Secure Health Monitoring in the Cloud Using Homomorphic Encryption, A Branching-Program Formulation

Scott Ames
Muthuramakrishnan Venkitasubramaniam
Dept. of Computer Science, University of Rochester

Alex Page
Övünç Kocabaş
Tolga Soyata
Dept. of Electrical and Computer Engineering, University of Rochester

ABSTRACT

Extending cloud computing to medical software, where the hospitals rent the software from the provider sounds like a natural evolution for cloud computing. One problem with cloud computing, though, is ensuring the medical data privacy in applications such as long term health monitoring. Previously proposed solutions based on Fully Homomorphic Encryption (FHE) completely eliminate privacy concerns, but are extremely slow to be practical. Our key proposition in this paper is a new approach to applying FHE into the data that is stored in the cloud. Instead of using the existing circuit-based programming models, we propose a solution based on Branching Programs. While this restricts the type of data elements that FHE can be applied to, it achieves dramatic speed-up as compared to traditional circuit-based methods. Our claims are proven with simulations applied to real ECG data.

Keywords: Homomorphic Encryption, Medical Cloud Computing, e-Health, Tele-medicine.

INTRODUCTION

Software as a Service (SaaS) provides an excellent alternative to any corporation looking to simplify their IT infrastructure. By renting Software as a Service (SaaS), rather than purchasing, the responsibility of software upgrades, as well as the infrastructure to run the software are transferred to the provider of the software. Upgrades on the software could be done instantly, since new patches and code improvements could be contained at the source, which resides within the servers of the provider of the software. While SaaS has been very successful in certain categories of applications, such as Salesforce.com (SalesForce.com, 2014), its adoption in the medical application arena has been very slow due to the strict rules and regulations introduced by Health Insurance Portability and Accountability Act - HIPAA (HIPAA, 2014). According to HIPAA regulations, private medical information should be treated with utmost care, and the penalties associated with the breach of HIPAA are steep and unacceptable. Despite the fact that a hospital can confidently switch its application hosting and file storage to cloud operators, save money, and simplify its IT infrastructure (Reichman, 2011; Good, 2013), this transition has been very slow.

A novel application introduced in (Kocabas, et al., 2013; Kocabas & Soyata, 2014)guarantees privacy of patient medical information during cloud computing. This technique owes its capability to using Fully Homomorphic Encryption (FHE) during its computations. FHE allows generalized operations on encrypted data (Gentry, 2009), without actually observing the underlying medical data, thereby completely eliminating privacy concerns due to processing sensitive medical information. While novel in theory, this technique is plagued by performance bottlenecks: FHE-based computations are orders of magnitude slower than their unencrypted counterparts, which confines the application space of FHE-based implementations to a very restricted set. Additionally, FHE-encrypted data takes up orders of magnitude larger storage space (Page, Kocabas, Soyata, Aktas, & Couderc, 2014). With this significant expansion in storage space, and extremely prolonged execution time, the cost-saving advantage of cloud outsourcing becomes questionable for FHE-based implementations.

This performance disadvantage of FHE motivated the launch of the large-scale DARPA PROCEED program (DARPA-PROCEED) to improve FHE performance. While the privacy advantages of FHE-based implementations are clear, substantial work has to be done before FHE can be practical. In this chapter, a reformulation of the idea introduced in (Kocabas, et al., 2013) is discussed, where FHE is not applied to the problem in a generalized way. Instead, a meaningful trade-off is presented between performance and range of input data. It is shown through simulations that, when a medical application is performing operations on data elements that lie within a well-defined range (e.g., 0.4 and 0.6 in the case of the QT_c value extracted from an ECG as will be described shortly in this chapter), comparisons can be made drastically faster. While most of the existing FHE implementations treat the arithmetic operations within a computer application as a set of operations that can be represented as a *circuit*, the formulation in (Page, Kocabas, Ames, Venkitasubramaniam, & Soyata, 2014) takes a radically different approach and is described in detail in this chapter.

In (Page, Kocabas, Ames, Venkitasubramaniam, & Soyata, 2014), a study is provided on a set of arithmetic (and logical) operations required for the execution of a medical application. These operations primarily consist of integer comparisons to determine the health state of a patient. These comparisons are performed on the vitals of a patient, such as the heart rate, or certain other metrics extracted from an Electrocardiogram (ECG). Rather than using the usual circuit-based representation of the operations, a *branching program* approach is taken, where each comparison is represented as a set of decisions applied to the bits of the compared values. Allowing the medical application to be represented as a branching program opens the door to borrowing from a rich body of research that exists for this computational model (Barrington, 1989; Sander, Young, & Yung, 1999; Ishai & Paskin, 2007). While the branching program approach restricts the applicability of FHE due to the limited values that the input data can have, the performance advantage of this approach which will be demonstrated in the Evaluation section far outweighs this disadvantage. Especially for medical applications that will be described in the next section, since the values are indeed in a restricted range, the disadvantages that the Branching Program formulation introduces can be mostly eliminated by a careful selection of the branching program that is used to replace the equivalent circuit.

This rest of this chapter is organized as follows: In the next section, background information is provided on FHE and medical applications of interest, followed by a specific case study medical application. For this case study, a detailed description of the medical condition that is being detected by the application is provided along with a functional infrastructure. Mapping of this functional infrastructure to an FHE-based implementation is the key contribution of this chapter, which is based on the Branching Program. A theoretical background is provided for this implementation, followed by an evaluation based on a simulated program and ECG data. Conclusions and future research directions are provided after this evaluation.

MOTIVATING APPLICATION

Due to the complexity of Fully Homomorphic Encryption (FHE), attempting to formulate a generalized framework for running a wide variety of cloud-based medical applications is not realistic at this point. While a mainstream adoption for FHE might take years or decades, the goal of this chapter is to investigate a set of applications that can be executed in a privacy-preserving setting through the use of open source FHE libraries such as HELib (HElib, 2014), based on the Brakerski-Gentry-Vaikuntanathan (BGV) encryption scheme (Brakerski, Gentry, & Vaikuntanathan, 2012). Target applications that can be adapted to HElib possess similar characteristics and we will identify them in this section. To determine what type of applications can be formulated to run on FHE-encrypted data, we first need to understand the limitations of FHE, which manifests itself on multiple fronts:

1. Each arithmetic operation (e.g., addition) that would normally take one cycle on regular numbers (e.g., integers) takes hundreds of thousands, and in some cases millions of cycles,
2. Representing each bit in FHE-encrypted format occupies hundreds of thousands, and in some cases millions of times larger storage area,
3. Usual data formats, such as integer and floating point types are not necessarily native to the FHE-style formulation, thereby making the application of any function non-trivial,
4. Due to the improvements made in the state-of-the-art BGV scheme (Brakerski, Gentry, & Vaikuntanathan, 2012), operations are performed SIMD-like, i.e., Single Instruction Multiple Data. This requires re-thinking how data elements should be represented/packed for native application to these SIMD operations.

While a broad set of applications might be suitable to work in an environment with such constraints, we specifically focus on long term patient health monitoring applications in this chapter. **Figure** 1 depicts the conceptualized long term cardiac health monitoring for patients outside the healthcare organization. In this application scenario, patients are given a sensor that is capable of acquiring and transmitting ECG signals, which is called an ECG patch (CardioLeaf-Pro). Since energy consumption is a top priority for the longevity of the device, a powerful processor cannot be incorporated into such an ECG patch. An example microcontroller that is suitable for such a device is a 16-bit Texas Instruments MSP430 (TI-MSP430). MSP430 consumes only 600μW during operation and including the peripheral acquisition circuitry, an ECG patch that is architected around MSP430 could be expected to consume around 1mW during continuous operation. Additionally, communicating the acquired samples over a WiFi or Zigbee link can consume an average of 1mW, resulting in a total of 2mW power drain. A typical coin battery has 675 mWh energy stored in it (CR2032), and can sustain this patch for almost two weeks, which is sufficient for long term cardiac monitoring. This patch has the capability to perform Digital Signal Processing (DSP) operations which will allow it to compute preliminary metrics on the acquired ECG data, such as the QT and RR values, as we will describe later in this chapter.

While pre-calculating the QT and RR values from the ECG signals might make sense for the patch, it doesn't make sense when the *bigger picture* is concerned, which involves privacy-preserving transfer of these values into the cloud. To ensure privacy of the acquired ECG samples, this chapter proposes to apply FHE into these samples. To run the long term health monitoring application in an FHE-based environment, the first step, *encryption*, must be performed at the source. This encryption operation is extremely computationally-intensive and the ECG patch has no way of performing it. Therefore, the data has to be transferred to a nearby computationally capable device. In Figure 1, we conceptualize this device to be either a smart-phone, with approximately 5 GFLOPS computational capability (iPhone5s) or a cloudlet with close to 100 GFLOPS capability (Soyata, Ba, Heinzelman, Kwon, & Shi, 2013; Soyata T. , et al., 2012; Soyata T. , Muraleedharan, Funai, Kwon, & Heinzelman, 2012; Wang, Liu, & Soyata, 2014; Alling, Powers, & Soyata, 2015; Powers, Alling, Gyampoh-Vidogah, & Soyata, 2014). Encrypted signals should include two different flavors: a traditional encryption, such as PGP or AES (NIST-AES, 2001), or FHE. While the traditionally-encrypted data occupies significantly lower amount of space in the cloud (and, therefore, during transmission through the internet), the FHE path is both computationally and communication-wise intensive. However, FHE-based data allows computations in the encrypted format. Our idea in this chapter is to use the AES or PGP based storage for permanent archiving, and the FHE version of the data for computation (i.e., health monitoring). Communication time of the FHE-based data is less of a concern when the cloudlet is transferring the data, rather than a 3G or 4G telecom network (Kwon, et al., 2014; Kwon M. , 2015).

Medical records must be stored for a period of time to comply with regulations pertaining to Electronic Medical Records (EMRs). In the redundant storage mechanism described above, this obligation is complied with. Also note that, storing the data in AES format permanently allows the conversion of this data to FHE at any point in time in the future using AES to FHE conversion techniques (Gentry, Halevi, & Smart, 2012), thereby allowing temporary processing on the data until it is no longer needed. When processing is done, the FHE version of the data can be discarded, since the AES version is permanently stored for future reference.

