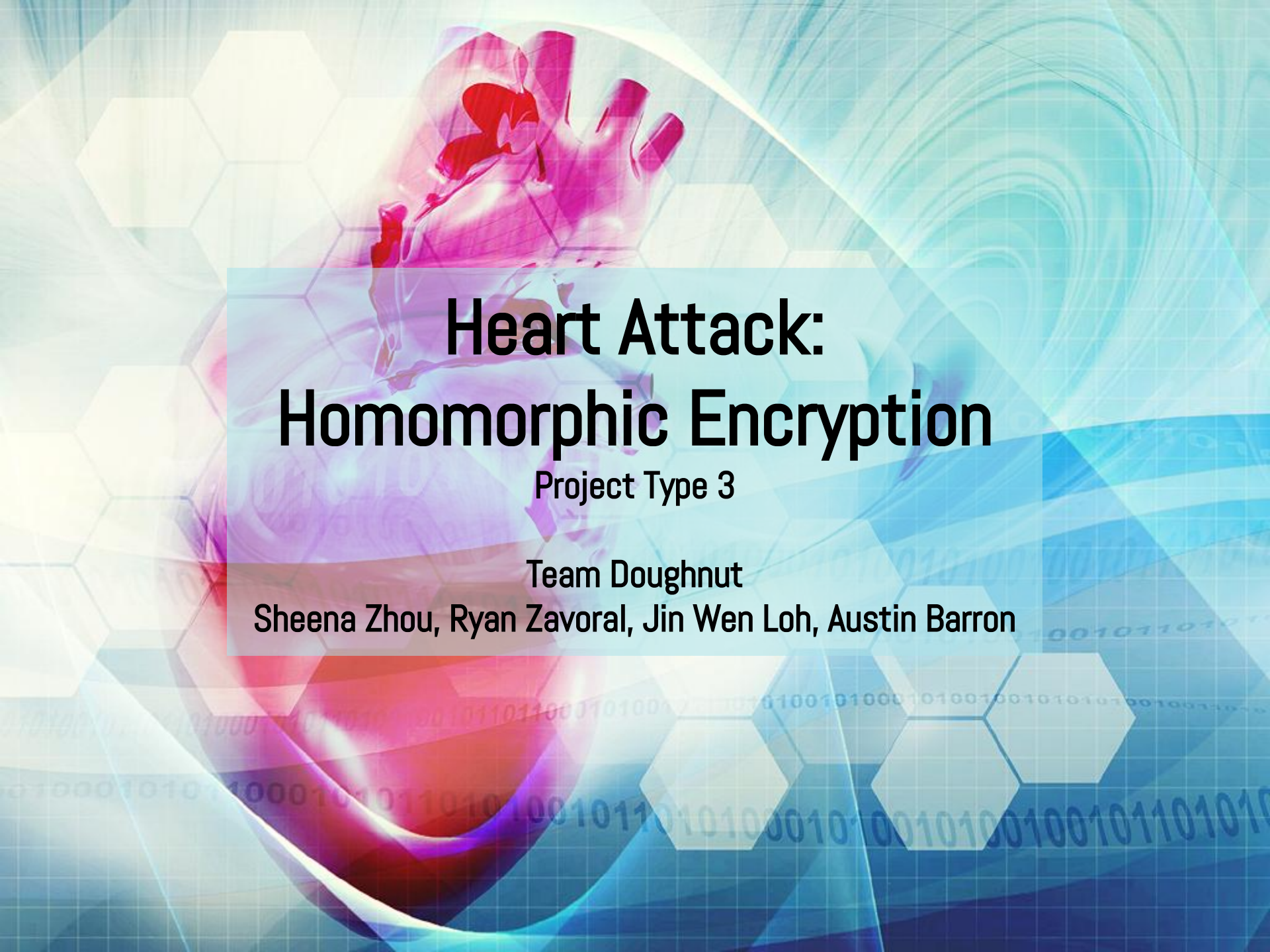


Heart Attack: Homomorphic Encryption

Project Type 3

Team Doughnut
Sheena Zhou, Ryan Zavoral, Jin Wen Loh, Austin Barron

Project Type 3: Goal

- Allow Alice to outsource her dataset using partially homomorphic encryption to a cloud server Carol
- Allow Alice to query Carol for a function to be computed over the stored encrypted data. This will be done by:
    1) Alice specifies query function to Carol
    2) Carol evaluates function over encrypted data
    3) Alice obtains output of the function returned by Carol and decrypts the data

