Introduction to Secure Multiparty Computation techniques

Claudio Orlandi

Università degli Studi di Firenze, Italy
Aarhus Universitet, Denmark
Outline

- Obfuscation
- Cryptocomputing
- Secure 2-party Computation
  - Yao’s garbled circuit
- Secure n-party Computation
  - Secret sharing-based arithmetic circuit
- Practical feasibility
Different Scenarios – Obfuscation

- \( P_1 \) wants to protect his function
- \( P_1 \) gives to \( P_2 \) the “encrypted” function
- \( P_2 \) computes the function on any input

\[ y = E(f)(x) \]
Obfuscation – state of the art

• What kind of obfuscation?
  – the attacker cannot learn more than from black-box access to the function

• General impossibility result
  – Barak et al. 2001

• Few positive results
  – Point functions, Re-encryption, …
Different Scenarios – Cryptocomputing

- $P_2$ gives to $P_1$ the encrypted input
- $P_1$ computes any function of it
- $P_1$ sends back the encrypted output
- $P_1$ decrypts his output

$E(x) = f(E(x))$

$y = D(E(y))$
Homomorphic Encryption

• It’s possible to compute on plaintexts just manipulating ciphertexts

\[ E_{pk}(x) - E_{pk}(y) = E_{pk}(x \odot y) \]
Multiplicative Homomorphic Encryption

\[ E_{pk}(x)E_{pk}(y) = E_{pk}(xy) \]

- RSA

\[ c_1 = x_1^e \mod n \quad c_2 = x_2^e \mod n \]

\[ c_1 c_2 = (x_1^e)(x_2^e) = (x_1 x_2)^e \mod n \]
Multiplicative Homomorphic Encryption

\[ E_{pk}(x)E_{pk}(y) = E_{pk}(xy) \]

- ElGamal

\[ c_1 = (g^{r_1}; x_1 h^{r_1}) \quad c_2 = (g^{r_2}; x_2 h^{r_2}) \]

\[ c_1 c_2 = (g^{r_1 + r_2}; x_1 x_2 h^{r_1 + r_2}) \]
Additive Homomorphic Encryption

\[ E_{pk}(x)E_{pk}(y) = E_{pk}(x + y) \]

\[ E_{pk}(x)^a = E_{pk}(ax) \]

- Modified ElGamal

\[ c_1 = (g^{r_1}; g^{x_1}h^{r_1}) \quad c_2 = (g^{r_2}; g^{x_2}h^{r_2}) \]

\[ c_1c_2 = (g^{r_1+r_2}; g^{x_1+x_2}h^{r_1+r_2}) \]

Inefficient decryption!
Additive Homomorphic Encryption

\[ E_{pk}(x)E_{pk}(y) = E_{pk}(x + y) \]
\[ E_{pk}(x)^a = E_{pk}(ax) \]

- Paillier

\[ c_1 = g^{x_1} r_1^n \mod n^2 \quad c_2 = g^{x_2} r_2^n \mod n^2 \]

\[ c_1 c_2 = g^{x_1 + x_2} (r_1 r_2)^n \mod n^2 \]
Cryptocomputing

• Fully Homomorphic Cryptosystem?

• State of the art
  – Non-interactive Cryptocomputing for NC¹
    Sander, Young 1999
  – the size of the ciphertext **doubles** after every operation
  – just for logarithmic-depth circuits
Interaction is needed?

- To compute any function in a secure way, you need to resort to Secure Multiparty Techniques

**Pros**
- General feasibility
- Strong security guarantees

**Cons**
- Computational overhead
- Communication overhead
- All parties need to cooperate online
Secure Multiparty Computation

- Parties agree on a function to be computed
- They want to protect their inputs

- Auction
- Voting
- …
Secure Multiparty Computation

P_1

P_2

P_3

P_4

output

output

output

output
Secure 2-party Computation

- Yao’s solution (1982):
  - $P_1$ “garbles” the circuit
  - $P_2$ evaluates the garbled circuit
Yao’s garbled circuits (1)

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

Diagram:

```
   A ---- B
   |     |
   |     |
   |     |
   C
```
Yao’s garbled circuits (2)

