HOMOMOPHIC ENCRYPTION FOR PALISADE USERS: TUTORIAL WITH APPLICATIONS

Boolean Arithmetic with Applications

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

• Boolean Algebra/Logic Review and Circuits
• Basic Examples from PALISADE
• Simple Circuit Examples Using the PALISADE Encrypted Circuit Emulator
• Additional Circuit Examples
Boolean Logic

A quick refresher
Boolean Algebra

- The basis of all digital logic
- Many equivalent representations:
  - Circuits of Gates
  - Logic Equations
  - Truth tables
- All are equivalent!
- Any logic circuit can be manipulated into many different forms.
  - Minimize # gates
  - Minimize depth
  - Use particular gate types

Other gate combinations are easily built from these

Thanks to https://www.secs.oakland.edu/~polis/Lectures/EGR240 D5.1 BasicLogicGates.ppt
Representing Systems of Logic Gates

- Circuit diagrams / Schematics (gates and wires)
- Net lists (BLIF, EDL, Bristol Format) not very readable
- Hardware Design Languages (VHDL, Verilog, System C)
- Other..
  - PALISADE lets you connect gates together via C++
  - Good for simple systems
  - PALISADE Encrypted Circuit Emulator lets you script your own circuits, and reads in some net list formats
    - Good for circuits with 1000’s of gates
Basic Examples

From the PALISADE distribution
The “cost” of Encrypted Logic

- Memory (32 bit implementation):
  - AP (STD128): Classical FHEW
    - Bootstrapping key: 1152 MB, Key switching key: 300 MB
  - GINX (STD128): TFHE
    - Bootstrapping key: 64 MB, Key switching key: 300 MB
- Note that the key switching key size can be reduced if desired (it is a controllable parameter).
- Each encrypted bit takes ~2KB storage
The “cost” of Encrypted Logic

• Execution time
  • Executing NOT is fast (100 nsec), since no bootstrapping is performed
  • Executing AND, OR has one bootstrap performed
    • 107 msec / thread (AP STD128)
    • 143 msec / thread (GINX STD128)
  • Two options for XOR:
    • XOR - 3x slower than AND (three bootstraps, gives same failure probability as AND <2\(^{-32}\))
    • XOR\_FAST - = AND but higher failure probability <2\(^{-15}\)
      • For more details, see https://eprint.iacr.org/2020/086
  • The execution time of an encrypted circuit is strictly a function of the gate time
    • Manipulating circuit to minimize # of gates (other than NOT) gives fastest runtime
C++ Examples provided in the PALISADE release

- Sample executables are in `${root}/build/bin/examples/binfhe`
- C++ source code for these examples are in `${root}/binfhe/examples`
  - **boolean**: simple collection of gates using the GINX method.
  - **boolean-ap**: same as above except using AP method.
  - **boolean-truth-tables**: example showing basic gate output for all input combinations
  - **boolean-serial-json**: how to serialize (save to disk) the components of a binfhe crypto-system (various keys and ciphertext)
  - **boolean-serial-binary**: same as above though with binary vs json storage (much smaller files)
- Source code for sample benchmarks are in `${root}/benchmark/src/binfhe*.cpp`
Listing of boolean.cpp

```cpp
#include "binfhecontext.h"
using namespace lbcrypto;
using namespace std;

int main() {

    // Step 1: Set CryptoContext
    auto cc = BinFHEContext();
    cc.GenerateBinFHEContext(STD128, AP); // set 128 bits of security, AP

    // Step 2: Key Generation
    auto sk = cc.KeyGen(); // Generate the secret key
    cc.BTKeyGen(sk); // Generate the bootstrapping keys

    // Step 3: Encryption
    // Encrypt two ciphertexts representing Boolean True (1)
    auto ct1 = cc.Encrypt(sk, 1);
    auto ct2 = cc.Encrypt(sk, 1);

    // Step 4: Evaluation
    auto ctAND1 = cc.EvalBinGate(AND, ct1, ct2); // Compute (1 AND 1) = 1
    auto ct2Not = cc.EvalNOT(ct2); // Compute (NOT 1) = 0
    auto ctAND2 = cc.EvalBinGate(AND, ct2Not, ct1); // Compute (1 AND (NOT 1)) = 0
    // Computes OR of the results in ctAND1 and ctAND2 = 1
    auto ctResult = cc.EvalBinGate(OR, ctAND1, ctAND2);

    // Step 5: Decryption
    LWEPlaintext result;
    cc.Decrypt(sk, ctResult, &result);
}
```