General Plan

1) **Alice's Dataset:** Heart.csv
2) **Alice's Desired Query Function:** Dataset entries matching a keyword (chosen by Alice)
3) **Method:**
   a) Create a program called "Alice" that encrypts input data using the homomorphic encryption scheme
   b) Create a program called "Carol" that computes matching on homomorphic encrypted data on query from a user i.e. age:42
   c) Create a program called "Darryl" that computes matching on non-encrypted data on query from a user and compare its performance against "Carol"

Heart.csv

age	sex	cp	trtbps	chol	fbs	restecg	thalachh	exng	oldpeak
63	1	3	145	233	1	0	150	0	2.3
37	1	2	130	250	0	1	187	0	3.5
41	0	1	130	204	0	0	172	0	1.4
56	1	1	120	236	0	1	178	0	0.8
57	0	0	120	354	0	1	163	1	0.6
57	1	0	140	192	0	1	148	0	0.4
56	0	1	140	294	0	0	153	0	1.3
44	1	1	120	263	0	1	173	0	0
52	1	2	172	199	1	1	162	0	0.5
57	1	2	150	168	0	1	174	0	1.6
54	1	0	140	239	0	1	160	0	1.2
48	0	2	130	275	0	1	139	0	0.2
49	1	1	130	266	0	1	171	0	0.6
64	1	3	110	211	0	0	144	1	1.8
58	0	3	150	283	1	0	162	0	1
50	0	2	120	219	0	1	158	0	1.6
58	0	2	120	340	0	1	172	0	0
66	0	3	150	226	0	1	114	0	2.6
43	1	0	150	247	0	1	171	0	1.5
69	0	3	140	239	0	1	151	0	1.8
59	1	0	135	234	0	1	161	0	0.5
44	1	2	130	233	0	1	179	1	0.4
42	1	0	140	226	0	1	178	0	0

Homomorphic Encryption - Paillier

- Sum homomorphic → used for calculating matching
  - **match = matched data - query data = 0**
  - **cell + match*random_value obfuscates rows that do not match**
- IND-CPA secure
- Extremely usable and secure library for python implementation - PHE

Alice

```python
import pandas as pd
import numpy as np
from phe import paillier
import pickle

def gen_keys():
    # generate paillier keys
    pubkey, prikey = paillier.generate_paillier_keypair()

    # save private and public keys
    with open('pubkey.pickle', 'wb') as handle:
        pickle.dump(pubkey, handle)
    with open('prikey.pickle', 'wb') as handle:
        pickle.dump(prikey, handle)
    return pubkey, prikey

# Read from csv
# store as numpy array, df_np
df = pd.read_csv("test.csv")
df = df.dropna()
df_np = df.to_numpy()

# create private and public keys
public_key, private_key = gen_keys()

cipher_vect = list()
for row in df_np:
    # encrypt each item and append row
        cipher_row = [public_key.encrypt(x) for x in row]
        cipher_vect.append(cipher_row)

# pickle data and columns
mydf = pd.DataFrame(cipher_vect, columns=df.columns)
pd.to_pickle(mydf, "encrypted.pickle")
```

Encrypts input data using Paillier homomorphic encryption scheme.

- Alice(Encrypt): Generates 3 files: public and private key files, and an encrypted csv
- Using Paillier, create public and private keys.
- Encrypt every column and row using our public key.
- Store it as a pickled csv.

left: Alice Encrypt

Alice (cont'd)

Alice (Decrypt):

Takes the results from Carol, decrypts them, and then prints them out.

right: Alice Encrypt

```python
import pandas as pd
import numpy as np
from phe import paillier
import pickle

def get_keys():
        # open paillier keys from files
        with open("pubkey.pickle", 'rb') as handle:
                public_key = pickle.load(handle)
        with open("prikey.pickle", 'rb') as handle:
                private_key = pickle.load(handle)
        return public_key, private_key

# read pickle file from carol
df = pd.read_pickle("encrypted_carol.pickle")
df = df.dropna()

# get valid index
validIndex = df.columns.get_loc('valid')

# convert to np
df_np = df.to_numpy()

# retrieve keys
public_key, private_key = get_keys()

# decrypt each cell in row and return if the row is valid
decrypt_vect = list()
for row in df_np:
        decrypt_row = [private_key.decrypt(cell) for cell in
        if int(decrypt_row[validIndex]) == 1:
                decrypt_vect.append(decrypt_row)

# print all valid rows
for valid_row in decrypt_vect:
        print(valid_row)
```

Carol

Computes matching on homomorphically encrypted data on query from a user i.e. (age:42).

