Using Fully Homomorphic Encryption To Ensure Voter Privacy

David Son, Elaina Huang, Kang In Park

Project Type 3: Homomorphic Encryption

Goal:

- Design a method for Alice to share encrypted data to Carol.
- Alice can then send queries to Carol, which will be computed by Carol on the encrypted data.
- The encrypted query result is sent back to Alice, who is able to decrypt it and get the correct query result.

Important Libraries Used

- Pyfhel - encryption
- pandas - read and store data from csv files
- numpy - convert data into arrays usable by Pyfhel
- psycopg2 - create and store data in a database people can connect to

Choosing Alice’s Dataset

The dataset we plan to use is extracted from Kaggle: https://www.kaggle.com/datasets/democracy-fund/2016-voter-survey?select=VOTER_Survey_December16_Release1.csv

- Contains the sampling of voters from 2016 presidential election.

The data set contains a lot of information; around 8000 rows and 668 columns.

We will mainly use two columns:

- “presvote16post_2016”
- “PARTY_AGENDAS_rand_2016”

C
PARTY_AGENDAS_rand_2016
Republican Party
Republican Party
Republican Party
Democratic Party
Republican Party
Democratic Party
Republican Party
Republican Party
Republican Party
Democratic Party
Republican Party
Republican Party
Republican Party

O
presvote16post_2016
Hillary Clinton
Donald Trump
Hillary Clinton
Gary Johnson
Donald Trump
Hillary Clinton
Hillary Clinton
Donald Trump
Hillary Clinton
Hillary Clinton
Hillary Clinton
Hillary Clinton
Donald Trump
Hillary Clinton

BFV Algorithm

- Parts
  - Secret Key — decryption
  - Public Key — encryption
  - Evaluation Key — homomorphic operations on ciphertexts
  - 2^n — keyspace
- Plaintext and ciphertext spaces in polynomial rings
- Secret Key generated as random ternary polynomial in key space
- Public Key generated in pair
  - $PK_1 = [-1(a*SK+e)]_q$
  - $PK_2 = a$

Encrypting with BFV Algorithm

- Encrypting message M generates 3 small random polynomials *u* from R_2 and e_1 and e_2 from error distribution and returns (C_1, C_2) as follows:

  $$C_1 = [PK_1 \cdot u + e_1 + \Delta M]_q$$
  $$C_2 = [PK_2 \cdot u + e_2]_q$$

  where

- ΔM is M/t
- t is the plaintext coefficient
- q is the ciphertext coefficient

Decrypting with BFV Algorithm

- Decrypting is done with the following algorithm

$$\mathsf{M} = \left[\left\lfloor \frac{t[\mathsf{C}_1 + \mathsf{C}_2 \cdot \mathsf{SK}]_q}{q} \right\rceil\right]_t$$

Intuition Behind Homomorphism

- Bearing in mind how the ciphertexts are created, an add function can be made simple as follows:

$$\mathsf{EvalAdd}(\mathsf{C}^{(1)}, \mathsf{C}^{(2)}) = ([\mathsf{C}_1^{(1)} + \mathsf{C}_1^{(2)}]_q, [\mathsf{C}_2^{(1)} + \mathsf{C}_2^{(2)}]_q) = (\mathsf{C}_1^{(3)}, \mathsf{C}_2^{(3)}) = \mathsf{C}^{(3)}$$

  This allows operations to be performed without ever having to decrypt the ciphertext

Implementation - Fully Homomorphic Encryption

```python
def HE_object():
    HE = Pyfhel()           # Creating empty Pyfhel object
    HE.contextGen(scheme='bfv', n=2**14, t_bits=20)#scheme?
    # Generate context for 'bfv'/'ckks' scheme
    # The n defines the number of plaintext slots.
    HE.keyGen() #generae a pair of public and secret keys
    return HE #return HE object

def HE_encryption(val, HE):
    vote = np.array([val], dtype=np.int64)#32?
    vote_ctxt = HE.encryptInt(vote) #encrypt it with using the public key
    return vote_ctxt

def HE_decryption(val, HE):
    vote_dtxt = HE.decryptInt(val)
    return vote_dtxt
```

In our implementation we chose to use integer encryption function of Pyfhel.

First, initialize an Pyfhel object instance, then fill in desired scheme and other parameters.

Then use the corresponding function to encrypt and decrypt. Very convenient.

