$
$$C\{2\} = A\{1\}B\{2\}^0 + A\{2\}B\{1\}^1 + A\{3\}B\{3\}^2$$
$$C\{3\} = A\{1\}B\{3\}^0 + A\{2\}B\{2\}^1 + A\{3\}B\{1\}^2$$

In this notation, concatenation indicates element-by-element multiplication, $+$ indicates element-by-element addition, and v^i indicates the rotation of vector v by i elements, namely the sequence $[v_i, v_{i+1}, \ldots, v_1, \ldots, v_{i-1}]$.

We extend this formula to say that when $C = AB$ and A and B are $n \times n$ matrices, then

$$C\{i\} = \sum_{j=1}^{n} A\{j\}B\{1 + (i - j \ mod \ n)\}^{j-1} \tag{6.4}$$

HElib supports all of these operations efficiently, so this algorithm is much

more efficient for computing the product of matrices homomorphically. In more detail, each of the diagonals are encrypted into packed ciphertexts. Given two matrices encrypted in diagonal format, we can compute the product in diagonal format using HElib operations. An apparent disadvantage of this method is that while homomorphic matrix multiplication has multiplication-depth of 1 and its operations can be bundled together efficiently, it still requires many homomorphic multiplications. However, these multiplications can be done in parallel. Furthermore, if the matrices are of a special sparse form, we can multiply them more efficiently.

6.2.4 Post-computation Phase

Recall that at the end of the Cloud Computation phase, after all the encrypted matrices are aggregated with matrix multiplication, the Cloud generates a single matrix that contains the value of the aggregated output. This matrix is downloaded by the health provider, such as the doctor who has the private-key associated with the encryption scheme, it can decrypt the matrix. Finally, to compute the answer, the determinant of this matrix needs to be evaluated. The multiplication evaluates an AND operation, while we need the OR operation. Hence, we consider the branching program for the function whose output is the negated output of the function f. This is denoted by $\neg f$. Now we recast the original problem with f using $\neg f$. More precisely,

$$\bigvee_i f(QT_i, RR_i) = \neg \left(\bigwedge_i \neg f(QT_i, RR_i) \right)$$

This means that upon receiving the final matrix, the determinant is evaluated and the output is negated to obtain the precise answer to the computation.

6.3 Optimizations and Scalability

Our proposed approach works for any boolean function that can be represented as a branching program. A branching program can compute any boolean function by simply branching on all n input bits so that each of $2n$ inputs result in a unique path. The size of this program is exponential in n. However, we are interested in what functions are representable by polynomial-size branching programs. As we have already seen from Lemma 2, any log-space computations can be represented as a branching program and can encompass a wide variety of problems. Therefore, this approach can be used for any log-space computable boolean functions. The size of the matrix corresponds to the size of the graph. For the equation in our case study, we construct a branching program of size 600.

If A is a band matrix, then there is a small w such that all nonzero entries are within w diagonals of the main diagonal, and we can multiply matrices much more efficiently using operations on bundled plaintexts. This w corresponds to the width of the branching program. By Barrington's theorem, any NC^1 circuit can be converted into an equivalent branching program of polynomial size and width 5. For any such circuit, we can convert it into a band matrix with band width 10. If we pack each entire diagonal into one ciphertext, then we can represent matrices of arbitrary size with just 21 ciphertexts. After multiplying two matrices with band width 10, we get a matrix with band width 11, and after we multiply two matrices with band width 11, we get a matrix with band width 12, and so on. Therefore the resulting matrices remain extremely sparse, even when iteratively multiplying many matrices together. This means that we get the advantage of sparseness for every round of matrix multiplications, not just the first one.

We note that homomorphic evaluation and combining of branching programs can be supported for any branching program using only homomorphic matrix multiplication. It is therefore effective for the server to use highly-optimized software

or even special hardware to enhance the efficiency of this operation. The branching program we construct for our case study has width 10, and the corresponding band matrix has width 9.

7 Evaluation

In this chapter, we will evaluate the performance of our proposed medical application. Our focus will be on the secure computation of medial applications using Fully Homomorphic Encryption based on two computational approaches: circuit and branching programs. We will also access the requirements for secure storage and sharing of the medical data in the public cloud. Designing cloud-based medical applications requires special attention to protect the privacy of the medial data, especially when public cloud resources are utilized. We analyze security requirements of the components of such a system, and review conventional and emerging cryptosystems for their applicability for each component.

The rest of this chapter is organized as follows. In Section 7.1, an overview of the encryption schemes that will be used during evaluation is provided. This section summarizes and compares the capabilities of the encryption schemes. Section 7.2 presents the experimental setup for the evaluation. In Section 7.3, security requirements for acquisition devices are reviewed. Section 7.4 details the performance and resource requirements of secure computation of the medical data with FHE. Finally, section 7.5 compares the conventional and emerging cryptography schemes for secure storage and sharing of medical data.

7.1 Overview of Encryption Schemes

Table 7.1 summarizes the secure computation and secure data sharing capabilities of the encryption schemes presented in Chapters 3 and 4. We note that computation and data sharing can be achieved in trusted storage such as a private cloud or a hospital datacenter. Yet, we evaluate encryption schemes based on their capabilities in public clouds, where service providers cannot be trusted.

Table 7.1: Comparison of encryption schemes based on secure computation and secure data sharing. (NA=Not Available for public cloud)

Type	Encryption	Computation	Data Sharing
Conventional	AES [52]	NA	Limited
	ECIES [112]	NA	Limited
ABE	KP-ABE [64]	NA	Fine-Grained
	CP-ABE [63]	NA	Fine-Grained
Homomorphic	BGV [15]	Full	Limited

Standard encryption schemes cannot provide computation, unless medical data is stored in a trusted private cloud where decryption is possible without violating the privacy. Data sharing is limited to the users who have the secret key and the private key of AES and ECIES, respectively. This requires either sharing the secret key of AES or encrypting data with several public-keys that creates duplicates.

Like standard encryption schemes, ABE cannot perform computations on data unless data is stored in a private cloud. However, ABE provides a fine-grained sharing of data stored in public cloud access.

Only Homomorphic encryption schemes can provide computation over encrypted data stored in public cloud. The BGV scheme performs arbitrary computations but requires more resources compared to the other schemes. Homomorphic encryption schemes limit data sharing to the users who have the private key, just like ECIES.

7.2 Experimental Setup

In our experiments, we use two libraries for implementation: Charm [152] and HElib [128]. The Charm library provides a high-level framework for designing cryptosystems. Charm is based on Python, but compute intensive operations are implemented in C, with a comparable performance to native C implementations. HElib is a state-of-the-art FHE library that implements the BGV scheme [15].

We use Charm for benchmarking the performance of standard and ABE encryption schemes covered in Chapter 3. Medical applications presented in Chapters 5 and 6 are implemented by extending HElib.

All experiments are run on a workstation computer with an Intel Xeon W3565 Processor with 4 cores and 8 threads with 24GB RAM and Ubuntu 15.04 64-bit as the operating system. Since the HElib library is not thread-safe at the moment, we report only single-thread run-time results.

7.2.1 Data Set

We use a sample patient medical data from the THEW ECG database [42] to simulate medical data collected with the acquisition devices (e.g., BAN sensors). The dataset contains raw ECG data captured from a patient during 24-hours via a 12-lead Holter monitor with 1000Hz sampling rate. The ECG data represents a summary of the each heart beat and provides information about the QT and RR intervals in terms of number of samples acquired (*toc*). 24-hour ECG data contains 87,896 samples, and each *toc* value is represented as a 16-bit unsigned integer.

7.2.2 Security Level of Encryption Schemes

We use 128-bit security for encrypting medical data, which is the recommended security level for federal government data by the National Institute of Standards and Technology (NIST) [153]. Table 7.2 presents the parameter selection of encryption schemes based on 128-bit security. For the BGV scheme, we use the analysis provided in [154] for setting the security parameters.

Table 7.2: Parameter selections of encryption schemes for 128-bit security.

ECIES [112]	Elliptic curve: $\mathbb{F}_p$ with p = 256-bit prime Symmetric-key encryption: AES-128 MAC: HMAC-SHA1 (160-bit)
CPABE [63]	Bilinear Pairing: Supersingular curve over $\mathbb{F}_p$, p = 1536-bit prime Access Policy: 10 attributes
KPABE [64]	Elliptic curve: $\mathbb{F}_p$ with p = 256-bit number p Symmetric-Key encryption: AES-128 MAC: HMAC-SHA1 (160-bit) Access Policy: 10 attributes

7.2.3 BGV Setup

Runtime and storage requirements of BGV are tightly coupled with the level L. The level L depends on the number of plaintext-slots allocated for a message (k) and the number of ciphertexts required for computation (N), as shown in Section 5.3. For the fastest execution time, we set the level L of the BGV scheme to the lowest possible value that allows the execution of all homomorphic operations without exceeding the noise threshold.

Table 7.3 presents the relevant parameters for calculating the minimum required level L for BGV. For all the computations `nMsgs` is equal to number ECG samples in our data set: 87,896. ECG samples in our data set are stored as 16-bit integers, yet our LQTS detection and Average HR computations require taking

cube of the samples and accumulating them, respectively. Therefore, to prevent incorrect results we allocate 32 plaintext slots per ECG sample for these applications. We pad each sample with 0's before assigning them to corresponding 32 plaintext slots. Since the maximum range of the medical data is 10-bits, using 32 plaintext slots will be sufficient without causing overflow. Minimum and Maximum HR computations are based on comparing numbers and selecting them, thus do not change the precision of the sample bit-length and 16 plaintext slots are allocated per ECG sample for the computation.

Table 7.3: Relevant BGV parameters for determining level L

Term	BGV Definition	Usage in Application
`nSlots`	# of slots in a plaintext	–
`nMsgs`	# of messages	# of ECG samples
k	# of slots allocated for a message	ECG sample bit-length
N	# of ciphertexts	to store all samples

The number of ciphertexts N required to encrypt the dataset depends on the number of plaintext slots (`nSlots`), which are determined by the BGV level. Table 7.4 presents the number of plaintext slots (`nSlots`) for different BGV levels. Each ciphertext can encrypt $\lfloor \texttt{nSlots}/k \rfloor$ messages that enables SIMD-like parallel homomorphic operations.

Table 7.4: Number of packed messages in a plaintext at various BGV levels for different message bit lengths.

BGV Level (L)	`nSlots`	16-bit messages ($k=16$)	32-bit messages ($k=32$)
$1 \leq L < 12$	630	39	19
$12 \leq L < 22$	682	42	21
$22 \leq L < 68$	1285	80	40
$68 \leq L < 77$	1650	103	51
$77 \leq L < 100$	2048	128	64

We use Table 7.4 to calculate required level L for our applications. For example, recall that LQTS detection with digital comparator requires $(\log_2 k + 1 +$

$\lceil \log_2 N \rceil$) levels. We derive the level L required to encrypt 87,896 ECG samples, where each sample is allocated into 32 plaintext slots, as follows:

$$L > (\log_2 k + 1 + \lceil \log_2 N \rceil) \implies L > (6 + \lceil \log_2 N \rceil) \implies$$
$$L > \left(6 + \left\lceil \log_2 \left\lceil \frac{87,896}{\lfloor \frac{\texttt{nSlots}}{32} \rfloor} \right\rceil \right\rceil\right) \implies \boxed{L{=}20} \tag{7.1}$$

L value can be iteratively computed from Equation 7.1 as $L = 20$, which implies packing 21 ECG samples in each plaintext containing 682 slots. Although $\frac{682}{32} = 21.3125$ messages can be packed into 682 slots, partial messages are not utilized in our implementation for simplicity, causing a wasted space of 10 slots in each plaintext. Since each plaintext can hold 21 messages, the number of ciphertexts required to encrypt our 24-hour ECG data is equal to:

$$N{=}\left\lceil \frac{\texttt{nMsgs}}{\lfloor \frac{\texttt{nSlots}}{k} \rfloor} \right\rceil \implies N{=}\left\lceil \frac{\texttt{87,896}}{\lfloor \frac{\texttt{682}}{32} \rfloor} \right\rceil \implies \boxed{N{=}4186}$$

7.3 Acquisition

Acquisition devices such as ECG patches and BAN devices have strict resource requirements, therefore the communication between acquisition devices and acquisition devices-to-Cloudlet must be secured with lightweight encryption schemes. We will use AES-128 for encrypting medical data captured with acquisition devices. Symmetric-key of AES-128 is shared via Elliptic Curve Diffie-Hellman (ECDH) key-exchange.

ECDH is used once to generate the same secret key between communicating parties. During the key exchange, two parties exchange a single ciphertext that represents a point in the elliptic curve. This ciphertext contains the (x, y) coordinates, each represented as a p-bit integer in $\mathbb{F}_p$. A 256-bit $\mathbb{F}_p$ is selected for

the elliptic curve to achieve 128-bit security. Therefore, the exchanged ciphertext has a size of $2 \cdot (256/8) = 64$ B. Both parties need to perform elliptic curve point multiplications to a generate secret key for AES. Our Charm library simulation for this shows a total run-time of 0.23 ms.

Once the secret key is generated, medical data can be securely transferred by using AES-128. Our Charm library simulation for AES-128 encryption and decryption times are 0.2 μs and 0.23 μs, respectively. These are the performance results for the AES-CBC mode of operation that is used in the OpenSSL library implementation.

The AES-GCM mode can be used to provide both confidentiality and integrity. AES-GCM mode can be implemented efficiently by using the techniques introduced in Section 3.2.1. By using the Intel AES-NI instruction set extensions, the optimized code that is published on Intel's website [104] resulted in AES-GCM encryption and decryption run times of 0.06 μs per 128-bit block.

The performance of AES-GCM mode can be further improved by using ASIC or FPGA implementations. A fully pipelined ASIC implementation of AES-GCM is presented in [103], which can run at 429.2 MHz and perform encryption/decryption in ≈ 2.3 ns per block.

Our proposed system uses AES encrypted version of the medical data for permanent storage. To store medical data for secure computation or sharing purposes, first a cloudlet will decrypt AES encrypted medical data and re-encrypt with the desired encryption scheme.

7.4 Computation

In this section, we provide the FHE-based implementation results of the three fundamental operations described in Section 5.3: 1) Average Heart Rate computation, 2) LQTS detection and, 3) Minimum/Maximum Heart Rate computation.

Specifically, we will provide the circuit based FHE implementation results for all the operations and branching program based FHE simulation results for LQTS detection.

7.4.1 Circuit Based Implementation Results

In Section 5.3, we analyzed the required minimum BGV level (L) for these three fundamental operations, and showed that L depends on two parameters: number of plaintext slots allocated per message (k) and the total number of ciphertexts (N). Table 7.5 summarizes the minimum required BGV level L for each operation for a given pair of k and N values. We note that, due to significant performance difference between two methods proposed for LQTS detection presented in Section 5.3.2.3, we only consider pre-computation based LQTS detection.

