Universität Heidelberg

Ist künstliche Intelligenz gefährlich?

SEMINAR

Homomorphic Encryption

Carine Dengler

SoSe17

Table of Contents

Introduction

Over the course of the seminar it has emerged that wast amounts of data are needed for an Artificial Intelligence to learn and evolve. However, not only has it become clear that providers, such as Yahoo! [yah] are unable to protect their users' data, in light of the massive governmental surveillance programs, the question has been raised to what extend they are (willingly or unwillingly) actively participating in undermining their users' privacy [nsa]. The only possible conclusion to this is to consider any and all data that is not secured by oneself or a trusted intermediary to be possibly compromised and at risk of public disclosure at any moment. To mitigate these threats to one's privacy, the most secure way is to encrypt as much data as possible. Doing so would however make it impossible to mine massive datasets, such as medical records, for possibly benevolent purposes, such as scientific studies. Therefore, the development, improvement and use of techniques that allow for processing encrypted data are becoming more and more important. The term paper presents two of these techniques, namely fully homomorphic encryption [Gen10] and CryptDB [PRZB11].

Cryptography

[KL08] defines (modern) cryptography as

> *the scientific study of techniques for securing digital information, transactions, and distributed computations.*

Cryptographic techniques are used to protect confidentiality by denying an unauthorised third party access to the data [Buc16, PS17].

Encryption Schemes

Encryption schemes provide secret communication between two parties. There are symmetric and asymmetric encryption schemes, which differ in the number and purpose of keys used [KL08, Buc16].

Symmetric Encryption Scheme A **symmetric encryption scheme** is a tuple $(K, P, C, KeyGen, Enc, Dec)$ where

- K is the set of **keys** k, and is called the **key space**
- $KeyGen$ is a probabilistic algorithm that, given a **security parameter** λ, generates a key $k \in K$ such that the bit-length $|k| \geq \lambda$
- P is the set of **plaintexts** m, and is referred to as the **plaintext space**
- C is the set of **ciphertexts** c, and is called the **ciphertext space**
- Enc is the **encryption algorithm** that, given a key k encrypts a plaintext m such that $Enc(k, m) = c$ with $c \in C$

- Dec is the deterministic **decryption algorithm**, that, given a ciphertext c and a key k decrypts c such that $Dec(k, c) = m$ with $m \in P$
- for every $k = KeyGen(\lambda)$ and $m \in P$ it holds that $Dec(k, Enc(k, m)) = m$

Enc may be either a deterministic or a probabilistic algorithm. If Enc is deterministic, the same plaintext is deterministically mapped to the same ciphertext each time Enc is executed on it, whereas if Enc is probabilistic, the same plaintext may be mapped to one of several different ciphertexts, out of which one is randomly chosen. A deterministic encryption scheme is considered to be less secure than a probabilistic encryption scheme, given that an attacker can determine whether a given ciphertext encrypts a given plaintext by running Enc.

To send Bob an message m, Alice uses the **secret key** k to encrypt it, and sends the resulting ciphertext c to Bob. Upon receiving the ciphertext, Bob in turn uses k to decrypt c which yields the plaintext message m. Since k is used to both encrypt and decrypt (hence the name **symmetric** encryption scheme), Alice and Bob have to share it beforehand. The Diffie-Hellman key exchange is an example for a method that can be used to share keys in a secure manner over an insecure channel [Buc16, Eck14, Gen10, KL08, PRZB11].

Asymmetric Encryption Scheme An asymmetric encryption scheme is closely related to a symmetric encryption scheme and is a tuple $(K, P, C, KeyGen, Enc, Dec)$ where

- given a security parameter $\lambda \in \mathbb{N}$, $KeyGen$ generates a pair of keys $(pk, sk) \in K$ such that $|pk|$ and $|sk| \geq \lambda$
- pk is called the **public key**, whereas sk is called the **secret key**
- Enc is the encryption algorithm that, given a key pk encrypts a plaintext $m \in P$ such that $Enc(pk, m) = c$ with $c \in C$
- Dec is the deterministic decryption algorithm that, given a secret key sk decrypts a ciphertext $c \in C$ such that $Dec(sk, c) = m$ with $m \in P$
- for every key pair $(pk, sk) = KeyGen(\lambda)$ and $m \in P$ holds that $Dec(sk, Enc(pk, m)) = m$

As with a symmetric encryption scheme, Enc is either a deterministic or a probabilistic algorithm.

