CSCI 531 Semester Project: Designing a Secure Decentralized Audit System
Tanmay Ghai

Note: For the semester project, I chose Option # 1: "Explore applicability of various cryptographic schemes" which is why for the core objectives, I only implemented a working prototype for the "privacy", "queries", and "immutability" goals (with identification/authentication in-built). For my exploration, I decided to look into Homomorphic and Fully Homomorphic Encryption with the help of the open source, lattice-based cryptographic library PALISADE. In particular, I explored the BFV and BGV (I switched CKKS from my proposal for BFV due to data applicability as CKKS is much more suitable for real-numbered, fractional data) schemes (rns variants) and their privacy + performance impact with encryption and decryption. Please check the section titled "**Homomorphic & Fully Homomorphic Encryption Discussion"** for more details on this component.

Introduction and Background

As stated in the project assignment, EHR data has become increasingly digital, and with that the security and privacy risks have also increased tremendously. Here's where an audit log can help: *audit logs* are a security mechanism, storing relevant and chronological records, acting as "evidence", indicating any transactions or operations that may have occurred by authorized entries over the course of time. The log file can thus be used to trace-back or restructure EHR data that may have been operated upon if need be, which is why, even the confidentiality and integrity of the log itself is of utmost importance.

For my implementation of this project, I focused on three main goals that must be achieved: *privacy*, *identification & authorization*, and *queries*, and *immutability* each are described briefly below:

- **Privacy**: since we are dealing with sensitive data, we must make sure that unauthorized entries and adversaries may not be able to access or retrieve patient-specific or sensitive data.
- **Identification & Authorization**: this goes hand in hand with privacy, but in particular, all users of the system must be able to provide identification that authorizes them to access the system. For the purposes of this project, we limit access to 5 different patients, and 2 audit companies, acting as "auditors".
- **Queries**: since we are an auditing system, users that wish to access records may be able to do so, so long as they are authorized. Patients and Auditors have separate authentication and are specified and authorized for different queries on the audit log data.
- **Immutability:** as with any secure system, as authorized users access the system, they leave a "trail". This allows us to verify, if at any point, an user (acting as an adversary) tries to delete or alter any existing audit data. The system should and will detect the nature of this transgression and will log it as well.

The overarching goal of this system is to defend the audit data of an assumed EHR data system, this does not guard the EHR data itself, rather it takes the audit log as input and operates via the above goals, over it. For simplicity purposes, we assume the audit log to come in .csv form (titled `audit_log.csv`), initially with ~20 records, and to have the following columns:

date_time: representing the date the EHR data was accessed
patient_id: representing the patient's unique id for whom the data was accessed (# from 1-5)
user_id: representing the username a user creates to access the auditing system
action_type: representing the action taken by the entity accessing the EHR data

Below is a snippet showing what 20 columns of a sample audit log may look like. For sample data, I used randomly generated

```
date_time, patient_id, user_id, action_type
2021-1-5,1,kdurant,create
2021-1-8,5,kbryant,query
2021-1-12,3,mjordan,print
2021-1-15,4,ljames,copy
2021-1-17,2,rwestbrook,delete
2021-1-29,1,scurry,change
2021-2-7,1,dlillard,query
2021-2-14,2,pgeorge,print
2021-2-18,3,jharden,delete
2021-2-26,4,djordan,create
2021-3-1,5,ldort,copy
2021-3-7,5,salexander,query
2021-3-8,5,njokic,query
2021-3-20,1,jembiid,print
2021-3-25,2,jbrown,copy
2021-4-3,3,jtatum,change
2021-4-4,5,tbryant,delete
2021-4-5,3,zlavine,create
2021-4-9,2,agordon,create
2021-4-13,4,jmurray,copy
```