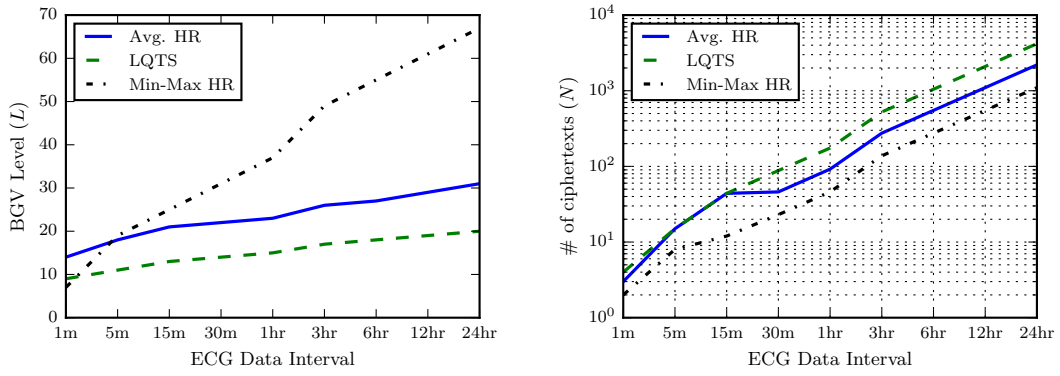
Table 7.5: BGV Level required for each operation.

Operation Type	Required BGV Level L
Average HR	$\left(\left\lceil \frac{\log_2(N/2)}{\log_2(3/2)} \right\rceil + 1\right) + (2\log_2 k + 1)$
LQTS	$\log_2 k + 1 + \lceil \log_2 N \rceil$
Min, Max HR	$(\log_2 k + 2) \times (\lceil \log_2 N \rceil)$

We used THEW ECG database [42] to simulate the acquired ECG samples (Phase I in Figure 2.1) which contains raw ECG data captured from a patient via a 12-lead Holter monitor at a 1000 Hz sampling rate. A 24-hour time period is processed to extract the heart beat information (i.e., the RR interval in Figure 2.6) in an ISHNE format annotation file [155] and can be readily used to simulate our Phase I, where our acquisition devices capture raw ECG data and then pre-process them to extract RR and QT intervals before sending them to cloud in FHE-encrypted format. We encrypted the 87,896 values in this annotation file using FHE, each of which is the temporal distance between two heart beats in number-of-samples acquired from Holter monitor during the RR interval (termed

toc). We perform operations over encrypted *toc* values, so our results will be in terms of *toc*. which can be trivially converted to "time" values by multiplying them with the sampling rate (1000 Hz) at the doctor's phone/tablet after decrypting the final result.

To evaluate the performance of the circuit-based implementation, we partition the 24-hour data interval to different time intervals. Each application operates on all the data from the given data interval and generates the summarized result. For example, Average HR computation with 1-min ECG data interval calculates the average heart rate for every minute, while 1 day ECG interval operates on all 24-hour data and returns a single result. ECG intervals can be adjusted to reflect condition of the patient, a patient with critical condition might require computations performed on every minute data while a healthy person just needs computations per day. Based on the selected ECG data interval, the BGV level L required for each application is shown in Figure 7.1(a). The levels are calculated by using the Tables 7.4 and 7.5. The total number of ciphertexts involved in each application for different ECG data interval are presented in Figure 7.1(b). As expected computations over longer ECG data intervals require more ciphertexts.



(a) BGV Level L for each application.

(b) Number of ciphertexts involved in the computation for each application.

Figure 7.1: BGV Level L and number of ciphertexts required for each application for different ECG data intervals.

We evaluate the performance of the proposed system based on three relevant metrics, each quantifying a different operational cost for cloud outsourcing [2]:

Computation Rate (Γ): We define Γ as:

$$\Gamma = \frac{\Gamma_{out}}{\Gamma_{in}} \tag{7.2}$$

where Γ_{in} is the time interval for the data being transmitted from the patient's house into the cloud, and Γ_{out} is the computation time in the cloud for this data. This "relative" definition allows us to determine whether FHE computations in the cloud can *catch up* with the rate of the incoming data ($\Gamma \leq 1$), or lag behind ($\Gamma > 1$).

The significance of the Γ metric is its ability to signal the necessity of additional storage space, and the added computational latency in providing the final result to the doctor. For example, $\Gamma = 2$ implies that, 1 hour patient data takes 2 hours to compute, causing a one hour delay in providing the results to the doctor, and additional storage space to buffer the incoming encrypted data (which is not a trivial amount, as we will shortly see).

Γ values of each medical application for different ECG data intervals is presented in Figure 7.2. Additionally, $\Gamma = 1$ is plotted in the figure to show the cut off point for catching up with the incoming data. The Average HR computation has $\Gamma < 1$ for all of the ECG data intervals, which makes it the most efficient application in terms of computation rate. This is due to very fast reduction operations performed by the CSA compression. Alternatively, both LQTS detection and Min-Max HR computation have $\Gamma > 1$ for the ECG intervals longer than 15 minutes. This indicates that either additional storage required to buffer incoming data or more processors needed to perform computations in parallel for the data intervals longer than 15 minutes.

Storage Expansion (Λ): As shown in Figure 4.4, FHE significantly expands

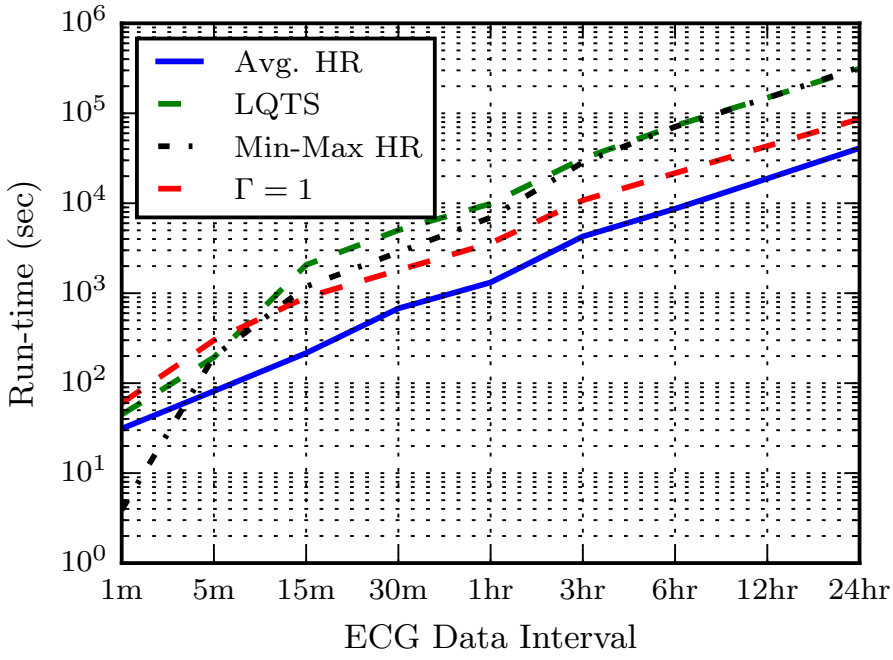


Figure 7.2: Run-time of the medical applications. $\Gamma = 1$ denotes the execution time same as data interval time.

the amount of storage required to store the encrypted incoming patient data as well as the FHE public keys. This *storage expansion* becomes worse for increased BGV levels, L. In our definition, $\Lambda = 10,000$ implies that, to store one byte of plain data in encrypted format, 10,000 bytes of cloud storage is required.

Storage expansion of the applications for different ECG intervals is presented in Figure 7.3. All applications observe the exponential increase in the storage with the longer ECG Data Intervals due to the more ciphertexts involved in the computation with higher BGV level L. The Min-Max HR computation has the worst Λ among the applications in Figure 7.3 which requires higher L as shown in Table 7.5.

Network Throughput (Υ): FHE-based computations strain the network bandwidth, since large amount of encrypted data must be transmitted over WAN connections. We define a third metric, Υ, that determines how much data is being

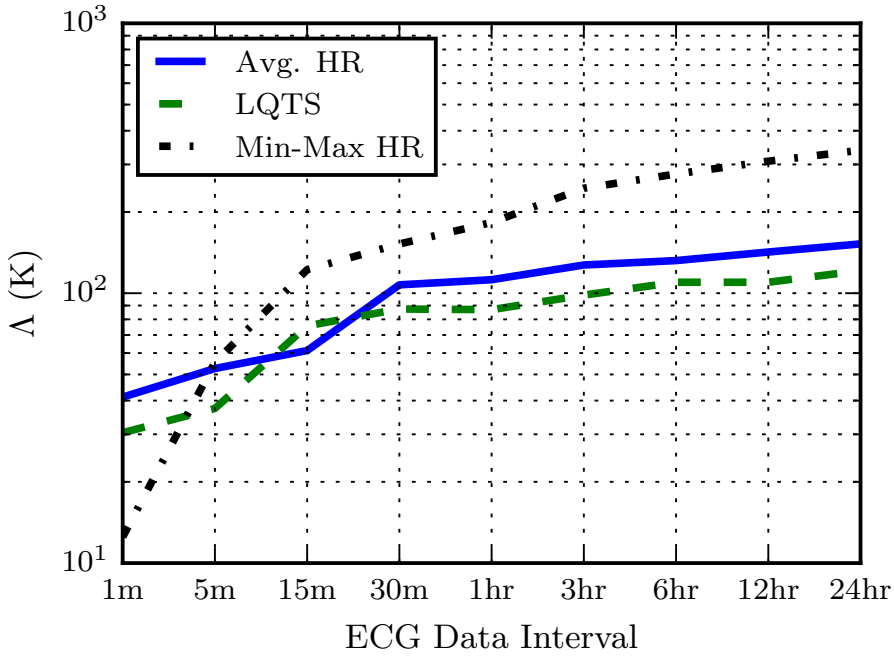


Figure 7.3: Storage extension of encrypted medical data with Homomorphic Encryption.

transmitted across the WAN during computations. Some cloud operators (e.g., AWS [2]) only charge for outgoing traffic, not for the incoming traffic. Therefore, we break Υ into two separate parts: $\Upsilon_{patient}$ is the amount of data transferred from front-end devices used by the patient, and Υ_{doctor} is the data transferred from cloud to back-end device used by the doctor.

$\Upsilon_{patient}$ of each medical application for different ECG data intervals is presented in Figure 7.4. The $\Upsilon_{patient}$ shows a similar trend as the storage expansion presented in Figure 7.3. Since all the data generated from the patient must be transferred to the cloud, the amount of the data increases with the ECG data interval.

Alternatively, Υ_{doctor} shows an exponential decrease in the amount of the data transferred out of the cloud with longer ECG intervals as presented in Figure 7.5. The doctor will receive a single summarized result for the given ECG data interval,

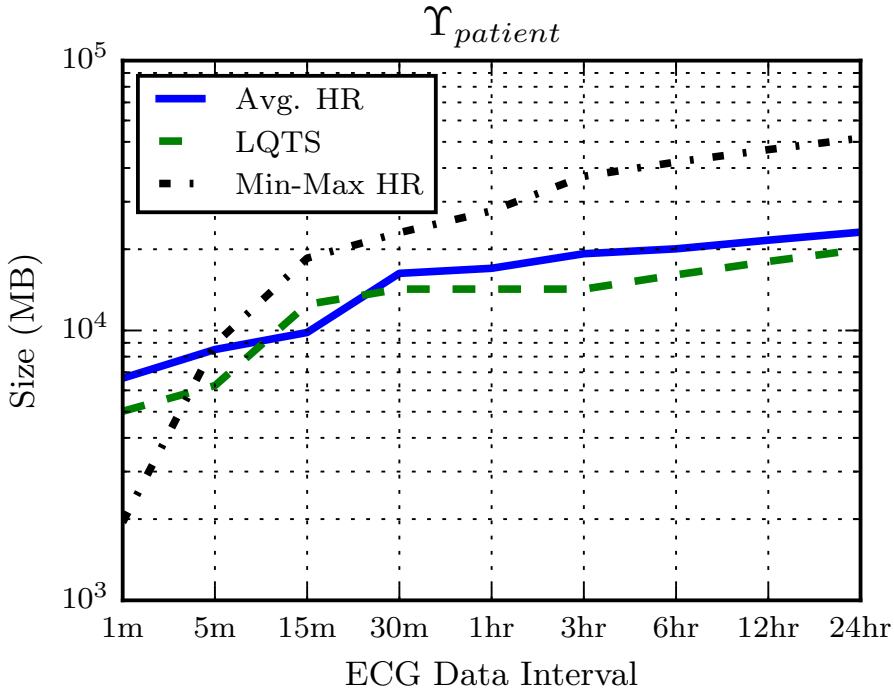


Figure 7.4: Data transferred from the cloud to doctor for different medical applications.

and computations on longer ECG intervals result in reduced data transfer from the cloud. For example, for the same 24-hour ECG data, the doctor will receive a single ciphertext if data interval is chosen as 24-hour, whereas 24 ciphertexts will be send for 1-hour data interval. The amount of data transfer for each application depends on the required BGV level which determines the ciphertext size. As shown in Figure 7.1(a), the Min-Max HR requires higher L, therefore the amount of the data needs to be transferred to the doctor is larger. Alternatively, LQTS has the lower L thus has the least network congestion.

We present the details of our experimental results in Table 7.6 for the aforementioned three fundamental operations over a 24 hours of ECG data, containing 87,896 samples. Rows of the table represents the partition of the data used in computations. For example, for the Average HR, the last row indicates N=2198, L=31, and Γ=0.47. These are the results we derived in Equation 5.9 at the end of

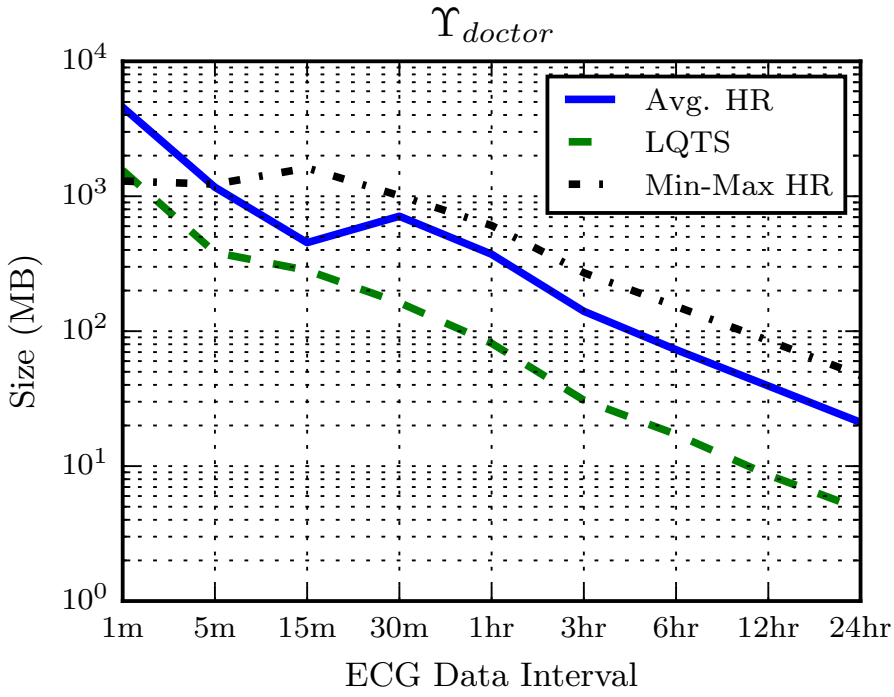


Figure 7.5: Data transferred from the cloud to doctor for different medical applications.

