PALISADE

HOMOMORPHIC ENCRYPTION FOR PALISADE USERS: TUTORIAL WITH APPLICATIONS

Homomorphic Encryption Serialization for Applications

Dr. David Bruce Cousins, Director Duality Labs
dcousins@dualitytech.com

Agenda

- The role of Serialization and Deserialization in encrypted applications
- Basic Examples from the PALISADE distribution
- Examples of Systems of Cooperating PALISADE Processes

The role of Serialization in Applications

Serialization and deserialization in C++

Serialization in Computing

- Serialization is the process of translating a **data structure** or **object**
  - into a **format** that can be **stored** or **transmitted** (i.e. bytes).
  - and **reconstructed** later (possibly on another system).
- This reconstruction is known as **deserialization** and produces a “clone” of that object.
- Applications of serialization include:
  - **Messaging** →transferring data between programs on the same or different computers
  - **Storage** → saving and recalling data in databases or files.
  - **Other computing models** → remote procedure calls (e.g. SOAP, etc), distributing objects as components (COM, CORBA, etc.)
- It is an essential tool for building real-world systems that use PALISADE.

Serialization in C++ is not so easy

- C++ has no standard serialization library
- Serializing and deserializing complicated objects that use pointers is not a simple task.
  - Objects can be made up of other objects and data structures
  - That data can be distributed throughout non-contiguous memory
  - Pointer References are not portable, you cannot just copy pointer values into a file
- Need to gather memory blocks and re-reference the pointers locally.
- Changes in object code may require changes in serialization code

Example of a Complicated Data Layout

- CKKS Ciphertext Object:
  - A handful integer members (to keep track of data dimensions, levels, multiplicative depth etc.)
  - Shared pointer to a **vector<DCRTPoly>**
    - **DCRTPoly** → handful members (dimensions, moduli) and pointers to **vector<shared_ptr<vector<uint64_t>>>**
    - Outer dimension is 2 or 3, inner dimension is **ring_size** (large)
  - Shared pointer to Metadata **map <string, shared_ptr<Metadata>>**
    - Metadata→ arbitrary parameters for application-level code
- Very complicated to serialize, you need to keep track of all these pointers to pointers to pointers….
- Fortunately, there are libraries that take most of the work out of this…

PALISADE uses Cereal for Serialization

- C++ header only library for serialization
- https://uscilab.github.io/cereal/
  - Supports STL objects and shared_ptr, unique_ptr
  - Serializes to JSON (human readable) or binary (smallest size)
  - Customizable to support PALISADE objects.
- We won't cover the inner details of Cereal
- Generally, PALISADE needs to add a pair of serialization/deserialization functions for complicated objects like CryptoContext and Ciphertext
  - Simple functions that allow Cereal to parse complex objects during compile time
  - Figures out serialization and deserialization routines automagically.
  - The penalty is those functions take a LONG time to compile – so separate functions that use serialization into a separate compilation module if you can.

Basic Examples from the PALISADE distribution

Serialization and deserialization functions

Example Code for Serialization in PALISADE

- Serialization is supported in all public key encryption schemes
- Functional unit tests (these can be hard to read):
  - src/core/unittest/UnitTestSerialize.cpp
    - Uses generic Serial::Serialize() and Serial::Deserialize() on various basic PALISADE data types.
  - src/pke/unittest/UnitTestSerialize*.cpp
    - BFV, BFVrns, BFVrnsB, BGVrns, CKKS, NULL, StSt
    - Uses Serial::Serialize() / Serial::Deserialize(), and additional specific functions to serialize complicated objects from an active CryptoContext
- Examples in pke/examples (often easier to understand):
  - Single thread programs that serialize and save CryptoContext, key and ciphertext objects to disk, then load them into new variables.
  - Simple-integers-serial.cpp (BFVrns)
  - Simple-integers-serial-bgvrns.cpp
  - Simple-real-numbers-serial.cpp (CKKS)

Examples of Systems of Cooperating PALISADE Processes

Serialization and deserialization between multiple heavyweight processes

Passing Objects Between Multiple Heavyweight Processes is Complicated

- A system composed of cooperating processes:
  - needs to pass serialized objects to each other through files, sockets or shared memory.
  - needs mechanisms to synchronize in addition to passing serialized data, such as file locks, mutexes or semaphores.
- These processes do not share the same memory map!
  - Unlike threads
- We've built a small repository to demonstrate sample systems
  - https://gitlab.com/palisade/palisade-serial-examples
  - Currently has one example but we will be constantly adding more
  - Separate repo so we can use Boost interprocess mechanisms

Client-Server for distributed secure computation

- src/real-server CKKS-based distributed encrypted computation

real-server.cpp represents secure repository of private data

real-client.cpp processes secure data remotely

Builds and serializes the CryptoContext, public key and various computation keys to files

Builds CryptoContext and keys from deserialized files

Sends data request

Receives data request, encrypts requested data and serializes it to files

Receives encrypted ciphertext

Computes on encrypted data and encrypts more data with public key

Sends resulting ciphertext to Server

Receives encrypted result

Decrypts and verifies result

time

Object transfer is done with file I/O. Synchronization done with Boost **named_mutex** → prevents files from being read before writing is completed by other process.

