Privacy-Preserving Functional Database Exploration with Lattigo

Jean-Philippe Bossuat

jeanphilippe.bossuat@gmail.com

FHE.org Fully Homomorphic Encryption Conference - 2024

Disclaimer: The views and opinions expressed in this talk are my own and they do not necessarily represent the views and opinions of my employer.

Table of Contents

INTRODUCTION

INTRODUCTION

- This presentation is aimed at showcasing how LATTIGO can be used to implement a very diverse circuit solving a practical use case
- The constructions used in this presentation are based on a joint work with **Malika Izabachène**, that will be accessible on ePrint in the coming days
- LATTIGO is an open-source Go module that implements RLWE based homomorphic-encryption primitives and Multiparty-Homomorphic-Encryption (MHE) based protocols
- LATTIGO was originally developed at the EPFL Laboratory for Data Security by **Jean-Philippe Bossuat** and **Christian Mouchet**, supervised by **Juan Troncoso**, and since 2022 it has been maintained by **Tune Insight SA**

INTRODUCTION

- A scientist (the Client) would like to conduct a large-scale medical study requiring patients with **specific combinations of attributes**
- To be funded, the scientist must first conduct a preliminary feasibility study to **assess if there are enough subjects available** in a patient database (the Server) meeting the study criteria
- To protect the **IP of the study** the Server should not learn anything about the selection criteria
- To protect the **privacy of the patients**, the Client should only learn a binary value: if there are enough patients meeting the selection criteria in the Server's database.

Given:

- A database $\mathcal{P}$ that consists in a $n \times h$ patient matrix
- A list of h scoring functions
- An individual score threshold t_0
- A global count threshold t_1

We want to answer the following question in a privacy preserving (and efficient) way:

Does $\mathcal{P}$ contains at least t_1 patients
whose individual score is at least equal to t_0?

THEORY & IMPLEMENTATION

Overview of the circuit:

$$\mathsf{Thresh}_{\mathsf{Enc}(t_1)}\left(\sum_{i=0}^{n-1}\mathsf{Thresh}_{\mathsf{Enc}(t_0)}\left(\sum_{j=0}^{h-1}\mathsf{Enc}(f_j)\otimes\mathcal{P}[i][j]\right)\right)$$

- $\mathcal{P}$: $n \times h$ database
- f_j: attribute scoring functions $\mathbb{R}_{[a,b]} \rightarrow \mathbb{Z}_p$
- t_0: individual selection threshold
- t_1: global selection threshold
- $\mathsf{Tresh}_t(x) = 1$ if $x \geq t$ else 0

IMPLEMENTATION - Overview

This circuit is interesting because it makes use of:

- **Encrypted Lookup Tables**: evaluation of the private scoring functions $\mathsf{Enc}(f_j)$
- **Ring-Packing**: repacking the individual private scores
- **Ring-Merging**: merging the small degree ciphertext of the individual scores into large degree ciphertexts
- **Scheme-Switching**: switching from coefficient packed messages to CKKS SIMD encoded messages
- **Private Thresholds with CKKS**: local and global private selection over the encrypted scores
- **Bootstrapping with CKKS** to evaluate the private thresholds

The implementation is available at

`https://github.com/Pro7ech/fhe-org-2024`

THEORY - Encrypted Scoring Function

Let

- $f : \mathbb{R}_{[a,b]} \to \mathbb{R}$
- $g : \mathbb{R}_{[a,b]} \to \mathbb{R}_{[0,1]} : y = \frac{1}{2}(\frac{2x-b-a}{b-a} + 1)$

Then, given a discretization factor of $1/N$, we can encode f on a polynomial of $\mathbb{R}[X]/(X^N + 1)$ as

$$\mathcal{F}(X) = f'(0) - \sum_{i=1}^{N-1} f'\left(g^{-1}\left(\frac{N-i}{N}\right)\right) \cdot X^i$$

and for $x \in [a, b]$, we have

$$X^{\lfloor Ng(x) \rceil} \cdot \mathsf{RLWE}(\mathcal{F}(X)) \approx \mathsf{RLWE}(f(x)X^0 + \star)$$

with an error bounded by $|f(x) - f(x \pm \frac{b-a}{N})|$

IMPLEMENTATION - Encrypted Scoring Function

This step is not a native operation in any of the HE schemes in LATTIGO .
But It can be implemented from the lower-level packages `rlwe` and `ring`:

1. Encode f on a `[]float64` of size N with the method described above
2. Encode the `[]float64` on an `rlwe.Plaintext` using the `hefloat.Encoder`
3. Encrypt the `rlwe.Plaintext` on an `rlwe.Ciphertext` using the `rlwe.Encryptor`

The code for this step is located in the files

- `function.go`
- `server.go` (for the API call)