Implementation Cont. - 1. Data Cleaning and Data Encoding

```
def encode_vote():
    df = pd.read_csv("voter_data.csv")
    print(' ')
    print("1. to learn what columns we want")
    print(df.head())

    print(' ')
    print("2. select only desired columns into new pandas frame")
    df_new = df[["case_identifier","PARTY_AGENDAS_rand_2016","presvote16post_2016"]]
    print(df_new.head())

    print(' ')
    print("3. restrict candidates to be only Hilary or Trump and remove NaN value")
    print(df_new['presvote16post_2016'].value_counts())

    df_new.replace(" ", float("NaN"), inplace=True)
    print(df_new.describe(include='all'))
    df_HT = df_new[df_new['presvote16post_2016'].isin(['Hillary Clinton','Donald Trump'])]
    df_HT.reset_index(inplace=True, drop=True)
    df_HT.info()
    print(df_HT["presvote16post_2016"].value_counts())

    print(' ')
    print("4. encode two candidates into 1-Hillary and 0-Trump")
    print("encode two parties into 1-Democratic and 0-Republican")
    df_HT['presvote16post_2016'] = df_HT['presvote16post_2016'].map({'Hillary Clinton':1,'Donald Trump':0})
    df_HT['PARTY_AGENDAS_rand_2016'] = df_HT['PARTY_AGENDAS_rand_2016'].map({'Democratic Party':1,'Republican Party':0})

    print(df_HT)
    print(' ')
    print("=> party count")
    print(df_HT['PARTY_AGENDAS_rand_2016'].value_counts())
    print(' ')
    print("=> candidate count")
    print(df_HT['presvote16post_2016'].value_counts())
    #df_HT = df_HT.reset_index()
    return df_HT
```

Step 1. Review the dataset

Step 2. Choose desired columns

Step 3. Eliminate unwanted candidates

Step 4. Encode 1-Hilary 0-Trump,

1-Democrat 0-Republican

Step 5. Summarize total count for later

reference purpose

Implementation Cont. - 2. Homomorphic Encryption

```python
print("----------Homomorphic Encryption------------")
# Alice wants confidentiality so she encrypts her dataset before sending out to Carol
# First Alice generates an Homomorphic Encryption object HE
HE = HE_object()

# Then Alice encrypts accordingly
# for efficiency purpose I'm only using the head of df to continue
df_head = df.head()
print("Before Encryption")
print(df_head)
for i, row in df_head.iterrows():
    ctxt = HE_encryption(int(row['PARTY_AGENDAS_rand_2016']), HE)
    df_head.loc[i,'PARTY_AGENDAS_rand_2016'] = ctxt
    ctxt = HE_encryption(int(row['presvote16post_2016']), HE)
    df_head.loc[i,'presvote16post_2016'] = ctxt
print(' ')
print("After Encryption")
print(df_head)
print(' ')
print("A full Pyfhel object sample: ")
vote = np.array([1], dtype=np.int64)#32?
vote_ctxt = HE.encryptInt(vote) #encrypt it with using the public key
print(vote_ctxt)
```

Use the HE function described before to encrypt all the data in **party** and **candidate** column.

seed@VM: ~/.../crypto_proj2

```
----------Homomorphic Encryption------------
Before Encryption
    case_identifier  PARTY_AGENDAS_rand_2016  presvote16post_2016
0               779                        0                    1
1              2108                        0                    0
2              2597                        0                    1
3              4460                        0                    0
4              5225                        1                    1
..              ...                      ...                  ...
95          6120266                        1                    1
96          6124499                        1                    1
97          6125228                        1                    1
98          6126617                        1                    0
99          6129137                        0                    0

[100 rows x 3 columns]

After Encryption
    case_identifier  ...                              presvote16post_2016
0               779  ...  <Pyfhel Ciphertext at 0x7f1ddd7949a0, scheme=b...
1              2108  ...  <Pyfhel Ciphertext at 0x7f1ddd794ae0, scheme=b...
2              2597  ...  <Pyfhel Ciphertext at 0x7f1dda871450, scheme=b...
3              4460  ...  <Pyfhel Ciphertext at 0x7f1dda8713b0, scheme=b...
4              5225  ...  <Pyfhel Ciphertext at 0x7f1dda871220, scheme=b...
..              ...  ...
95          6120266  ...  <Pyfhel Ciphertext at 0x7f1ddd92d400, scheme=b...
96          6124499  ...  <Pyfhel Ciphertext at 0x7f1ddd92d4a0, scheme=b...
97          6125228  ...  <Pyfhel Ciphertext at 0x7f1ddd92d540, scheme=b...
98          6126617  ...  <Pyfhel Ciphertext at 0x7f1ddd92d5e0, scheme=b...
99          6129137  ...  <Pyfhel Ciphertext at 0x7f1ddd92d680, scheme=b...

[100 rows x 3 columns]

A full Pyfhel object sample:
<Pyfhel Ciphertext at 0x7f1ddd92df90, scheme=bfv, size=2/2, noiseBudget=361>
```