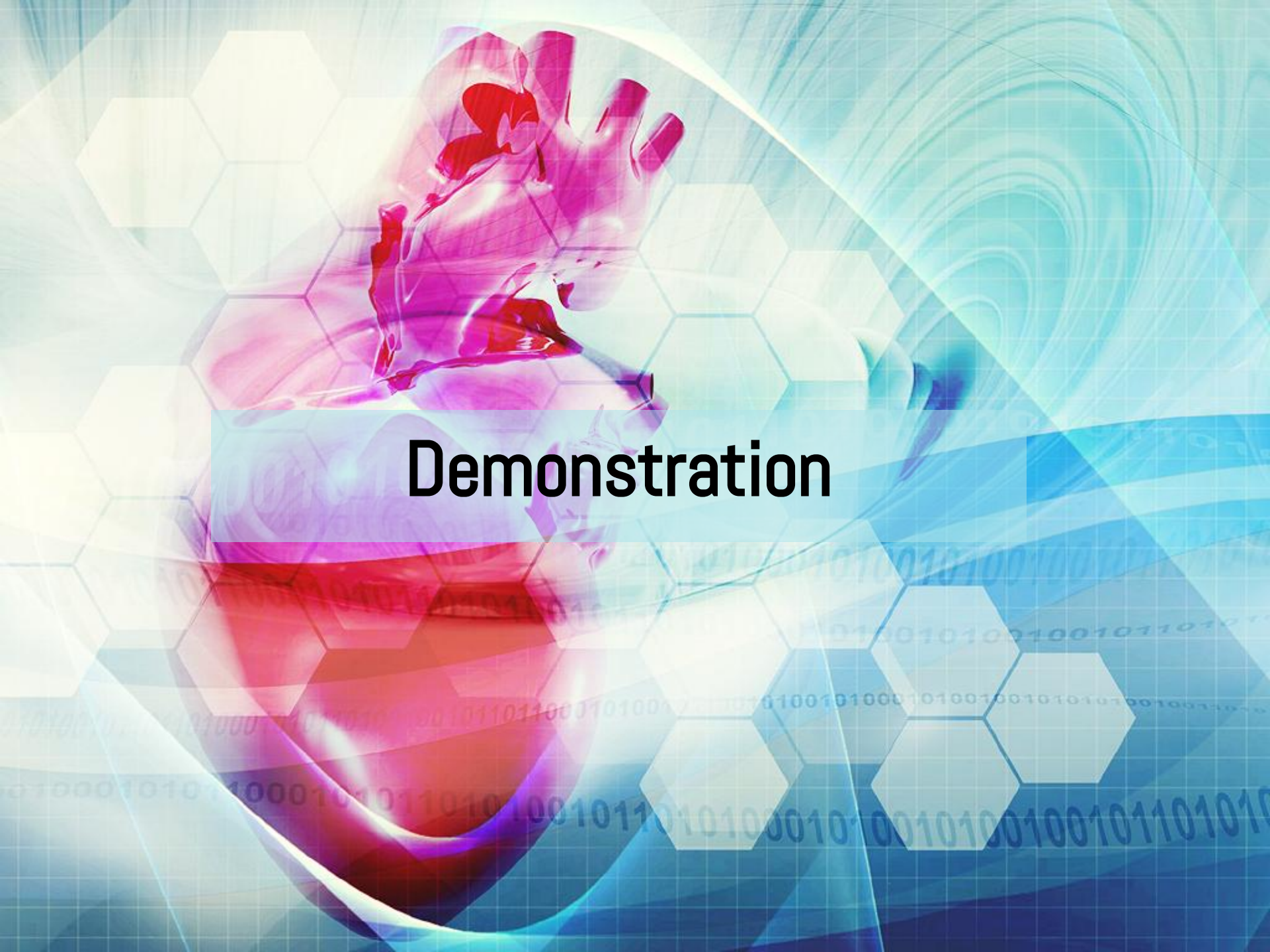
Returns encrypted data that matches the query in a pickled file.

```python
''' Carol will compute matching on encrypted data on query
from Alice and compare'''
import argparse
import copy
import csv
import math
import pandas as pd
import phe.encoding
from phe import paillier
import pickle
import random
import sys

def randomize():
    pos_or_neg = 1 if random.random() < 0.5 else -1
    return random.randint(0, 10000000000)*pos_or_neg

def get_rows(pklfile):
    # get csv as dataframe
    df = pd.read_pickle(pklfile)
    df = df.dropna()
    header = df.columns
    rows = df.to_numpy()

    return header, rows

def compute_matching(header, rows, category, data):
    if category not in header:
        sys.stderr.write('Error: "{}" is not a valid category\n'.format(category))
        sys.exit(1)

    # get index to apply to
    catIndex = header.get_loc(category)

    # apply subtracting method to identify matching row
    for i in range(0, len(rows)):
        catcell = rows[i][catIndex]

        # subtract to extract correct answer
        temp = catcell - float(data)

        # multiply by random number to hide values (also ruins the 'valid' cell)
        rows[i] = [temp*randomize() + cell for cell in rows[i]]

    return rows, header

# Carol finding Alice's query in the encrypted data without leaking any additional info to Alice
def Carol (category, data):
    header, rows = get_rows('encrypted.pickle')
    return compute_matching(header, rows, category, data)

def parse(category, data):
    # create parser for data and category