IMPLEMENTATION - Encrypted Scoring Function

The evaluation step of the encrypted scoring function is done as:

$$\mathsf{RLWE}\Big(\sum_{j=0}^{h-1} f_j(x_j)X^0 + \star\Big) \approx \sum_{j=0}^{h-1} \mathsf{NTT}\big(X^{\lfloor Ng(x_j)\rceil}\big)\cdot \mathsf{RLWE}(\mathcal{F}_j(X))$$

and can be carried out with basic polynomial arithmetic of the `ring` package:

- `ring.NTT(p0, p1 ring.Poly)`
- `ring.MulCoeffsMontgomery(p0, p1, p2 ring.Poly)`
- `ring.MulCoeffsMontgomeryThenAdd(p0, p1, p2 ring.Poly)`

THEORY - Ring-Packing

- Each evaluation of **Encrypted Scoring Function** step returns an RLWE whose first coefficient is the evaluation of $f(x_i)$
- We can homomorphically repack N of those coefficients in a single RLWE ciphertext:

$$\sum_{i=0}^{N-1} \mathsf{RLWE}(m_i + \star) \cdot X^i \xrightarrow{\text{Repack}} \mathsf{RLWE}\left(\sum_{i=0}^{N-1} m_i \cdot X^i\right)$$

- This reduces the number of ciphertexts by a factor of N

IMPLEMENTATION - Ring-Packing

LATTIGO has a native support of **Ring-Packing** through the API of the `rlwe.Evaluator` which enables to pack a hash map containing indexed `rlwe.Ciphertext` from 0 to $N-1$ into a single `rlwe.Ciphertext`:

- Method: `rlwe.Evaluator.Pack`
- Input: `map[int]*rlwe.Ciphertext`
- Output: `rlwe.Ciphertext`

The code for this step is located in the files

- `repacking.go`
- `repacking_keys.go`
- `server.go` (for the API call)

THEORY - Ring-Merging

- The output of the **Ring-Packing** step returns RLWE ciphertexts whose ring degree is small, e.g. $N = 2^{12}$
- **Scheme-Switching** to CKKS makes uses of the CKKS bootstrapping, which requires a ring degree $N' = 2^{16}$ (for security)
- We can merge N'/N RLWE ciphertexts of degree N into a single RLWE ciphertext of degree N':

$$\sum_{j=0}^{N'/N-1} \mathsf{RLWE}\left(\sum_{i=0}^{N-1} m_{jN+i} \cdot X^i\right) \cdot X^j$$

$$\downarrow \text{Merge}$$

$$\mathsf{RLWE}\left(\sum_{j=0}^{N'/N-1}\left(\sum_{i=0}^{N-1} m_{jN+i} \cdot X^{iN'/N}\right) \cdot X^j\right)$$

IMPLEMENTATION - Ring-Merging

LATTIGO has native support for switching RLWE ciphertext dimensions through key-switching and `Ring-Merging` can be implemented in only a few lines of code:

$$\mathsf{RLWE}_{2N} \leftarrow \mathsf{KeySwitch}_{\pi(s)\rightarrow s'}(\pi(\mathsf{RLWE}_N) + X \cdot \pi(\mathsf{RLWE}_N))$$

- $\pi : X \rightarrow X^2$: `rlwe.SwitchCiphertextRingDegreeNTT`
- KeySwitch : `rlwe.Evaluator.ApplyEvaluationKey`

The code for this step is located in the files:

- `repacking.go`
- `repacking_keys.go`
- `server.go` (for the API call)

- The **Scheme Switching** step homomorphically encodes the values and raises the modulus to enable further computations

- This is equivalent to a CKKS bootstrapping that skips the last step (SlotsToCoeffs):

  1. **ModRaise**: raises the ciphertext to the largest modulus
  2. **CoeffsToSlots**: homomorphically encodes the ciphertext
  3. **EvalMod**: homomorphically evaluates the modular reduction
  4. (**SlotsToCoeffs**: homomorphically decodes the ciphertext)

LATTIGO supports for CKKS bootstrapping with out of the box default parameters, as well as advanced parameterization, enabling custom instantiation as well as custom circuit composition:

- `bootstrapping.Evaluator`
- `.ScaleDown`: scales the ciphertext down to Q_0
- `.ModUp`: raises the ciphertext modulus from Q_0 to Q_L
- `.CoeffsToSlots`: homomorphic encoding
- `.EvalMod`: homomorphic modular reduction
- `.SlotsToCoeffs`: homomorphic decoding

The code for this step is located in the files

- `bootstrapping.go`
- `scheme_switching.go`
- `server.go` (for the API call)