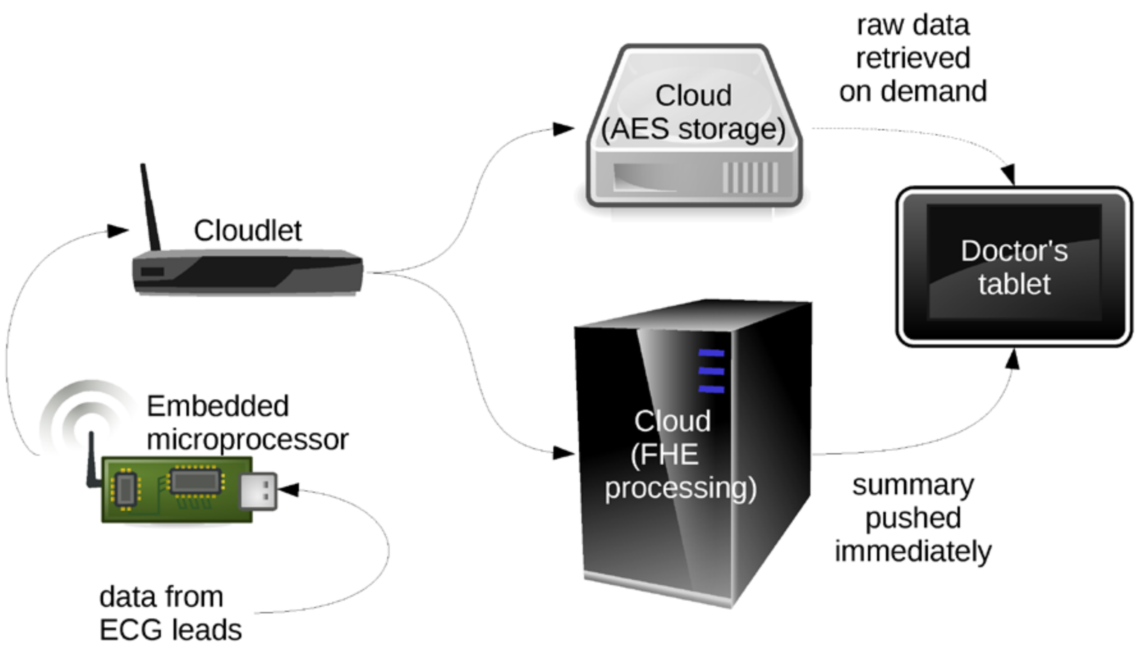


Figure 1. *A system for monitoring cardiac-related health vitals of a patient at home. An ECG patch is given to the patient* **(CardioLeaf-Pro)** *which transmits its data to a nearby smart-phone or cloudlet.*

APPLICATION CASE STUDY : LONG-TERM CARDIAC HEALTH MONITORING

One specific function that can be performed by the system in Figure 1 is *QT monitoring*. The QT interval – illustrated in Figure 2 – is an important marker for the onset of Torsades de Pointes (TdP), a potentially-fatal arrhythmia (Priori, Bloise, & Crotti, 2001). Because QT varies with heart rate, clinicians prefer to look at *corrected* QT, known as QTc. Many QT correction formulas exist; the most popular is Fridericia's (Fridericia, 1920):

$$QTcF = \frac{QT}{\sqrt[3]{RR/\sec}} \tag{1}$$

where 'F' denotes that this is the Fridericia correction, and QT and RR are the durations of the intervals illustrated in Figure 2. *Prolongation* of QTcF is a warning sign for TdP. This prolongation may occur as a result of genetic mutations, or as a reaction to certain medications (Shah, 2004). Patients who are at risk due to any of these factors are frequently monitored via ECG, particularly when adjusting prescriptions. Based on a patient's gender, medications, and history, a cardiologist will assign some threshold for QTc. If the patient's QTc goes above this threshold, the doctor should be notified immediately. Thresholds are typically around 470ms.

While monitoring of QTc is incredibly important for at-risk patients, it is usually only conducted in hospitals. This occurs in real-time, but does not provide a good picture of a patient's QTc throughout a typical day/week. One solution is to discharge the patient with a *Holter monitor*. This device can record the patient's ECG for several days. It is then returned to the hospital for analysis. This provides a good long-term view of the patient's cardiac health, but does not allow for instant notifications of potential problems. Ideally, monitoring should be both long-term *and* real-time. Such a monitoring system would involve the patient's ECG data being continuously uploaded to a server for immediate analysis, which poses many challenges to hospitals in terms of privacy and administration (Patel & Shah, 2005).

The application (and problem) we've just described has a few features that make it ideal for FHE: (1) It involves only simple calculations, (2) privacy of the data is paramount, and (3) the algorithm cannot be released. (In this case, the algorithm isn't secret; by 'cannot be released' we really mean that it cannot be remotely updated on the patient's hardware, for security reasons.) We therefore envision a system where FHE-encrypted ECG data is uploaded to a cloud-based server for QTc analysis. The server then pushes the encrypted results to the doctors' phones, which will decrypt them and raise an alarm if necessary. We will now look at some of the details of how data will be passed around in such a system.

ECG samples are generally taken with 16-bit resolution, on three or more leads, at 200-1000Hz. The resulting data stream is on the order of 10KB/sec. However, this stream can be preprocessed by a microprocessor on the patient to output only the QT and RR value associated with each heartbeat. This limits the upload stream to under 20 bytes per second, depending on the chosen data type for QT and RR, and on the patient's heart rate. For example, if we choose to represent QT and RR as 32-bit floating-point numbers, and heart rate is 60bpm, we will only need to upload and process two 4-byte values per second. On the download side of the system, we really only need to convey 1 bit of information to the doctor for each heartbeat: 'sick' or 'not sick'.

Because we intend to use HElib for processing (HElib, 2014), all values must be stored as integers. We could, for example, store the time durations in 'samples' or 'milliseconds' rather than 'seconds', or use a fixed-point notation to accomplish this. Also, because processing speed can be greatly increased by reducing the number of bits per value, it would be reasonable to store QT and RR as `short ints`, i.e. 16-bit values. And while the unencrypted QT and RR values are only generated at a rate of a few bytes per second, homomorphic encryption will explode them to several megabits per second. One way to avoid transferring QT and RR to the cloud at this exploded data rate is to use AES encryption for the upload, and an AES→FHE circuit in the cloud (Gentry, Halevi, & Smart, 2012). The consequences of using the AES→FHE technique will be extra computation in the cloud, and the need to distribute or generate the AES key without revealing it to the server.

We have described the input and output data rates and types, and now need to define the cloud-based function that will be reading and generating this data. The function to compute is given by Equation (1), which can be rewritten as:

$$\mathrm{QT}^3 > t^3 * (\mathrm{RR/sec}) \qquad (2)$$

where t is the threshold QTc value above which a warning should be raised. Note that t^3/sec can be pre-computed, rather than actually performing the cube and division operations under FHE. Or, in the case where t=500 ms, we see that multiplication by t^3 will become a right shift operation. These types of simplifications will be useful when translating the equation to the FHE domain. Later sections will explain how to rewrite this function as a matrix of FHE-encrypted values.

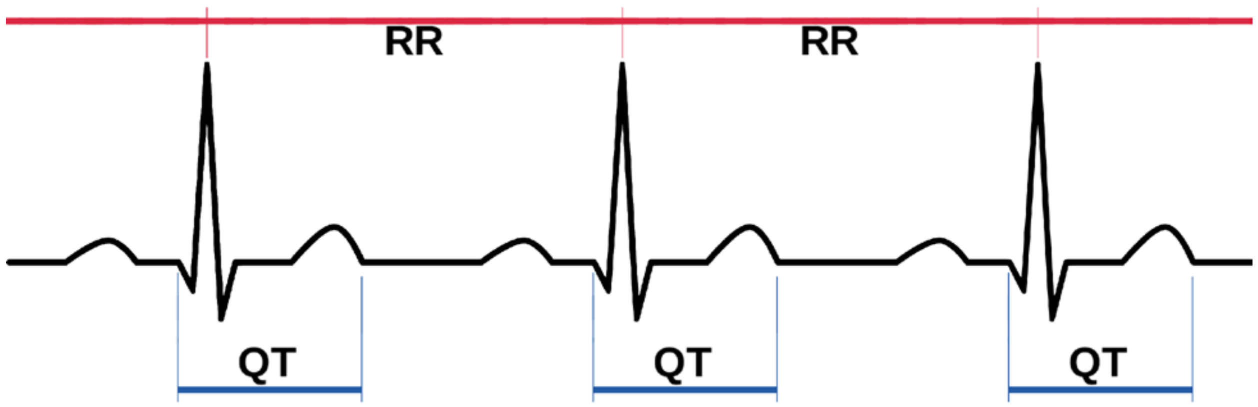


Figure 2. *QT and RR intervals in an ECG recording. Prolongation of the QT interval points to potentially hazardous cardiac events (known as Long QT syndrome). Image based on "SinusRhythmLabels" by Anthony Atkielski.*

THEW ECG DATA REPOSITORY

The telemetric and holter ECG warehouse initiative (THEW) is a worldwide repository hosted by the University of Rochester (Couderc, 2010). This library contains patient-identification-removed ECG recordings for healthy and unhealthy patients with certain known cardiac problems (e.g., LQTS). Figure 3 shows an example ECG recording from the THEW library which contains multiple Normal (N) heart beats, and multiple abnormal beats denoted as V and S types. To determine the correctness of the algorithms proposed in this chapter, simulations can be performed based on the data obtained from the THEW library. The storage of the recordings in the THEW library follow the ISHNE format (Badilini, 1998). In this format, there is a standard header that contains information such as the sampling rate and the number of samples between two beats.

Each beat is recorded as a 16-bit voltage value and the number of samples between two beats can be used to determine the temporal distance between two beats. This allows us to work with a summarized yet realistic ECG database, since in our concept system shown in Figure 1, we assume that the QT and RR values and the distance between two consecutive QT and RR values are being transferred to the cloud in FHE-encrypted format. Therefore, the knowledge of every single sample within the ~100 to 1000 samples between two heart beats contains no additional useful information for our algorithm. Knowing the QT and RR values, the QTcF value mentioned in the previous section can be computed in the cloud and compared against a known "hazard" value, such as 500 ms as previously described. As in some of our previous work **(Kocabas & Soyata, 2014)**, we will use a long term ECG recording data set, spanning approximately 24 hours. This dataset contains 87,896 heart beats (i.e., QT and RR values). Therefore, 87,896 comparisons will be necessary to determine any existing cardiac hazard conditions.