Server: Sending the CryptoContext and Keys

- For flexibility, serializing a CryptoContext does not share everything within that context (for example, keys)
  - The server must serialize multiple components
  - The client must deserialize these and may also regenerate other components
- Most objects can be serialized directly with SerializeToFile()
- Server::writeData() serializes the following components:
  - CryptoContext with Serial::SerializeToFile()
  - publicKey with Serial::SerializeToFile()
  - EvalMultKey/Relinearization key with CryptoContext::SerializeEvalMultKey()
  - RotationKeys with CryptoContext:: SerializeEvalAutomorphismKey

PALISADE

Server: Notes on Sending Keys

- EvalMult and rotation (automorphism) keys are handled with special functions because there are several objects of each type associated with the CryptoContext, and their content is application dependent.
  - For example, there is one key for each index of rotation used by the application
- These functions serialize to std::ostream, so you need to open and close ofstream in the code.
- Note that EvalSumKey is not used in this example but may be required in your application.

PALISADE

Client: Receiving and Reconstructing CryptoContext and Keys

- The details are in real_client.cpp receiveCCAndKeys()
- Several key steps are needed in addition to transferring objects:
- We must clear out any PALISADE data objects when we deserialize and assemble the client CryptoContext.
  - Use CryptoContextFactory<DCRTPoly>::ReleaseAllContexts() before creating a new CC.
  - Load client_CC, (our CryptoContext) → Serial::DeserializeFromFile())
- Clearing the keys is important before loading them
  - clientCC->ClearEvalMultKeys()
  - clientCC->ClearEvalAutomorphismKeys()
- Load keys
  - publicKey →Serial::DeserializeFromFile())
  - evalMultKey →client_CC->DeserializeEvalMultKey()
  - Rotation keys →clientCC->DeserializeEvalAutomorphismKey()
  - Eval sum keys would also need to be loaded if used.

PALISADE

Sending/Receiving Ciphertexts

- Server sends Ciphertexts to client with Serial::SerializeToFile()
- Client receives Ciphertexts with Serial::DeserializeFromFile()
- We could also use Serial::Serialize() / Serial::Deserialize() to give us more flexibility in building systems
  - This lets us Serialize to any ostream object
    - File → ofstream
    - Local memory → stringstream
    - Sockets → Boost socket stream
    - Shared memory → Boost interprocess shared memory.

Object sizes in this example

- FHE objects are large

ciphertext	Binary size	JSON
CryptoContext	3.1 K Bytes	66 K Bytes
ciphertext	5.6 M Bytes	44 M Bytes
Public key	5.6 M bytes	44 M Bytes
EvalMult keys	23 M Bytes	226 M Bytes
Rotation keys	91 M Bytes	1.1 G Bytes

- Their size is related to security requirements, and multiplicative depth desired.
- Only use JSON when you must (Like for human debugging)

PALISADE

Running the Example Code

- Clone, build and install PALISADE from the development repo
- Clone the PALISADE/serial-examples repo. Detailed build instructions are found in README.md

palisade@ubuntu: ~/Documents/palisade-serial-examples/b...

```
palisade@ubuntu:~/Documents/palisade-serial-examples/build$ bin/real-server
This program requres the subdirectory `demoData' to exist, otherwise you will ge
t an error writing serializations.
SERVER: creating and acquiring server lock
SERVER: computing crypto context and keys
SERVER: sending cryptocontext
SERVER: sending Public key
SERVER: sending EvalMult/reliniarization key
SERVER: sending Rotation keys
SERVER: Writing data to: demoData
SERVER: sending encrypted data
SERVER: ciphertext1 serialized
SERVER: ciphertext2 serialized
SERVER: Releasing server lock
SERVER: Acquiring client lock
SERVER: Acquiring Server lock
SERVER: Receive and Verify data
SERVER: Deserialized all processed encrypted data from client
Mult correct: Yes
Add correct: Yes
Vec encryption correct: Yes
Rotation correct: Yes
Negative rotation correct: Yes
SERVER: Releasing Server lock
SERVER: Releasing Client lock
SERVER: Cleaning up
SERVER: Exiting
palisade@ubuntu:~/Documents/palisade-serial-examples/build$
```

palisade@ubuntu: ~/Documents/palisade-serial-examples/b...

```
palisade@ubuntu:~/Documents/palisade-serial-examples/build$ bin/real-client
This program requires the subdirectory demoData' to exist, otherwise you will ge
t an error writing serializations.
CLIENT: Open server lock
CLIENT: create and acquire client lock
CLIENT: acquire server lock
CLIENT: Acquired server lock. Getting serialized CryptoContext and keys
CLIENT: public key deserialized
CLIENT: Relinearization keys from server deserialized.
CLIENT: Getting ciphertexts
CLIENT: Computing and Serializing results
CLIENT: Applying operations on data
CLIENT: encrypting a vector
CLIENT: Releasing Client lock
CLIENT: Acquiring Server lock
CLIENT: Acquired server lock. Server is done
CLIENT: Released server lock. Cleaning up
CLIENT: Exiting
palisade@ubuntu:~/Documents/palisade-serial-examples/build$
```

Exploring Further

- There is a complicated example of three heavyweight processes participating in proxy-key re-encryption
  - PALISADE src/pke/examples/pre_server uses simple file-based synchronization for ultra portability
  - A trusted Server builds CryptoContext and shares it with Alice and Bob
  - Alice send her decryption key to the server
  - Bob sends his public key to server, who generates a re-encryption key from Alice's and Bob's keys, and sends it to Bob
  - Alice can then send her encrypted data directly to Bob, who can decrypt it with a combination of the re-encryption key and his decryption key.
  - Note Alice's data could have been stored somewhere and later read by Bob.
  - Bob does not need Alice's decryption key!
- We will add this and other examples of cooperating-process PALISADE applications to the serial-examples repository

Thank you for attending!

Any Questions?