Practical Privacy Preserving Medical Diagnosis using Homomorphic Encryption

Sergiu Carpov,
Thanh Hai Nguyen,
Renaud Sirdey
CEA, LIST
Point Courrier 172, 91191 Gif-sur-Yvette Cedex, France

Gianpiero Costantino,
Fabio Martinelli
IIT-CNR
Via G.Moruzzi 1, 56127 Pisa

Abstract—The use of remote services offered by cloud providers has been popular in the last lustrum. Services allow users to store remote files, or to analyze data for several purposes, like health-care or message analysis. However, when personal data are sent to the Cloud, users may lose privacy on the data-content, and on the other side cloud providers may use those data for their own businesses. In this paper, we present our solution to analyze users’ health-data directly into the Cloud while preserving users’ privacy. Our solution make use of Fully Homomorphic Encryption (FHE) to protect users’ data during the analysis. In particular, we developed a mobile application that offload users’ data into the Cloud, and a Fully Homomorphic Encryption algorithm that processes those data without leaking any information to the Cloud provider. Performed empirical tests show that our FHE algorithm is able to evaluate users’ data in reasonable time proving the feasibility of this emerging way of private-data evaluation.

I. INTRODUCTION

Nowadays, exploiting remote services to store files or to analyze data is becoming more and more an usual fact. Users choose the opportunity to store a file into the Cloud, exploiting a cloud service like Dropbox [1], iCloud [2], One Drive [3], and so on, to have the same file always accessible from any device at any place. Cloud services have in the most of cases free plans that allow users to exploit the “power of the Cloud”. However, data-content is a relevant worry that hits customers. In fact, data are sent to the Cloud and providers may have free access to them. Thus, in recent years software companies have developed programs to increase users’ privacy. An example is Boxcryptor [4], which is a software that encrypts users’ files before sending them to the Cloud. Boxcryptor is fully compatible with the most important clouds providers, and files stored in the Cloud are only accessible to the users since they are the only ones to have the decryption key.

SpiderOak [5] provides a solution in spirit similar to Boxcryptor. SpiderOak is a cloud provider that considers users’ privacy a fundamental property. As Boxcryptor, they developed a client application that encrypts users’ file with a private key, and then the files are sent to the Cloud, which is managed by SpiderOak that, however, does not have the decryption key to access files in clear.

In this paper, we propose a solution to analyze health-users data in the Cloud preserving customers’ privacy. The solution that we propose make use of Homomorphic Encryption to evaluate data content. In fact, data are encrypted with the private key of the user and then sent to the Cloud. Here, the Cloud provides the functionality to elaborate and analyse the data that it gets exploiting the Homomorphic Encryption. So, data evaluation is done directly on encrypted data and results obtained remain still unknown to the Cloud. Only, the user, who sent the data, has the key to decrypt the information given by the data analysis.

To show the effectiveness of our integrated solution. In this paper, we present our method to process users’ data using the Fully Homomorphic Encryption (FHE). In particular, we discuss a use case in which a patient sends health-data to the Cloud, and an algorithm that homomorphically calculates a risk factor of the cardiovascular disease. Then, the result generated by the algorithm is sent to the patient, who is able to know his current health situation.

To show the feasibility of our solution, we developed a mobile application that sends patients’ data to the Cloud. The latter exposes Application Program Interfaces (APIs), which allows clients to communicate with the Cloud, and also to run the algorithm to analyze patient’s data. Empirical tests say that a patient is able to retrieve the result of his health-analysis within 4 seconds with a quite small resulting FHE-ciphertext.

The motivation to use Homomorphic Encryption comes out from the need to analyze health data keeping totally private the content of patients’ data. This approach finds different field of applications. For instance, a Hospital can allow its patients to collect health data, using wearable devices, and send them to the Cloud to be homomorphically processed. This latter step, however, is not traditionally done (i.e., working on clear data) but it is executed on data previously encrypted by the patients. Only after the analysis, a doctor, which was previously chosen by the patient, will be able to read the data-result and communicate the diagnosis to the patient.

The paper is structured as follow: in Section II we begin with a description of Homomorphic Encryption together with components and tools used in this work. Afterwards, Section III and IV describe the deployed cloud architecture and an e-Health application we have developed. In Section V and

* This work has been funded in part by EIT Digital project HC@WORKS.
VI, we finalize the paper by presenting performance results and conclusions.