Section 5.3.1. From Table 7.6, we see that, Γ=0.47, translating to a computation time of $24 \times 0.47 = 11.28$ hours, or, 40,997 seconds, as indicated in the "Run-time" column. Since $\Gamma < 1$, we deduce that, the FHE computations can be done faster than the rate of data arrival, thereby eliminating the necessity to buffer large amounts of data in the cloud. However, the 24-hour ECG data still takes up $152,300\times$ more space than the original raw data, as indicated in the Λ column. Computing Average HR requires shuttling 22.6GB of encrypted data from the patient's house to the cloud (the $\Upsilon_{patient}$ column), and requires transferring 21.1MB of resulting ciphertext from the cloud to the doctor (the Υ_{doctor} column). The significant disparity is due to the substantial amount of aggregation performed during the computation, leaving only the highly-summarized results that need to be transferred to the doctor's smartphone. This is good news in the sense that, cloud operators, such as AWS, only charge for outgoing data, which is orders of

magnitude lower in this case.

Let us now focus on the remaining rows of Table 7.6. The specific row we just described was based on 24 hours of accumulated patient data, acquired, transmitted, and computed as a whole (indicated as "24 hr" in the "ECG Data Interval" column). This row assumes that, the Average HR computaion will operate on 24-hour data and produce single result at the end. Alternatively, the second row of the Average HR (indicated as "5 min") displays N=15, L=18 and Γ=0.27. This can be interpreted as: When the entire 24-hour data is transmitted 5 minutes at a time, the amount of each chunk is significantly smaller: 5 minute chunks require only 15 ciphertexts (N=15), each of which is encoded at an FHE level of L=18. The computation time for each chunk is 81.6 seconds, corresponding to $\Gamma=\frac{81.6}{300}\frac{s}{s}$=0.27. Although this row looks computationally-advantageous based on the computation metric (Γ), we see that, it hurts the Υ_{doctor} metric, since the total amount of data to transfer 24 hours of data in 5 minute chunks ends up being 1166.4 MB. To state alternatively, the smaller the granularity of the results, the worse the Υ_{doctor} becomes, and the better the Γ is. The storage required for these results improves when the chunk size is smaller due to the reduced level, L, to encrypt these chunks. For example, while Λ=52,600 for 5 minute chunks, it increases to Λ=152,300 for a single large 24-hour chunk. Note that, for the Average HR, 1 minute chunks perform worse than 5 minute chunks. Since only 3 ciphertexts are used for the computation, efficient CSA compression is only used once and run-time is dominated by the Kogge-Stone Adder.

The trade-offs involving these three parameters become less obvious when the cloud operators' billing is based on availability of the resources. For example, if the application has the flexibility to wait for the results longer, different rows in Table 7.6 might provide different cost alternatives for the healthcare organization, which is the rationale for our detailed analysis.

Table 7.6: Operational cost of medical applications based-on: Computation Rate (Γ), Storage Expansion (Λ) and Network Throughput (Υ).

Operation	ECG Data Interval	N	L	Enc. (sec)	Dec. (sec)	Ctxt (MB)	Pkey (MB)	Run-time (sec)	Γ	Λ	$\Upsilon_{patient}$ (GB)	Υ_{doctor} (MB)
	1 min	3	14	0.30	0.24	3.17	274.95	31.2	0.52	41.2K	6.5	4564.8
	5 min	15	18	0.41	0.31	4.05	352.23	81.6	0.27	52.6K	8.3	1166.4
Average	15 min	44	21	0.45	0.39	4.72	399.12	215.9	0.24	61.3K	9.6	453.1
Heart Rate	30 min	46	22	1.88	0.77	14.81	1721.55	677.7	0.38	107.4K	15.9	710.9
	60 min	92	23	1.96	0.80	15.49	1783.98	1317.4	0.37	112.2K	16.6	371.7
(k=32)	3 hr	275	26	2.21	0.90	17.55	2038.95	4272.7	0.40	127.2K	18.8	140.4
	6 hr	550	27	2.30	0.94	18.24	2093.75	8679.8	0.40	132.1K	19.6	72.9
	12 hr	1099	29	2.46	1.05	19.63	2247.22	18803.9	0.44	142.1K	21.1	39.3
	24 hr	2198	31	2.69	1.15	21.03	2414.25	40997.4	0.47	152.3K	22.6	21.1
	1 min	4	9	0.06	0.02	1.08	101.91	44.7	0.75	30.4K	4.9	1555.2
Long-QT	5 min	15	11	0.08	0.02	1.33	115.81	193.7	0.65	37.4K	6.1	383.1
Syndrome	15 min	44	13	0.19	0.15	2.94	274.13	2064.1	2.29	75.7K	12.2	282.3
(LQTS)	30 min	88	14	0.22	0.18	3.39	325.14	5014.3	2.79	87.4K	13.9	162.8
Detection	60 min	175	15	0.22	0.18	3.39	325.73	9885.1	2.75	86.9K	13.9	81.3
	3 hr	524	17	0.24	0.19	3.85	350.43	31341.1	2.90	98.4K	15.7	30.8
(k=32)	6 hr	1047	18	0.26	0.23	4.30	399.83	72845.3	3.37	109.8K	17.6	17.2
	12 hr	2093	19	0.27	0.23	4.30	399.87	149083.3	3.45	109.8K	17.6	8.6
	24 hr	4186	20	0.31	0.25	4.75	425.53	313188.1	3.62	121.2K	19.5	4.75
	1 min	2	7	0.08	0.04	0.90	74.32	3.9	0.07	12.5K	1.98	1296.1
	5 min	8	19	0.31	0.32	4.27	365.07	198.1	0.66	55.4K	8.73	1229.7
Min, Max	15 min	12	25	2.12	0.86	16.86	1984.95	1185.4	1.32	122.3K	18.09	1618.6
Heart Rate	30 min	23	31	2.58	1.06	21.03	2414.25	2852.8	1.58	152.2K	22.57	1009.5
	60 min	46	37	3.09	1.27	25.27	2880.37	6946.8	1.93	182.8K	27.12	606.5
(k=16)	3 hr	138	49	4.28	1.73	33.86	3850.81	28459.1	2.64	244.9K	36.34	270.9
	6 hr	275	55	4.75	1.88	38.21	4372.26	70849.2	3.28	276.5K	41.01	152.8
	12 hr	550	61	4.99	2.05	42.57	4918.28	148189.6	3.43	308.4K	45.69	85.2
	24 hr	1099	67	5.04	2.29	46.94	5362.67	325331.9	3.77	339.7K	50.38	46.9

7.4.2 Branching Program Simulation Results

In this section, we will provide simulated results for branching program method and will compare them to the circuit-based method. Note that, while the circuit method allows a significantly more generalized application domain, we will demonstrate in this section that, once the function to achieve is very well defined (i.e., a Yes/No answer to a pre-determined question), the branching-programs method (which will be referred to as the "matrix method") provides much more improved performance results.

Our branching program method first generates the matrix encoding in encrypted form for the i^{th} input sample QT_i and RR_i that is provided in encrypted form. Given then matrices for all the samples, the aggregated output is simply the product of these matrices. To generate a matrix encoding, we need a branching program to compute $QT^3 > RR$. Towards this we pursue the following approach:

We now describe how to construct the branching program and the ideas behind it. Suppose that the j^{th} bit of QT is 1. Then we know that $QT^3 \geq 2^{3i}$. Therefore if any of the first $2n/3$ bits of QT are set then the equation will be true, because the lowest value QT^3 can take will be bigger than the largest value RR can take. By checking through the first $2n/3$ bits sequentially we can determine whether this is the case. Now it remains to verify the equation when this is not the case. Here we need to compare the cube of the low order bits of QT with RR. Next we observe in a branching program consider a sequence of vertices reachable from the start vertex through exactly one path. This path can effectively "remember" the path that led up to them and therefore all the bits that were tested along that path in the sequence of vertices. This allows a branching program to simulate a table lookup. By branching on each of the $2n/3$ least-significant bits, we get a path for each value of QT below $2n/3$. Once we have this there is a vertex for each possible value which remembers its associated QT value (and therefore

know its associated QT^3 value). Next from each such vertex we scan through the bits of RR to determine which is larger. Next we explain how the comparison is performed in the last step.

Before we explain how to compare numbers in a branching program we will briefly recall how to compare numbers in plaintext, such as 254 and 249. To compare two numbers, we start from the most significant digit and look for the index where the digits of the given numbers are different. The result is determined by the first index where the digits differ. If there is no such index, then the two numbers are equal. For our example of comparing 254 and 249, the most significant digits are the same so we look at the second digit, where 5 is greater than 4, therefore $254 > 249$.

The diagram in Figure 7.6 contains an abbreviated form of the circuit $x^3 < y$ for 9-bit inputs x and y. We use x and y to emphasize the fact that this is not the same circuit because we ignore the constant multiplicative factor to the right-hand side of the equation in the original formula. The leftmost vertex is the start vertex and the rightmost vertex is the end vertex. After traversing the edges labeled with $x_3 = 1, x_2 = 0$, and $x_1 = 1$ (in that order) we know that $x = 5$ and $x^3 = 125$. We compare the bits of y with the bits of 125 from the most significant bit to the least significant bit. At the first difference in the bits of y and 125 we can determine the result, so if were comparing a particular bit we know that the higher order bits of y were the same as those in 125 (the two bit strings are equal so far). If a bit j is the first bit where y and 216 differ then we can determine which one is bigger: if $y_j = 1$ then $x^3 > y$, otherwise $x^3 \leq y$. If the two bits are the same then we continue comparing them until we get to a difference, or until we get to the end. If the last bits are the same then $x^3 = y$, otherwise $x^3 > y$.

Similarly, after traversing the edges labeled with $x_3 = 1, x_2 = 1$, and $x_1 = 1$ (in that order) we know that $x = 7$, so $x^3 = 343$. We can proceed in the same way with the different value of x^3. Each path will compare y with a value of x^3

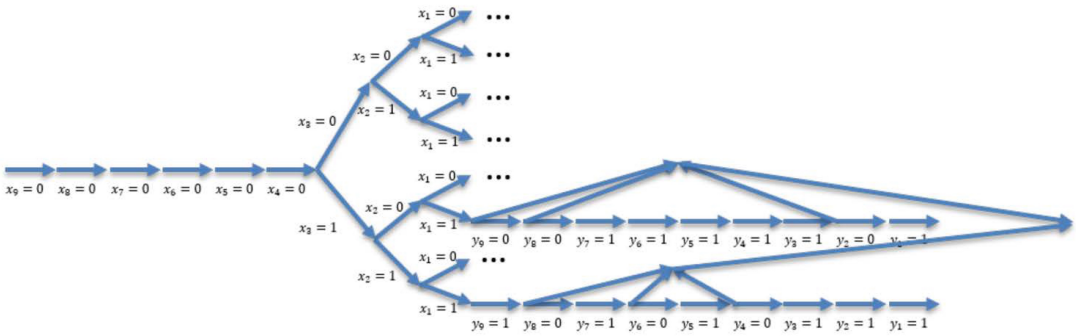


Figure 7.6: Diagram of the Branching Program for computing $254 > 249$?

"known" to that path.

By considering all possible outcomes, we can generate a branching program based on the bit values of x and y. Based on the branching program, a symbolic matrix is generated, where each bit value will be replaced by the BGV encrypted version of the bit. Since BGV ciphertexts, we will use the rotate and select operations, and we to generate only 10 packed ciphertexts for our branching program. We evaluate the product of the matrices using Equation 6.4. Computing each diagonal of the resulting matrix requires at most 10 rotate and 10 multiply operations and 10 additions.

7.4.2.1 Performance Estimation

In Chapter 6, we showed that branching program reduces the required depth of the LQTS detection compared to circuit model. Furthermore, branching program evaluates the LQTS detection computation as matrix multiplication, which can be computed in parallel. Therefore, our goal is to compare the performance of the branching program and circuit model with multi-processor setting.

However, as of this moment the HElib library [128] is not thread-safe and does not allow parallel computation. Instead, we will simulate the parallel computation process for both approaches to compare their performance. To accomplish this,

we first perform real benchmarks of each individual operation on single-thread machine, and then we will use performance of each operation to estimate the computational costs for our parallel experiments. Since the performance of the operations in BGV scheme depends on the level L, we compute the required level L for both methods during the benchmark.

In the circuit method required level L depends on the depth of both computation and aggregation. Computation requires $\log_2 k + 1$ multiplication-depth with digital comparator as presented in Section 5.2.5.1. Aggregation requires $\log_2 N$ multiplications for aggregating N samples. In the matrix method, there is no multiplication involved in setting up the matrix and only the aggregation involves multiplication. Since every matrix multiplication involves only one sequential multiplication, the overall depth to compute and aggregate is simply $\log_2 N$.

Finally, we wrote a compiler that took a sequence of operations to be performed on the samples and used a greedy heuristic to assign the computations to the different parallel processors taking into account the dependencies. The greedy heuristic maintains a list of free processors and assigns the next computation to the first available processor in the list. A clock was simulated to determine when the next processor will be available and the processor list was updated accordingly. In our estimate we assumed that each parallel machine had instantaneous access to the input encryptions and results of computations from other machines. We deliberately ignored the data transfer and sharing costs since it would be hard to assess and incorporate. Nevertheless, we argue that the estimates are conservative in the sense that circuit method requires a lot of data transfers than branching program. Taking into account the data transfer costs will only improve the relative timing of branching program.

7.4.2.2 Results

To compare both methods, we considered computing and aggregating LQTS with two different numbers of samples: 10 and 10240. Figure 7.7 presents the performance of processing 10 and 10240 samples in parallel with different number of processors. We can see that both approaches improve with the number of processors. However, the branching program approach is consistently better than the circuit approach by a factor of 20. The main reason for this is that the cost of computation increases exponentially with the depth of the computation and the depth of the circuit approach is significantly higher than that of the branching program approach.