    parser = argparse.ArgumentParser(description='Given a category and data, query with carol and
    # parse if no input
    if not category:
        parser.add_argument('--category', help='The category to match. "valid" cannot be searched
    if not data:
        parser.add_argument('--data', help='The data to match')
    cmdline = parser.parse_args()

    if not category:
        # check for category
        if cmdline.category:
            category = cmdline.category
        else:
            category = input('Enter category: ')

    if not data:
        # check for data
        data = ''
        if cmdline.data:
            data = cmdline.data
        else:
            data = input('Enter data: ')

    # prevent 'valid' from being searched as a category
    if category in ['valid']:
        sys.stderr.write('Error: "valid" cannot be searched\n')
        parser.print_help()
        sys.exit(1)

    # prevent invalid data from being searched
    try:
        float(data)
    except ValueError:
        sys.stderr.write('Error: "{}" is not a float or int\n'.format(data))
        parser.print_help()
        sys.exit(1)

    return category, data

# This is mimic Alice asking Carol for information
def main(category=None, data=None):
    pklfile = 'encrypted_carol.pickle'

    category, data = parse(category, data)

    # search category and data on alice's encrypted pickle file
    rows, header = Carol(category, data)

    # write to carol's enrypted pickle file
    mydf = pd.DataFrame(rows, columns=header)
    pd.to_pickle(mydf, pklfile)

if __name__ == '__main__':
    main()
```