THEORY - Threshold

Let

$$\mathsf{Thresh}_t(x) : \begin{cases} 1 & \text{if } x \geq t \\ 0 & \text{otherwise} \end{cases} \quad \text{and} \quad \mathsf{Sign}(x) : \begin{cases} 1 & \text{if } x > 0 \\ 0.5 & \text{if } x = 0 \\ -1 & \text{otherwise} \end{cases}$$

then $\mathsf{Thresh}_t(x) = \frac{1}{2}\mathsf{Sign}(x-t) + \frac{1}{2}$ and we can evaluate Sign in the interval $[-1, -2^{-\alpha}] \cup [2^{-\alpha}, 1]$ with precision β by using a composition of successive minimax approximations[1]

This approach requires to scales $x - t$ to the interval $[-1, 1]$, which is done during the local threshold by a normalization factor $1/\sum \max f_j$ and during the global threshold by a normalization factor $1/p$

[1] Lee et al, *Minimax Approximation of Sign Function by Composite Polynomial for Homomorphic Comparison*

IMPLEMENTATION - Threshold

LATTIGO has native support for comparison, through the `hefloat.ComparisonEvaluator`. The user can generate and provide a custom mimimax composite approximation that suits the application needs

- **Local Private Threshold**: $[-1, -2^{-8} \cup [2^{-8}, 1]$ with precision $\beta = 14$, total depth of 12
- **Global Private Threshold**: $[-1, -2^{-16}] \cup [2^{-16}, 1]$ with precision $\beta = 9.4$, total depth of 20
- `hefloat.ComparisonEvaluator.Step`

To avoid $x = t_i$ in both cases, which would returns 0.5, we add a small bias to $x - t$ that is half the smallest difference between two possible values of x

The code for this step is located in the files

- `server.go`

PUTTING IT ALL TOGETHER

1. The client generates:
   - The necessary evaluation keys
   - The h encrypted scoring functions $[\mathsf{Enc}(f[0]), \ldots, \mathsf{Enc}(f[h-1])$
   - The local encrypted threshold $\mathsf{Enc}(t_0)$ and normalization factor $\mathsf{Enc}(1/\sum \max(f_j))$
   - The global encrypted threshold $\mathsf{Enc}(t_1)$

   and sends them to the server
2. The server evaluates $\mathsf{Enc}(y[i]) = \sum X^{\mathcal{P}[i][j]} \cdot \mathsf{Enc}(f[j])$
3. The server packs the $\mathsf{Enc}(y[i])$ into RLWE low-degree ciphertexts
4. The server merge the low-degree RLWE ciphertexts into large-degree RLWE ciphertexts
5. The server scheme-switches the large-degree RLWE ciphertexts into CKKS ciphertexts
6. The server evaluates the private local-thresholds and aggregates all values
7. The server evaluates the final private global-threshold

PERFORMANCE

PERFORMANCE - PARAMETERS

- Set I: Encrypted Lookup-Table & Ring-Packing
- Set II-V: Ring-Merging
- Set V: Local & Global Thresholds
- Set VI: Scheme-Switching & Bootstrapping

Set	$\log(N)$	$\log(Q)$	$\log(P)$	$\log(QP)$	h	λ
I	12	60	48	108	$2N/3$	128
II	13	60	48	108	$2N/3$	> 128
III	14	60	48	108	$2N/3$	> 128
IV	15	60	48	108	$2N/3$	> 128
V	16	$60 + 8 \cdot 45$	158	578	192	> 128
VI	16	$60 + 8 \cdot 45 + 3 \cdot 39$ $+8 \cdot 60 + 4 \cdot 56$	$5 \cdot 61$	1546	192	128

PERFORMANCE - CLIENT DATA

Set	Size [MB]
Ring Packing Keys	3.25
Ring Merging Keys	7.5
Bootstrapping Keys	7395
RLWE(f_j)	2.01
RLWE(t_0) & RLWE$(1/\sum \max(f_j))$	18
RLWE(t_1)	9
Total	≈ 7435

PERFORMANCE - TIMINGS

- Timings for a database of 2^{19} entries single threaded[2]

Operation	Total	Amortized
Client Initialization	22.7s	43μs
Server Initialization	10.3s	19μs
Private Functions & Packing	436.6s	832μs
Scheme-Switching	64.2s	122μs
Private-Threshold Local	327.1s	624μs
Private-Threshold Global	49.9s	95μs
Total	910.8s	**1.737ms**

- Peak of 22GB of RAM, most of it being due to the size of the bootstrapping keys and the plaintext matrices for the homomorphic encoding/decoding during the bootstrapping

[2] i9-12900K, 64GB DDR5, Windows 11, Go 1.22

OTHER CONSIDERATIONS

- In many real-world settings, data are distributed across several databases that may be independently managed. Computing the circuit without leaking becomes a secure multiparty computation (MPC) problem

- LATTIGO supports multiparty homomorphic encryption (MHE) schemes that enable an efficient MPC protocol. We have recently released **Helium**, a first implementation of such a protocol

`https://github.com/tuneinsight/lattigo`
`https://github.com/ChristianMct/helium`