II. HOMOMORPHIC ENCRYPTION

In a nutshell, Fully Homomorphic Encryption is a new kind of cryptographic techniques, which on top of allowing the scrambling of data in order to protect their confidentiality, also provides the necessary mathematical building blocks for the execution of general algorithms directly on encrypted data. As such, FHE is a unique ground breaking software-only technology allowing to enforce the confidentiality of data when they are manipulated by untrusted servers without decryption and without disclosing any secret to those servers. Since the seminal work of Gentry [6], introducing the first fully homomorphic encryption (FHE) many other simpler and more efficient schemes have been proposed [7]–[9]. The spectrum of applications of homomorphic encryption is rather large, in particular in the domain of cloud-based applications [10]–[12]. The ciphertext and plaintext in FHE schemes are either integer or polynomial ring elements. According to the literature the schemes over polynomial rings are asymptotically more efficient than the schemes based on integer rings [13]. If the ciphertext sizes in both cases are roughly the same then the computations are heavier and the additional data (public and evaluation keys) have larger sizes for schemes over integer rings. In return the learning with errors (LWE) problem, on which is based the security of integer ring schemes, is better understood than the Ring-LWE problem [14]. The majority of FHE schemes have a common set of operations: parameter generation, key generation, encryption/decryption, homomorphic addition and multiplication of ciphertexts. Addition and multiplication operations can also be performed with one non-encrypted input, in this case the homomorphic operations are much lighter. Usual applications of homomorphic encryption use Boolean plaintext space. In this case the homomorphic addition corresponds to logical XOR and homomorphic multiplication to logical AND. Such a fully homomorphic encryption scheme allows to execute any Boolean circuit directly on encrypted data.

FHE schemes are necessarily probabilistic and this means that some noise components is added in the encryption process to make sure that each plaintext message has a large number of possible ciphertexts. This is a fundamental property that is necessary to provable semantic security. Still, one of the issue is that with FHE, ciphertexts are intrinsically significantly larger than plaintexts. Primarily for this necessity of noise add, all known FHE are intrinsically unstable: the noise amplitude grows with the homomorphic calculations until decryption is no more possible. Usually, noise growth is faster with multiplications than with additions. That is why many authors consider only the multiplicative depth\(^1\) of evaluated circuits when FHE schemes are parametrized. The multiplicative depth is of the utmost importance: the higher the multiplicative depth is, the larger the parameter of the FHE systems are (with direct impact on the performances).

It is important to recall the security model underlying the use of FHE. Indeed, in the most basic settings, two parties are involved: the user (owner of some private data) and the server (owner of an algorithm and possibly some data which it is willing to inject in the calculation). The issue that is addressed by FHE is that of protecting the confidentiality of user data with respect to threats coming from the server. As such, in proper cryptographic terminology, we are in the so-called honest-but-curious server setting. In particular, the server has complete control over the algorithm it executes, so the user must trust that the server will perform consistently to a functional specification although it has no access to the algorithm details. Also, the degree to which algorithms or server data privacy is achieved with respect to threats from the user depends on the functional specification of the algorithm itself. So the model is not symmetric with respect to security guarantees.

A. Armadillo compilation chain

A FHE cryptosystem with binary plaintext space can be described by the following API:

- \( \text{enc}_{pk} : \mathbb{Z}_2 \to \Omega \)
- \( \text{dec}_{sk} : \Omega \to \mathbb{Z}_2 \)
- \( \text{add}_{pk} : \Omega \times \Omega \to \Omega \)
- \( \text{mul}_{pk} : \Omega \times \Omega \to \Omega \)

Here, \( \Omega \) is a large cardinality set (e.g. \( \mathbb{Z}_q, \mathbb{Z}_q[X] / \phi(X) \) etc.). Key properties for all \( m_1 \in \mathbb{Z}_2 \) and \( m_2 \in \mathbb{Z}_2 \):

- \( \text{dec}_{sk} (\text{add}_{pk} (\text{enc}_{pk} (m_1), \text{enc}_{pk} (m_2))) = m_1 \oplus m_2 \)
- \( \text{dec}_{sk} (\text{mul}_{pk} (\text{enc}_{pk} (m_1), \text{enc}_{pk} (m_2))) = m_1 \otimes m_2 \)