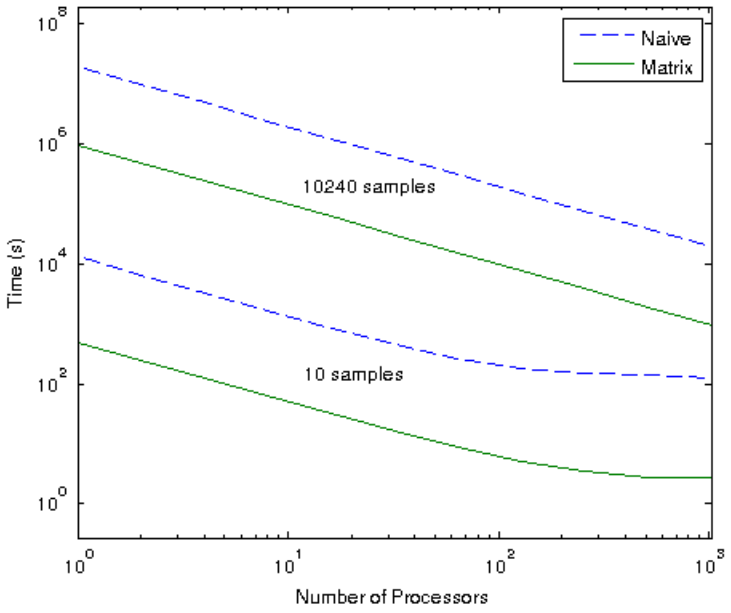


Figure 7.7: Simulated parallel computation time for matrix (branching program) method vs. naïve(circuit) method of Long QT detection. The matrix method is consistently better by a factor of 20x.

One trade-off that is not represented in the graph is the amount of network bandwidth used. The communication from the patient's end will be the same for both approaches, but at the doctor's end, an entire matrix (i.e. roughly 20

ciphertexts) will have to be downloaded in branching program approach while only 1 ciphertext needs to be downloaded in the circuit approach. However, in our medical application, this will only be done once every day (or few hours), so this should not be a significant problem. Another difference with the branching program approach is that the doctor's computer must compute the determinant of the decrypted matrix in order to get the result; in the circuit approach, the result will be directly available (or be a simple "OR" of the decrypted bits).

7.5 Storage and Sharing

Once the medical data is captured, it is transferred to a more computationally capable device such as mobile phone or cloudlet. The data can be encrypted with different encryption schemes based on desired capability (i.e., sharing, computation) before transferred to public cloud by first decrypting AES-128 and re-encrypting the data with the selected encryption scheme.

Table 7.7 lists execution times and storage requirements for ciphertexts for different encryption schemes. Encryption (Enc.) and Decryption (Dec.) columns list the required time to encrypt/decrypt 24-hr ECG data, consisting of 87,896 samples as described in Section 7.2.1. Ctxt column shows the space required for storing encrypted data. For BGV scheme, we consider the resource requirement for computing Average HR over 24-hr (Last row of "ECG Data Interval" in Table 7.6).

Table 7.7: Requirements of encrypting 24-hr ECG data with different schemes.

Encryption	Enc. (sec)	Dec. (sec)	Ctxt (MB)
ECIES [112]	40.3	38.7	8.4
KP-ABE [64]	439.5	615.3	56.7
CP-ABE [63]	58 K	32.5 K	708.1
BGV [15]	5912	2527	46.2 K

We detail the resource requirements of each encryption scheme in the following subsections.

7.5.1 ECIES

For ECIES, we select AES-128 for symmetric-key cryptography and HMAC-SHA1 for HMAC. The ciphertext generated by the ECIES encryption has three components: a point on the elliptic curve, an AES-128 encrypted message and a *tag* generated by HMAC-SHA1. A point on the elliptic curve has two 256-bit coordinates, the AES-128 encrypted message is 128-bits and the *tag* from HMAC-SHA1 is 160-bits. Therefore total ciphertext size is equal to $(2{\cdot}256{+}128{+}160)/8 = 100\,\text{B}$. Encryption and decryption operations using ECIES require 0.46 ms and 0.44 ms, respectively based on Charm results.

7.5.2 Attribute-Based Encryption (ABE)

For ABE, we consider two candidates: CP-ABE scheme from [63] and the recent KP-ABE scheme from [64]. We evaluate both schemes based on an access policy P, consisting of 10 attributes. Table 7.8 presents the execution times of the operations for the ABE schemes.

Table 7.8: Average execution time (ms) of ABE operations

ABE Scheme	Setup	KeyGen	Encrypt	Decrypt
KP-ABE [64]	5	4	5	7
CP-ABE [63]	140	136	660	700

The ciphertext and key sizes change with policy P and we report the sizes for both based on our chosen policy based on 10 attributes.

A ciphertext in the CP-ABE scheme consists of the set C', C, C_y, C'_y, where C_y and C'_y are generated for each attribute in the policy P. Each element in the

ciphertext is a point on the elliptic curve, which is represented as two coordinates in the 1536-bit prime field $\mathbb{F}_p$. Therefore, the total size of a ciphertext in the CP-ABE scheme is $(2 \cdot (1 + 1 + 10 + 10) \cdot 1536)/8 = 8448\,\text{B}$. Encryption and decryption operations are performed in 660 ms and 700 ms, respectively based on Charm results.

In the KP-ABE scheme, a ciphertext consists of the set C, *tag*, and C_i, where a different C_i is generated for each attribute in the policy P. C is the 128-bit ciphertext, encrypted using AES-128. The *tag* is generated using HMAC-SHA1 and 160-bits. Each C_i is a point on the elliptic curve, which is represented as two coordinates in the 256-bit prime field $\mathbb{F}_p$. The total size of a ciphertext in the KP-ABE scheme is $(128 + 160 + (2 \cdot 10 \cdot 256))/8 = 676\,\text{B}$. Encryption and decryption operations are performed in 5 ms and 7 ms, respectively based on Charm results.

While the KP-ABE scheme is more efficient and has less storage requirements compared to the CP-ABE scheme, in general KP-ABE schemes put more pressure on the key-generator. Since the ciphertexts are generated based on the attribute set and the private keys contain the access policy, the key-generator has to specify carefully which users can access the data. On the other hand, in CP-ABE schemes, users are associated with attributes based on their credentials and ciphertexts contain the access policy. This makes converting already widely used Role Based Access systems to CP-ABE schemes almost without any extra effort.

7.5.3 BGV

In the BGV scheme, ciphertext sizes depend on the level L. The resource requirements are calculated from the results reported in Table 7.6. To calculate Average Heart Rate during 24-hours, we need to set level as $L = 31$. A plaintext at this level has 1285 slots and packs $\lfloor 1285/32 \rfloor = 40$ 32-bit samples. Therefore, $\lceil 87,896/40 \rceil = 2198$ ciphertexts required to encrypt all ECG samples. Each ci-

phertext is 21.03MB, thus 46GB storage space is needed store 2198 ciphertexts. Encryption and decryption operations are performed in 2.69 sec and 1.15 sec, respectively.

We summarize the comparison of encryption schemes for secure storage and sharing of the medical data in Table 7.9. Encryption/Decryption time and ciphertext size present values normalized to AES.

Table 7.9: Comparison of encryption schemes. Enc. Time, Dec. Time and Ctxt Size values are normalized to AES.

Encryption	Enc. Time	Dec. Time	Ctxt Size
AES [52]	1	1	1
ECIES [112]	2.3 K	1.9 K	6.3
KP-ABE [64]	25 K	30.4 K	42
CP-ABE [63]	3.3 M	1.6 M	528
BGV [15]	9 M	3.6 M	1.3 M

The standard encryption schemes provide efficient and low-cost storage for storing medical data in the public clouds. Unfortunately, these schemes only protects the privacy of the data. More complex operations such as secure data sharing more advanced and emerging cryptography schemes. ABE based schemes offer fine-grained data access of multiple users with reasonable overhead, especially by using recent KP-ABE scheme [64]. However, these schemes cannot perform secure computations when data is stored in public cloud. Homomorphic encryptions on the other hand can perform computations on the encrypted data with significant overhead without providing secure sharing.

8 Cost Analysis of Implementation in the Cloud

In this chapter, we will analyze the cost of outsourcing our medical applications to the public cloud service providers. The cost analysis will be based on the implementation of the applications with circuit based FHE presented in Section 5. We will show that despite the substantial resource intensity of the applications, they can still be run on a reasonable budget in the cloud [6].

Figure 8.1 highlights the resource requirement of the proposed medical applications from the cloud pricing perspective. We recall from Chapter 2.3 that medical applications will allow patient health monitoring at home through medical data acquisition devices, such as, ECG patches [11], or multi-sensory acquisition devices [156]. The acquired patient data is encrypted (top of Figure 8.1) and then transmitted to the cloud (middle of Figure 8.1). To ensure PHI privacy, a novel encryption mechanism called, Fully Homomorphic Encryption [15]) is used, which requires encrypting the data PHI encryption at home, after which point the data is unrecognizable to the cloud, however, it remains executable [7, 157]. Note that, this patient transmission is one-way. On the bottom of Figure 8.1, the HCO personnel and HCO administrators can access the application with their mobile device such as a smart-phone or tablet, and transmit their preferences (i.e., two-way). The medical application runs in the cloud, which operates on the encrypted

PHI without ever observing actual medical data.

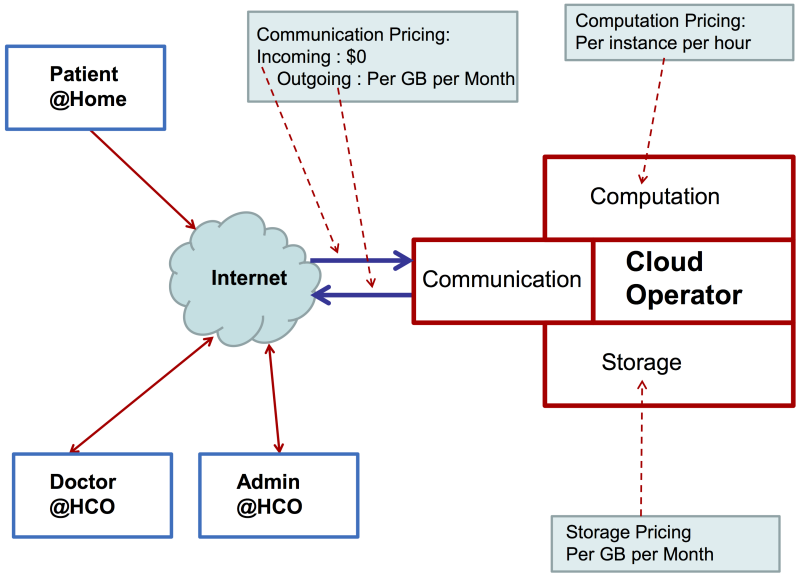


Figure 8.1: Proposed medical application environment and related cloud pricing metrics. HCO = Healthcare Organization.

As we showed in Chapter 7, compared to conventional cloud applications, the significant difference of this application scenario is its resource intensity. We describe *resource* as one of the three entities that a cloud operator charges for as follows:

Storage: This most widely utilized resource involves renting a shared storage space from the cloud operator. The pricing of this resource varies widely based on the allocated space. The most meaningful product to purchase for an HCO is based on per-GB-per-month pricing.

Computation: The execution of the medical algorithms based on Fully Homomorphic Encryption puts a substantial strain on cloud computational resources. While it is possible to cut costs by using shared computational instances, it makes a lot more sense to use dedicated *boxes* (i.e., comput-

ers) to perform the type of computations. The pricing for these instances is based on per-hr usage of that instance.

Communication: This resource involves data traffic from and into the cloud operator. The data transfer pricing is based on per-GB-per-month data traffic. Some of the cloud service providers [36] do not charge for the incoming-data, but rather charge for the outgoing-data.

The rest of this chapter is organized as follows. In Section 8.1, a brief background for Mobile Cloud Computing is provided. Section 8.2 presents pricing metrics for major public cloud service providers. Section 8.3 provides details of Service Level Agreements for cloud services. Finally, in Section 8.4 monthly costs of running medical applications is calculated.

8.1 Mobile Cloud Computing

Cloud is the platform of multiple servers over a widely distributed geographic area, connected by the Internet for the purpose of serving data or computation [158]. Mobile Cloud Computing (MCC) can be described as a model for enabling convenient, on-demand network access to a shared pool of configurable computing resources [159] from mobile devices. MCC therefore refers to both the applications delivered as services over the Internet and the hardware and system software in datacenters that provide those services. The services themselves have long been referred to as Software as a Service (SaaS). Some vendors use terms such as Infrastructure as a Service (IaaS) and Platform as a Service (PaaS) and others to describe their products, but we abstain from these because accepted definitions for them still differ widely. The datacenter hardware and software is what we will call a cloud. When a cloud is made available in paying costs as they occur to the general public, it is called a public cloud and the service being sold is utility

computing. Private cloud on the other hand refers to internal data centers of a business or other organization [160].

The point at which these internal data centers are large enough to enable organizations to benefit from the advantages of cloud computing are the subject of much debate. In [161], authors described cloud computing as the sum of SaaS and utility computing, but does not include small or medium-sized datacenters, though some of these rely on virtualization for management. People can be users or providers of SaaS, or utility computing. The focus here is on SaaS providers (cloud users) and cloud providers, who have received less attention than SaaS users. Mobile computing is the delivery of services, software and processing capacity over the Internet, reducing cost, increasing storage, automating systems, decoupling of service delivery from underlying technology, and providing flexibility and mobility of information. However, the actual realization of these benefits is far from being achieved for mobile applications [161].

MCC is introduced as an integration of cloud computing into the mobile environment and represents a relatively new and fast growing segment of the cloud computing paradigm [162]. It is a model for enabling convenient, on-demand network access to a shared pool of configurable computing resources [163]. MCC is a model for transparent elastic augmentation of mobile devices via ubiquitous wireless access to cloud storage and computing resources, with context-awareness and dynamic adjusting of offloading in respect to change in operating conditions, while preserving available sensing and interactive capabilities of mobile devices [164].

In view of the inherent advantages of this technology, enterprises today are looking to cloud computing to help them better deliver existing as well as new, innovative services on demand across network, computing, and storage resources at reduced cost [165]. A recent white paper [166] titled “Economics of the Cloud” states that the mobile computing industry is moving towards the cloud driven by three important economies of scale:

1. large data centers can deploy computational resources at significantly lower costs than smaller ones

2. demand pooling improves utilization of resources

3. multi-tenancy lowers application maintenance and labor costs for large public clouds

The cloud also provides an opportunity for IT professionals to focus more on technological innovation rather than thinking of the budget of waiting to force things to move. However, many organizations find it difficult to determine the total operating costs of using cloud services [166].