Alice now uses Bob's public key pk to encrypt the message m, and sends the resulting ciphertext c to Bob. Bob then decrypts c using his secret key sk to get the original message m. Given that now the public key pk is used to encrypt, while the secret key is used to decrypt (hence **asymmetric** encryption scheme), Bob only has to keep sk secret, while he may openly distribute pk to anyone wishing to communicate confidentially with him [Buc16, KL08].

Homomorphic Cryptography

Motivation

Widespread use of cloud computing raises the question whether it is possible to delegate processing of data without giving access to it. Encrypting one's data with a conventional encryption scheme to protect one's privacy seems to undermine the benefits of cloud computing, since it is impossible to process the data without the decryption key. However, encrypting the data with a so-called **homomorphic encryption scheme** allows for (some) meaningful manipulations on the encrypted data. A **fully homomorphic encryption scheme** on the other hand imposes no limitations on the manipulations that can be performed [Gen10].

Fully Homomorphic Cryptography

A given (asymmetric) encryption scheme is extended by an algorithm *Eval* and a set $\mathcal{F}$ of **permitted functions** f such that

$$\begin{aligned}&\text{if } Eval(pk, f, c_1, \dots, c_n) = c \text{ then } Dec(sk, c) = f(m_1, \dots, m_n)\\&\text{with } c_i = Enc(pk, m_i),\ m_i \in P,\ i = 1, \dots, n,\ n \in \mathbb{N}\end{aligned}$$

A function $f \in \mathcal{F}$ can be **handled** by the encryption scheme. *Eval* is generally undefined for any function $f \notin \mathcal{F}$. The **compact ciphertexts** requirement states that the amount of computation to decrypt c as well as the size of c are completely independent of f. The encryption scheme is said to be **fully homomorphic** if it can handle any function f, fulfills the compact ciphertexts requirement and *Eval* is efficient, i.e. polynomial in the security parameter $\lambda \in \mathbb{N}$ [Gen10].

Complexity

The complexity of *Eval* is on the one hand dependent on the security parameter λ and on the other hand on the complexity of the function f that is being evaluated. The complexity of f can be measured as the number of steps T_f of a Turing machine that computes f. Furthermore, if f can be computed in T_f steps on a Turing machine, it can be expressed as a boolean circuit with $S_f \cong T_f$ gates. These gates can be evaluated by adding, subtracting and multiplying mod2:

$$\begin{aligned}&\forall x, y \in \{0, 1\}\\&\texttt{AND}(x, y) = x \cdot y\\&\texttt{OR}(x, y) = 1 - (1 - x)(1 - y)\\&\texttt{NOT}(x) = 1 - x\end{aligned}$$

Eval's runtime is at least linear in the number of input ciphertexts n, since it has to touch all of them to prevent leaking information about the underlying plaintexts' relation to f. In addition, given that f is represented as a fixed circuit, the number of output wires and therefore the size of the output must be fixed in advance, so that the output must be either truncated or padded [Gen10].

Somewhat Homomorphic Encryption Scheme

To construct an encryption scheme that can handle a limited set of permitted functions $\mathcal{F}$, the symmetric encryption scheme is modified such that *KeyGen*, *Enc* and *Dec* operate as outlined below.

$$KeyGen(\lambda) = p \text{ s.t. } p \text{ is a } \lambda^2\text{-bit odd integer}$$

For a bit $m \in \{0, 1\}$,

$$Enc(p, m) = m' + pq$$

s.t. m' is a random λ-bit integer with $m' = m \bmod 2$, and q is a random λ^5-bit number.

$$Dec(p, c) = c \bmod p \bmod 2$$

$c \bmod p$ is the **noise** associated to the ciphertext c; if c is a ciphertext output by *Enc* (in contrast to a ciphertext output by *Eval*), it is called a **fresh** ciphertext, since the associated noise is small (λ). Given such a fresh ciphertext, the decryption algorithm *Dec* is correct because $c \bmod p = m'$, which has the same parity as m.