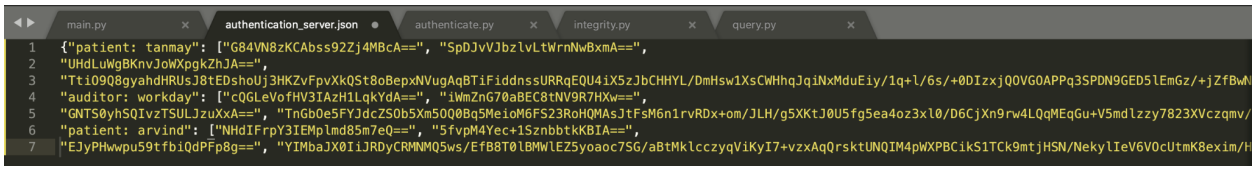
dates between the start of 2021 and the current date, and for user_id's I used first initials + full last names of existing and current NBA players.

System Design & Cryptographic Components

My auditing system is broken up into various files, each completing a particular concern/goal the system seeks to address. **Note: to run the files, make sure to start up a python virtual-environment with Python 3.7.3+ and pycryptodom >= 3.6 & < 3.10(some functionality is broken on their end in newer versions).** In the main component (disregarding work in PALISADE for now), the following files exist:

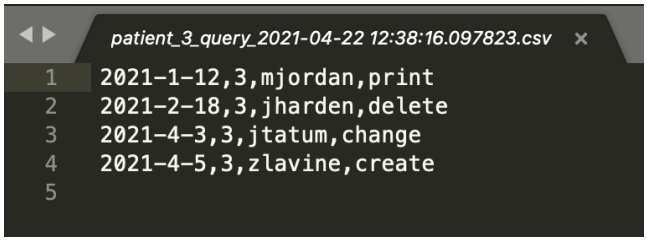
- **main.py:** this is the file to execute when trying to access the auditing system. It will start with a welcome message and will prompt the user to select from a list of following commands. It is configured to allow for: 'registering a user', 'retrieving a system identifier` (to demonstrate how encryption/decryption works for authentication, `querying audit records`, `immutability verification`, or `exit the system`.
    - Uses the python library **inquirer** (https://pypi.org/project/inquirer/) to help with end-user specification and prompting
    - Also, encrypts the existing `audit_log.csv` file that comes in "clear" to the system and stores it in a file called `audit_log.csv.enc`. It uses AES-128 to do the encryption process with the help of **pycryptodome** (https://pypi.org/project/pycryptodome/) as a library for encrypt/decrypt functionality.
- **authenticate.py:** this file handles all the authentication functionality that exists within the system and will be called upon at various times to support the other goals. It contains methods for registering a user, retrieving a user's patient or auditor id, authenticating a patient, authenticating an auditor, and storing authentication data in a "server" (server stubbed as a file called `athentication_server.json`). **In a nutshell, cryptographic-ally, this component is composed of RSA signatures ("PKCS#1" standard) for signing/verifying users, SHA-512 hashes to create digital signatures to be verified, and AES-128 encryption used to "store" sensitive authentication data in a stubbed server.** I chose this form of authentication because for RSA digital signatures, I found the **PKCS#1 standard** to be the most widely adopted standard and it is described in RFC 8017. For my purposes, I used the **PKCS#1 v1.5** version and referenced this tutorial for an introduction to the concepts: https://cryptobook.nakov.com/digital-signatures/rsa-sign-verify-examples.
    - The system requires a 16-byte or 128 bit key for AES-128, which was randomly generated once, but is stored to maintain consistency across encryption and decryption.
    - register_user(): prompts the user to provide a user_id as well as to describe whether their intended use of the system is as a `patient` or an `auditor`. Based on this, the system generates a digital signature, which uses pycryptodome to hash of the user's input user_id (via SHA-256) and an RSA key-pair generated via **pycryptodome RSA** (https://pycryptodome.readthedocs.io/en/latest/src/public_key/rsa.html). The digital signature is of the form of a 1024-bit integer and corresponds to the RSA key size, and as mentioned above it follows the PKCS# v1.5 scheme. AES-128 is then applied in CBC mode (padding and un-padding functionality also provided by the crypto-utils of the pycryptodome library) and depending on whether the user input patient or auditor, a random identifier between (1 and 5) or between (1 and 2) is assigned to them, respectively. Additionally, the use of base 64 encoding (https://docs.python.org/3/library/base64.html) was helpful in converting bytes to store in `utf-8` format in the authentication server, for easy read/write access.
    - All this data is encrypted via the AES cipher and is stored in a dictionary "authentication_keys" where the key is a string containing "patient: " or "auditor: " + the user's user_id, and the value is a tuple of the cipher's iv, the encrypted patient/auditor id, the encrypted user id, and the encrypted digital signature established. Note, in addition, the RSA key-pair is stored for later access (i.e.