8.2 Cloud Operator Pricing

In this section, we provide details of pricing metrics for three major cloud service providers: Amazon Web Services (AWS), Google Cloud Platform and Microsoft Azure. For each provider the pricing will be based on the US East region. Operating system is assumed to be Linux.

8.2.1 Amazon Web Services (AWS) Pricing

The cost of running applications in AWS is determined by three factors: computation, storage and data transfer. We use AWS Elastic Compute Cloud (EC2) instances to calculate computation cost and AWS Simple Storage Service (S3) to calculate storage cost. Data transfer cost is included for only outgoing data transfers from EC2 and S3 to the Internet, while the incoming data transfer is free of charge.

Table 8.1 presents EC2 instance types and their configurations. EC2 instances are grouped into categories based on their capabilities. `t2` and `m3` instances are

Table 8.1: AWS EC2 Instance Types with different configurations. The cost of EC2 instances differ with usage type. Reserved instances include a one-time upfront fee but offer lower monthly cost compared to On Demand instances.

AWS EC2 Instance Configurations						On Demand	3-Year Reserved	
Instance Name	vCPU	Memory (GB)	Storage (GB)	Physical Processor	Freq. (GHz)	Hourly Rate	Upfront Fee	Hourly Rate
t2.micro	1	1	-	Xeon family	2.5	0.013	109	0.002
t2.small	1	2	-	Xeon family	2.5	0.026	218	0.004
t2.medium	2	4	-	Xeon family	2.5	0.052	436	0.008
m3.medium	1	3.75	1×4 SSD	Xeon E5-2670	2.5	0.070	337	0.015
m3.large	2	7.5	1×32 SSD	Xeon E5-2670	2.5	0.140	673	0.03
m3.xlarge	4	15	2×40 SSD	Xeon E5-2670	2.5	0.280	1345	0.06
m3.2xlarge	8	30	2×80 SSD	Xeon E5-2670	2.5	0.560	2691	0.12
c3.large	2	3.75	2×16 SSD	Xeon E5-2680	2.8	0.105	508	0.022
c3.xlarge	4	7.5	2×40 SSD	Xeon E5-2680	2.8	0.210	1016	0.045
c3.2xlarge	8	15	2×80 SSD	Xeon E5-2680	2.8	0.420	2031	0.09
c3.4xlarge	16	30	2×160 SSD	Xeon E5-2680	2.8	0.840	4063	0.18
c3.8xlarge	32	60	2×320 SSD	Xeon E5-2680	2.8	1.680	8126	0.359
g2.2xlarge	8	15	1×60 SSD	Xeon E5-2670	2.6	0.650	6307	0.06
r3.large	2	15.25	1×32 SSD	Xeon E5-2670	2.5	0.175	1033	0.026
r3.xlarge	4	30.5	1×80 SSD	Xeon E5-2670	2.5	0.350	2066	0.052
r3.2xlarge	8	61	1×160 SSD	Xeon E5-2670	2.5	0.700	4132	0.104
r3.4xlarge	16	122	1×320 SSD	Xeon E5-2670	2.5	1.400	8264	0.208
r3.8xlarge	32	244	2×320 SSD	Xeon E5-2670	2.5	2.800	16528	0.416
i2.xlarge	4	30.5	1×800 SSD	Xeon E5-2670	2.5	0.853	2740	0.121
i2.2xlarge	8	61	2×800 SSD	Xeon E5-2670	2.5	1.705	5480	0.241
i2.4xlarge	16	122	4×800 SSD	Xeon E5-2670	2.5	3.410	10960	0.482
i2.8xlarge	32	244	8×800 SSD	Xeon E5-2670	2.5	6.820	21920	0.964
hs1.8xlarge	16	117	24×2,048	Xeon Family	2.0	4.600	16924	0.76

for general purpose applications. `c3` instances are optimized for computation with high-performance processors. `g2` instances contain GPUs for graphics and general purpose GPU applications. `r3` instances are optimized for memory-intensive applications and provide lowest price per GB of RAM. `i2` and `hs1` instances are optimized for storage and provide lowest price per GB of storage on the instance. The pricing of EC2 instances depends on the usage as shown in Table 8.1. On demand instances are charged by the hour with no commitments. Reserved instances charge for one-time up front fee but in return provide lower hourly cost. Reserved instances differ in commitment time (1 year vs 3 years) and utilization (light, medium and heavy). Additionally, EC2 provides Spot Instances, which can be purchased by bidding on unused EC2 instances. Pricing of the Spot Instances

is set by EC2 and may change based on availability of these instances.

Table 8.2 presents the AWS S3 storage and Data Transfer Out cost based on amount of data used per month. AWS only charges for data transferred out from AWS to Internet.

Table 8.2: AWS S3 Storage and Data Transfer Out Pricing.

AWS S3 Storage		Data Transfer Out	
Amount (TB)	Price	Amount (TB)	Price
First 1	0.0300	First 1 (GB)	0.00
Next 49	0.0295	Up to 10	0.12
Next 450	0.0290	Next 40	0.09
Next 500	0.0285	Next 100	0.07
Next 4000	0.0280	Next 350	0.05
Over 5000	0.0275		

8.2.2 Google Cloud Platform Pricing

Google Cloud Platform offers IaaS services through Google Compute Engine (GCE) as shown in Table 8.3. GCE billing includes a minimum of 10 minutes of usage. After this 10 minute minimum interval, each instance is charged at 1-minute increments (rounded up to the nearest minute). Additionally, GCE offers discounts based on sustained (continuous) usage. The `f1-micro` and `g1-small` instances are for non resource-intensive tasks that remain active for long periods of time.

Google charges network differently for networking inside the cloud infrastructure and outside communication via the Internet as shown in Table 8.4. Incoming data into the cloud services and outgoing data for same zone or to a different cloud service in the same region are free. Alternatively, outgoing data to a different Zone in the same Region or different Region within the US are charged at $0.010 per GB of data movement. Finally storage costs are $0.04 per GB per month for standard disks and $0.325 per GB per month for SSD disks.

Table 8.3: Google Compute Engine Pricing

Instance Type	Cores	Memory (GB)	Price	Discounted (25% - 50%)	Discounted (50% - 75%)	Discounted (75% - 100%)
n1-standard-1	1	3.75	$0.06	$0.05	$0.04	$0.03
n1-standard-2	2	7.5	$0.13	$0.10	$0.08	$0.05
n1-standard-4	4	15	$0.25	$0.20	$0.15	$0.10
n1-standard-8	8	30	$0.50	$0.40	$0.30	$0.20
n1-standard-16	16	60	$1.01	$0.81	$0.61	$0.40
n1-highmem-2	2	13	$0.15	$0.12	$0.09	$0.06
n1-highmem-4	4	26	$0.30	$0.24	$0.18	$0.12
n1-highmem-8	8	52	$0.59	$0.47	$0.36	$0.24
n1-highmem-16	16	104	$1.18	$0.95	$0.71	$0.47
n1-highcpu-2	2	1.8	$0.08	$0.06	$0.05	$0.03
n1-highcpu-4	4	3.6	$0.16	$0.13	$0.10	$0.06
n1-highcpu-8	8	7.2	$0.32	$0.26	$0.19	$0.13
n1-highcpu-16	16	14.4	$0.64	$0.51	$0.38	$0.26
f1-micro	1	0.6	$0.01	$0.01	$0.01	$0.01
g1-small	1	1.7	$0.03	$0.03	$0.02	$0.01

Table 8.4: Google Cloud Platform Network Pricing

Outbound data transfers (per GB)	Cost
0 1 TB	$0.12
1 10 TB	$0.11
>10 TB	$0.08

8.2.3 Microsoft Azure Pricing

Microsoft Azure offers pricing for two tiers: basic and standard. While both tiers are similar in terms of virtual machine configurations, standard tier offers additional capabilities such as load balancing and auto-scaling for performance improvement. Table 8.5 and 8.6 demonstrate the pricing for basic and standard tiers respectively, which provide all possible machine (instance) configurations. Compute intensive Instances A8 and A9 contain the Intel Xeon E5-2670 CPU @2.6 GHz.

Table 8.5: Basic Tier Pricing for A and D series of virtual machines

Instance Type	Cores	RAM (GB)	Disk (GB)	Price (per hr)
A0	1	0.75	20	$0.018
A1	1	1.75	40	$0.044
A2	2	3.5	60	$0.088
A3	4	7	120	$0.176
A4	8	14	240	$0.352
D1	1	3.5	10	$0.077
D2	2	7	40	$0.154
D3	4	14	100	$0.308
D4	8	28	200	$0.616
D11	2	14	40	$0.195
D12	4	28	100	$0.390
D13	8	56	200	$0.702

Table 8.6: Microsoft Azure Standard Tier Pricing for A and D series of virtual machines

Instance Type	Cores	RAM (GB)	Disk (GB)	Price (per hr)
A0	1	0.75	20	$0.020
A1	1	1.75	70	$0.060
A2	2	3.5	135	$0.120
A3	4	7	285	$0.240
A4	8	14	605	$0.480
A5	2	14	135	$0.250
A6	4	28	285	$0.500
A7	8	56	605	$1.000
A8	8	56	382	$1.970
A9	16	112	382	$4.470
D1	1	3.5	50	$0.094
D2	2	7	100	$0.188
D3	4	14	250	$0.376
D4	8	28	500	$0.752
D11	2	14	100	$0.238
D12	4	28	200	$0.476
D13	8	56	400	$0.857
D14	16	112	800	$1.542

Storage prices are for persistent storage and billed by GB per month. Table 8.7 tabulates the storage pricing when the data is stored with 99.9% read/write availability SLA [4]. Data transfers are charged based on the direction to Azure data centers. Inbound data transfers, which are the data transfers into the data centers are free of charge. Outbound data transfers are charged based on the data amount

shown in Table 8.7.

Table 8.7: Microsoft Azure Storage and Data Transfer Pricing

Azure Storage		Data Transfer Out	
Amount (TB)	Price (per GB)	Amount (GB)	Price
0 -1	0.0240	First 5 GB	0.00
1 - 50	0.0236	5 GB - 10 TB	0.087
50 - 500	0.0232	Next 40 TB	0.083
500 - 1000	0.0228	Next 100 TB	0.07
1000 - 5000	0.0224	Next 350 TB	0.05

8.3 Cloud Operator Service Level Agreements

In all instances, with the information provided by service providers, the nature and content of service level agreement can influence the choice of the provider. A Service Level Agreement (SLA) is used as a formal contract between the service provider and a consumer to ensure service quality [167]. An SLA should specify the details of the service usually in quantifiable terms. The goal of an SLA is therefore to establish a scalable and automatic management framework that can adapt to dynamic and real time environmental changes using multiple qualities of service (QoS) parameters [168]. The SLA for mobile cloud computing should have terms that include multiple domains with heterogeneous resources. In addition, consumers should be involved in the management process of SLA to a certain extent especially regarding reliability and trust/security. In particular, QoS parameters must be updated dynamically over time due to the continuing changes in the mobile application operating environments [169].

Amazon and Microsoft Azure offer a tiered service credit plan that gives users credits based on the discrepancy between SLA specifications and the actual service levels delivered. These providers typically offer cloud storage SLA that articulates precise levels of service such as 99.9% uptime and recourse or compensation to

the user should the provider fail to provide the service as described. Another normal cloud storage SLA detail is service availability, which specifies the maximum amount of time a read request can take, how many retries are allowed and so on.

8.3.1 Microsoft Azure SLA

Azure Active Directory Premium service is available in the following scenarios such as: Users are able to login to the service, login to access applications on the access panel and reset passwords. IT administrators are able to create, read, write and delete entries in the directory or provision or de-provision users to applications in the directory. Windows Azure has separate SLAs for compute and storage. No SLA is provided for free tier of Azure Active Directory. Availability is calculated over a monthly billing cycle [4].

8.3.2 Google SLA

Google has implemented industry standard systems and procedures for cloud SLA to ensure the security and confidentiality of an application and customer data, protect against anticipated threats or hazards to the security or integrity of an application and customer data, and protect against unauthorized access. Customers have the ability to access, monitor, and use or disclose their data submitted by end users through the service. Customer data and applications can only be used to provide the services to customer and its end users and to help secure and improve the services. For instance, this may include identifying and fixing problems in the services, enhancing the services to better protect against attacks and abuse, and making suggestions aimed at improving performance or reducing costs [3].

8.3.3 Amazon EC2 SLA

Amazon AWS SLA policy governs the use of Amazon Elastic Compute Cloud. This SLA applies separately to each account using Amazon EC2 or Amazon EBS. AWS is committed to using commercially reasonable efforts to make Amazon EC2 and Amazon EBS each available with a monthly uptime percentage of at least 99.95%, in each case during any monthly billing cycle. The monthly uptime percentage is less than 99.95% but equal to or greater than 99.0% and service credit of 10% and less than 30% [2].

8.4 Operational Cost of Running Applications in the Cloud

We estimate the cost of running the medical applications with Amazon Web Services, Google Compute Engine and Microsoft Azure by using the simulation results on the cluster node presented in Table 7.6. For each application we use the ECG data interval as 5 minutes such that the cloud will perform computations and transfer the results for every 5 minutes of data acquisition. We choose 5 minute interval based on Γ metric presented in Table 7.6. 5 minute interval is the maximum interval rate where all applications have $\Gamma < 1$, thus all computations can be performed faster than arrival rate of data.

Table 8.8 presents the summary of resource requirements for the medical applications. The Run-time column shows the execution time of the applications with the cluster node we used in Chapter 7. The Duty-Cycle indicates % of time, where CPU is active. Finally storage and data transfer out present the total size required for storing ciphertexts and the data transferred to doctor respectively.

Table 8.8: Performance of medical applications on a cluster node over a period of one month.

Medical Application	Run-time (hours)	Duty Cycle (%)	Storage (GB)	Data Transfer Out (GB)
Average HR	197.7	27	252.9	34.74
LQTS	475.8	65	181.8	11.41
Min. & Max. HR	483.2	66	266.5	36.63

8.4.1 Amazon Web Services (AWS)

To minimize the cost of running our application with AWS, we will select 3-year Heavy Utilization Reserved Instances that yield lowest monthly cost. We look at our application characteristics to select an EC2 instance type. LQTS and Average Heart Rate (HR) computation is less compute-intensive compared to Minimum and Maximum HR. A general-purpose instance of `m3.large` with 2 CPU's is enough to compute both LQTS and Average HR in parallel. On the other hand Minimum and Maximum HR are compute-intensive with 67% duty cycle. Therefore a `c3.large` instance optimized for compute-intensive applications could compute both Minimum and Maximum HR in parallel.