Due to the low-level formalism of FHE, a compiler chain targeting FHE execution has to manipulate Boolean circuits. That is a directed graph \( G = (V, A) \) which vertices are either inputs, outputs or operators (XOR, AND) and which arcs corresponds to data transfers. The following constraints are imposed to a compiler targeting FHE execution:

- No ifs (unless regularized by conditional assignment).
- No data dependant loop termination (need upper bounds).
- Array dereferencing/assignment in \( O(n) \) (vs \( O(1) \)).
- Algorithms always realize (at least) their worst-case complexity!
- Can handle only a priori (multiplicative) bounded-depth programs.
- Multiplication (AND gate) is much more expensive than addition (XOR gate).

The Armadillo compilation chain [15] provides an easy to use compiler which builds a privacy-preserving binary for an application written in a high-level language using homomorphic encryption as back-end. The compilation chain is composed of 3 layers: a front-end, a middle-end and a back-end. The front-end transforms code written in C++ into its Boolean circuit representation. The middle-end layer optimizes the Boolean circuit produced by the front-end using ABC. ABC is a well-known open source System for Sequential

\(^1\)Multiplicative depth is the number of sequential homomorphic multiplications which can be done on freshly encrypted ciphertexts in order to be able to decrypt and retrieve the result of multiplications.
Synthesis and Verification developed by the Berkeley Logic Synthesis and Verification Group [16]. The back-end constructs a binary which homomorphically executes the Boolean circuit over encrypted data. Two FHE are addressed by the Armadillo compiler: (i) an in-house implementation of [8] and (ii) the publicly available library HElib [17], [18].

B. Efficient communication towards FHE-enabled servers

As already emphasized, there are a number of issues with respect to transmitting FHE-encrypted data, mainly: FHE-encryption is a computationally heavy operation and FHE-encrypted data are much larger than their associated plaintexts. However, FHE is quite powerful and allows to perform a trick known as trans-ciphering, by means of which it is possible to switch from some data encrypted with some cryptosystem (e.g. a classic overhead-free symmetric cryptosystem) to the same data encrypted under FHE, without this data to ever be in clear form. If AES is used as system for encrypting messages, then the trans-ciphering from AES to FHE can be performed like:

\[
\text{AES}^{-1}([x]_{\text{key}})_{\text{FHE}} \cdot [\text{key}]_{\text{FHE}} = [x]_{\text{FHE}}
\]

This corresponds to the homomorphic execution of the AES decryption circuit on AES encrypted data using an FHE encryption of the AES key. Homomorphically executing an AES decryption still takes a lot of time (18 minutes per decryption were reported in [15]) as the algorithm has a multiplicative depth of 40, which is quite large.

To cope with this, stream ciphers have recently been acknowledged as more appropriate candidates for practical trans-ciphering. Indeed, when a block-cipher usually is a relatively low degree function iterated a significant number of times (e.g. 10 times for AES-128 or more to the notable exception of Prince and the more recent LowMC), a design which is intrinsically not FHE-friendly, stream ciphers (when not based on block-ciphers) follow different design patterns, some of them friendlier for efficient FHE execution.

What we need is a stream cipher where keystream bits must be multiplicatively bounded. This is the case if keystream bits are independent by chunks. Also, when using a stream cipher, keystream bits can be homomorphically “mined” independently of the data. Hence, trans-ciphering induces almost no latency (homomorphic XORs are fast) as long as keystream mining has been done in advance. So, the most appropriate pattern appears to be the use of an IV-based stream cipher in counter mode. In this context, the authors of [19] analysed and proposed stream ciphers which are by design “friendlier” for efficient FHE execution. They found that the Trivium algorithm is a very good candidate as a respected 80-bits key lightweight stream cipher. Still, in order to increase the overall security to 128-bits, they have proposed a design of a 128-bits key extension of Trivium, Kreyvium, which also retains the FHE-friendliness. Using trans-ciphering for FHE computations in a cloud environment has the following key advantages:

- Avoid the computational burden of FHE-encryption on the client device.
- Avoid the intrinsic bandwidth inflation of transmitting FHE-encrypted data from the device.
- Almost transparently interface the client device with the remote “cryptocomputer”.
- Almost standard cryptography on the client side.