The encryption scheme supports the operations

$$\begin{aligned} Add(c_1, c_2) &= c_1 + c_2 \\ Sub(c_1, c_2) &= c_1 - c_2 \\ Mult(c_1, c_2) &= c_1 \cdot c_2 \end{aligned}$$

such that to evaluate a boolean function f with input ciphertexts $c_1, \ldots, c_n$

1. f is expressed as a boolean circuit C with `XOR` and `AND` gates
2. the circuit $C^\dagger$ is constructed by replacing the `XOR` and `AND` gates in C by addition and multiplication gates
3. the multivariate function $f^\dagger$ corresponding to $C^\dagger$ is computed on the ciphertexts $c_1, \ldots, c_n$ and its result c is output

It must however be pointed out that the *Add*, *Sub* and *Mult* operations increase the noise associated to the output ciphertext. For ciphertexts $c_1, \dots c_n$ with associated noises $m'_1, \dots, m'_n$ and some integer q'

$$f^\dagger(c_1, \dots, c_n) = f^\dagger(m'_1, \dots, m'_n) + pq'.$$

If $|f^\dagger(m'_1, \dots, m'_n)| < \frac{p}{2}$, then

$$\begin{aligned}&Dec(p, f^\dagger(c_1, \dots, c_n)) \\ &= f^\dagger(c_1, \dots, c_n) \bmod p \bmod 2 \\ &= f^\dagger(m'_1, \dots, m'_n) \bmod 2 \\ &= f(m_1, \dots, m_n).\end{aligned}$$

Whereas *Dec* is no longer correct if the noise $|f^\dagger(m'_1, \dots, m'_n)|$ is greater than $\frac{p}{2}$. Therefore, the encryption scheme can only handle the functions f for which $|f^\dagger(m'_1, \dots, m'_n)|$ is always less than $\frac{p}{2}$ if $c_1, \dots, c_n$ are of bit-length of at most λ bits [Gen10].

Somewhat Homomorphic Asymmetric Encryption Scheme

To use the somewhat homomorphic scheme to construct a fully homomorphic scheme, it is turned into an asymmetric encryption scheme such that

- the secret key *sk* is p
- the public key *pk* is a list of encryptions of 0 with length polynomial in λ
- *Enc* encrypts $m \in \{0, 1\}$ by adding a random subset sum of *pk* to m

If the ciphertexts in *pk* have small enough noise, the ciphertext resulting from *Enc*(*pk*, m) will also have small enough noise and *Dec* is still correct [Gen10].

Bootstrappable Encryption Scheme

An homomorphic encryption scheme that is able to handle its own decryption function is said to be **bootstrappable**. As will be seen further below, this property can be used to construct a fully homomorphic encryption scheme.

The decryption algorithm *Dec* of the somewhat homomorphic asymmetric encryption scheme is

$$\begin{aligned}Dec(p, c) &= (c \bmod p) \bmod 2 \\ &\equiv \texttt{LSB}(c)\ \texttt{XOR}\ \texttt{LSB}(\lfloor \frac{c}{p} \rceil)\end{aligned}$$

where $\lfloor \cdot \rceil$ rounds to the nearest integer. Given that both `LSB` and `XOR` are trivial to implement in a boolean circuit, the encryption scheme is bootstrappable if it can handle the function $f = \lfloor \frac{c}{p} \rceil$. The noise of the output ciphertext of $c\frac{1}{p}$ however is too large, so that to get a bootstrappable encryption scheme, the decryption algorithm has to be simplified.

To this end, a new key generation algorithm $KeyGen^*$ with parameters α and β is introduced:

1. $KeyGen(\lambda)$ is run to get key pair (pk, sk)
2. a set $Y = \{y_1, \ldots y_\beta\}$ of rational numbers in $[0, 2)$ is generated such that there is a subset $S \subset \{1, \ldots, \beta\}$ of size α such that $\sum_{i \in S} y_i \approx \frac{1}{p} \bmod 2$
3. sk^* is set to the vector $\in \{0, 1\}$ with Hamming weight α that encodes S
4. pk^* is set to (pk, Y)

$KeyGen^*$ therefore includes a hint (the set Y) about the secret integer p to the key pair output by $KeyGen$.