Table 8.9 presents the cost for outsourcing one patient's monitoring to AWS for a month. EC2 costs include both fixed cost fee and run-time fee. Fixed cost fee is calculated by dividing one-time upfront fee into equal monthly payments. Run-time fee is calculated by multiplying the run-time of the applications with hourly rate. Since two applications are executed in parallel we select application run-time that takes longer to finish. S3 costs are the storage costs derived from the application storage requirements presented in Table 7.6. Minimum and maximum HR computation uses same data set therefore their storage amount is counted only once. The transfer cost is based on the data transferred out to doctor's smartphone/tablet from AWS. The overall cost of running applications with AWS services is $108 per month.

Table 8.9: Monthly cost of running medical applications with AWS services.

Medical Application	Instance Type	EC2 Cost	S3 Cost	Transfer Cost	Total Cost
LQTS, Avg. HR	`m3.large`	$32.94	$13.04	$4.36	$50.35
Min. & Max. HR	`c3.large`	$43.15	$7.86	$6.31	$57.32

8.4.2 Google Compute Engine

We select a Google Compute Engine machine type based on the type of medical computation to be performed. LQTS and Average Heart Rate (HR) are less compute-intensive compared to Minimum and Maximum HR computation. We select an `n1-standard-2` machine type with 2 CPUs to compute LQTS and Average HR. For the Minimum and Maximum HR we select `n1-highcpu-2` machine with 2 CPUs. Based on the duty cycle information provided in Table 8.8, we use appropriate sustained discount price provided by Google Compute Engine. The total cost of running these computations are reported in Table 8.10. The overall cost of running applications with GCE services is $114 per month.

Table 8.10: Monthly Cost of Running Medical Applications with Google Compute Engine (GCE)

Medical Application	GCE Instance	GCE Cost	Storage Cost	Transfer Cost	Total Cost
LQTS, Avg. HR	`n1-standard-2`	$47.98	$17.38	$5.54	$70.90
Min. & Max. HR	`n1-highcpu-2`	$23.23	$10.66	$8.79	$42.68

8.4.3 Microsoft Azure

Based on the application characteristics, we choose `D2` instance for LQTS and Average HR computation and `D11` instance for Minimum and Maximum HR computation. Since LQTS and Average HR is less compute-intensive `D2` instance with 2 cores and 7 GB RAM is enough to compute them in parallel. `D11` instance with 2 cores and 14 GB RAM is chosen for more compute-intensive Minimum and Max-

imum HR in parallel. Table 8.11 presents the cost for outsourcing one patient's monitoring for a month with Microsoft Azure services. The overall cost of running applications with Microsoft Azure services is $194 per month.

Table 8.11: Monthly Cost of Running Medical Applications with Microsoft Azure

Medical Application	VM Type	VM Cost	Storage Cost	Transfer Cost	Total Cost
LQTS, Avg. HR	D2	$73.1	$10.4	$3.6	$87.1
Min. & Max. H	D11	$94.4	$6.4	$5.9	$106.7

9 Conclusions and Future Work

9.1 Conclusions

In this dissertation, a long-term health monitoring system is proposed to achieve the end goal of detecting patient health issues by continuously monitoring the ECG data acquired outside the Healthcare Organization (HCO). The proposed system consists of ECG acquisition devices, a cloud-based medical application, and back-end devices that display the monitoring results. Privacy issues related to Personal Health Information (PHI) during its acquisition, storage and processing are addressed by utilizing Fully Homomorphic Encryption (FHE), which permits operations on encrypted data. Despite the well-known intense computational requirements of FHE, an efficient implementation of medical applications is proposed by modeling them based on two computational models: circuit and branching programs. Compared to conventional and Attribute Based Encryption schemes, the FHE-based implementation allows secure computations with associated overheads for storing and sharing medical data. However, these associated overheads are manageable, and a cost analysis of running the proposed FHE-based solution shows that the implementation cost of running the proposed system with public cloud services can be achieved with a reasonable budget.

The contributions of this dissertation are as follows:

- We proposed a cloud-based long-term health monitoring system that enables monitoring of patients based on ECG data. The proposed system utilizes FHE to allow secure computations over the encrypted medical data in a public cloud.
- We implemented our medical applications with circuit-based FHE. We designed circuit-based building blocks that can be used to convert generic algorithms to their FHE counterpart. Methods for reducing multiplication-depth of the applications are proposed to allow their efficient computation.
- We proposed a branching program based solution as an alternative to the circuit-based approach. The branching program reduces the multiplication-depth further compared to the circuit-based approach. Performance improvements up to 20x can be achieved for certain types of applications with parallel processors.
- We compared our circuit-based solution to conventional and Attribute Base Encryption (ABE) schemes. We analyzed the associated overheads related to performance and storage for secure a) storage, b) computation, and c) sharing of the medical data.
- We calculated the running cost of our applications with major public cloud providers. We showed that our proposed system can be implemented with a reasonable cost.

9.2 Future Work

Since its first introduction with Gentry's scheme [14], the performance of the FHE schemes has been the most crucial problem that still prevents FHE schemes from

becoming practical. Recent research has improved the performance of FHE exponentially, therefore in near future FHE is expected to become practical for some real-time applications [170]. While out of the scope of this research, investigating the performance of applying FHE to applications such as face recognition [171] is clearly beneficial. Adaptation of a practical FHE scheme to our proposed system will significantly improve its performance and reduce the running cost in the public cloud. Additional future work related to this research are listed as follows:

- Our proposed medical application is designed for long-term patient monitoring based on ECG signals. Our application could be extended to work with other types of medical data or operations. We've already provided implementation of basic building blocks with circuit based FHE in Section 5.2. These blocks could be used for extending our medical application or proposing new applications that require secure computation.

- We showed that circuit-based FHE puts a strain on network communication when encrypted patient data is transferred to the cloud due to the size of the ciphertexts that are magnitudes larger than the medical data itself. One way to improve the communication is to transmit AES encrypted data to the cloud and then convert it to FHE encrypted version. This process involves evaluating the AES decryption operation homomorphically as presented in [154], where the authors show that the conversion can be done in 6 seconds per AES block. Their implementation uses plaintext space $\mathbb{GF}(2^8)$ for BGV, which is the same as the AES reduction polynomial. However, using $\mathbb{GF}(2^8)$ as the plaintext space in BGV does not permit computing arbitrary functions efficiently. Instead, an AES to FHE conversion can be redesigned to work with plaintext space $\mathbb{GF}(2)$. Then, our medical application can use AES to FHE conversion to reduce network strain. Still, this conversion will increase the computation time and required level L for the

applications, and analyzing the trade-offs between the two approaches could be beneficial.

- We implemented our medical application with circuit based FHE and summarized the results in Table 7.6. LQTS detection and Minimum/Maximum Heart Rate require buffering (i.e., $\Gamma > 1$), if the computation is performed on chunks of medical data with longer than a 5-minute interval. The performance of these operations could benefit from running them in parallel. Once a thread-safe version of HElib is available, it would be beneficial to perform the experiments summarized in Table 7.6 with a multi-processor environment and compare the results.

- We proposed a Branching Program based LQTS Detection in Chapter 6 and provided simulation results for its performance in Section 7.4.2. We did not implement the branching program with HElib, because this method benefits from the availability of multiple-processors to perform homomorphic multiplications in parallel. It would be beneficial to implement the branching program with HElib once a thread-safe version is available and compare it with our simulation results.

- We compared our circuit based FHE results with conventional encryption schemes and Attribute-Based Encryption in terms of secure storage, processing and sharing of medical data. While FHE is the *only* encryption mechanism to perform secure computations, it is inefficient in terms of storage and sharing. The recent efforts of embedding attribute-based properties into FHE [172] could improve our medical application's secure sharing with FHE. Applicability of the methods presented in [172] to the BGV scheme could be investigated and incorporated into our medical application.

- The back-end of our application (i.e., displaying results to a doctor from a mobile device) has not yet been implemented. This requires porting HElib

to work on mobile operating systems such as Android or iOS. Once a mobile-compatible version is implemented, there are exciting research possibilities to combine secure computation with data visualization on mobile devices to improve healthcare.

- FHE has been studied within the context of Medical Cyber Physical Systems (MCPS) [12]. Expanding the application of FHE to other more generalized Cyber Physical Systems (CPS) could allow the construction of autonomous field systems [173–175] that are capable of transmitting low volume data that is FHE-encrypted; in these applications, the goal is to transmit summarized data from a remote monitoring device after FHE-encrypting it, with the intention to provide further elaborate processing in a cloud environment. Visual sensors network based surveillance is one such application.

Bibliography

[1] J. Couderc, J. Xia, X. Xu, S. Kaab, M. Hinteeser, and W. Zareba, "Static and dynamic electrocardiographic patterns preceding torsades de pointes in the acquired and congenital long QT syndrome," in *Computing in Cardiology, 2010*, pp. 357–360, 2010.

[2] "Amazon Web Services." `http://aws.amazon.com`.

[3] "Google Cloud Platform." `https://cloud.google.com/`.

[4] "Microsoft Windows Azure." `http://www.microsoft.com/windowazure`.

[5] "Saleforce." `http://www.salesforce.com`.

[6] O. Kocabas, R. Gyampoh-Vidogah, and T. Soyata, "Operational Cost of Running Real-Time Mobile Cloud Applications," in *Enabling Real-Time Mobile Cloud Computing through Emerging Technologies* (T. Soyata, ed.), ch. 10, pp. 294–321, IGI Global, 2015.

[7] O. Kocabas, T. Soyata, J. Couderc, M. K. Aktas, J. Xia, and M. Huang, "Assessment of Cloud-based Health Monitoring using Homomorphic Encryption," in *Proceedings of the 31st IEEE International Conference on Computer Design (ICCD)*, (Ashville, VA, USA), pp. 443–446, Oct 2013.

[8] A. Page, O. Kocabas, T. Soyata, M. K. Aktas, and J. Couderc, "Cloud-Based Privacy-Preserving Remote ECG Monitoring and Surveillance," *Annals of Noninvasive Electrocardiology (ANEC)*, vol. 20, no. 4, pp. 328–337, 2014.

[9] US Department of Health and Human Services, "Health Insurance Portability and Accountability Act." `http://www.hhs.gov/ocr/privacy/`.

[10] S. Lobodzinski and M. Laks, "New devices for every long-term ECG monitoring," *Cardiology Journal*, vol. 19, no. 2, pp. 210–214, 2012.

[11] "Clearbridge VitalSigns CardioLeaf PRO." `http://www.clearbridgevitalsigns.com/pro.html`, 2013.

[12] O. Kocabas, T. Soyata, and M. K. Aktas, "Emerging Security Mechanisms for Medical Cyber Physical Systems," *IEEE/ACM Transactions on Computational Biology and Bioinformatics (TCBB)*, 2016.

[13] O. Kocabas and T. Soyata, "Utilizing Homomorphic Encryption to Implement Secure and Private Medical Cloud Computing," in *IEEE 8th International Conference on Cloud Computing (CLOUD)*, (New York, NY), pp. 540–547, June 2015.

[14] C. Gentry *et al.*, "Fully homomorphic encryption using ideal lattices.," in *STOC*, vol. 9, pp. 169–178, 2009.

[15] Z. Brakerski, C. Gentry, and V. Vaikuntanathan, "(Leveled) fully homomorphic encryption without bootstrapping," in *ITCS*, pp. 309–325, 2012.

[16] J. W. Bos, K. Lauter, and M. Naehrig, "Private predictive analysis on encrypted medical data," *Journal of biomedical informatics*, 2014.

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

[18] W. Wang, Y. Hu, L. Chen, X. Huang, and B. Sunar, "Exploring the feasibility of fully homomorphic encryption," *IEEE Transactions on Computers*, Aug 2013.

[19] W. Dai, Y. Doroz, and B. Sunar, "Accelerating NTRU based Homomorphic Encryption using GPUs," in *High Performance Extreme Computing Conference*, 2014.

[20] D. A. Barrington, "Bounded-Width Polynomial-Size Branching Programs Recognize Exactly Those Languages in NC^1," *Journal of Computer and System Sciences*, vol. 38, no. 1, pp. 150–164, 1989.

[21] T. Sander, A. L. Young, and M. Yung, "Non-Interactive CryptoComputing For NC^1," in *FOCS*, pp. 554–567, 1999.

[22] Y. Ishai and A. Paskin, "Evaluating branching programs on encrypted data," in *TCC*, pp. 575–594, 2007.

[23] O. Kocabas and T. Soyata, "Towards Privacy-Preserving Medical Cloud Computing Using Homomorphic Encryption," in *Enabling Real-Time Mobile Cloud Computing through Emerging Technologies* (T. Soyata, ed.), ch. 7, pp. 213–246, IGI Global, 2015.

[24] S. Ames, M. Venkitasubramaniam, A. Page, O. Kocabas, and T. Soyata, "Secure Health Monitoring in the Cloud Using Homomorphic Encryption, A Branching-Program Formulation," in *Enabling Real-Time Mobile Cloud Computing through Emerging Technologies* (T. Soyata, ed.), ch. 4, pp. 116–152, IGI Global, 2015.

[25] W. E. Encinosa and J. Bae, "Health information technology and its effects on hospital costs, outcomes, and patient safety," *Inquiry*, vol. 48, no. 4, pp. 288–303, 2011.

[26] U.S. News and Fidelity Investments, "Hospital Executives Survey." `http://health.usnews.com/health-news/best-practices-in-health/articles/2011/07/18/healthsurveytables`.

[27] W. Shanklin, "Impact of electronic medical records discussed." `http://archive.sph.harvard.edu/now/10302009/impact-of-electronic-medical-records.html`.

[28] A. Reichman, "File storage costs less in the cloud than in-house," tech. rep., Forrester, August 2011.

[29] A. Page, T. Soyata, J. Couderc, and M. K. Aktas, "An Open Source ECG Clock Generator for Visualization of Long-Term Cardiac Monitoring Data," *IEEE Access*, vol. 3, pp. 2704–2714, Dec 2015.

[30] A. Page, T. Soyata, J. Couderc, M. Aktas, B. Kantarci, and S. Andreescu, "Visualization of Health Monitoring Data acquired from Distributed Sensors for Multiple Patients," in *IEEE Global Telecommunications Conference (GLOBECOM)*, (San Diego, CA), Dec 2015.

[31] A. Page, M. K. Aktas, T. Soyata, W. Zareba, and J. Couderc, "QT Clock to Improve Detection of QT Prolongation in Long QT Syndrome Patients," *Heart Rhythm*, vol. 13, pp. 190–198, Jan 2016.

[32] Care Cloud. `http://www.carecloud.com/`, 2013.

[33] Dr Chrono. `https://drchrono.com/`, 2013.