Before encryption

After encryption

Implementation Cont. - 3. Pass Into Database

```
print("----------Pass Into Database-----------")
# Next Alice stores the ciphertext into Carol
createTable() # create table if such table no exist

insertDB(df_head) # insert Pyfhel.PyCtxt.PyCtxt object into Postgres database
# convert it to bytes stores in BYTEA datatype

checkDB(HE) # Pull out all inserted records to confirm information integrity is preserved
# pull out bytes convert it back to Pyfhel.PyCtxt.PyCtxt decrypted object
```

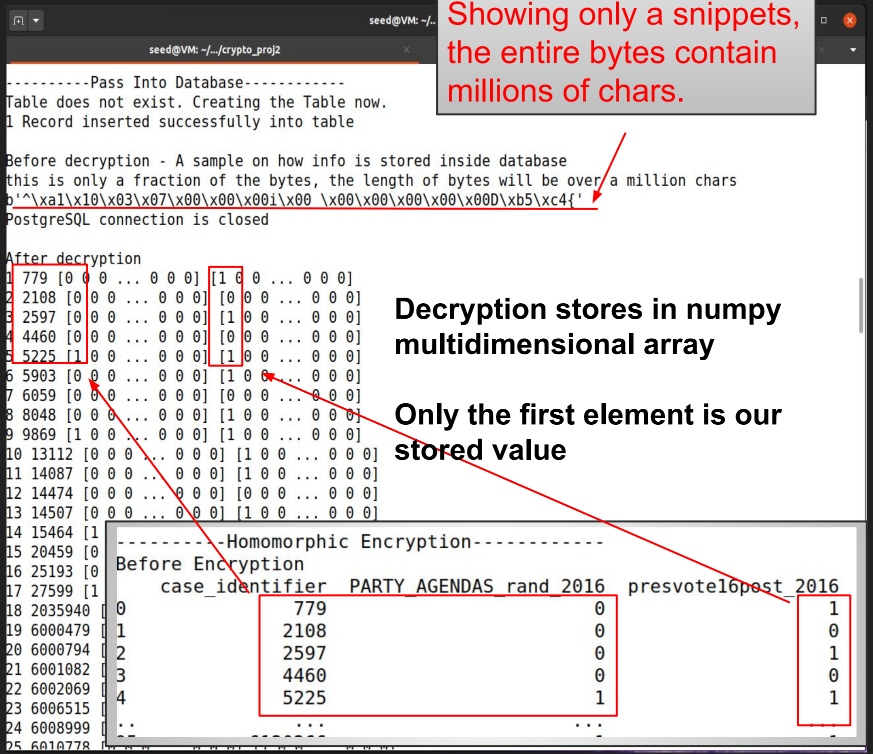
Step 1. Create the table if not existed

Step 2. Insert Pyfhel.PyCtxt.PyCtxt object into Postgres database. Because Postgres only allows certain data types, therefore we convert the Pyfhel object into bytes through its built-in function to_bytes() to store inside the database.

Step 3. Check the data integrity, make sure nothing mess up the object -> bytes -> object conversion process.

How the data actually being stored inside database.

Showing only a snippets, the entire bytes contain millions of chars.



Decryption stores in numpy multidimensional array

Only the first element is our stored value

Implementation Cont. - 4. Perform Queries

```
print(' ')
print("----------Perform Queries------------")
print(' ')
print("1. [Addition] Carol, give me the total votes of Hilary and Trump individually")
queryDB(HE,1)

print(' ')
print("2. [Conditional Addition] Carol, how many republican voted for Hilary?")
queryDB(HE,2)

print(' ')
print("3. [Conditional Division] Carol, whats the percentage of republican voted for Hilary?")
queryDB(HE,3)
```

```
----------Perform Queries-----------

1. [Addition] Carol, give me the total votes of Hilary and Trump individually
Hilary:  60
Trump:  40
PostgreSQL connection is closed

2. [Conditional Addition] Carol, how many republican voted for Hilary?
1 id                                                  1
caseid                                              779
party        [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, ...
candidate    [1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, ...
Name: 0, dtype: object
2 id                                                  3
caseid                                             2597
party        [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, ...
candidate    [1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, ...
Name: 2, dtype: object
3 id                                                  6
caseid                                             5903
party        [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, ...
candidate    [1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, ...
Name: 5, dtype: object
4 id                                                  8
caseid                                             8048
party        [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, ...
```

Result is based on 100 rows of dataset

0 meaning they are from republican party.

```
28 id                                                 93
caseid                                           6116600
party        [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, ...
candidate    [1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, ...
Name: 92, dtype: object
29 id                                                 94
caseid                                           6116972
party        [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, ...
candidate    [1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, ...
Name: 93, dtype: object
The total count is:  29
PostgreSQL connection is closed

3. [Conditional Division] Carol, whats the percentage of republican voted for Hilary?
72.5 %
PostgreSQL connection is closed
[12/10/22]seed@VM:~/.../crypto_proj2$
```

In our queries, we perform Addition, Conditional Addition and Conditional Division.