The hint Y introduced above is now used by the new encryption algorithm Enc^* to postprocess a ciphertext c output by Enc:

1. run $Enc(pk, m)$ to get ciphertext c
2. for $i \in \{1, \ldots, \beta\}$, $z_i = cy_i \bmod 2$, keeping only $\sim \log \alpha$ bits of precision after the binary point
3. set c^* to $(c, \vec{z})$ with $\vec{z} = \langle z_1, \ldots, z_\beta \rangle$

Finally, the new decryption function Dec^* has less remaining work due to the postprocessing of c:

$$Dec^*(sk^*, c^*) = \texttt{LSB}(c) \texttt{ XOR } \texttt{LSB}(\lfloor \sum_i s_i z_i \rceil)$$

If α is set small enough, the encryption scheme can handle $\lfloor \sum_i s_i z_i \rceil$ and is therefore bootstrappable [Gen10].

Fully Homomorphic Encryption Scheme

As has been mentioned, the accumulation of noise eventually causes the decryption function to no longer be correct. For the encryption scheme to be fully homomorphic, it must therefore be able to keep the noise from getting large enough for this to happen.

Given distinct keys (pk_i, sk_i) with $i = 1, \ldots$ and the encryption c_1 of the bit m under pk_1, as well as the vector of the encryptions of the bits of sk_1 under pk_2 $\bar{sk_1}$

$$Recrypt(pk_2, Dec, \bar{sk_1}, c_1) = c$$

such that

$\bar{c_1}$ is the vector of the encryptions $Enc(pk_2, c_{1j})$
over the bits c_{1j} of c_1 and
$c = Eval(pk_2, Dec, \bar{sk_1}, \bar{c_1})$.

Since the encryption scheme is bootstrappable, *Recrypt* outputs a new encryption of m under the public key pk_2 while removing the inner encryption. Decrypting the inner encryption removes the noise associated to it, but *Recrypt* simultaneously adds new noise. If the newly added noise is however less than the noise that has been removed and is still small enough to allow performing another operation, the encryption scheme can be made fully homomorphic.

To this end, *Dec* is augmented by a further gate (*Add*, *Mult*, *Sub*). The public key of the encryption scheme is composed of a series of public keys pk_1, pk_2 and the encryptions of the associated secret keys $\bar{sk_1}, \bar{sk_2}, \ldots$ such that $\bar{sk_i} = Enc(pk_{i+1}, sk_i)$. Any function f may now be represented as a circuit whose gates are arranged into levels that are being stepped through sequentially.

The modified somewhat homomorphic asymmetric encryption scheme is finally assumed to also be **circular-secure**, which means that it is safe to reveal the encryption of the secret key sk under pk. That way, the public key can be composed of a single public key pk for all levels as well as a single secret key $\bar{sk}$ encrypted under pk instead of distinct public keys pk_i for each circuit level i and a chain of encrypted secret keys sk_i. Without this property, the complexity of the key generation algorithm would be growing linearly with the maximum circuit depth, which violates the requirement that it is efficient, that is to say, polynomial in the security parameter λ [Gen10].

Summary

The answer to the question whether it is possible to delegate processing data without giving access to it may is mixed. Although it has been shown that arbitrarily processing encrypted data is possible in theory, it is prohibitively expensive and therefore unfit for practical use [Gen10, PRZB11].

CryptDB

Motivation

Applications using database management systems (DBMS), in particular online applications, are vulnerable to data theft, be it through an malicious database administrator or an attacker exploiting vulnerabilities in the underlying software. CryptDB provides data confidentiality for such applications [PRZB11].

Threat Models

Two threats are addressed; the first threat is an attacker with administrative access to the DBMS, such as, but not limited to, the database administrator. The attacker is assumed to be passive; they change neither issued queries, nor query results or data in the DBMS. The second threat is an attacker that not only gains control over the DBMS, but possibly over the application server and the proxy as well [PRZB11].