Each “wire” is a ciphertext
Each “gate” is a function call
Inputs are “encrypted”
Outputs are “decrypted”
Simple Circuit Examples

From the PALISADE Encrypted Circuit Emulation repository
PALISADE Encrypted Circuit Emulator

- GitLab repo: [https://gitlab.com/palisade/palisade-encrypted-circuit-emulator](https://gitlab.com/palisade/palisade-encrypted-circuit-emulator)
  - Build instructions are in README.md, requires you to install PALISADE
- Contains prototype C++ code to
  - Parse circuit representation input files
  - Analyze and Assemble them into an intermediate form for circuit emulation (*.out file)
  - Run C++ test fixtures to generate input and test output for various circuits
  - Executes resulting logic circuit in plaintext and encrypted form – uses Open MP to evaluate encrypted gates in parallel on all available threads.
  - Stores minimum number of circuit ciphertexts
    - once all gates have fired that are fed by a node, it is deleted.
PALISADE Encrypted Circuit Emulator

- Current limitations (Aug 2020)
  - Input currently limited to “Bristol Format Circuits”
    - Format is primarily used for Garbled Circuit R&D
    - As a result it has limited I/O, limited gate types, and these circuits prefer XOR over all other gates – also not the smallest circuits possible but are a good representation.
    - Intermediate file format (*.out) is a bit primitive and fiddly, but you can use it to write your own circuits
  - Circuit management / scheduling code is done brute force
    - Executes encrypted gates on parallel threads, minimize circuit ciphertext
    - Overhead high for small circuits, negligible (2%) on large circuits.
PALISADE Encrypted Circuit Emulator

- Planned extensions
  - Add New Bristol Fashion Circuit format
  - Improve I/O definitions to allow more complex circuits
  - Optimize circuit management and internal netlist generation (currently takes a long time for large circuits)
  - Follow improvements in PALISADE binfhe performance as they get released
Example Simple Circuits

- There are two simple circuits that are provided in the distribution examples folder that we will review in this session
  - Simple Circuits - examples/simple_ckts/*
- Hand assembled circuits that demonstrate capability with minimal run time
  - **adder_2bit** - 4 bit input, three bit output adder with carry
  - **parity** - 8 bit parity generator/checker
- We will next review the process of building the description by hand
adder_2bit circuit

- Start with basic circuit
  - 
  - 
  - 
  - 

\[ Q = a + b \]

CO is carry out

https://i.stack.imgur.com/TpBpr.gif
adder_2bit circuit

- Start with basic circuit
- Label nodes
  - Input nodes need to be numbered sequentially
  - Nodes need to be R# starting from zero, but can be any order

https://i.stack.imgur.com/TpBpr.gif

https://i.stack.imgur.com/TpBpr.gif
adder_2bit circuit

- Start with basic circuit
- Label nodes
  - Input nodes need to be numbered sequentially
  - Nodes need to be R# starting from zero, but can be any order
- Label Inputs and Outputs
  - Input registers are numbered In1,0 is bit 0 of first input
    In2,1 is bit 1 of second input
    there can be one or two inputs
  - Outputs bits are labeled Out0, Out1, etc. -- there is only one output register
- These constraints are based on historical requirements of the Bristol circuit format and will be lifted in future revisions.

https://i.stack.imgur.com/TpBpr.gif
Resulting adder_2bit assembly code:

```assembly
# number input1 bits 2
# number input2 bits 2
# number output1 bits 3
# Do not edit the top 3 lines!
# 2 bit adder
# Q = a + b
# CO is carry out
#
# inputs:
# In0,0 In0,1 are a0 , b0
# In1,0 In1,1 are a1 , b1
# outputs
# Out0 is Q0, Out1 is Q1, Out2 is CO

R0 = LOAD(In1,0)
R1 = LOAD(In1,1)
R2 = LOAD(In2,0)
R3 = LOAD(In2,1)
R4 = XOR(R0, R2)
R5 = AND(R0, R2)
Out0 = STORE(R4)
R6 = XOR(R1, R3)
R7 = AND(R1, R3)
R8 = XOR(R5, R6)
Out1 = STORE(R8)
R9 = AND(R5, R6)
R10 = OR(R9, R7)
Out2 = STORE(R10)
```