querying) in a file [user_id].pem as it uses PEM encoding utilized by pycryptodome. An example of what this would look like in the authentication server is shown below:



- retrieve_user(): this is mainly for use-case screenshots in the **System "In Use" Examples** section, but theoretically can be used in situations where a user would like to identify themselves and retrieve back their patient/auditor id
    - It uses the same pycryptodome library for decryption purposes by checking against the authentication server and provides back the corresponding identifier based on the user_id and context provided.
- authenticate_patient(user_id, patient_id) & authenticate_auditor(user_id, auditor_id) both individually authenticate a given tuple of user_id and patient/auditor id that is provided to the system. These methods are particularly useful in `query.py`, but essentially asks for user id, patient/auditor id, and then applies the digital signature logic to determine validity of the user. This is done by decrypting the stored digital signature via AES (created when the user was registered), and using a "verifier" to verify the validity of the new hash of the originally signed "message", which in this case is the user_id (also decrypted via AES).
- store_in_server() simply does a local json dump of the authentication_keys dictionary and stores it in the json file titled `authentication_server.json`.

- **query.py:** this file handles querying of data by patients and auditors. Execution starts by prompting the user to identify themselves as a `patient` or an `auditor` and based on that input either calls query_as_a_patient() or query_as_an_auditor(). Additionally, there is a method called decrypt_audit_log() which is a utility function that locally decrypts the encrypted audit log file `audit_log.csv.enc` and stores the temporary results in a temporary file `temp_dec.csv` which is used for calculating the queried data results, and is deleted by the os once the operation is finished. The querying methods call two helper methods: query_audit_log_for_patient(patient_id) and query_audit_log_for_auditor(auditor_id, patient_ids)
    - query_as_a_patient() requires 2 inputs from the user: public_key_file, and patient id, and uses these as parameters for the authenticate_patient(public_key_file, patient_id) in the authenticate.py file. If the authorization passes, it allows for the user to pick (y/n) whether they would like to query the audit log; if yes, then decrypt_audit_log() is called and query_audit_log_for_patient(patient_id) is called
        - query_audit_log_for_patient(patient_id) uses the **csv** (https://docs.python.org/3/library/csv.html) in-built library to parse the `temp_dec.csv` file and build (write) a collection of audit log records that match the patient_id input. Here is a stack-overflow reference I used for building the query via csv writer



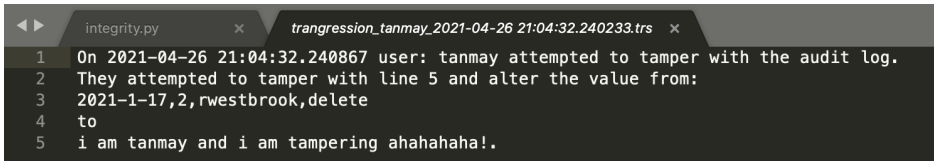
(https://stackoverflow.com/questions/41625986/how-to-filter-out-specific-data-from-a-csv-via-python). This is because, for privacy reasons, patients are only allowed to query for their own data. All audit records pertaining to their unique patient identifier will be

returned. This queried data is stored in a csv file titled “patient_” + [patient_id] + “_query_” + [current_date_time].csv. On the right is an example screenshot of a query for a patient with patient_id = “3”.