Analyzing Performance

- Fully Homomorphic Encryption (FHE) is known to be slow.
- How much slower is it compared to queries made to unencrypted data?

We used varying input sizes to test the average runtime of each query.

- 50 rows
- 100 rows
- 200 rows

Ran both FHE version and unencrypted versions multiple times to get average runtime for each query.

Example Program Output Showing Query Times

FHE

```
----------Perform Queries------------
1. [Addition] Carol, give me the total votes of Hillary and Trump individually
Hillary:  27
Trump:  23

2. [Conditional Addition] Carol, how many republican voted for Hillary?
The total count is:  17

3. [Conditional Division] Carol, whats the percentage of republican voted for Hillary?
73.91 %

----------Execution Time for Each Query------------
Query 1: 1.806 seconds
Query 2: 1.795 seconds
Query 3: 1.820 seconds
[12/10/22]seed@VM:~/.../applied-crypto-project-2$
```

Unencrypted

```
----------Perform Queries------------
1. [Addition] Carol, give me the total votes of Hillary and Trump individually
Hillary:  27
Trump:  23

2. [Conditional Addition] Carol, how many republican voted for Hillary?
The total count is:  17

3. [Conditional Division] Carol, whats the percentage of republican voted for Hillary?
73.91 %

----------Execution Time for Each Query------------
Query 1: 0.008 seconds
Query 2: 0.008 seconds
Query 3: 0.012 seconds
[12/10/22]seed@VM:~/.../applied-crypto-project-2$
```

50 Rows

FHE

	Trial 1	Trial 2	Trial 3	Trial 4	Trial 5	**AVERAGE**
Query 1	1.806	1.797	2.062	1.973	1.815	**1.891**
Query 2	1.795	1.907	1.956	1.829	1.781	**1.854**
Query 3	1.820	1.824	1.856	1.953	1.929	**1.876**

Total Average: 1.874

Unencrypted

	Trial 1	Trial 2	Trial 3	Trial 4	Trial 5	**AVERAGE**
Query 1	0.008	0.008	0.010	0.007	0.008	**0.008**
Query 2	0.008	0.010	0.016	0.009	0.010	**0.011**
Query 3	0.012	0.016	0.019	0.012	0.017	**0.015**

Total Average: 0.011

100 Rows

FHE

	Trial 1	Trial 2	Trial 3	Trial 4	Trial 5	**AVERAGE**
Query 1	3.563	3.472	3.456	3.523	3.475	**3.485**
Query 2	3.471	3.495	3.508	3.516	3.432	**3.479**
Query 3	3.465	3.432	3.463	3.439	3.486	**3.452**

Total Average: 3.472

Unencrypted

	Trial 1	Trial 2	Trial 3	Trial 4	Trial 5	**AVERAGE**
Query 1	0.008	0.009	0.009	0.010	0.012	**0.010**
Query 2	0.009	0.010	0.010	0.012	0.012	**0.011**
Query 3	0.012	0.010	0.010	0.011	0.011	**0.011**

Total Average: 0.011

200 Rows

FHE

	Trial 1	Trial 2	Trial 3	Trial 4	Trial 5	**AVERAGE**
Query 1	12.698	11.372	10.560	10.156	10.091	**10.975**
Query 2	11.315	10.644	10.019	9.938	10.211	**10.426**
Query 3	10.813	10.156	9.983	10.075	10.531	**10.311**

Total Average: 10.571

Unencrypted

	Trial 1	Trial 2	Trial 3	Trial 4	Trial 5	**AVERAGE**
Query 1	0.008	0.008	0.008	0.009	0.010	**0.009**
Query 2	0.013	0.012	0.013	0.013	0.015	**0.013**
Query 3	0.014	0.015	0.013	0.013	0.013	**0.014**

Total Average: 0.012

Analyzing Performance: Observations

FHE

As input size grew from 50 → 100 → 200

Average query time increased from 1.874 → 3.472 → 10.571

+85% +204%

Unencrypted

As input size grew from 50 → 100 → 200

Average query time increased from 0.011 → 0.011 → 0.012

+0% +9%

Analyzing Performance: Insight

- FHE is not scalable. As input size increases, time taken for each query drastically increases.
- We were not able to use all 8000 rows of the dataset since the queries took too long and eventually crashed the program. 250 rows was the highest we could go without crashing.
- Queries that didn’t use homomorphic encryption was faster and scalable
    - Showed no significant difference in runtime even as input size increased.
- The upside of using FHE is that the confidentiality of data is preserved, even to those that store the data AND execute the queries (in this case, Carol).
- Alice can query for data from Carol, and still get accurate results.