Although trans-ciphering allows to send data to the Cloud at no overhead, receiving computation results from the Cloud must be done using FHE ciphertexts (with the incurred FHE ciphertext size overhead). Still, FHE message packing techniques described in [10], [20] can be used to pack several homomorphic ciphertexts into a single one.

III. Health Data Analysis as Cloud Service: Architecture

In this section we introduce the architecture that we use to analyze patients’ data in the Cloud. Data are directly submitted by patients to the Cloud since it provides the homomorphic evaluation as Software as a Service (SaaS). Services are exposed through APIs that can be remotely invoked to execute specific operations, like submitting patients’ data. Thus, the Hospital and patients do not have to worry about confidentiality of the information. In fact, patients’ data are encrypted before sending them to the Cloud. So, the Cloud provider does not have access to clear data and cannot infer anything from the data it stores and processes.

Fig. 1 shows a top-level flow of the health-data from the moment in which they are submitted, processed and sent back
to the user. In our architecture, we assume that a patient will use a client application in which she sets up values about her current health status, for example blood pressure level. Then, the application offloads patient’s data into the Cloud, exploiting a specific API (Step 1). Here, patient’s data are stored in the Cloud and they are ready to be processed using an algorithm that homomorphically evaluates the data (Step 2). When the homomorphic algorithm completes its job, results of the evaluation are sent back to the patient (Step 3).

A. Architecture Deployment

As we presented above, our reference scenario focuses on cloud as SaaS. In particular, the homomorphic encryption functions are hosted in the Cloud and are remotely accessible thought APIs. The Hospital provides to the patient a mobile application to offload health-data into the Cloud. Only after homomorphically evaluating the data, results are sent back to the application to show the current health status of the patient. The Cloud that we deployed is composed by a single Virtual Machine with 8 cores, 16 GByte of RAM and 300 Gbyte of storage. The operating system used is an Ubuntu Server 14.04 LTS [21], while APIs are developed using the RESTful [22] technology combined with the web-server JBOSS 7.1 [23]. APIs are built as simple HTTP or HTTPTs calls that can be executed by a client specifying the Cloud-IP plus the signature of the API to call. Moreover, when a client calls a remote API, it must specify the kind of request of the HTTP call, e.g., GET, POST and so on, plus the typology of submitted and received data, e.g. JSON format. APIs are developed in JAVA [24] and they wrap-up the implemented services exposed by the Cloud, such as our algorithm that evaluates the health data in the fully homomorphic encryption context.

B. Hardening the Cloud

The Cloud we set up to execute homomorphic encryption functions is totally exposed to the Internet and as consequence to security threats. For this reason, we implemented some solutions to protect it from malicious actions. Our protection focus on data confidentiality, cloud protection and authentication. Regarding confidentiality, each API invocation is performed using the HTTP protocol combined with the Transport Layer Security (TLS) [25]. HTTPs communications allow clients and Cloud to protect messages exchanged between them and to avoid that messages could be eavesdropped by a potential attacker. Moreover, the use of TLS entails a digital certificate that the server must expose before establishing a connection with a client. The certificate proves the real identity of the Cloud since our certificate is issued by a trusted Certification Authority (CA). Thus, the use of a digital certificate reduces the possibility that an attacker may perform the Man-in-the-Middle (MiM) attack. In fact, the client is able to verify the validity of the issued certificate of the Cloud, and it can communicate directly with the desired remote identity.

The Firewall is an additional protection block that we applied to our cloud to reduce the number of security threats. As described before, APIs communicate using the HTTPs protocol, this means that the Cloud manages incoming connection only through the 8443 port. Another additional port that we use for management purpose is the 22, i.e. Secure Shell (SSH) [26]. So, we configured the IpTables [27] Firewall to open incoming connection only from the 22 and 8443 ports. Moreover, to reduce as much as possible the attempt of brute-force attack2, we decided to configure our Firewall to block persistent attempt of connections coming from the same user that fails.

Finally, to authenticate client devices that perform connections to the Cloud, we adopt the basic authentication technique. Here clients, which attempt the connections, use credentials made of username:password during the authentication phase. In this way, we reduce the risk of exposing APIs functionalities to clients that are not authorized to use them.