- query_as_an_auditor(), similarly, requires 2 inputs from the user: public_key_file, and patient id and uses these as parameters for the authenticate_auditor(public_key_file, patient_id) in the authenticate.py file. If authentication passes, it allows for the auditor to pick (y/n) if they would like to query the audit log; if yes, it requires a space delimited list of patient ids that the auditor would like to query (i.e. 1 2 3). It uses these in addition to the auditor_id and passes it as parameters to query_audit_log_for_auditor(auditor_ids, patient_ids)

  - query_audit_log_for_auditor(auditor_ids, patient_ids), maintaining consistency, builds and writes the corresponding/matching audit records that correspond to the list of patient ids passed in. The results are stored in a csv file titled “auditor” + [auditor_id] + “_query_” + [current_date_time].csv. On the left is an example screenshot of a query for an auditor with auditor_id = “1” who queried for patients 1, 3, and 5.

- **integrity.py:** this file walks through an immutability verification check a user may try out, where the user is authorized and authenticated, but when they try to alter (either delete or modify) the existing audit log, the transgression is detected and logged accordingly. It does this with the help of a Merkle-Tree data structure, and utilizing “audit-trails” that are established for any data-chunk (in this case, a data chunk corresponds to a line in the audit-log). Verify_audit_trail (chunk_hash, audit_trail) verifies the authenticity of any chunk from the audit trail and can be used to ensure against any tampered chunk. If a chunk is modified/deleted and then checked against the audit trail, it will be detected and reported since it wasn’t originally present in the Merkle-Tree. This design paradigm utilizes Merkle Trees for data and consistency verification, two abilities & features that make them tremendously useful data structures for security and privacy proofs.
  - The **majority of the code** (the MerkleNode, MerkleTree classes as well as verify_audit_trail(chunk_hash, audit_trail) are from the following source: https://www.codementor.io/blog/merkle-trees-5h9arzd3n8#merkle-trees-in-action. The file integrity.py also contains a reference to this blog.
  - The execution of the file starts by again asking the user whether they are a `patient` or an `auditor` and depending on that, calls authenticate_patient() or authenticate_auditor(). It also does a local decryption of the audit log and builds a Merkle Tree for each “data chunk” (i.e. each individual line).
    - It allows for the user to provide a line to “audit”, first by constructing an audit trail of the lined-chunk, and then by verifying the audit trail to ensure the chunk was indeed originally part of the tree and the trail is consistent.
    - Then, it allows the user to “modify”/”delete” the data chunk by inputting a new tampered_chunk_hash, which then is checked against the audit trail. If the value the user inputs is different from the original value, then the markle_tree.get_audit_trail function will return false, and if this is the case,



an instance of “tampering” of the audit logs has just occurred. The system will detect this, and report it by logging a transgression in the file: “transgression_” + [user_id] + “_” + [current_date_time].trs. An example of a transgression where a user named “tanmay” tried to alter the record `2021-1-17,2,rwestbrook,delete` to `my name is tanmay ahahaha.` is shown to the right:

System “In-Use” Examples

In this section, I will briefly walk through examples of the different functionalities present in my system:

Register a patient and an auditor:

```
~/us/csci53/secure-audit-system python3 main.py
Welcome to Tanmay Ghai's Decentralized Secure Audit System! Please choose from the following choices:
[?] What would you like to do?: Register a user
 > Register a user
   Retrieve patient/auditor id
   Query audit records
   Immutability verification
   Time Encryption/Decryption for Audit Log
   Exit system

Please enter your user_id to be registered: tanmay
Please enter 'patient' if you are a patient or 'auditor' if you are an audit company: patient
You have identified yourself as a patient, your patient id is: 1
The following user: tanmay has been registered successfully.
Your RSA keypair has been stored in: tanmay.pem

[?] What would you like to do?: Register a user
 > Register a user
   Retrieve patient/auditor id
   Query audit records
   Immutability verification
   Time Encryption/Decryption for Audit Log
   Exit system

Please enter your user_id to be registered: workday
Please enter 'patient' if you are a patient or 'auditor' if you are an audit company: auditor
You have identified yourself as an auditor, your auditor id is: 2
The following user: workday has been registered successfully.
Your RSA keypair has been stored in: workday.pem
```