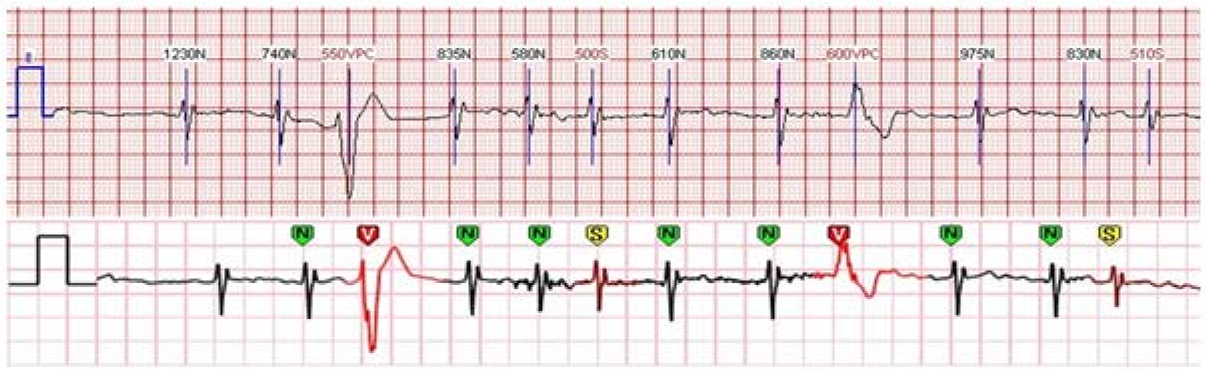


Figure 3. *Sample patient ECG data obtained from the THEW library* **(Couderc, 2010)***.*

FULLY HOMOMORPHIC ENCRYPTION (FHE)

Homomorphic Encryption schemes provide a mechanism to compute over encrypted data. The first homomorphic encryption schemes supported adding or multiplying of the values encrypted but not both operations at the same time. The Goldwasser-Micali scheme (Goldwasser & Micali, 1982) and Paillier (Paillier, 1999) schemes supported Modulo N addition operations, making them *additively homomorphic*; while the ElGamal scheme (El Gamal, 1985) was *multiplicatively homomorphic*. None of these techniques could support simultaneous addition and multiplication operations. Boneh, Goh, and Nissim (Boneh, Goh, & Nissim, 2005) was the first scheme to support multiple operations, allowing arbitrary homomorphic additions operations along with a single homomorphic multiplication operation. As an example, a vector dot product operation could be performed homomorphically using the (Boneh, Goh, & Nissim, 2005) scheme. Sander, Young, Yung (Sander, Young, & Yung, 1999) showed how to compute any shallow-depth circuit (NC^1) but their construction required a significant blow-up in the ciphertext size. The Dåmgard-Jurik scheme (Damgard, Jurik, & Nielsen, 2010) was additively homomorphic and could offer an efficiency guarantee: For a fixed public key, it can encrypt plaintexts of any size, and the ciphertext associated with a plaintext is only additively larger than the plaintext. Ishai and Paskin (Ishai & Paskin, 2007) showed how to use this property to construct a homomorphic encryption scheme that could evaluate a branching program over encrypted data. Constructing a homomorphic encryption scheme that could perform arbitrary computations over encrypted data was a long-standing open problem.

Constructed on lattice-based cryptography, Craig Gentry (Gentry, 2009) achieved a major breakthrough by introducing the first provably secure fully-homomorphic encryption scheme in 2009. Much like other public-key cryptosystems (Rivest, Adleman, & Shamir, 1978; Diffie & Hellman, 1976), lattice-based cryptography is based on an intractable Closest Vector Problem (CVP): a lattice could have an infinite number of base vectors and it is computationally hard to find the closest vector unless a proper set of basis vectors are known associated with the lattice. Based on this Closest Vector Problem (CVP), Gentry scheme can support arbitrary number of addition and multiplication operations on encrypted ciphertexts.

Since his work, there has been tremendous progress in Gentry’s work by reducing the necessary hardness assumptions to just the Learning with Errors problem on lattices (LWE) and improving the efficiency of the construction (Brakerski & Vaikuntanathan, 2011; Brakerski, Gentry, & Vaikuntanathan, 2012; Dijk, Gentry, Halevi, & Vaikuntanathan, 2010). HElib (HElib, 2014) is an open source library implementing the best known (leveled) fully-homomorphic encryption system due to Brakerski, Gentry and Vaikuntanathan, which we will refer to as BGV in the rest of the chapter.

HElib LIBRARY

The HELib library that we will use in our analysis relies on the BGV FHE encryption scheme (Brakerski, Gentry, & Vaikuntanathan, 2012) and is an open source implementation (HElib, 2014) by Halevi and Shoup (Halevi & Shoup, 2014). 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 exponentially. 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 (Smart & Vercauteren, 2014) 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. For example, in our experiments, we were able to pack 682 bits into plaintext slot, which can hold floor(682/16)=42 messages (i.e., ECG values). If we take the homomorphic addition operation, then each operation is applied to the previously mentioned 42 messages in a bitwise manner. Unfortunately, this leaves 10 message slots wasted, however, this is the artifact of the BGV scheme.

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. To store the 87,896 ECG recordings that we previously mentioned, we would need ceil(87896/42)=2093 ciphertexts. Our goal is to perform the comparison operation on these ciphertexts, where each QTc value stored in a message can be compared against the "danger threshold" of 500 ms. that we previously described. This will allow us to detect the Long-QT syndrome (LQTS) on a beat-by-beat basis.

Available Operations in HElib: In the BGV library, a rich set of operations exist. Within this set, we picked a functionally-orthogonal set to perform the LQTS comparison function. 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:

Encryption: This operation converts a plaintext into a ciphertext. Since each plaintext contains multiple "slots," the encrypted ciphertext is the representation of every slot in the plaintext, stored in encrypted form. Let ***Enc()*** denote the encryption operation. Also, let ***A*** and ***B*** denote two plaintexts with 682 slots in GF(2) each. We will denote the encrypted ciphertexts ***Enc(A)*** and ***Enc(B)***. Indeed, the size of the ciphertexts ***Enc(A)*** and ***Enc(B)*** are significantly larger then the corresponding plaintext sizes, ***A*** and ***B***. Furthermore, the ciphertext sizes for ***Enc(A)*** and ***Enc(B)*** depend on the level at which A and B are being stored.

The dependence of the ciphertext sizes on the "BGV level" is an extremely important concept, which is the dominating factor in determining the speed and storage requirements of FHE. For example, take a plaintext ***A*** containing 682 slots in GF(2). Based on our experiments, storing the corresponding ciphertext, i.e., ***Enc (A)***, requires 1MB at Level=10. In other words, to store 682 bits worth of data, 1MB must be used in the encrypted domain, translating to a storage expansion of 1024*1024*8/682=12,000x. This four-order-of-magnitude storage expansion might not sound incredibly bad when we take a look at what happens when the BGV level goes up to 20. At this Level=20, the same plaintext requires 10MB of storage, corresponding to a 120,000x expansion. This explosive growth eventually makes the ciphertext size go up to 100MB at Level=100, corresponding to a 1,200,000x expansion at level 100.

Homomorphic Evaluation: The primary reason for the storage expansion is the fact that, a lot more information has to be stored to represent the same encrypted number at higher levels. The intuition behind this is as follows: As we described before, the "level" indicates the "multiplicative depth," i.e., the maximum number of multiplications that can be performed using a ciphertext before the noise becomes too high. Assume a plaintext ***A***, whose Level=1 representation is a ciphertext **Enc(A)** of size 200KB. This means that, 682 slots (bits in GF(2)) require 200KB of storage, i.e., 300B for each bit at Level=1 (i.e., a 2,400x storage expansion). This size goes up to 300KB for the ciphertext, translating to 450B for each bit when the Level=2 (i.e., 3,600x storage expansion). The intriguing questions are: 1) what is being stored in 300B ? and, 2) why is the storage growing so fast when the level increases.

To answer the first question, we need to understand what is needed for *evaluating* a ciphertext. While addition operations are not as computationally-intensive, multiplication operations increase the noise exponentially, and requires a noise-reduction procedure. This procedure is computationally expensive and dominates the runtime of homomorphic evaluation. Performing evaluation on ciphertexts involves a massive amount of bitwise multiplications of the ciphertext bits with the appropriate public key bits. In FHE, a public key is composed of many parts (thousands or millions). Each part is needed to be stored within the public-key array for runtime evaluation, which increases the storage required.

Decryption: Homomorphic Decryption operation, denoted as ***Dec()***, that on input a ciphertext and the private key, turns a ciphertext back to its corresponding plaintext. Note that, both for the ***Enc()*** and ***Dec()*** operations, the level is known a priori.

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). Also assume that, our messages are 16 bits each. As previously mentioned, this will allow us to store floor(682/16)=42 messages in each plaintext. The corresponding ciphertexts are ***Enc(A)*** and ***Enc(B)***, which will be 100's of KB, or even MB depending on the BGV level. Now assume that, we are interested in performing a homomorphic addition on ***Enc(A)*** and ***Enc(B)***. The result is ***Enc(C)***=***Enc(A)***+***Enc(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*** *XOR* ***B*** in the un-encrypted domain, which is bitwise *XOR* or every plaintext slot (bit) individually. More specifically, we performed ***C[n]***=***A[n]*** *XOR* ***B[n]***, where ***A[n]***, ***B[n]***, and ***C[n]*** are the nth individual slot (i.e., nth bit in GF(2)) of ***A***, ***B***, and ***C*** plaintexts.

As previously discussed, our messages are 16 bits each, which occupy 16 slots of plaintext space. Since we have 682 slots in each plaintext in the example described earlier, only 42*16=672 slots are meaningful to us, leaving 10 slots unused (wasted). Regardless of this waste, the homomorphic operation still performs bitwise XOR operations on all 682 slots of the plaintext, 10 results of which will be ignored as an artifact of the "packing" concept. This highlights some important points: 1) Continuous additions on the same plaintext will eventually cause carry on 16-bit messages, which is something that our algorithm has to deal with by manually accounting with the carry results of the messages, 2) clearly, this can be prevented if a string of additions will never cause the message to exceed 16 bits, 3) more importantly, even the carry from one bit to the other has to be accounted for, since bitwise-XOR works only on individual bits.

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(A)*** and ***Enc(B)*** are their corresponding ciphertexts. Therefore, ***Enc(C)***=***Enc(A)********Enc(B)*** operation on ciphertexts corresponds to the ***C*** =***A*** *AND* ***B*** in the un-encrypted domain, which is bitwise *AND* or every plaintext slot (bit) individually. More specifically, we performed ***C[n]***=***A[n]*** *AND* ***B[n]***, where ***A[n]***, ***B[n]***, and ***C[n]*** are the nth individual slot (i.e., nth bit in GF(2)) of ***A***, ***B***, and ***C*** plaintexts. Exactly like the homomorphic addition operation, this operation will perform 10 multiplications that we will ignore. By properly choosing BGV parameters, the number of plaintext slots can be somehow manipulated to partially (or completely) avoid this waste.

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. Since these two operations work in GF(2), the carry functionality must be taken care of by using other operations: Both the Rotation or Shift operations can be used to take care of carry propagation and are both available in HElib. 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(A)*** is its encrypted version. ***RotL(Enc(A))*** operation on the ciphertext ***Enc(A)*** is equivalent to ***RotL(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 1 681****]***, 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 4 shows the runtime of the homomorphic multiplication, addition, and rotation 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.