The first 3 lines need to list the Number of bits for inputs and outputs
Comment lines have # in column 1

LOAD and STORE are used to Indicate inputs and outputs into the Circuit (they trigger encrypt/decrypt)

Use this format, no extra spaces
adder_2bit sample output

- Program name bin/TB_adder_2bit
- Size of input and output registers.
- Number of internal nodes
- Security parameters used
- Test input and correct output
- Plaintext circuit runtime 1 msec
- Circuit size
- Encrypted circuit run time 2113 msec, Efficiency 99.9053%
- Circuit verified correctly
Parity generator circuit

8 bits for input (9th bit can be set to 0 or used for chaining multiple circuits together for wider words)

2 bits output appropriate bit signals even or odd parity
Parity assembly code: examples/simple_ckts/parity

R9 = NOT(R0)
R10 = NOT(R1)
R11 = NOT(R2)
R12 = NOT(R3)
R13 = NOT(R4)
R14 = NOT(R5)
R15 = NOT(R6)
R16 = NOT(R7)
R17 = NOT(R8)

R18 = XOR(R9, R10)
R19 = XOR(R11, R12)
R20 = XOR(R13, R14)
R21 = XOR(R15, R16)
R22 = XOR(R18, R19)
R23 = XOR(R20, R21)
R24 = XOR(R22, R23)
R25 = XOR(R24, R25)
R26 = NOT(R25)

Out0 = STORE(R25)
Out1 = STORE(R26)
Parity test: examples/simple_ckts/parity

- Test program performs the following operations, using the same circuit as both a parity generator and parity checker. Note all data is encrypted. Decrypting the output determines the parity of the encrypted word.
Parity sample output

- Program name bin/TB_parity
- Circuit size 10 NOT, 8 XOR
- Encrypted circuit run time **3648 msec**, Efficiency 99.9%
- Circuit verified correctly
Additional Complex Circuit Examples

From the PALISADE Encrypted Circuit Emulation repository
Complex Circuit Examples for Further Exploration

Currently the repository has examples taken from https://homes.esat.kuleuven.be/~nsmart/MPC/old-circuits.html

- The arithmetic examples are relatively quick, crypto examples have >10K gates so can take a long time to process and run

<table>
<thead>
<tr>
<th>Arithmetic</th>
<th>Function</th>
<th>File</th>
<th># ANDs</th>
<th># XORs</th>
<th># NOTs</th>
<th># ORs</th>
</tr>
</thead>
<tbody>
<tr>
<td>32-bit Adder</td>
<td>TB_adders</td>
<td>127</td>
<td>61</td>
<td>187</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>64-bit Adder</td>
<td>TB_adders</td>
<td>256</td>
<td>115</td>
<td>379</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>32x32-bit Multiplier</td>
<td>TB_multipliers</td>
<td>5926</td>
<td>1069</td>
<td>5379</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>32-bit comparisons</td>
<td>TB_comparators</td>
<td>150</td>
<td>0</td>
<td>150/162</td>
<td>0</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Crypto</th>
<th>Function</th>
<th>File</th>
<th># ANDs</th>
<th># XORs</th>
<th># NOTs</th>
<th># ORs</th>
</tr>
</thead>
<tbody>
<tr>
<td>Md5 hash</td>
<td>TB_crypto</td>
<td>29084</td>
<td>14150</td>
<td>34627</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>SHA-256</td>
<td>TB_crypto</td>
<td>90825</td>
<td>42029</td>
<td>103258</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>AES 128 (No Key Expansion)</td>
<td>TB_aes</td>
<td>6800</td>
<td>25124</td>
<td>1692</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>AES 128 (Key Expanded)</td>
<td>TB_aes</td>
<td>5440</td>
<td>20325</td>
<td>1927</td>
<td>0</td>
<td></td>
</tr>
</tbody>
</table>
THANK YOU

dcousins@dualitytech.com