Querying as a valid patient and auditor:

```
Welcome to Tanmay Ghai's Decentralized Secure Audit System! Please choose from the following choices:
[?] What would you like to do?: Query audit records
   Register a user
   Retrieve patient/auditor id
 > Query audit records
   Immutability verification
   Time Encryption/Decryption for Audit Log
   Exit system

Are you attempting to query as a 'patient' or an 'auditor'? Please enter: patient
Please provide your RSA public key file: tanmay.pem
Please provide your patient ID: 1
Your Digital Signature is valid.
You have been successfully authenticated as a registered patient.
Would you like to query your patient records? (y/n): y
Your queried data has been stored for your usage here: patient_1_query_2021-05-05 10:12:04.307464.csv

[?] What would you like to do?: Query audit records
   Register a user
   Retrieve patient/auditor id
 > Query audit records
   Immutability verification
   Time Encryption/Decryption for Audit Log
   Exit system

Are you attempting to query as a 'patient' or an 'auditor'? Please enter: auditor
Please provide your RSA public key file: workday.pem
Please provide your auditor ID: 2
Your Digital Signature is valid.
You have been successfully authenticated as a registered auditor.
Would you like to query some patient records? (y/n): y
Please provide a list of patient id's you would like to query for (separated by whitespace): 1 2 3
Your queried data has been stored for your usage here: auditor_2_query_2021-05-05 10:12:35.371411.csv
```

Invalid re-registration:

```
Welcome to Tanmay Ghai's Decentralized Secure Audit System! Please choose from the following choices:
[?] What would you like to do?: Register a user
 > Register a user
   Retrieve patient/auditor id
   Query audit records
   Immutability verification
   Time Encryption/Decryption for Audit Log
   Exit system

Please enter your user_id to be registered: tanmay
Please enter 'patient' if you are a patient or 'auditor' if you are an audit company: patient
This user is already registered in our system. Please use your digital signature and select a different option. If you forgot, misplaced or
 lost your patient/auditor id, you may choose the retrieve option.
```

Retrieval of system identifier (only to show decryption of AES works):

```
Welcome to Tanmay Ghai's Decentralized Secure Audit System! Please choose from the following choices:
[?] What would you like to do?: Retrieve patient/auditor id
   Register a user
 > Retrieve patient/auditor id
   Query audit records
   Immutability verification
   Time Encryption/Decryption for Audit Log
   Exit system

Please enter your user_id to be checked against our system: tanmay
Please enter 'patient' if you are a patient or 'auditor' if you are an audit company: patient
Since you identified as a patient, your patient id is: 1
```

Unauthorized querying (authentication & privacy):