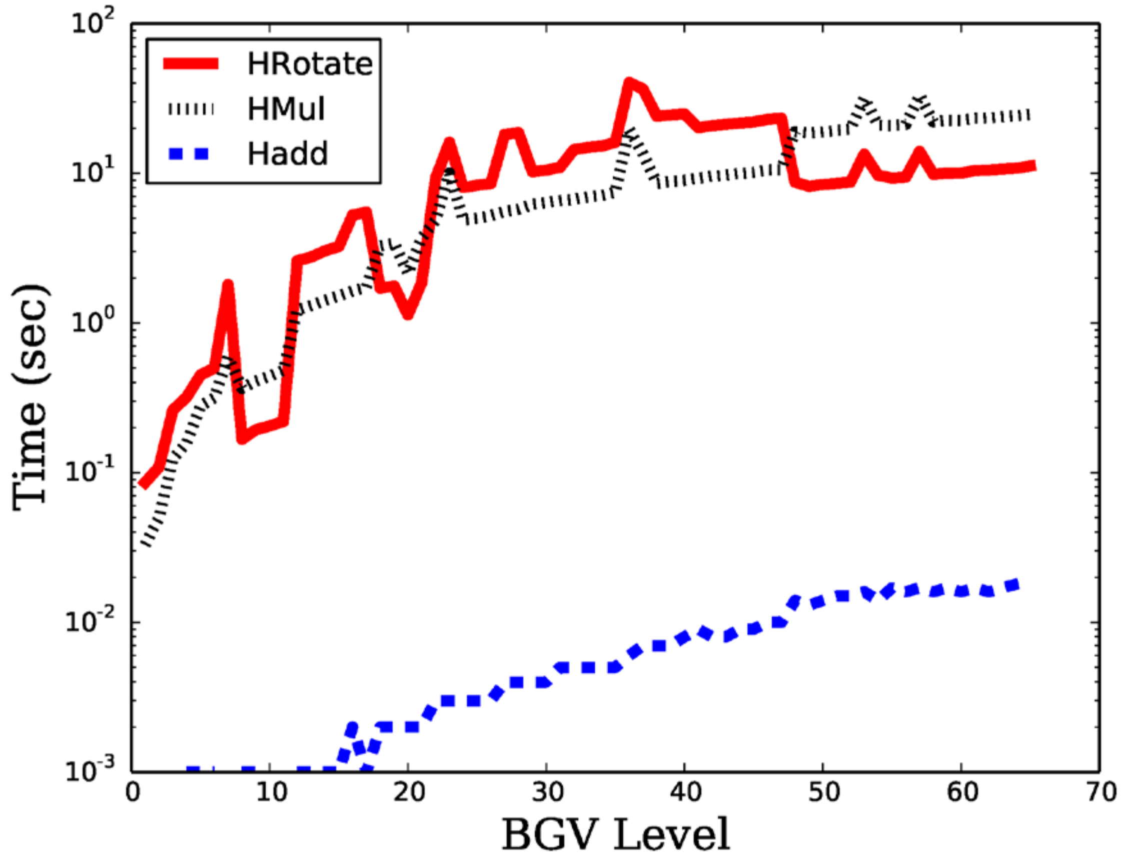


Figure 4 *Available operations in HElib and their runtimes. HAdd, HMul, and HRotate are the addition, multiplication, and rotation operations*

PROPOSED SOLUTION - OVERVIEW

In our motivating case study, we are interested in performing a simple computational task on encrypted data. More generally, in the context of cloud computing, our mechanism will be useful for securely implementing a streaming algorithm in the cloud. Informally, streaming algorithms involve performing the same simple operation on a stream of data arriving at a processing center, and then somehow aggregating the results. Examples include searching a database, indexing or collecting statistics.

More formally, in a streaming algorithm, we have a stream of data x_1, x_2, x_3 ... arriving at the processing center and the goal is to compute a function of the stream. In our motivating case study, we want to detect if there exists an element x_i such that $f(x_i) = 1$ for the simple function f described in Equation (1), where $x_i = f(QT_i, RR_i)$. Concisely, we wish to compute

$$\bigvee_i f(QT_i, RR_i)$$

Since FHE allows for computing over encrypted data, the obvious approach is to send encryptions of the data elements to the cloud, homomorphically evaluate f on each element in the incoming stream, and then compute the "OR" of the result of the computations (again, homomorphically). As we show in our experimental results, this solution is computationally costly. This is because homomorphic operations are inherently expensive. As pointed out earlier, the 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 (Goldreich, 2008) with the multiplication depth (i.e., a cascaded set of multiplications). A simple calculation will show that in order to do this following the naïve approach we need a depth $d = depth_f + \log n$ to process n data elements (each $\vee$ requires 1-depth), where $depth_f$ is the multiplication depth of f. The main contribution of our work is to show how we can significantly improve the computational efficiency by relying on an alternative representation of the computation that will significantly reduce the depth of the overall computation, i.e. function evaluation and aggregation. In addition, our method will be easily parallelizable and have small input locality, i.e. our sequence of homomorphic operations can be broken down to smaller sets each only dependent on a few ciphertexts. Most FHE implementations show how to compute any 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, in a sense, a form of combinatorial optimization (Soyata, Friedman, & Mulligan, 1997; Soyata & Friedman, 1994). The result of the computation is true if there is a path from the start to the final node traversing only edges for which the assignment sets the value on the edge true. Next, we show how to use an FHE scheme to evaluate a branching program and aggregate the results of the computation over streaming data. Using elementary linear algebra we first show that evaluating a branching program is equivalent to evaluating the determinant of a particular matrix. More precisely, the determinant will be $f(x)$ for the matrix corresponding to input x. Given the matrix representation of two inputs x_1 and x_2, computing the "AND" of $f(x_1)$ and $f(x_2)$ now reduces to simply multiplying the matrices corresponding to the inputs, since

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

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 elements in the data stream. 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 multiplicative depth of our computation. In fact, the depth of our computation will be $\log n$. An overview of the process we've just described is shown in **Figure 5**. We will now explain our approach in detail.

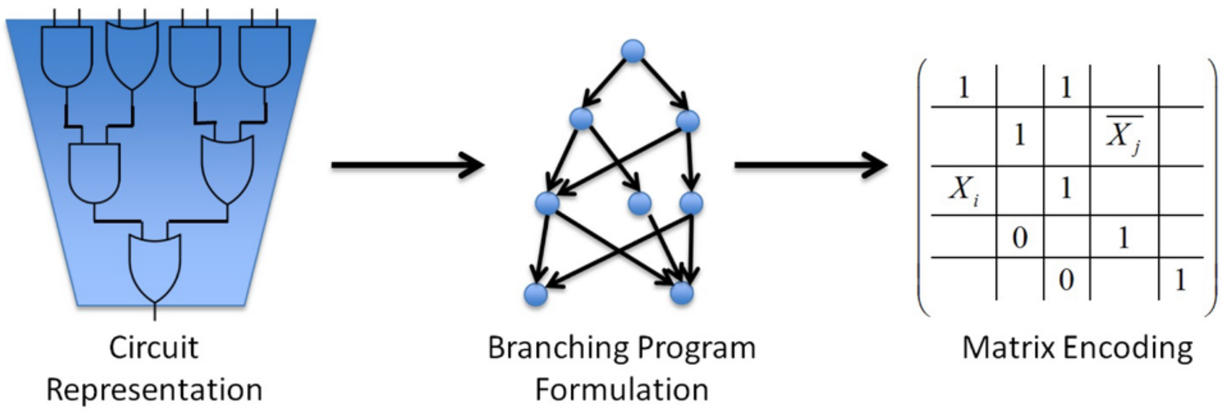


Figure 5. *Converting a circuit (representing a function, which is a part of an algorithm) into matrix form.*

COMPUTATIONAL MODELS

Oded Goldreich (Goldreich, 2008) 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* our computational task and the best environment to *securely* evaluate it. Towards this, we first discuss the computational models relevant to our discussion. We assume familiarity with Turing Machines and polynomial-time computation.

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 on providing a formal framework for our circuit model.

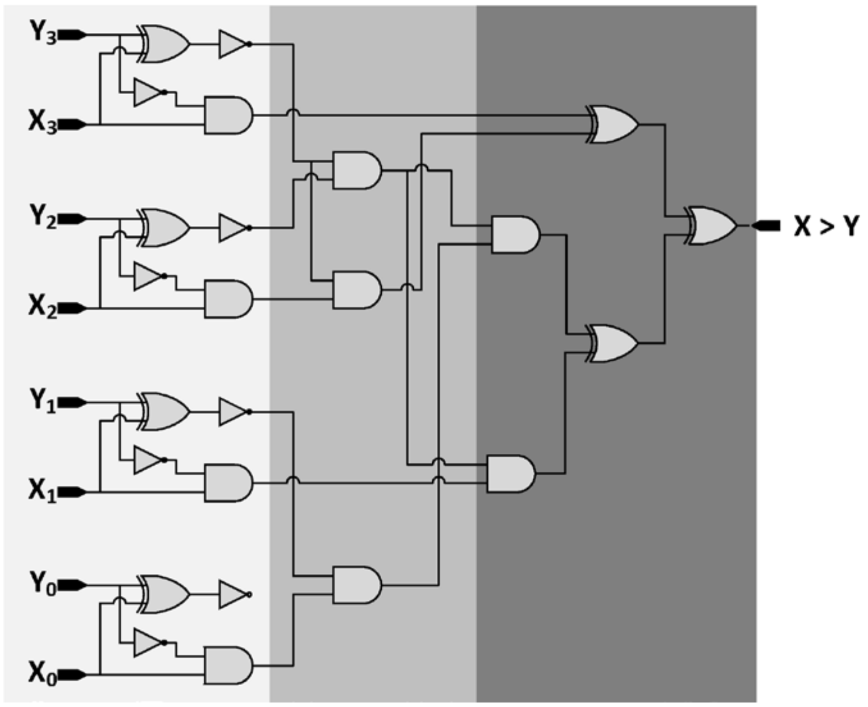


Figure 6. *The circuit for the comparison of two 4-bit numbers, denoted as X and Y.*

Definition 1. A circuit C is a 6-tuple $(n, G, S, \varphi, \theta, v^*)$ 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, $\varphi : S \rightarrow \{i\}_{1 \le i \le n}$ labels vertices in S with an input bit, $\theta : V \setminus S \rightarrow \{\wedge, \oplus\}$ labels the rest of the vertices with a Boolean operation, and $v^* \in V$ 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 $x_{\varphi(s)}$, and the output of any work gate $v \in V \backslash S$ 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^* 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^*. We denote this number by $Depth(C)$. We define $Depth_{\wedge}(C)$ similarly, except that $\oplus$ gates do not increase $\wedge$-depth: the $\wedge$-depth of a $\oplus$ work gate is defined as the maximum of the $\wedge$-depth of its two parent gates.

Definition 2. A family of circuits $\{C_i\}_{i \in N}$ 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 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^1 circuits is already a rich class which can compute basic arithmetic operations such as addition, multiplication and division on n-bit integers. In particular, we this model will allow computation of our function f from Equation (1).

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 5. A branching program is a tuple (V,E,φ,s,t,n), where (V,E) is a directed acyclic graph, $\varphi: E \rightarrow \{T\} \cup \{x_i, \overline{x_i}\}_{i\in V}$ 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.

We show below a branching program for the comparison operator on 2-bit inputs X and Y. In order to compare two numbers, first, we compare the most significant bits in X and Y and then work down to the least significant digit. If the most significant bit of X is larger than the most significant bit of Y, then $X > Y$. If the most significant bit of X is smaller then we know $X > Y$ is false. If they are equal then we move onto the next significant bit. Once we find a position where the bits in $X > Y$ are, we don't have to look further. We implement this idea as a simple directed (acyclic) graph as shown in **Figure 9**.

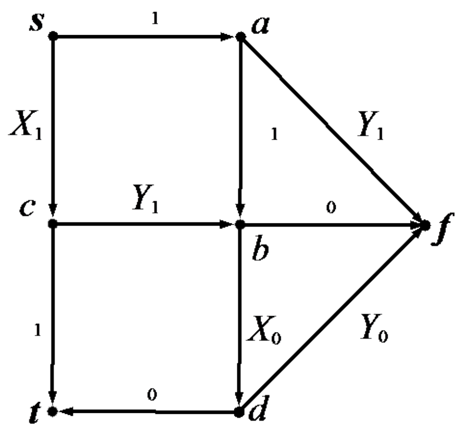


Figure 7. *A Branching Program for Comparison.*

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. In **Figure 10**, 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.