Architecture

The typical architecture of database-backed applications, namely an DBMS server and a separate application server, is extended by a database proxy. The proxy intercepts and rewrites all SQL queries the application server issues so that the DBMS can execute them on encrypted data. The proxy anonymizes the table and column names and encrypts the constants. It then sends the query to the DBMS server, from which it afterwards receives the encrypted query results, which it decrypts and sends to the application [PRZB11].

SQL-aware encryption strategy

CryptDB takes advantage of the fact that a SQL query is composed of a set of well-defined primitive operators to encrypt data in a way that allows the DBMS to execute the queries using standard SQL [PRZB11].

Random (RND) RND is a probabilistic encryption scheme. While it provides the maximum security of all the encryption schemes used in CryptDB, it does not allow any computation to be performed efficiently on the ciphertext.

> *For example, if the application issues no queries that compare data in a column, or that sort a column, the column should be encrypted with RND.*

suggests however that a `SELECT` query without a filter can be executed nonetheless [PRZB11].

Deterministic (DET) DET is, as the name suggests, a deterministic encryption scheme. As such it provides a slightly weaker security than the probabilistic RND, since it generates the same ciphertext for the same plaintext and consequently leaks which encrypted values correspond to the same data item. This on the other hand allows to perform equality checks on the encrypted values and therefore allows to execute queries such as e.g. `SELECT` with equality predicates [PRZB11].

Order-preserving encryption (OPE) The OPE encryption scheme guarantees that, for any values x and y, and any secret key k, if $x < y$, then $OPE_k(x) < OPE_k(y)$. That way order relations between encrypted values can be established and queries such as e.g. `MIN`, `MAX` and `SORT` may be run on the data. It is however a weaker encryption scheme than both DET and RND [PRZB11].

Homomorphic encryption (HOM) HOM is a further probabilistic encryption scheme and allows for performing certain operations on the encrypted data. To support summation, the Paillier cryptosystem is implemented, which permits summation over encrypted values, since the product of two values encrypted with it equals the sum of the values, such that $HOM(x) \cdot HOM(y) = HOM(x + y)$. That way, `SUM` aggregates and averages may be computed, as well as values incremented. Alongside RND, it is considered one of the strongest encryption schemes [PRZB11].

Join (JOIN and OPE-JOIN) Joins involving equality and order checks (equi-joins and range joins respectively), are supported to a degree. To minimize revealing information about the encrypted data, it should not be possible for the DBMS server to join columns for which a join was not requested. Columns that are never joined should therefore be encrypted with different keys. In case the queries are known beforehand, the concerned columns may be encrypted with matching keys if the queries are known beforehand. If this is not the case, to join two columns, they have to be re-encrypted with the same key. Unfortunately, it is not possible to re-adjust the key for range joins, requiring these queries to be declared ahead of time. The encryption scheme providing equi-joins on the other hand allows for re-adjusting the column keys [PRZB11].

Word search (SEARCH) SEARCH provides full-word keyword searches on encrypted text. The columns' texts are split into keywords, from which the duplicates are removed and whose positions are then randomly permutated. The words are then encrypted and padded to the same size. While the encryption scheme is nearly as secure as RND, it leaks the number of encrypted keywords, making it possible for an attacker to guess the number of distinct or duplicate words [PRZB11].

Adjustable query-based encryption

CryptDB uses onions of encryption to adjust the encryption scheme depending on queries issued by the application at runtime. The data should be encrypted with the most secure encryption scheme that allows for executing a specific query on it.

Each data item is encrypted in one or more onions of encryption, each of which consists of nested encryption layers. The outer layers provide better security, while the inner encryption layers provide more functionality. The layers of each onion provide different operations which may not be ordered. Due to this and to performance considerations, multiple onions are needed. The Eq onion supports `SELECT` queries with equality conditions as well as equality joins, while the Ord onion e.g. supports range joins, `ORDER BY` and `SORT`. The Search onion allows for keyword searches and the Add onion provides e.g. `SUM` aggregates.

In a single-principal setup, all values in the same column are encrypted with the same key. Different keys are used across tables, columns, onions and onion layers. That way, the proxy does not have to compute separate keys for each row in a column, and the DBMS server cannot learn additional relations between the columns.