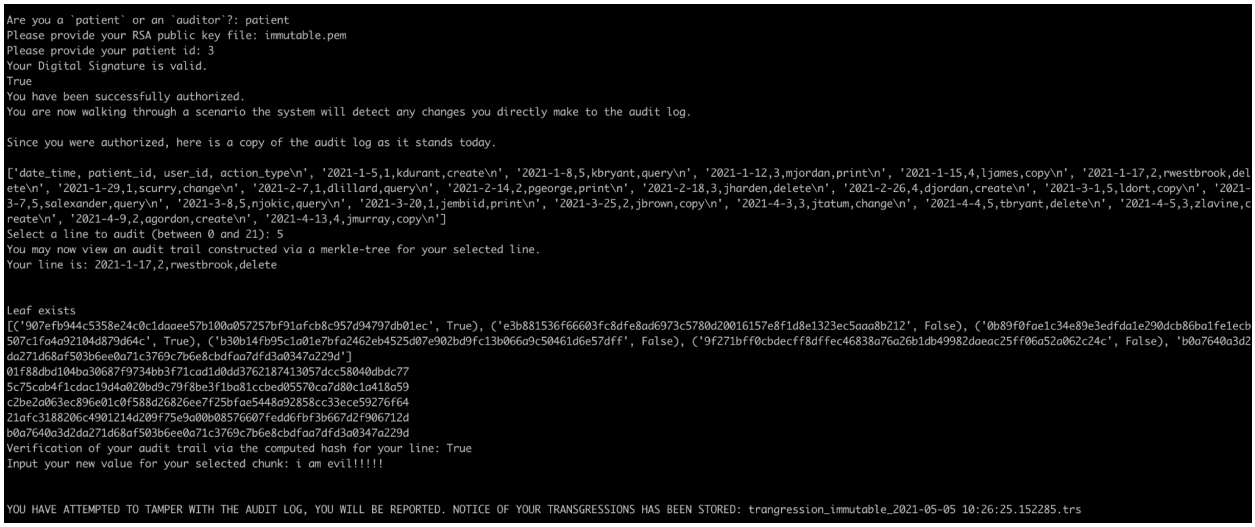
```
[?] What would you like to do?: Query audit records
   Register a user
   Retrieve patient/auditor id
 > Query audit records
   Immutability verification
   Time Encryption/Decryption for Audit Log
   Exit system

Are you attempting to query as a 'patient' or an 'auditor'? Please enter: auditor
Please provide your RSA public key file: afsafasfaf.pem
Please provide your auditor ID: 1
You are unauthorized to perform this action!
```

```
[?] What would you like to do?: Query audit records
   Register a user
   Retrieve patient/auditor id
 > Query audit records
   Immutability verification
   Time Encryption/Decryption for Audit Log
   Exit system

Are you attempting to query as a 'patient' or an 'auditor'? Please enter: patient
Please provide your RSA public key file: tanmay.pem
Please provide your patient ID: 5
You are unauthorized to perform this action!
```

Immutability Verification:



System Limitations and Assumptions

As with any system, there are limitations and vulnerabilities. In the way that I devised my system, here are some assumptions and thus limitations incurred as a result:

- First and foremost since I chose option #1, this implementation does not involve the decentralization, but I did go ahead and implement identification and authorization.
- For identification and authorization, as I mentioned, I used the help of RSA digital signatures, which as we talked about in lecture, can be really useful for identification, but maybe aren't the best direct approach for authentication as the "goals" of the two are different: with authentication, the end goal is to tend to have a user "around" for an entire session; this is one limitation in my system (which I address via another means). Register_user() function acts as the signer of the digital signature based on the input from the user (their user_id) as well as the place that does the public key distribution (which in practice would not be a great design, but for prototyping purposes works efficiently and clearly), and authentication of the user is done via the digital signature verification process. **Note**: for identification and authorization, once identified, I do assign a unique identifier for a patient or auditor such that they can access the system via that identifier rather than their digital signature, once they have been initially authenticated, however, the propagation of this verification with all options within the system (querying, immutability, etc) is not completely synchronized.