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.

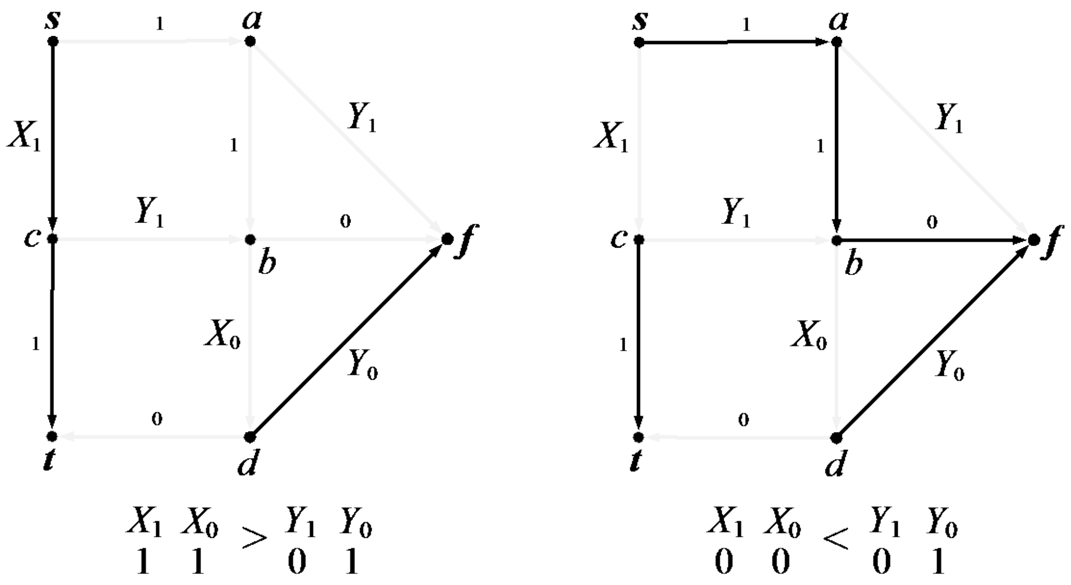


Figure 8. *Evaluation of a Branching Program.*

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 logspace Turing machine can do.

Definition 6. A logspace 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 1. Any logspace 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 logspace 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.

METHODOLOGY

Our proposed system has three phases: pre-computation, cloud computation, and post computation. In the pre-computation phase, input data elements are encrypted under FHE scheme 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 values via matrix multiplication. In the post-computation phase, the final matrix, that is the product of all matrices, is downloaded and decrypted at the client. To compute the final outcome of the computation in the post-computation phase, the client simply evaluates the determinant of the decrypted matrix.

Pre-computation Phase

Recall that the QT and RR values from the ECG signals are computed in the ECG patch. Since encrypting and transmitting is computationally expensive, the data is transferred to a nearby computationally capable device. The values received at the edge device is first encrypted and then transmitted to the cloud. Typically, QT and RR are 16-bit values. Since we will be employing the BGV encryption scheme to manipulate the data in the cloud, these values will be encrypted using the same scheme at the device. Towards this a public-key/private-key is generated for the appropriate level, 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 extended BGV scheme allows multiple slots, we encrypt as many of the values in a single ciphertext. As mentioned before, we will use the scheme with 682 slots it can pack $21\ (\times\ 32)$ samples of QT and RR with 10 slots of wastage.

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 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 $\varphi(\mathrm{i}, \mathrm{j})$. The matrix representation of an input $x = b_1 b_2 \ldots b_n$ will be the adjacency matrix will be the matrix M obtained from $A(G)$ by replacing the variables x_i with their respective values b_i.

We explain below how the adjacency matrix will help encode the computation of $f(x)$. Suppose we have a direct acyclic graph $G = (V, E)$ with n vertices. First, we show that the number of paths of length t from i to j is equal to $A(G)^t_{i,j}$.

Lemma 2. Let A 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 to be $(I - A)^{-1}$. We prove that $P_{i,j}$ is the number of paths from i to j.

Corollary 1. 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 $(I - A(G))^{-1}_{i,j}$.

Proof. Applying this lemma, 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,

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

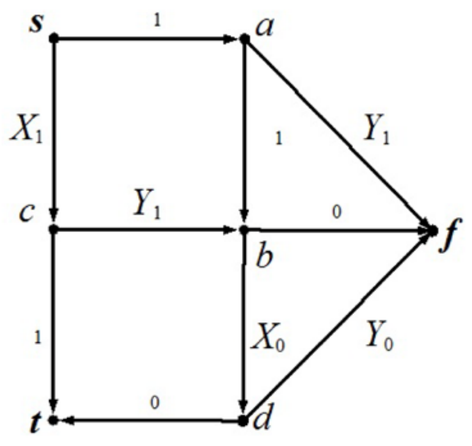
which is equivalent to

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

Let $P(G) = (I - A(G))^{-1}$ be the path counting matrix of G. To determine $P(G)_{i,j}$ 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 says that if $Bx = b$ and $det(B) \neq 0$ then $x_i = \frac{\det(B')}{\det(B)}$ where B' is B, except its ith column is replaced with b. Therefore if we replace the ith column of $I - A$ with the jth 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 $P_{i,j} = \det(A')\ (mod\ p)$. 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.

Once we know the branching program we want to evaluate, we can compute $(I - A'(G))^{-1}$ as defined in Corollary 1 to get a matrix such that its determinant is equal to $f(x)$. We compute this matrix symbolically for all x then plug in the values of the bits of x according to the formula to get the "literal" matrix. However, the server only gets the encrypted input bits, i.e. it receives the encryption of each bit of the input. The server can construct the matrix by replacing literal zero and one elements with encryptions of zero and one and replacing elements labeled with input bit indices with the appropriate input ciphertext, negating the ciphertext if the matrix calls for it. Negation of a ciphertext can be performed by adding 1 (modulo 2) and is an efficient homomorphic operation. Hence the matrix in encrypted form can be efficiently computed from the encrypted input using homomorphic operations.

Recall our comparison example from **Figure 9**. We apply the preceding idea to obtain the corresponding matrix where the rank of the matrix will determine the output of the function. This matrix corresponding to the graph for 2-bit comparison is also displayed in **Figure 9**. We remark that this idea is inspired by the work of Ishai and Kushilevitz (Kushilevitz & Ishai, 2000).



$$\begin{array}{c|cccccc}
 & f & t & a & c & b & d \\
\hline
f & 1 & 0 & 0 & 0 & 0 & 0 \\
s & 0 & 0 & \overline{X}_1 & X_1 & 0 & 0 \\
a & Y_1 & 0 & 1 & 0 & \overline{Y}_1 & 0 \\
c & 0 & \overline{Y}_1 & 0 & 1 & Y_1 & 0 \\
b & \overline{X}_0 & 0 & 0 & 0 & 1 & X_0 \\
d & Y_0 & \overline{Y}_0 & 0 & 0 & 0 & 1
\end{array}$$

Figure 9. *Matrix representation of the Comparison Function.*

Aggregation

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

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

Recall that multiple data elements can be encrypted in a single 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 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.

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

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.

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

Then using a single operation of $Row[1] \times Col[1] = (a_{1,1}b_{1,1} \quad a_{1,2}b_{2,1} \quad a_{1,3}b_{3,1})$ which looks promising as they are required to compute $C[1,1]$. However HElib does not provide operations to directly sum the elements of a packed ciphertext. 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 $\left(1, 1 + (i \bmod n)\right), \left(2, 1 + (i + 1 \bmod n)\right), \dots, \left(n, 1 + (i + n - 1 \bmod n)\right)$. 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

$$\begin{aligned} A\{0\} &= (a_{1,1} \quad a_{2,2} \quad a_{3,3}) \\ A\{1\} &= (a_{1,2} \quad a_{2,3} \quad a_{3,1}) \\ A\{2\} &= (a_{1,3} \quad a_{2,1} \quad a_{3,2}) \end{aligned}$$

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

$$\begin{aligned} C\{1\} &= A\{1\}B\{1\}^0 + A\{2\}B\{3\}^1 + A\{3\}B\{2\}^2 \\ C\{2\} &= A\{1\}B\{2\}^0 + A\{2\}B\{1\}^1 + A\{3\}B\{3\}^2 \\ C\{3\} &= A\{1\}B\{3\}^0 + A\{2\}B\{2\}^1 + A\{3\}B\{1\}^2 \end{aligned}$$

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 $[v_i, v_{i+1}, \dots, v_1, \dots, v_{i-1}]$.

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

$$C\{i\} = \sum_{j=1}^{n} A\{j\}\, B\{1 + (i - j \bmod n)\}^{j-1}$$

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 ∧-depth 1 and its operations can be bundled together efficiently, it still requires many homomorphic multiplications. However, these can be done in parallel. Furthermore, if the matrices are of a special sparse form, we can multiply them more efficiently.

Post-computation Phase

Recall that at the end of the Cloud Computation phase, after all the encrypted matrices are aggregated, the Cloud possesses a single matrix that contains the value of the aggregated output. This matrix is downloaded by the health provider, such as the doctor. Since the doctor 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. 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,

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

A branching program for $\neg f$ is considered. This means upon receiving the final matrix, the determinant is evaluated and the output is negated to obtain the precise answer to the computation.

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 1, any logspace computations can be represented as a branching program and can encompass a wide variety of problems. Therefore, this approach can be used for any logspace 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.

PERFORMANCE EVALUATION

In this section, we will provide simulated results for our proposed method and will compare them to the circuit-based method, which will be referred to as the "naïve 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.

Experimental Setup

In the previous section, we described our methodology. We compare the performance of our method with the naïve method of implementing the circuit that computes the function in entirety. We first describe the naïve method and how the two methods were compared.

a) Naïve method:

First, we explain how we use the SIMD-like operations in HElib. An example operation is shown in **Figure 10**, where two 4-bit numbers, A_1=13 and B_1=17 are compared. Simultaneously, two other 4-bit numbers A_0=9 and B_0=11 are compared. These two simultaneous comparisons are an example of how multiple identical operations can be performed in a SIMD environment as a benefit of the "packing" concept introduced previously.

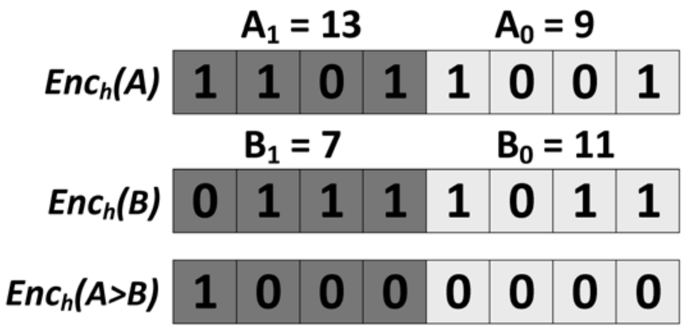


Figure 10. *An example comparison of two 4-bit numbers.*

Each individual operation, applied to A_i and B_i is a 4-bit SIMD addition, for which the circuit in **Figure 7** can be used. Note that, this circuit is composed of only XOR and AND gates, since the NOT gates (i.e., inverters) can be implemented by XOR gates. The reason for restricting the set of available gates to only these two is previously mentioned: We are only using the GF(2) homomorphic addition and homomorphic multiplication operations, which correspond to bitwise XOR and bitwise AND operations, respectively. Therefore, drawing the circuit for any function we are trying to implement allows us to use two of the four previously mentioned HElib primitives.