IV. HEALTH DATA ANALYSIS AS CLOUD SERVICE: APPLICATION

The cloud provides a privacy preserving cardiac risk factor assessment service based on homomorphically encrypted user health data. Fig. 3 presents an overview of deployed cardiac risk factor algorithm. In the following sections we shall describe the execution phases of the algorithm.

A. Cardiac risk factor algorithm

The user fills a medical data form on the device and the Cloud runs an representative application of a cardiac risk factor assessment. The algorithm represents a set of rules which when activated increment the risk factor. The rules used in the algorithm are shown in Fig. 2. The result of the algorithm is an integer between 0 and 9, where larger value means a bigger risk factor of cardiology disease. This algorithm was implemented in C++ and compiled using the Armadillo compilation chain. The underlined terms in Fig. 2 are manipulated in encrypted form by means of specific variable types (i.e. “encrypted” data types) in Armadillo. Although its relatively modest in terms of computation, this algorithm has been chosen because it represents a certain reality in an e-Health context: individual-centric relatively simple algorithms manipulating intrinsically sensitive data.

An Android tablet is used as a client device. The Android tablet holds a (secret) symmetric encryption key SYMK and the secret key of the FHE scheme FHESK, which allows the client device to decrypt the results generated by the homomorphic cloud. The cloud (optionally) holds the FHE scheme public key FHEPK as well as a public token needed for trans-ciphering that corresponds to the symmetric secret SYMSK encrypted under the public key of the FHE scheme [SYMSK]FHE, recall Sec. II-B.

B. Trans-ciphering

The medical data from the user is encrypted using a keystream generated by Kreyvium cipher with the private key

2Here, an attacker tries all possible combinations of login credentials to obtain the access to the resource.
1: +1 if man and age > 50 years
2: +1 if woman and age > 60 years
3: +1 if cardiology disease in family history
4: +1 if smoking
5: +1 if diabetic
6: +1 if high blood pressure
7: +1 if HDL cholesterol less than 40
8: +1 if weight > height-90
9: +1 if daily physical activity less than 30 minutes
10: +1 if man and alcohol consumption more than 3 glasses/day
11: +1 if woman and alcohol consumption more than 2 glasses/day

Fig. 2. The set of rules used in the cardiac risk factor assessment algorithm.

SYM_{SK} and an initialization value (IV). The encryption is simply the XOR between medical data and the corresponding key-stream. On the Cloud side, the Kreyvium algorithm is executed homomorphically using the FHE encrypted private key $[SYM_{SK}]_{FHE}$ and the same IV. In this way, the client side and the Cloud side will have the same key-stream except that on the cloud it will be in the FHE domain. The Kreyvium cipher algorithm was implemented in Armadillo also and the Armadillo runtime was used to run it. We shall note that the same C++ code was used on the user side in order to generate the key-stream, but this time natively compiled and executed.

This operation on the Cloud is called key-stream mining and is performed in advance in order to hide the latency of Kreyvium homomorphic execution. Mined key-stream are stored on the Cloud for further use. In our current implementation we store a backlog of key-stream which allows to execute one further cardiac risk computation. The synchronization between user and cloud IVs is done once when the user registers for the Cloud service. Subsequently the IV is incremented on both sides after a computation. One can also imagine other synchronization possibilities (e.g. hashing the last IV). In any case the IV is a public information and can be send to the Cloud as plaintext without compromising system security.

The first time the user wants to use cloud service for computing his risk factor a registration to FHE services phase must be accomplished (after the authentication phase described in Sec. III). The user sends to the Cloud its Kreyvium secret key encrypted with FHE to the Cloud. A zero initialization value for Kreyvium is used by both sides.

C. Algorithm execution details

Once the Cloud receives user data encrypted using a key-stream generated by the Kreyvium cipher, the Cloud looks up for the corresponding key-stream computed homomorphically and XORs homomorphically the user data with it. In this way the cloud obtains user medical data encrypted homomorphically. User data is fed to the cardiac risk factor computation algorithm described earlier and it is executed using the Armadillo runtime. The result of the computation (risk factor) is represented by 4 ciphertexts, one per bit of risk factor. We recall that homomorphic encryption allows to execute only boolean operations that is why we need a compiler in order to transform C++ code into its boolean representation. The resulting ciphertexts are packed into a single one using the coefficient packing method described in [10], [20].