- Additionally, one potential security issue with my system design is that it would potentially be susceptible to man-in-the-middle attacks where an adversary was able to intercept a user's public key file. However, this is why, for any operation, I also ask for the system identifier (which could also be potentially compromised, but at least it acts as a 2nd barrier of defense).
- Also, for RSA digital signatures, I am using the **PKCS#1 v1.5** standard, which as we discussed in lecture has known attacks (the Bleichenbacher attack) against it. I used this standard from the pycryptodome library because of its convenience (they also have support for **PKCS#1 v2.0 - OAEP** based on optimal asymmetric encryption padding, but that was not working with my machine's version of python3 due to some dependency issues.
- To address confidentiality, I used AES-128 to encrypt and store the audit log data as well as store all the key/authentication server data in an encrypted fashion. My 128 bit key is only generated pseudo-randomly once and then is stored locally for use. I decided on this approach because it allows for synchrony of encryption/decryption without having to manage decentralization or distribution, but is definitely not the beast approach in terms of security as it acts as a single point of failure (once the key is compromised, there goes the encryption).
- The use of digital signatures also provide my auditing system message integrity and non-repudiation as the signature ensures the message is coming from the signer and was not altered (verified via the hash-then-sign paradigm discussed in class).
- Exploration of using HE & FHE potentially to increase security guarantees of my system is located below, with emphasis on the performance overhead and cost it might take to achieve that higher security goal.

Homomorphic & Fully Homomorphic Encryption Discussion

Homomorphic Encryption (HE) is a representation of a public key, cryptographic scheme that essentially allows for the use of computation to be done over encrypted data itself. More formally, given n different plaintexts *(p_1 ... p_n)*, and a function f, HE allows for the function *f* to be evaluated directly on the *n* corresponding ciphertexts *(c_1 ... c_n)*. Note, it does this while not needing any decryption for the underlying data or any leakage of the decryption key. HE, however, can also be looked at as a "set" or "class" of various types of schemes that correspond to certain operations, computational levels, and theoretical guarantees (i.e. some are based on integer arithmetic, real numbers, lattices, RWLE, etc). There is, for example, *partial* or *somewhat* HE, where partial relates to schemes that can operate only one type of circuit gate (HE is based on arithmetic and boolean circuitry), i.e. addition OR multiplication, and somewhat can perhaps evaluate both gates, but only for a subset of all possible circuit combinations. There is also *full* and *leveled*, which are talked about below. It is also important to note that HE is based on the RWLE problem and as of today is currently resistant to cryptanalysis as well as quantum attacks.

Fully Homomorphic Encryption (FHE) is a more stringent version of HE and represents schemes that allow for an "arbitrary" composition of arithmetic and boolean circuitry, and can allow for an "unbounded depth" of computation. As a quick aside, in many practical applications, FHE schemes are applied as leveled versions of themselves, meaning that they can also evaluate the same arbitrary circuits, however, they are limited by some level of pre-determined computational depth. In particular, FHE also requires that we uphold circuit privacy, which essentially means that in evaluation of the circuit, the evaluation function *f* should also be "hidden" from leakage in addition to the calculations *f(c_1 ... c_n).* Additionally, it should be noted that the first FHE scheme was introduced in 2009 by Craig Gentry who based his construction on ideal lattices.

For my exploration into these two sections of cryptography, I focused on understanding the performance of encrypting and decrypting my audit log (a .csv file) with two schemes, the he *Brakerski-Fan-Vercauteren* (BFV) and the *Brakerski-Gentry-Vaikuntanathan* (BGV) FHE constructions as compared to AES-128. Cryptographically, BFV (in terms of what made it a novel construction), is that it introduces a new way of dealing with error growth, without

needing a change of modulus. This allows it to be more elegant and less memory consuming as compared to BGV (w.r.t certain input), while also being constrained in some ways as it only allows integer input for many operations. BGV, on the other hand, was introduced by Gentry as being the first "real, practical" FHE scheme. It introduced improvements to Gentry's original FHE scheme such as a relinearization technique to avoid quadratic growth for cipher sizes, a switch modulus to reduce noise after multiplication, and a double chinese remainder theorem ciphertext form (DCRT) as well as batching such that multiple plaintexts could be "packed" or slotted into one ciphertext (this greatly reduces computational overhead).