A close observation of **Figure 7** reveals that, just the bitwise XOR and AND operations will not be sufficient to perform the 4-bit comparison function, which can be denoted as

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

In this formulation of the comparison of X and Y (i.e., computation of X>Y), when "slot indices" match for the bits of two numbers (e.g., x_3 and y_3), no rotation is necessary, however, we will use the previously mentioned second set of HElib operations, namely Rotate and Select) to perform alignment operations on the bits. In **Figure 7**, the output of the bottom inverter has no connection. The XOR and NOT gates connected to the Y_0 node symbolically depict how SIMD operations actually perform these extraneous operations, even though the result will not be used for anything useful. As shown in **Figure 11**, if we define an intermediate value M as the rotated versions of the E, where E is the equality. Therefore,

$E = XNOR(X,Y)$ and $M = (1 \quad e_3 \quad e_3e_2 \quad e_3e_2e_1)$, where e_i is the ith bit of E.

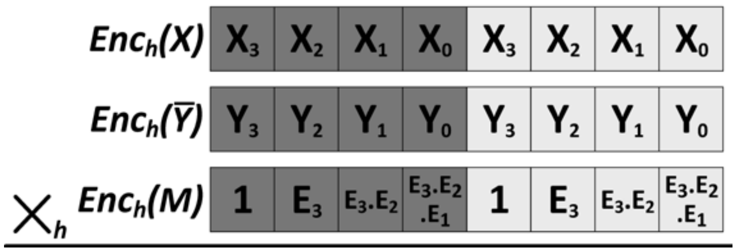


Figure 11. *Reformulating the X>Y operation in terms of an intermediate variable M*

As can be intuitively visualized from **Figure 11**, the four operations we are using from HElib will be sufficient to calculate the M and E values, and, therefore the Boolean value of (X>Y).

To compute Equation (1), we need to compute $QT^3 - RR$ and check whether it is positive or negative. We evaluate QT^3 as follows. We first compute the n addends in computing QT^2. This can be done by computing

$$Q_i = (QT \times QT_i) \ll i \text{ for } i = 1, \dots, n$$

where $\ll$ is the left-shift operator. This is the first step in the long-multiplication form of multiplication. Next we multiply each of these n addends by QT to obtain n^2 addends required to compute QT^3. More precisely,

$$A_{j*n+i} = \left(Q_j \times QT_i\right) \ll i \text{ for } i = 1, .., n \text{ and } j = 1, \dots, n$$

Next we add the value $\sim RR$ and 1 as addends, i.e. A_{n^2+1} is set to the bitwise negation of RR. In two's complement, these two final summands add up to $-RR$ (the additive inverse of RR, not to be confused with the bitwise negation of RR), so it is appropriate to include them as summands in order to subtract RR from QT^3. To sum the $n^2 + 2$ addends, we first reduce the number of addends by using a 3-2 compressor. In more detail, this method replaces 3 addends with 2 addends using the following approach. Let a, b, c be three n-bit addends. We replace them by d and e where

$$d = a \oplus b \oplus \mathrm{c}$$

$$e = (a \wedge b \oplus b \wedge c \oplus c \wedge a) \ll 1$$

where each of the operations are done bitwise. The result of this computation reduces the problem of adding three n-bit numbers a, b, c to adding two $n + 1$-bit numbers d and e. Given than we have $n^2 + 2$ n-bit addends, we divide them into groups of 3 and perform this operation to get a total of $2\left\lceil\frac{n^2+2}{3}\right\rceil$ addends, each $n + 1$-bits long. We now repeat this process to get $2\left\lceil\frac{2\left\lceil\frac{n^2+2}{3}\right\rceil}{3}\right\rceil$ addends where each addend is now $n + 2$-bits long. We continue this process until we only have two $n + O(\log_{3/2} n)$-bit addends remaining. We then add them using a standard carry look-ahead adder which we describe next.

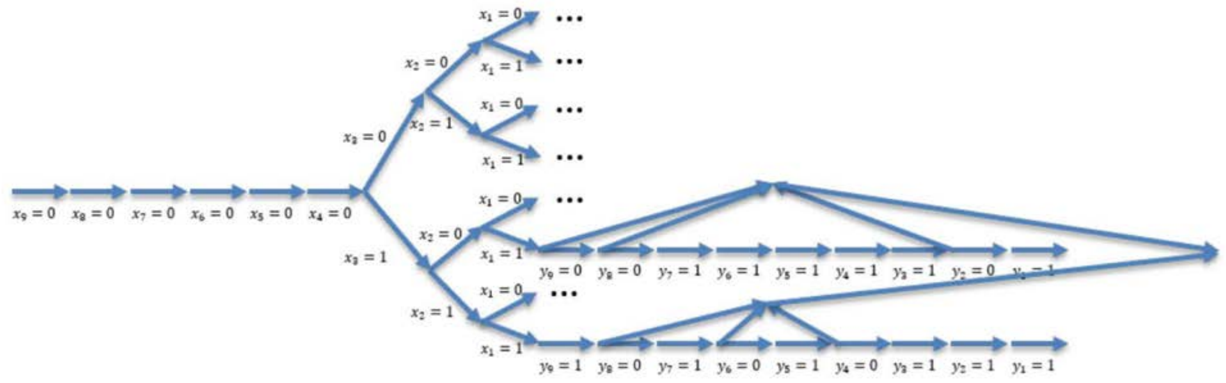
Let a and b be two m-bit numbers. In the ripple-carry adder, we first compute the propagate and generator bits $p = a \cdot b$ and $g = a \oplus$ b where the operations are performed bit-wise. Next the carry bits c are computed bitwise $c_{i+1} = g_i + p_i \cdot c_i$ where c_1 is set to 0 and $i = 1, \ldots, m$. Finally the sum is calculated as $s_i = p_i \oplus c_i$ for $i = 1, \ldots, m + 1$ where p_{i+1} is set to 0. Computed this way, will lead to m cascading multiplications. However, by appropriately computing products of p_i efficiently we can bring it down to log (m) cascading multiplications. In fact, this is what is done in a carry look-ahead adder.

Given this final sum s, to obtain the bit that represents whether $QT^3 > RR$ is obtained by selecting the most significant bit of the s.

b) Matrix method: For the i^{th} input sample $QT[i]$ and $RR[i]$ that is provided in encrypted form, we first generate the matrix encoding in encrypted form. Given then matrices for all the samples, the aggregated output is simply the product of these matrices. To generated 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 $QT^3 \geq 2^{3j}$. 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 $2^{n/3}$ least-significant bits, we get a path for each value of QT below $2^{n/3}$. 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. Start from the most significant digit and look for the first difference in the digits of the two numbers. The result is determined by the first place where the digits differ, and if the digits never differ then the two numbers are equal. The first digits are the same so we look at the second digit. Five is more than four, so $254 > 249$.



$x_9 = 0$
$x_8 = 0$
$x_7 = 0$
$x_6 = 0$
$x_5 = 0$
$x_4 = 0$
$x_3 = 0$
$x_3 = 1$
$x_2 = 0$
$x_2 = 1$
$x_2 = 0$
$x_2 = 1$
$x_1 = 0$
$x_1 = 1$
$y_9 = 0$ $y_8 = 0$ $y_7 = 1$ $y_6 = 1$ $y_5 = 1$ $y_4 = 1$ $y_3 = 1$ $y_2 = 0$ $y_1 = 1$
$y_9 = 1$ $y_8 = 0$ $y_7 = 1$ $y_6 = 0$ $y_5 = 1$ $y_4 = 0$ $y_3 = 1$ $y_2 = 1$ $y_1 = 1$

Figure 12. *Diagram of the Branching Program for computing 254 > 249 ?*

The diagram in **Figure 12** contains an abbreviated form of the branching program $x^3 < y$ 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 $x = 5$, so $x^3 = 125$. 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 we're 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 and none of the earlier bits are 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 "known" to that path.

Since we know the symbolic matrix, we can generate the packed ciphertexts homomorphically. This simply uses the rotate and select operations, and we need to generate only 10 packed ciphertexts for our branching program. We evaluate the product of the matrices using Equation 2. Computing each diagonal of the resulting matrix requires at most 10 rotate and 10 multiply operations and 10 additions.

Time Estimation via Simulation

Since the HELib library is not thread-safe, to compare the performance of the two approaches we simulated the computation to estimate the time taken on parallel machines. To accomplish this, we first performed real benchmarks of each individual operation on a single thread, and then used that time to estimate the computational costs for our parallel experiments. We set the length of QT and RR to be 16-bit.

Depth calculation: In order to use HElib, we need to generate keys for the BGV encryption scheme for a particular level. The level determines the number of sequential multiplications we can perform.

In the naïve method, since we need to both calculate and aggregate the overall depth and for 16-bit inputs, the depth is 44. Aggregation requires $\log T$ multiplications for aggregating T samples. We performed the experiment for $T = 1{,}50{,}100{,}500,$ and 1000. Depending on the value of T we instantiated the BGV scheme appropriately.

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 aggregate and compute is simply $\log T$ = 12.

Then 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 accordingly the list was updated. 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 the naïve method requires a lot of data transfers since it involves more dependencies, while the proposed approach will have significantly less and taking into account the data transfer costs will only improve the relative timing of our approach.

Results

To evaluate our approach we considered processing and aggregating several different numbers of samples. In **Figure 13**, we provide our results for processing 10 and 10240 samples. We can see that both approaches improve with the number of processors. However, the matrix approach is consistently better than the naïve 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 naïve approach is significantly higher than that of the matrix approach. One tradeoff 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 our approach while only 1 ciphertext will need to be downloaded in the naïve approach. However, in our application, this will only be done once every day (or few hours), so this should not be a significant problem. Another difference with the matrix approach is that the doctor's computer must compute the determinant of the decrypted matrix in order to get the result; in the naïve approach, the result will be directly available (or be a simple "OR" of the decrypted bits).

CONCLUSIONS AND FUTURE WORK

Medical cloud computing is the unstoppable next revolution in healthcare. However, its widespread adoption depends on addressing Protected Health Information (PHI) privacy issues during its acquisition, storage and processing. The system described by the authors before and expanded in this paper provides a way to completely solve data privacy issues while the patient medical data is being stored and processed in the cloud. The enabling component of this system, Fully Homomorphic Encryption (FHE), creates a computational environment, where it is possible to compute on encrypted data. Data privacy issues are automatically eliminated when the only information that is available during transmission and processing is encrypted by a *provably secure* encryption mechanism.