[34] C. D. Patel and A. J. Shah, *Cost Model for Planning, Development and Operation of a Data Center*. HP Laboratories Palo Alto, June 2005.

[35] D. Munro, "HIPAA Support Widens In Cloud Vendor Community," *Forbes*, May 2013.

[36] "Amazon Web Services Compliance." `https://aws.amazon.com/compliance/#hipaa`, March 2014.

[37] "HIPAA Compliance with Google Apps." `https://support.google.com/a/answer/3407054?hl=en&ctx=go`, March 2014.

[38] A. Page, M. Hassanalieragh, T. Soyata, M. K. Aktas, B. Kantarci, and S. Andreescu, "Conceptualizing a Real-Time Remote Cardiac Health Monitoring System," in *Enabling Real-Time Mobile Cloud Computing through Emerging Technologies* (T. Soyata, ed.), ch. 1, pp. 1–34, IGI Global, 2015.

[39] M. Hassanalieragh, T. Soyata, A. Nadeau, and G. Sharma, "Solar-Supercapacitor Harvesting System Design for Energy-Aware Applications," in *Proceedings of the 27th IEEE International System-on-Chip Conference (IEEE SOCC)*, (Las Vegas, NV), pp. 280–285, Sep 2014.

[40] T. Soyata, L. Copeland, and W. Heinzelman, "RF Energy Harvesting for Embedded Systems: A Survey of Tradeoffs and Methodology," *IEEE Circuits and Systems Magazine*, vol. 16, pp. 22–57, Feb 2016.

[41] M. Hassanalieragh, A. Page, T. Soyata, G. Sharma, M. K. Aktas, G. Mateos, B. Kantarci, and S. Andreescu, "Health Monitoring and Management Using Internet-of-Things (IoT) Sensing with Cloud-based Processing: Opportunities and Challenges," in *2015 IEEE International Conference on Services Computing (SCC)*, (New York, NY), pp. 285–292, June 2015.

[42] J. Couderc, "The telemetric and holter ECG warehouse initiative (THEW): A data repository for the design, implementation and validation of ECG-related technologies," in *EMBC*, pp. 6252–6255, IEEE, 2010.

[43] V. Shusterman, A. Goldberg, D. M. Schindler, K. E. Fleischmann, R. L. Lux, and B. J. Drew, "Dynamic tracking of ischemia in the surface electrocardiogram," *Journal of electrocardiology*, vol. 40, no. 6, pp. S179–S186, 2007.

[44] R. L. Woosley, "Drugs That Prolong the QT Interval and/or Induce Torsades de Pointes," tech. rep., Torsades.org, October 2001.

[45] R. R. Shah, "Drug-Induced QT Interval ProlongationRegulatory Guidance and Perspectives on hERG Channel Studies, in The hERG Cardiac Potassium Channel," in *The hERG Cardiac Potassium Channel: Structure, Function and Long QT Syndrome, Novartis Foundation Symposium*, p. 251, 2005.

[46] M. Fink, D. Noble, L. Virag, A. Varro, and W. R. Giles, "Contributions of herg k+ current to repolarization of the human ventricular action potential.," *Prog Biophys Mol Biol*, vol. 96, pp. 357–376, 2008.

[47] R. Shah, "Drug-induced QT interval prolongation: regulatory perspectives and drug development," *Annals of medicine*, vol. 36, no. S1, pp. 47–52, 2004.

[48] S. M. Straus, J. A. Kors, M. L. De Bruin, C. S. van der Hooft, A. Hofman, J. Heeringa, J. W. Deckers, J. H. Kingma, M. C. Sturkenboom, B. H. C. Stricker, *et al.*, "Prolonged QTc interval and risk of sudden cardiac death in a population of older adults," *Journal of the American College of Cardiology*, vol. 47, no. 2, pp. 362–367, 2006.

[49] S. Fanoe, D. Kristensen, A. Fink-Jensen, H. K. Jensen, E. Toft, J. Nielsen, P. Videbech, S. Pehrson, and H. Bundgaard, "Risk of arrhythmia induced by psychotropic medications: a proposal for clinical management," *European Heart Journal*, 2014.

[50] J.-P. Couderc, C. Garnett, M. Li, R. Handzel, S. McNitt, X. Xia, S. Polonsky, and W. Zareba, "Highly Automated QT measurement techniques in 7

thorough QT studies implemented under ICH E14 guidelines," *Annals of Noninvasive Electrocardiology*, vol. 16, no. 1, pp. 13–24, 2011.

[51] L. Fridericia, "Die systolendauer im elektrokardiogramm bei normalen menschen und bei herzkranken," *Acta Medica Scandinavica*, vol. 54, no. 1, pp. 17–50, 1921.

[52] National Institute of Standards and Technology, "Advanced encryption standard (AES)," Nov. 2001. FIPS-197.

[53] SORIN GROUP, "SMARTVIEW Remote Monitoring System." `http://www.sorin.com/product/smartview-remote-monitoring-system`, 2013.

[54] Alivecor, "ECG screening made easy." `http://www.alivecor.com/`, 2013.

[55] A. Fahad, T. Soyata, T. Wang, G. Sharma, W. Heinzelman, and K. Shen, "SOLARCAP: Super Capacitor Buffering of Solar Energy for Self-Sustainable Field Systems," in *Proceedings of the 25th IEEE International System-on-Chip Conference (SOCC)*, (Niagara Falls, NY), pp. 236–241, Sep 2012.

[56] T. Soyata, R. Muraleedharan, C. Funai, M. Kwon, and W. Heinzelman, "Cloud-Vision: Real-Time Face Recognition Using a Mobile-Cloudlet-Cloud Acceleration Architecture," in *Proceedings of the 17th IEEE Symposium on Computers and Communications (ISCC)*, (Cappadocia, Turkey), pp. 59–66, Jul 2012.

[57] T. Soyata, R. Muraleedharan, S. Ames, J. H. Langdon, C. Funai, M. Kwon, and W. B. Heinzelman, "COMBAT: mobile Cloud-based cOmpute/coMmunications infrastructure for BATtlefield applications," in *Proceedings of SPIE*, vol. 8403, pp. 84030K–84030K, May 2012.

[58] T. Soyata, H. Ba, W. Heinzelman, M. Kwon, and J. Shi, "Accelerating Mobile Cloud Computing: A Survey," in *Communication Infrastructures for Cloud Computing* (H. T. Mouftah and B. Kantarci, eds.), ch. 8, pp. 175–197, IGI Global, Sep 2013.

[59] B. Schneier, "Homomorphic Encryption Breakthrough." `https://www.schneier.com/blog/archives/2009/07/homomorphic_enc.html`, July 2009.

[60] R. L. Rivest, L. Adleman, and M. L. Dertouzos, "On data banks and privacy homomorphisms," *Foundations of secure computation*, vol. 4, no. 11, pp. 169–180, 1978.

[61] A. Sahai and B. Waters, "Fuzzy identity-based encryption," in *Advances in Cryptology–EUROCRYPT 2005*, pp. 457–473, Springer, 2005.

[62] V. Goyal, O. Pandey, A. Sahai, and B. Waters, "Attribute-based encryption for fine-grained access control of encrypted data," in *Proceedings of the 13th ACM conference on Computer and communications security*, pp. 89–98, 2006.

[63] J. Bethencourt, A. Sahai, and B. Waters, "Ciphertext-policy attribute-based encryption," in *IEEE Symposium on Security and Privacy, 2007. SP'07*, pp. 321–334, 2007.

[64] X. Yao, Z. Chen, and Y. Tian, "A lightweight attribute-based encryption scheme for the internet of things," *Future Generation Computer Systems*, vol. 49, pp. 104–112, 2015.

[65] N. Powers, A. Alling, R. Gyampoh-Vidogah, and T. Soyata, "AXaaS: Case for Acceleration as a Service," in *Globecom Workshops (GC Wkshps)*, (Austin, TX), pp. 117–121, Dec 2014.

[66] N. Powers and T. Soyata, "AXaaS (Acceleration as a Service): Can the Telecom Service Provider Rent a Cloudlet ?," in *Proceedings of the 4th IEEE International Conference on Cloud Networking (CNET)*, (Niagara Falls, Canada), pp. 232–238, Oct 2015.

[67] N. Powers and T. Soyata, "Selling FLOPs: Telecom Service Providers Can Rent a Cloudlet via Acceleration as a Service (AXaaS)," in *Enabling Real-Time Mobile Cloud Computing through Emerging Technologies* (T. Soyata, ed.), ch. 6, pp. 182–212, IGI Global, 2015.

[68] N. Powers, A. Alling, K. Osolinsky, T. Soyata, M. Zhu, H. Wang, H. Ba, W. Heinzelman, J. Shi, and M. Kwon, "The Cloudlet Accelerator: Bringing Mobile-Cloud Face Recognition into Real-Time," in *Globecom Workshops (GC Wkshps)*, (San Diego, CA), Dec 2015.

[69] M. Pouryazdan, B. Kantarci, T. Soyata, and H. Song, "Anchor-Assisted and Vote-based Trustworthiness Assurance in Smart City Crowdsensing," *IEEE Access*, vol. 4, pp. 529–541, Mar 2016.

[70] W. Diffie and M. Hellman, "New directions in cryptography," *IEEE Trans. Inf. Theor.*, vol. 22, no. 6, pp. 644–654, 2006.

[71] C. Poon, Y. Zhang, and S. Bao, "A novel biometrics method to secure wireless body area sensor networks for telemedicine and m-health," *IEEE Communications Magazine*, vol. 44, no. 4, pp. 73–81, 2006.

[72] K. K. Venkatasubramanian, A. Banerjee, and S. Gupta, "PSKA: usable and secure key agreement scheme for body area networks," *IEEE Transactions on Information Technology in Biomedicine*, vol. 14, no. 1, pp. 60–68, 2010.

[73] D. F. Ferraiolo and D. R. Kuhn, "Role-based access controls," *arXiv preprint arXiv:0903.2171*, 2009.

[74] M. Li, W. Lou, and K. Ren, "Data security and privacy in wireless body area networks," *IEEE Wireless Communications*, vol. 17, no. 1, pp. 51–58, 2010.

[75] O. Goldreich, *Foundations of cryptography: volume 2, basic applications*. Cambridge university press, 2004.

[76] N. Cao, C. Wang, M. Li, K. Ren, and W. Lou, "Privacy-preserving multi-keyword ranked search over encrypted cloud data," *Parallel and Distributed Systems, IEEE Transactions on*, vol. 25, no. 1, pp. 222–233, 2014.

[77] D. Boneh, C. Gentry, S. Halevi, F. Wang, and D. J. Wu, "Private database queries using somewhat homomorphic encryption," in *Applied Cryptography and Network Security*, pp. 102–118, Springer, 2013.

[78] V. Nikolaenko, U. Weinsberg, S. Ioannidis, M. Joye, D. Boneh, and N. Taft, "Privacy-preserving ridge regression on hundreds of millions of records," in *Security and Privacy (SP), 2013 IEEE Symposium on*, pp. 334–348, IEEE, 2013.

[79] R. Gennaro, C. Gentry, and B. Parno, "Non-interactive verifiable computing: Outsourcing computation to untrusted workers," in *Advances in Cryptology–CRYPTO 2010*, pp. 465–482, Springer, 2010.

[80] D. Boneh and D. M. Freeman, "Homomorphic signatures for polynomial functions," in *Advances in Cryptology–EUROCRYPT 2011*, pp. 149–168, Springer, 2011.

[81] S. Dziembowski and K. Pietrzak, "Leakage-resilient cryptography," in *Foundations of Computer Science, 2008. FOCS'08. IEEE 49th Annual IEEE Symposium on*, pp. 293–302, IEEE, 2008.

[82] Y. Zhou and D. Feng, "Side-channel attacks: Ten years after its publication and the impacts on cryptographic module security testing.," *IACR Cryptology ePrint Archive*, vol. 2005, p. 388, 2005.

[83] P. C. Kocher, "Timing attacks on implementations of Diffie-Hellman, RSA, DSS, and other systems," in *Advances in Cryptology—CRYPTO'96*, pp. 104–113, Springer, 1996.

[84] P. L. Montgomery, "Speeding the pollard and elliptic curve methods of factorization," *Mathematics of computation*, vol. 48, no. 177, pp. 243–264, 1987.

[85] J. López and R. Dahab, "Fast multiplication on elliptic curves over GF(2^m) without precomputation," in *Cryptographic Hardware and Embedded Systems*, pp. 316–327, Springer, 1999.

[86] P. Kocher, J. Jaffe, and B. Jun, "Differential power analysis," in *Advances in Cryptology—CRYPTO'99*, pp. 388–397, Springer, 1999.

[87] T. S. Messerges, "Securing the AES finalists against power analysis attacks," in *Fast Software Encryption*, pp. 150–164, Springer, 2001.

[88] J.-S. Coron, "Resistance against differential power analysis for elliptic curve cryptosystems," in *Cryptographic Hardware and Embedded Systems*, pp. 292–302, Springer, 1999.

[89] D. Boneh, R. A. DeMillo, and R. J. Lipton, "On the importance of checking cryptographic protocols for faults," in *Advances in Cryptology—EUROCRYPT'97*, pp. 37–51, Springer, 1997.

[90] R. Karri, K. Wu, P. Mishra, and Y. Kim, "Fault-based side-channel cryptanalysis tolerant Rijndael symmetric block cipher architecture," in *Defect*

and Fault Tolerance in VLSI Systems, 2001. Proceedings. 2001 IEEE International Symposium on, pp. 427–435, IEEE, 2001.

[91] G. Bertoni, L. Breveglieri, I. Koren, P. Maistri, and V. Piuri, "Error analysis and detection procedures for a hardware implementation of the advanced encryption standard," *Computers, IEEE Transactions on*, vol. 52, no. 4, pp. 492–505, 2003.

[92] I. Biehl, B. Meyer, and V. Müller, "Differential fault attacks on elliptic curve cryptosystems," in *Advances in Cryptology—CRYPTO 2000*, pp. 131–146, Springer, 2000.

[93] D. J. Bernstein, "Cache-timing attacks on AES," 2005.

[94] D. A. Osvik, A. Shamir, and E. Tromer, "Cache attacks and countermeasures: the case of AES," in *Topics in Cryptology–CT-RSA 2006*, pp. 1–20, Springer, 2006.

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

[96] B. B. Brumley and R. M. Hakala, "Cache-timing template attacks," in *Advances in Cryptology–ASIACRYPT 2009*, pp. 667–684, Springer, 2009.

[97] D. McGrew and J. Viega, "The Galois/counter mode of operation (GCM)," *Submission to NIST. http://csrc. nist. gov/CryptoToolkit/modes/proposedmodes/gcm/gcm-spec.pdf*, 2004.

[98] A. A. Group, "Armv8 instruction set overview," 2011.