Each onion is encrypted with the most secure encryption scheme at the beginning. For the Eq and Ord onions, the outermost layer is therefore RND, HOM for the Add onion and SEARCH for the Search onion. With each query the proxy receives, it determines the onion layer needed to run the respective query, and strips off encryption layers accordingly by sending the corresponding key(s) to the DBMS server. It should however be noted that the least-secure encryption layer is never stripped off; it is also possible to specify a different threshold layer. Onion decryption happens only when a query requests a new class of computation on a column. Once an encryption layer has been stripped off, the column remains in that state [PRZB11].

Key Chaining

In case of a single-principal setup, the proxy stores a secret master key, from which all keys are derived. In a multi-user setup on the other hand, all keys are derived from user passwords. Given an access control policy, the proxy derives the onion keys from the password provided by the user, such that the DBMS server may only execute queries on data the respective user has access to. The user's key is deleted when the user logs out, ensuring that an attacker cannot decrypt the user's data without knowing their password.

Access to data items is specified using entities called **principals**, who can explicitly delegate privileges between themselves. Each principal is an instance of a principal type defined beforehand (such as e.g. users or groups). External principals are associated with end users who authenticate themselves to the application in order to obtain the privileges of their principal. Internal principals on the other hand can only obtain privileges through delegation.

For any subset of data items the principal(s) having access to it may be specified by annotating the SQL schema. Furthermore, rules for how to delegate privileges, as well as the conditions under which the delegations may occur, can be determined. The privileges are delegated using the speaks-for relation; if Bob speaks for Alice, he has access to Alice's key and therefore to the same data items as Alice.

Each principal is associated with a secret key, which consists of a symmetric key and a public-private key pair. The external principals' keys are stored encrypted with the respective end user's password. As has been stated above, if Bob speaks for Alice, he needs to have access to Alice's key. To this end, an encryption of Alice's key under Bob's key is separately stored.

Usually, a principal's symmetric key is used to encrypt data. But if they are not logged in, CryptDB does not have access to the user's key. In this case, CryptDB may encrypt any data using the concerned user's public key. Upon logging in, the user may then decrypt the data using their private key and re-encrypt it using their symmetric key if need be [PRZB11].

Summary

CryptDB attempts to balance the trade-off between privacy and performing operations on encrypted data. While on the one hand relaxing data confidentiality requirements to allow for processing the data, CryptDB on the other hand ensures that

- sensitive data is never decrypted past the least-secure encryption layer and therefore never available in plaintext
- the DBMS server cannot execute queries involving computation classes for which the concerned encryption layer(s) have not been stripped off yet
- the data of inactive users is safeguarded in case the application server, DBMS server and/or proxy are compromised

CryptDB therefore guarantees strong and practical confidentiality concerning the discussed threats [PRZB11].

Conclusion

Both FHE and CryptDB address the raised concerns. While FHE is still largely theoretical due to performance issues, it proves that processing encrypted data in an arbitrary manner is in fact possible. CryptDB on the other hand provides a practical approach to protect end user's privacy [Gen10, PRZB11].

References

[Buc16] Johannes Buchmann. *Einführung in die Kryptographie.* Springer Spektrum, 2016.

[Eck14] Claudia Eckert. *IT-Sicherheit.* Oldenbourg Wissenschaftsverlag GmbH, 2014.

[Gen10] Craig Gentry. Computing arbitrary functions of encrypted data. *Communications of the ACM*, 2010.

[KL08] Jonathan Katz and Yehuda Lindell. *Introduction to Modern Cryptography.* Chapman & Hall/CRC, 2008.

[nsa] Nsa prism program taps in to user data of apple, google and others. https://www.theguardian.com/world/2013/jun/06/us-tech-giants-nsa-data.

[PRZB11] Raluca Ada Popa, Catherine M. S. Redfield, Nickolai Zeldovich, and Hari Balakrishnan. Cryptdb: Protecting confidentiality with encrypted query processing. 2011.

[PS17] Ronald Petrlic and Christoph Sorge. *Datenschutz.* Springer Fachmedien Wiesbaden GmbH, 2017.

[yah] Yahoo hack: 1bn accounts compromised by biggest data breach in history. https://www.theguardian.com/technology/2016/dec/14/yahoo-hack-security-of-one-billion-accounts-breached.