Despite this unparalleled advantage of FHE, its intense computational requirements make it applicable to only a limited set of medical applications. While extensive research is underway to speed up generalized FHE, in this chapter, we focused on a specific cloud computing scenario, where the goal of computation is to detect a set of known cardiac hazard conditions from streamed ECG data. This ECG data, streamed from the patient's house through ECG patches into the cloud, allows long term health monitoring for certain cardiac conditions that are very hard (if not impossible) to detect at a healthcare organization (HCO). Additionally, allowing monitoring outside the HCO has a significant cost-saving and improved diagnosis potential due to the additional information that is being provided to the healthcare professionals.

In this chapter, an extensive analysis and evaluation of the application of FHE into long term healthcare monitoring is presented. Rather than using a circuit-based approach to evaluate the required functions for monitoring, a Branching Programs (BP) based approach is investigated. While the BP approach narrows the application area of FHE, it is shown that it promises a significant speed up for detection applications, where the required result is a Yes/No Boolean value. A Branching Program model is developed in this chapter which detects the Long-QT Syndrome (LQTS) from a set of streamed ECG interval data. A 20x speed-up is observed as compared to the circuit-based methods. This is a promising step towards improving the speed of FHE, which can eventually become a key ingredient of future medical applications.

The solution presented in this chapter detects LQTS by using a formula $\mathrm{QTc}^3 > 2\,RR$, which is restricted to the QTc value of 0.5 (i.e., 500 ms). Research is underway to generalize this to a broad set of potential QTc values that are of clinical relevance, such as 450 ms or 470 ms. Although it looks like a simple extension of the existing work, arbitrarily changing the comparison value requires the re-construction of the Branching Program, and, therefore, is the focus of our future work.

ACKNOWLEDGMENT

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

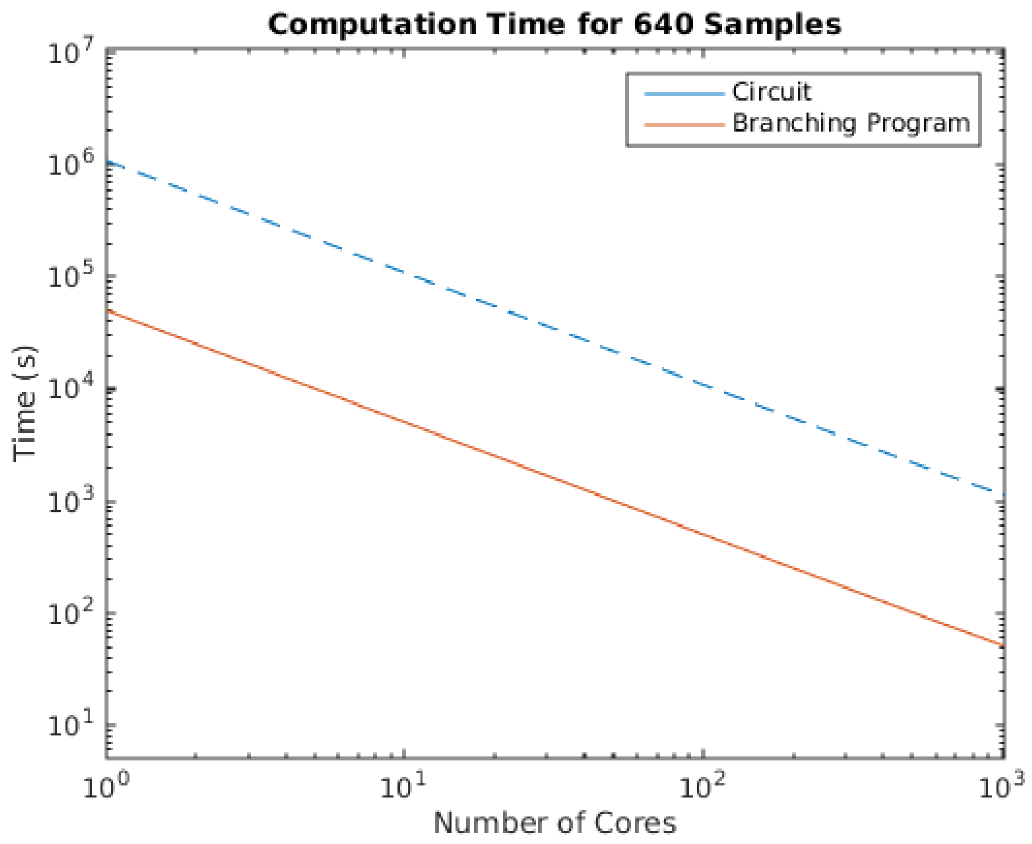


Figure 13. *Simulated computation time for matrix method vs. naïve method of Long QT detection. The matrix method is consistently about 20x faster.*

REFERENCES

(2014). From SalesForce.com: http://www.salesforce.com

(2014). From HIPAA: http://www.hhs.gov/ocr/privacy/

Alling, A., Powers, N., & Soyata, T. (2015). Face Recognition: A Tutorial on Computational Aspects. In *Emerging Research Surrounding Power Consumption and Performance Issues in Utility Computing.* IGI.

Badilini, F. (1998). The ISHNE Holter Standard Output File Format. *Annals of Noninvasive Cardiology*, 263-266.

Barrington, D. (1989). Bounded-Width Polynomial Size Brancing Programs Recognized Exactly Those Languages in NC1. *Journal of Computer System Sciences, 38*(1), 150-164.

Boneh, D., Goh, E.-J., & Nissim, K. (2005). Evaluating 2-DNF Formulas on Ciphertexts. *Proceedings of the 2nd International Conference on Theory of Cryptography*, (pp. 325-341).

Brakerski, Z., & Vaikuntanathan, V. (2011). Efficient Fully Homomorphic Encryption from Standard LWE. *Foundations of Computer Science*, 97-106.

Brakerski, Z., Gentry, C., & Vaikuntanathan, V. (2012). Leveled Fully Homomorphic Encryption without Bootstrapping. *Proceedings of the 3rd Innovations in Theoretical Computer Science Conference*, (pp. 309-325).

CardioLeaf-Pro. (n.d.). From http://www.clearbridgevitalsigns.com/

Couderc, J. (2010). The telemetric and holter ECG warehouse initiative (THEW). A data repository for the design, implementation and validation of ECG-related technologies. *Annual International Conference of the IEEE Engineering in Medicine and Biology Society (EMBC)*, 6252-6255.

CR2032. (n.d.). From http://en.wikipedia.org/wiki/CR2032_battery

Damgard, I., Jurik, M., & Nielsen, J. B. (2010). A Generalization of Paillie's Public Key System with Applications to Electronic Voting. *International Journal of Information Security*, 371-385.

DARPA-PROCEED. (n.d.). From DARPA-PROCEED: http://www.darpa.mil/Our_Work/I2O/Programs/PROgramming_Computation_on_EncryptEd_Data_(PROCEED).aspx

Diffie, W., & Hellman, M. (1976). New directions in cryptography. *IEEE Transactions on Information Theory, 22*(6), 644-654.

Dijk, M., Gentry, C., Halevi, S., & Vaikuntanathan, V. (2010). Fully Homomorphic Encryption over the Integers. *Advances in Cryptology (EUROCRYPT)*, 24-43.

El Gamal, T. (1985). A Public Key Cryptosystem and a Signature Scheme based on Discrete Logarithms. *IEEE Transactions on Information Theory*, 469-472.

Fridericia, L. S. (1920). Die Systolendauer im Elektrokardiogramm bei normalen Menschen und bei Herzkranken. *Acta Medica Scandinavica, 53*, 469-486.

Gentry, C. (2009). *A Fully Homomorphic Encryption Scheme.* Stanford University.

Gentry, C., Halevi, S., & Smart, N. (2012). Homomorphic Evaluation of the AES Circuit. In R. Safavi-Naini, & R. Canetti (Eds.), *Advances in Cryptology – CRYPTO 2012* (Vol. 7417, pp. 850-867). Springer Berlin Heidelberg. From http://dx.doi.org/10.1007/978-3-642-32009-5_49

Goldreich, O. (2008). *Computational complexity - a conceptual perspective.* London: Cambridge University Press.

Goldwasser, S., & Micali, S. (1982). Probabilistic Encryption - How to Play Mental Poker Keeping Secret all Partial Information. *Proceedings of the 14th Annual ACM Symposium on Theory of Computing*, (pp. 365-377).

Good, S. (2013). *Why Healthcare Must Embrace Cloud Computing.* Forbes.

Halevi, S., & Shoup, V. (2014). Algorithms in HElib. *Proceedings, Part I, Advanced in Cryptology - {CRYPTO} 2014 - 34th Annual Cryptology Conference*, (pp. 554-571). Santa Barbara, CA.

HElib. (2014). *HElib Fully Homomorphic Encryption Library*. From http://www.github.com/shaih/HElib

iPhone5s. (n.d.). From http://www.anandtech.com/print/7335/the-iphone5s-review

Ishai, Y., & Paskin, A. (2007). Evaluating Branching Programs on Encrypted Data. *Theory of Cryptography, 4th Theory of Cryptography Conference, (TCC) 2007*, (pp. 575-594). Amsterdam.

Kocabas, O., & Soyata, T. (2014). Medical Data Analytics in the cloud using Homomorphic Encryption. In P. R. Deka, *Handbook of Research on Cloud Infrastructures for Big Data Analytics* (pp. 471-488). Hershey, PA, USA: IGI Global. doi:10.4018/978-1-4666-5864-6.ch019

Kocabas, O., Soyata, T., Couderc, J.-P., Aktas, M., Xia, J., & Huang, M. (2013). Assessment of Cloud-based Health Monitoring using Homomorphic Encryption. *Proceedings of the 31st IEEE International Conference on Computer Design (ICCD)*, (pp. 443-446). Ashville, VA, USA.

Kushilevitz, E., & Ishai, Y. (2000). Randomizing Polynomials: A New Representation with Applications to Round-Efficient Secure Computation. *41st Annual Symposium on Foundations of Computer Science* (pp. 294-304). Redondo Beach: ACM.

Kwon, M. (2015). A Tutorial on Network Latency and its Measurements. IGI Global.

Kwon, M., Dou, Z., Heinzelman, W., Soyata, T., Ba, H., & Shi, J. (2014). Use of Network Latency Profiling and Redundancy for Cloud Server Selection. *Proceedings of the 7th IEEE International Conference on Cloud Computing (IEEE CLOUD 2014)*, (pp. 826-832). Alaska. doi:10.1109/CLOUD.2014.114

NIST-AES. (2001). *Advanced encryption standard (AES).*

Page, A., Kocabas, O., Ames, S., Venkitasubramaniam, M., & Soyata, T. (2014). Cloud-based Secure Health Monitoring: Optimizing Fully-Homomorphic Encryption for Streaming Algorithms. *IEEE Globecom 2014 Workshop on Cloud Computing Systems, Networks, and Applications (CCSNA).* Austin, TX.

Page, A., Kocabas, O., Soyata, T., Aktas, M., & Couderc, J.-P. (2014). Cloud-Based Privacy-Preserving Remote ECG Monitoring and Surveillance. *Annals of Noninvasive Electrocardiology (ANEC)*.