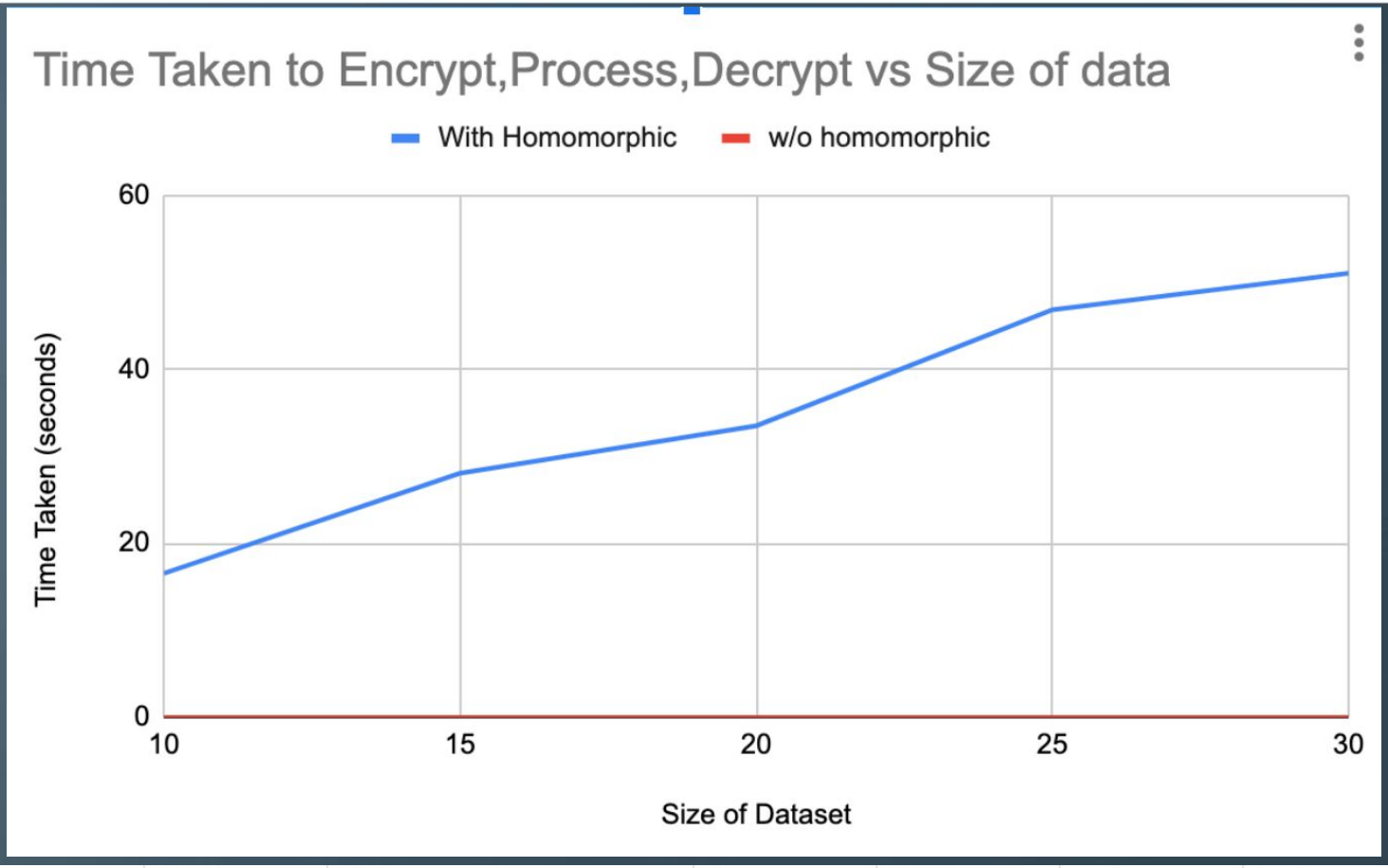
Darryl

- Performs matching on unencrypted data
  - **insecure, leakage of information**

```
# Darryl finding Alice's query in the unencrypted data
def Darryl (category, data):
    answer = list()
    csvfile = 'Heart.csv'

    # read csvfile
    with open(csvfile, newline='') as csvhandle:
        reader = csv.reader(csvhandle)
        rows = list(reader)

    header = rows[0]

    if category not in header:
        sys.stderr.write('Error: "{}" is not a valid category\n'.format(category))
        sys.exit(1)

    # get index
    catIndex = rows[0].index(category)

    rows = rows[1:]
    for row in rows:
        # append if string matches
        if float(row[catIndex]) == float(data):
            answer.append(row)
    return answer
```


Demonstration

Carol (with homomorphic) vs Darryl (no homomorphic) Performance



n	With Homomorp	w/o homomorphic	alice_enc	alice_dec	carol	daryl
10	16.55061483	0.000486135482	12.51676798	3.532279015	0.5015678406	0.000486135482
15	28.06184602	0.000483274459	21.17903304	6.074609041	0.8082039356	0.000483274459
20	33.49507999	0.000452041626	25.40655494	7.133583069	0.954941988	0.000452041626
25	46.84245682	0.000530958175	35.08054996	10.44200802	1.319898844	0.000530958175
30	51.03205252	0.000551939010	38.77906609	10.78343225	1.469554186	0.000551939010

Conclusion

Successes:

- Encryption and string matching were successful
- Team Doughnut was amazing

Improvements:

- Decryption takes too long - could decrypt first column, filter data, and decrypt just the necessary rows
- Encryption should be signed

Sources:

PHE: Python 3 Library For Partially Homomorphic Encryption Using Paillier: https://python-paillier.readthedocs.io/en/develop/usage.html#role-1