The packed ciphertext result is sent back to the user’s tablet which decipheres it using FHE scheme secret key FHE_{SK}. The deciphered ciphertext is a polynomial whose 4 lowest degree coefficients are the bits of risk factor. The application reconstructs the risk factor from these bits and shows it to the user.

D. Third-party result receiver

In the use case described above the owner of the data is the same as the recipient of homomorphic algorithm results. One can imagine another use case in which the recipient of results is not the owner of input data. For example, a doctor which accesses the result of a computation performed on his patient’s data. It is easy to adapt our cloud architecture to this case. The only thing to change is the initialization phase. The patient, instead of providing to the cloud the symmetric key under in his FHE domain, will send to the Cloud his symmetric key encrypted with the FHE key of his doctor. The trans-ciphering process transforms patient’s data to the FHE domain of the doctor he had previously authorized.

This process can be seen as an authorization given by a patient to a doctor for accessing results of algorithms applied on his data. In the security model for this use case the Cloud and the doctor must not collude, otherwise the Cloud will be able to decrypt patient user data having access to doctor’s FHE secret key.

In the next section we describe our results in terms of network traffic and computing power utilization.

V. IMPLEMENTATION DETAILS AND PERFORMANCE

As a summary, the demonstrator platform described previously executes the following technical scenario:

- Algorithm implementation, compilation and deployment on the server.
- Homomorphic pre-calculation of Kreyvium keystream on the server.
- The Android tablet sends the Kreyvium-encrypted private user health data.
- The server receives and homomorphically “trans-ciphers” to FHE.
- The server homomorphically executes the diagnostic algorithm and sends back the encrypted answer to the tablet.
- As the FHE secret key owner, the tablet is the only party able to decrypt and thus interpret the server reply.

The Client side Android application was developed in Java. Fig. 4 presents a print-screen of the application and in particular the form in which the user introduces his data.

In our implementation (server side) we have used the FHE scheme [8] implemented in Armadillo. We have configured
it to 128-bits of security. The Kreyvium [19] stream cipher used for trans-ciphering has a 128-bits key length. In this configuration the Cloud service provides end-to-end 128-bits security.

Kreyvium stream cipher has a multiplicative depth of 12. The cardiac risk factor algorithm has 8 multiplicative levels. The FHE scheme we use was dimensioned to support 20 levels of multiplicative depth, in order to allow error free decryption of algorithm results. A FHE ciphertext has size approximately 250 kB in our implementation.

During the first interaction between the user and the Cloud, the user sends the key $|SYM_{SK}\rangle_{FHE}$ to the Cloud. This corresponds to 128 FHE ciphertexts (1 ciphertext per Kreyvium key bit). A total of $\sim 32$ MB are sent to the Cloud during this phase. We did not used any packing method as this step is performed only once.

The next times the user sends data to the Cloud Kreyvium key-stream is employed for encrypted health data. There is no expansion in the sent data size. Once the homomorphic computations are done a single ciphertext ($\sim 250$ kB) is received by the user from the Cloud with algorithm results. The total response time from the moment the user tablet sends the data to the moment the computation result is received and decrypted is less than 4 seconds, of which the homomorphic execution of risk factor algorithm takes $\sim 3.2$ seconds. We shall note that the FHE encryption was implemented in Java on the Android platform without any particular optimizations – if needed, a native implementation would be much more efficient). The homomorphic execution of the Kreyvium algorithm takes approximately 8 minutes (on the hardware configuration we use), that is why it is interesting to perform the key-stream mining in advance.

VI. Conclusion

In this paper, we have reported on one of the first attempt to build an operational cloud platform prototype embedding Fully Homomorphic Encryption to intrinsically ensure user data confidentiality with respect to threats from the platform itself. In particular, our platform integrates the state-of-the-art components (trans-ciphering, automatic compilation and parallelisation, as well as message packing) that allows to get homomorphic encryption much closer to practice.

Still, in its present form, our integrated platform is suitable for either applications that are strongly dis-symmetric in terms of uplink and downlink volumes of exchanged data or applications that simply do not send any results directly back to the client device (which is often the case in e-Health scenarios). Nevertheless, our platform is a first step in demonstrating the feasibility of operationally deploying homomorphic cryptography in order to address lightweight-enough yet real-world applications in which guaranteeing the confidentiality of sensitive data justifies the increased computing cost.
REFERENCES