As I did with AES-128, I encrypted each record in the .csv individually (Note: I used the help of the **csvstream** (https://github.com/awdeorio/csvstream**)** library to read in the csv file in C++), and you can see my findings for performance cost below:

As you can see, the delta between simple AES, and implementations in PALISADE are stark in terms of time cost. The HE schemes take approximately 3-10x the time cost of AES-128 (which is to be expected given the large overhead incurred by applying homomorphic operations). It is also important to note that HE (as of today), only really applies in confidentiality settings and is not great for addressing non-repudiation and integrity. My work for this part is in the C++ file `secure_audit_system_fhe_explorations.cpp` and **to run this make sure you follow build instructions from PALISADE here: https://gitlab.com/palisade/palisade-development.**

Time cost using AES-128 (encrypt/decrypt):

```
time taken to encrypt audit log data (aes-128): 1.0828971862792969 ms
time taken to encrypt audit log data (aes-128): 1.2958049774169922 ms
```

Time cost using BFVrns in PALISADE (encrypt/decrypt):

```
~/De/palisade-development/build | master#v1.10.2-dev +289 !22 ?11    bin/examples/pke/secure_audit_system_fhe_explorations    ✔ | venv
time taken to encrypt audit log data (bfvrns): 1017.43 ms
time taken to decrypt audit log data (bfvrns): 115.681 ms
~/De/palisade-development/build | master#v1.10.2-dev +289 !22 ?11    ✔ | venv
```

Time cost using BGVrns in PALISADE (encrypt/decrypt):

```
~/De/palisade-development/build | master#v1.10.2-dev +289 !22 ?11    bin/examples/pke/secure_audit_system_fhe_explorations
time taken to encrypt audit log data (bgvrns): 307.418 ms
time taken to decrypt audit log data (bgvrns): 70.2139 ms
~/De/palisade-development/build | master#v1.10.2-dev +289 !22 ?11    ✔ | venv
```

Additionally, as I mentioned HE and FHE can be very useful for applying computations over encrypted data, which is why I also sought to see if there were ways to see if patient records could be aggregated while they were encrypted. I did this by extracting the encrypted patient id's and subtracting them while in encrypted form. If, when decrypted, the result of the subtraction was == 0, then we know we have found a match. Below you can see an example of how I aggregated records for where the patient_id's matched (a step in the process of achieving the querying goal). Here is a sample output of a run:

References (beyond those referenced within the doc)

https://palisade-crypto.org/

https://eprint.iacr.org/2011/277

https://sites.math.washington.edu/~morrow/336_14/papers/mitchell.pdf

https://crypto.stanford.edu/craig/craig-thesis.pdf

https://eprint.iacr.org/2016/381.pdf

https://tools.ietf.org/html/rfc8017

https://getrfc.com/rfc6594

https://homomorphicencryption.org/standard/


```
patient: 0 and patient: 5 are the same!
patient: 0 and patient: 6 are the same!
patient: 0 and patient: 13 are the same!
patient: 1 and patient: 10 are the same!
patient: 1 and patient: 11 are the same!
patient: 1 and patient: 12 are the same!
patient: 1 and patient: 16 are the same!
patient: 2 and patient: 8 are the same!
patient: 2 and patient: 15 are the same!
patient: 2 and patient: 17 are the same!
patient: 3 and patient: 9 are the same!
patient: 3 and patient: 19 are the same!
patient: 4 and patient: 7 are the same!
patient: 4 and patient: 14 are the same!
patient: 4 and patient: 18 are the same!
patient: 5 and patient: 0 are the same!
patient: 5 and patient: 6 are the same!
patient: 5 and patient: 13 are the same!
patient: 6 and patient: 0 are the same!
patient: 6 and patient: 5 are the same!
patient: 6 and patient: 13 are the same!
patient: 7 and patient: 4 are the same!
patient: 7 and patient: 14 are the same!
patient: 7 and patient: 18 are the same!
patient: 8 and patient: 2 are the same!
patient: 8 and patient: 15 are the same!
patient: 8 and patient: 17 are the same!
patient: 9 and patient: 3 are the same!
patient: 9 and patient: 19 are the same!
patient: 10 and patient: 1 are the same!
```