Paillier, P. (1999). Public Key Cryptosystems Based on Composite Degree Residuosity Classes. *Advances in Cryptology*, (pp. 223-238).

Patel, C. D., & Shah, A. J. (2005, June). *Cost Model for Planning, Development and Operation of a Data Center.* Retrieved from http://www.hpl.hp.com/techreports/2005/HPL-2005-107R1.pdf

Powers, N., Alling, A., Gyampoh-Vidogah, R., & Soyata, T. (2014, Dec). AXaaS: Case for Acceleration as a Service. *IEEE Globecom 2014 Workshop on Cloud Computing Systems, Networks, and Applications (CCSNA).* Austin, TX.

Priori, S. G., Bloise, R., & Crotti, L. (2001). The long QT syndrome. *Europace, 3*(1), 16-27. Retrieved from http://europace.oxfordjournals.org/content/3/1/16.short

Reichman, A. (2011). *File storage costs less in the cloud than in-house.* Forrester.

Rivest, R., Adleman, L., & Shamir, A. (1978). A method for obtaining digital signatures and public-key cryptosystems. *Communications of the ACM, 21*(2), 120-126.

Sander, T., Young, A. L., & Yung, M. (1999). Non-Interactive CryptoComputing For NC1. *40th Annual Symposium on Foundations of Computer Science* (pp. 554-567). New York: ACM.

Shah, R. (2004). Drug-induced QT interval prolongation: regulatory perspectives and drug development. *Annals of medicine, 36*(S1), 47-52.

Smart, N. P., & Vercauteren, F. (2014). Fully homomorphic {SIMD} operations. *Des. Codes Cryptography, 71*(1), 57-81.

Soyata, T., & Friedman, E. G. (1994). Retiming with Non-Zero Clock Skew, Variable Register and Interconnect Delay. *Proceedings of the IEEE Conference on Computer-Aided Design (ICCAD)*, (pp. 234-241).

Soyata, T., Ba, H., Heinzelman, W., Kwon, M., & Shi, J. (2013, Sep). Accelerating Mobile Cloud Computing: A Survey. In H. T. Mouftah, & B. Kantarci (Eds.), *Communication Infrastructures for Cloud Computing* (pp. 175-197). Hershey, PA, USA: IGI Global.

Soyata, T., Friedman, E. G., & Mulligan, J. H. (1997, January). Incorporating Interconnect, Register, and Clock Distribution Delays into the Retiming Process. *IEEE Transactions on Comptuer Aided Design of Integrated Circuits and Systems, 16*(1), 105-120.

Soyata, T., Muraleedharan, R., Ames, S., Langdon, J. H., Funai, C., Kwon, M., & Heinzelman, W. B. (2012, May). COMBAT: mobile Cloud-based cOmpute/coMmunications infrastructure for BATtlefield applications. *Proceedings of SPIE*, *8403*, pp. 84030K-84030K.

Soyata, T., Muraleedharan, R., Funai, C., Kwon, M., & Heinzelman, W. (2012, Jul). {Cloud-Vision:} {Real-Time} Face Recognition Using a {Mobile-Cloudlet-Cloud} Acceleration Architecture. *Proceedings* *of the 17th IEEE Symposium on Computers and Communications (IEEE ISCC 2012)*, (pp. 59-66). Cappadocia, Turkey.

TI-MSP430. (n.d.). *Overview for MSP430F1x*. From http://www.ti.com/lsds/ti/microcontrollers_16-bit_32-bit/msp/ultra-low_power/msp430f1x/overview.page

Wang, H., Liu, W., & Soyata, T. (2014, Mar). Accessing Big Data in the Cloud Using Mobile Devices. In P. R. Chelliah, & G. Deka (Eds.), *Handbook of Research on Cloud Infrastructures for Big Data Analytics* (pp. 444-470). Hershey, PA, USA: IGI Global.

KEY TERMS & DEFINITIONS

Public-Key Encryption: An encryption type, where a public key and private key pair are required for the encryption and decryption to work. Public key is known to everyone, whereas the private key is only known to parties that are receiving secret messages through encryption. While encryption is possible by anyone by using the public key, decryption is only possible when the private key is known.

Additive Homomorphism: A type of homomorphism, where addition operations on unencrypted data correspond to the same addition operations in the encrypted version. There are many encryption mechanisms which are additively homomorphic, but not multiplicatively homomorphic.

Multiplicative Homomorphism: A type of homomorphism, where multiplication operations on unencrypted data correspond to the same multiplication operations in the encrypted version. There are many encryption mechanisms which are multiplicatively homomorphic, but not additively homomorphic.

Homomorphic Encryption (HE): A public key encryption algorithm where addition or multiplication operations in the "unencrypted domain" correspond to the identical addition or multiplication operations in the "encrypted" domain, thereby making it possible to compute on encrypted data. However, in HE, both addition and multiplication operations do not necessarily correspond to the same operations in their encrypted version *simultaneously*.

Fully Homomorphic Encryption (FHE): The encryption mechanism that achieve both additive and multiplicative homomorphism. With this type of encryption, it is possible to perform an arbitrary sequence of operations, since addition and multiplication operations are sufficient to represent any form of computation which can be written in terms of these two fundamental operations. Therefore, FHE can be used to perform operations at an arbitrary complexity on encrypted data, without observing the underlying data.

Message: A string of bits constituting a certain amount of information to send. Depending on the type of encryption, bit of this message could have different locations in its encrypted version. A message gets encrypted and decrypted during secure transportation.

Plaintext: The bit stream containing the unencrypted message. In the case of FHE, a plaintext has multiple slots, which correspond to multiple messages, not necessarily the same quantity.

Ciphertext: The encrypted message. In the case of FHE, a plaintext has multiple slots, corresponding to multiple messages, however, the encrypted message (i.e., the ciphertext) bits or the locations of the encrypted bits are no longer recognizable due to the way FHE encrypts each slot.

Single Instruction Multiple Data (SIMD): A computational mechanism, where a single instruction (e.g., ADD) is applied to a set of data, (e.g., 128). This type of computation requires an underlying computer architecture that is suitable for executing such SIMD instructions. Also, the type of problems that are suitable to this type of computation are limited. Image Processing operations fit perfectly to this model.

BGV Scheme: An FHE scheme that was introduced by three researchers, Brakerski, Gentry, and Vainkuntanathan, which introduces many optimizations to the original Gentry FHE scheme. One of the most important optimizations is the adoption of a SIMD-like processing structure.

HElib: An FHE library based on the BGV scheme which introduces SIMD type operation on encrypted data. These operations are ADD, MULTIPLY, SHIFT, and BIT SELECT. By using these SIMD operations, a more generalized set of operations can be performed. SIMD nature was introduced to HElib to drastically speed up the FHE execution, since the original version was too slow to be practical.

Band Matrix: A type of matrix where only the main diagonal and a band of "d" additional diagonals above and below the main diagonal are non-zero. The rest of the matrix is made up of zeros. For example, assume an NxN matrix consisting of N^2 elements, where only the diagonal is non-zero. Then, N elements are non-zero and N^2-N elements are zero. If the "band" extends one diagonal above and below the main diagonal, then, N+2*(N-1) = 3N-2 elements will be non-zero, and N^2-(3N-2) elements will be zero. One can extend the band to the point, where there are N-1 bands, at which point the entire matrix is non-zero (i.e., N^2 elements), and it is no longer a band matrix.

Circuit Model: A computational model, where each operation performed on data elements is represented as one of the circuit elements, such as, XOR gates, AND, OR, NOT gates.

Branching Program: A branching program is a computational model, where the decisions that start from a starting node "s" and end up at a terminal node "t" are modeled as branches.

Directed Acyclic Graph (DAG): A type of graph which does not contain any cycles (i.e., nodes that start and terminate at the same vertex). Each edge in this graph is directed, i.e., it goes from one specific vertex into another specific vertex.

Galois Field 2 - GF(2): The field of integer numbers, containing only 2 numbers, 0 and 1. Any operation performed in this field can only produce one of two values, 0 and 1. For more sophisticated computations spanning far beyond a single bit, a careful planning of the operations is necessary to route the carry from one bit to the neighboring bits.

Determinant of a Matrix: Denoted as det(A) which is the determinant of an NxN square matrix A, det(A) contains crucial information about a matrix. For example, if det(A) is zero, this matrix is singular, i.e., it does not represent a solution that is unique for the underlying system of linear equations. Additionally, the determinant can be used to calculate the value of a vector using the Cramer's rule.

Cramer's Rule: This rule can be used to determine the value of a specific a variable X_i in the systems of equations AX=B, where A is an NxN square matrix, and X^T and B are column vectors. To calculate X_i, it suffices to calculate $\det(A_i)/\det(A)$, where $\det(A)$ is the determinant of the entire A matrix, and $\det(A_i)$ is the determinant of the same A matrix where the i^{th} column is replaced with the B matrix.

Log-Space Computation: In computational complexity, Log-space class constitutes a class of problems requiring log(N) space for N entries.

NC^1 Circuits: In complexity theory, NC class (Nick's class) is a class of problems which can be solved in poly-logarithmic time, $O(\log^c N)$ by using polynomial computational elements, $O(N^k)$. NC^1 specifically implies that, a circuit of depth log(N) can be solved in $O(N^k)$ time.

Electrocardiogram (ECG): The diagram produced by an ECG machine by turning the voltages in the body, produced by heartbeats, into sketches on an ECG paper. This provides quick visual information for the operation of the patient's heart.

ECG Patch: A sensor attached to the patient to collect ECG data. This device attaches to the body, typically close to the heart, and provides data to a recorder and/or monitoring device.

RR Interval: The interval between two adjacent R points in an ECG waveform. Since the heart beat is periodic, 60/RR gives the *heart rate* in beats per minute.

QT Interval: The QT interval of each heartbeat delineates the ventricular recovery phase of the heart.

QTc Value: A corrected version of the QT interval, which adjusts for heart rate. Bazett suggested almost 100 years ago to use $QTc=QT/\sqrt{RR}$. This is known as Bazett's formula. However, an alternative suggestion from Fridericia proved to be more accurate for a wider range of heart rates. Fridericia's equation has the same form as Bazett's, but replaces the square root of RR with the cube root. These two formulas can be written as:

$$QTcB = QT/(RR/sec)^{1/2} \qquad QTcF = QT/(RR/sec)^{1/3}$$

The divisions of RR by 1 second are in place to preserve units between QT and QTc.

Long QT syndrome (LQTS): The symptom of prolonged QT interval. This is an indicator of incorrect heart operation. When corrected QTc values are used, the variation among different people should be small. This allows the definition of "safe" QTc values. Typically, values under 500 ms are acceptable, although 440 ms is typically considered a good QTc value. Patients with values over 500 ms are under risk of serious cardiac hazards.

Torsades de pointes (TdP): A heart failure (arrhythmia) that can cause death. The risk of TdP increases with LQTS.

Telemetric and Holter ECG Warehouse (THEW): A warehouse of ECG data from healthy patients and patients with various cardiac conditions, such as Long QT Syndrome, LQTS1, LQTS2. This data consists of 24 to 48 hour Holter monitoring. In this chapter, ECG data from THEW is used as simulated patient data.