[99] S. Morioka and A. Satoh, "An optimized S-Box circuit architecture for low power AES design," in *Cryptographic Hardware and Embedded Systems-CHES 2002*, pp. 172–186, Springer, 2003.

[100] D. Canright, *A very compact S-box for AES*. Springer, 2005.

[101] Y. Nogami, K. Nekado, T. Toyota, N. Hongo, and Y. Morikawa, "Mixed bases for efficient inversion in $F((2^2)^2)^2$ and conversion matrices of subbytes of AES," in *Cryptographic Hardware and Embedded Systems, CHES 2010*, pp. 234–247, Springer, 2010.

[102] X. Zhang and K. K. Parhi, "On the optimum constructions of composite field for the AES algorithm," *Circuits and Systems II: Express Briefs, IEEE Transactions on*, vol. 53, no. 10, pp. 1153–1157, 2006.

[103] A. Satoh, T. Sugawara, and T. Aoki, "High-performance hardware architectures for galois counter mode," *Computers, IEEE Transactions on*, vol. 58, no. 7, pp. 917–930, 2009.

[104] S. Gueron and M. E. Kounavis, "Intel® carry-less multiplication instruction and its usage for computing the gcm mode," *White Paper*, 2010.

[105] N. Koblitz, "Elliptic curve cryptosystems," *Mathematics of computation*, vol. 48, no. 177, pp. 203–209, 1987.

[106] V. Miller, "Use of elliptic curves in cryptography," in *Advances in Cryptology—CRYPTO'85 Proceedings*, pp. 417–426, 1986.

[107] J. M. Pollard, "Monte carlo methods for index computation," *Mathematics of computation*, vol. 32, no. 143, pp. 918–924, 1978.

[108] T. El Gamal, "A public key cryptosystem and a signature scheme based on discrete logarithms," in *Advances in cryptology*, pp. 10–18, 1985.

[109] D. Boneh and M. Franklin, "Identity-based encryption from the weil pairing," in *Advances in Cryptology—CRYPTO 2001*, pp. 213–229, 2001.

[110] M. Abdalla, M. Bellare, and P. Rogaway, "DHAES: An Encryption Scheme Based on the Diffie-Hellman Problem," *IACR Cryptology ePrint Archive*, vol. 1999, p. 7, 1999.

[111] D. Hankerson, A. J. Menezes, and S. Vanstone, *Guide to elliptic curve cryptography*. Springer Science & Business Media, 2006.

[112] V. G. Martínez, E. L. Hernández, A. C. Sánchez, *et al.*, "A survey of the elliptic curve integrated encryption scheme," *ratio*, vol. 80, no. 1024, pp. 160–223, 2010.

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

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

[115] J. D. Cohen and M. J. Fischer, "A robust and verifiable cryptographically secure election scheme," in *Proceedings of the 26th Annual Symposium on Foundations of Computer Science*, pp. 372–382, 1985.

[116] P. Paillier, "Public-key cryptosystems based on composite degree residuosity classes," in *Advances in cryptology – EUROCRYPT*, pp. 223–238, 1999.

[117] I. Damgård and M. Jurik, "A Generalisation, a Simplification and Some Applications of Paillier's Probabilistic Public-Key System," in *Public Key Cryptography*, pp. 119–136, 2001.

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

[119] M. van Dijk, C. Gentry, S. Halevi, and V. Vaikuntanathan, "Fully homomorphic encryption over the integers," in *EUROCRYPT*, pp. 24–43, 2010.

[120] Z. Brakerski and V. Vaikuntanathan, "Fully Homomorphic Encryption from Ring-LWE and Security for Key Dependent Messages," in *CRYPTO*, vol. 6841, p. 501, 2011.

[121] Z. Brakerski and V. Vaikuntanathan, "Efficient fully homomorphic encryption from (standard) LWE," in *FOCS*, pp. 97–106, 2011.

[122] J.-S. Coron, A. Mandal, D. Naccache, and M. Tibouchi, "Fully Homomorphic Encryption over the Integers with Shorter Public Keys," in *CRYPTO*, pp. 487–504, 2011.

[123] C. Gentry and S. Halevi, "Fully homomorphic encryption without squashing using depth-3 arithmetic circuits," in *Foundations of Computer Science (FOCS), 2011 IEEE 52nd Annual Symposium on*, pp. 107–109, IEEE, 2011.

[124] N. P. Smart and F. Vercauteren, "Fully homomorphic encryption with relatively small key and ciphertext sizes," in *PKC*, pp. 420–443, 2010.

[125] D. Stehlé and R. Steinfeld, "Faster fully homomorphic encryption," in *ASIACRYPT*, pp. 377–394, 2010.

[126] C. Gentry, S. Halevi, and N. Smart, "Fully homomorphic encryption with polylog overhead," in *EUROCRYPT*, pp. 465–482, 2012.

[127] C. Gentry, S. Halevi, and N. P. Smart, "Better Bootstrapping in Fully Homomorphic Encryption," in *PKC*, pp. 1–16, 2012.

[128] S. Halevi and V. Shoup. `https://github.com/shaih/HElib`.

[129] A. Page, O. Kocabas, S. Ames, M. Venkitasubramaniam, and T. Soyata, "Cloud-based Secure Health Monitoring: Optimizing Fully-Homomorphic

Encryption for Streaming Algorithms," in *Globecom Workshops (GC Wkshps)*, (Austin, TX), pp. 48–52, Dec 2014.

[130] N. P. Smart and F. Vercauteren, "Fully homomorphic SIMD operations," *Designs, codes and cryptography*, vol. 71, no. 1, pp. 57–81, 2014.

[131] O. Goldreich, S. Goldwasser, and S. Halevi, "Public-key cryptosystems from lattice reduction problems," in *Advances in Cryptology-CRYPTO*, pp. 112–131, 1997.

[132] L. Babai, "On Lovász lattice reduction and the nearest lattice point problem," *Combinatorica*, vol. 6, no. 1, pp. 1–13, 1986.

[133] D. Micciancio, "Improving lattice based cryptosystems using the hermite normal form," in *Cryptography and Lattices*, pp. 126–145, Springer, 2001.

[134] C. Gentry and S. Halevi, "Implementing Gentry's fully-homomorphic encryption scheme," in *Advances in Cryptology–EUROCRYPT*, pp. 129–148, 2011.

[135] Z. Brakerski, "Fully homomorphic encryption without modulus switching from classical GapSVP," in *Advances in Cryptology–CRYPTO 2012*, pp. 868–886, Springer, 2012.

[136] A. López-Alt, E. Tromer, and V. Vaikuntanathan, "On-the-fly multiparty computation on the cloud via multikey fully homomorphic encryption," in *Proceedings of the forty-fourth annual ACM symposium on Theory of computing*, pp. 1219–1234, 2012.

[137] J. Hoffstein, J. Pipher, and J. H. Silverman, "NTRU: A ring-based public key cryptosystem," in *Algorithmic number theory*, pp. 267–288, Springer, 1998.

[138] V. Lyubashevsky, C. Peikert, and O. Regev, "On ideal lattices and learning with errors over rings," in *EUROCRYPT*, pp. 1–23, 2010.

[139] O. Goldreich, *Computational complexity - a conceptual perspective*. Cambridge University Press, 2008.

[140] J. E. Savage, *Models of Computation: Exploring the Power of Computing*. Addison Wesley, 1st ed., 1997.

[141] P. M. Kogge and H. S. Stone, "A parallel algorithm for the efficient solution of a general class of recurrence equations," *IEEE Trans. Comput.*, vol. 22, no. 8, pp. 786–793, 1973.

[142] NCSU, "Free PDK 45nm Standard Cell Library." `http://www.eda.ncsu.edu/wiki/FreePDK45:Contents`, 2014.

[143] C. S. Wallace, "A suggestion for a fast multiplier," *Electronic Computers, IEEE Transactions on*, no. 1, pp. 14–17, 1964.

[144] L. Dadda, "Some schemes for parallel multipliers," *Alta frequenza*, vol. 34, no. 5, pp. 349–356, 1965.

[145] T. Soyata and J. Liobe, "pbCAM: probabilistically-banked Content Addressable Memory," in *Proceedings of the 25th IEEE International System-on-Chip Conference (SOCC)*, (Niagara Falls, NY), pp. 27–32, Sep 2012.

[146] T. Soyata, "Probabilistically-banked content addressable memory and storage," 2013.

[147] X. Guo, E. Ipek, and T. Soyata, "Resistive Computation: Avoiding the Power Wall with Low-Leakage, STT-MRAM Based Computing," in *Proceedings of the International Symposium on Computer Architecture (ISCA)*, vol. 38, (Saint-Malo, France), pp. 371–382, Jun 2010.

[148] T. Soyata and E. G. Friedman, "Incorporating Interconnect, Register and Clock Distribution Delays into the Retiming Process," *IEEE Transactions on Computer Aided Design of Circuits and Systems (TCAD)*, vol. CAD-16, pp. 105–120, Jan 1997.

[149] T. Soyata and E. G. Friedman, "Retiming with Non-Zero Clock Skew, Variable Register and Interconnect Delay," in *Proceedings of the IEEE Conference on Computer-Aided Design (ICCAD)*, pp. 234–241, Nov 1994.

[150] T. Soyata, E. G. Friedman, and J. H. Mulligan, "Integration of Clock Skew and Register Delays into a Retiming Algorithm," in *Proceedings of the International Symposium on Circuits and Systems (ISCAS)*, pp. 1483–1486, May 1993.

[151] Y. Ishai and E. Kushilevitz, "Private simultaneous messages protocols with applications," in *Theory of Computing and Systems*, pp. 174–183, 1997.

[152] J. A. Akinyele, C. Garman, I. Miers, M. W. Pagano, M. Rushanan, M. Green, and A. D. Rubin, "Charm: a framework for rapidly prototyping cryptosystems," *Journal of Cryptographic Engineering*, vol. 3, no. 2, pp. 111–128, 2013.

[153] E. Barker and A. Roginsky, "Transitions: Recommendation for transitioning the use of cryptographic algorithms and key lengths," *NIST Special Publication*, vol. 800, p. 131A, 2011.

[154] C. Gentry, S. Halevi, and N. P. Smart, "Homomorphic evaluation of the AES circuit," in *CRYPTO*, pp. 850–867, 2012.

[155] F. Badilini, "The ISHNE Holter Standard Output File Format," *Annals of Noninvasive Electrocardiology*, vol. 3, pp. 263–266, October 2006.

[156] M. M., "The new age of medical monitoring," *Scientific American*, vol. 308, pp. 33–34, 2013.

[157] O. Kocabas and T. Soyata, "Medical Data Analytics in the cloud using Homomorphic Encryption," in *Handbook of Research on Cloud Infrastructures for Big Data Analytics* (P. R. Chelliah and G. Deka, eds.), ch. 19, pp. 471–488, IGI Global, Mar 2014.

[158] N. Bansal, "Cloud computing technology (with bpos and windows azure).," *IJCC*, vol. 2, no. 1, pp. 48–60, 2013.

[159] W. Shanklin, "Revisiting cloud computing: how has it changed and changed us?." `http://www.gizmag.com/revisiting-cloud-computing/26768/`.

[160] M. Armbrust, A. Fox, R. Griffith, A. D. Joseph, R. Katz, A. Konwinski, G. Lee, D. Patterson, A. Rabkin, I. Stoica, and M. Zaharia, "A view of cloud computing," *Commun. ACM*, vol. 53, no. 4, pp. 50–58, 2010.

[161] D. Kovachev, Y. Cao, and R. Klamma, "Mobile cloud computing: A comparison of application models," *CoRR*, vol. abs/1107.4940, 2011.

[162] B. P. Rimal and E. Choi, "A service-oriented taxonomical spectrum, cloudy challenges and opportunities of cloud computing," *International Journal of Communication Systems*, vol. 25, no. 6, pp. 796–819, 2012.

[163] H. T. Mouftah and B. Kantarci, *Communication Infrastructures for Cloud Computing*. IGI Global, 2013.

[164] D. Fesehaye, Y. Gao, K. Nahrstedt, and G. Wang, "Impact of cloudlets on interactive mobile cloud applications," in *Enterprise Distributed Object Computing Conference (EDOC), 2012 IEEE 16th International*, pp. 123–132, 2012.

[165] C. Chappell, "Unlocking Network Value: Service Innovation in the Era of SDN," *HeavyReading, white paper*, 2013.

[166] R. Harms and M. Yamartino, "The economics of the cloud," *Microsoft whitepaper, Microsoft Corporation*, 2010.

[167] L. Wu, *SLA-based resource provisioning for management of Cloud-based Software-as-a-Service applications*. PhD thesis, Citeseer, 2014.

[168] P. Glasser, O. Kocabas, B. Kantarci, T. Soyata, and J. Matthews, "Energy efficient VM migration revisited: SLA assurance and minimum service disruption with available hosts," in *12th International Conference on High-capacity Optical Networks and Emerging/Enabling Technologies (HONET)*, pp. 22–27, Dec 2015.

[169] M. Kwon, Z. Dou, W. Heinzelman, T. Soyata, H. Ba, and J. Shi, "Use of Network Latency Profiling and Redundancy for Cloud Server Selection," in *Proceedings of the 7th IEEE International Conference on Cloud Computing (CLOUD)*, (Anchorage, AK), pp. 826–832, Jun 2014.

[170] S. Halevi, "Homomorphic encryption tutorial." `http://www.iacr.org/conferences/crypto2011/slides/Halevi.pdf`, 2011. CRYPTO 2011 Conference Presentation.

[171] A. Alling, N. Powers, and T. Soyata, "Face Recognition: A Tutorial on Computational Aspects," in *Emerging Research Surrounding Power Consumption and Performance Issues in Utility Computing*, ch. 20, pp. 405–425, IGI Global, 2016.

[172] C. Gentry, A. Sahai, and B. Waters, "Homomorphic encryption from learning with errors: Conceptually-simpler, asymptotically-faster, attribute-based," in *Advances in Cryptology–CRYPTO 2013*, pp. 75–92, Springer, 2013.

[173] M. Hassanalieragh, T. Soyata, A. Nadeau, and G. Sharma, "UR-SolarCap: An Open Source Intelligent Auto-Wakeup Solar Energy Harvesting System for Supercapacitor Based Energy Buffering," *IEEE Access*, vol. 4, pp. 542–557, Mar 2016.

[174] G. Honan, N. Gekakis, M. Hassanalieragh, A. Nadeau, G. Sharma, and T. Soyata, "Energy Harvesting and Buffering for Cyber Physical Systems: A Review," in *Cyber Physical Systems - A Computational Perspective*, ch. 7, pp. 191–218, CRC, Dec 2015.

[175] T. Soyata, *Enabling Real-Time Mobile Cloud Computing through Emerging Technologies*. IGI Global, Aug 2015.