- $P_1$ selects a random string for every values, for all wires
Yao’s garbled circuits (3)

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>a₀</td>
<td>b₀</td>
<td>( E_{a₀,b₀}(c₀) )</td>
</tr>
<tr>
<td>a₀</td>
<td>b₁</td>
<td>( E_{a₀,b₁}(c₀) )</td>
</tr>
<tr>
<td>a₁</td>
<td>b₀</td>
<td>( E_{a₁,b₀}(c₀) )</td>
</tr>
<tr>
<td>a₁</td>
<td>b₁</td>
<td>( E_{a₁,b₁}(c₁) )</td>
</tr>
</tbody>
</table>

- \( P₁ \) encrypts the output using the inputs as a key
- \( P₁ \) permutes the table randomly
Yao’s garbled circuits (4)

<table>
<thead>
<tr>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>E_{a1,b0}(c_0)</td>
</tr>
<tr>
<td>E_{a1,b1}(c_1)</td>
</tr>
<tr>
<td>E_{a0,b0}(c_0)</td>
</tr>
<tr>
<td>E_{a0,b1}(c_0)</td>
</tr>
</tbody>
</table>

- $P_1$ sends to $P_2$ the garbled table
- $P_1$ sends the string corresponding to his input
  - It appears just as a random string to $P_2$
- $P_2$ needs the string associated to his input
Yao’s garbled circuits (5)

• $P_2$ needs the string associated to his input

• $P_2$ doesn’t want to reveal his input to $P_1$

• $P_1$ doesn’t want to reveal both strings to $P_2$
  – Computing $g(0,B)$ and $g(1,B)$ $P_2$ will learn $B$

• Solution? Oblivious Transfer
1 out of 2 Oblivious Transfer

- **Sender** doesn’t know which secret is chosen
- **Receiver** doesn’t learn the other secret
A simple OT protocol

Receiver

\[ c_0 = E(1 \mid b) \]
\[ c_1 = E(b) \]
\[ (1,0) \circ (0,1) \]

Sender

\[ d = c_0 \circ c_1 \]

\[ x_b = D(d) \]

\[ d = c_0^x c_1^x = E(1 \mid b)^x E(b)^x = E((1 \mid b)x_0 + bx_1) = E(x_b) \]
Yao’s garbled circuits – Final protocol

- \( P_1 \) inputs: \((A,C) = (0,1)\)
- \( P_2 \) inputs: \((B,D) = (1,1)\)
Yao’s garbled circuits – Setup

- $P_1$ prepares the garbled circuit
  - Assign a pair of secret strings to each wire
  - Encrypt the output of each gate with secret strings

- $P_1$ sends the garbled circuit to $P_2$
Yao’s garbled circuits – Inputs exchange

- $P_1$ sends to $P_2$ the strings corresponding to his inputs,
Yao’s garbled circuits – Inputs exchange

- $P_1$ sends to $P_2$ the strings corresponding to his inputs,

- $P_1-P_2$ run Oblivious Transfer
  - $P_2$ obtains secret strings corresponding to his inputs
Yao’s garbled circuits – Evaluating

- $P_2$ uses the secret strings to decrypt the output of the first layer
Yao’s garbled circuits – Evaluating

- $P_2$ uses the secret strings to decrypt the output of the first layer

- $P_2$ uses these strings to decrypt the second layer
Yao’s garbled circuits – Decoding

- \( P_1 \) sends to \( P_2 \)
  - \( <H(g0),0> \)
  - \( <H(g1),1> \)
  (\( H \) some hash function)

- \( P_2 \) evaluates \( f \) on the obtained string and learns the actual output

- \( P_2 \) communicates to \( P_1 \) the output
Yao’s garbled circuits

- $P_1$ generates the garbled circuit
  - Assign random strings for each wire
  - Encrypt
  - Permute

- $P_2$ obtains random strings for his inputs with OT
  - Oblivious Transfer

- $P_2$ evaluate the circuit
  - Decoding layer by layer

- $P_2$ recover the outputs and sends it to $P_1$
  - Decoding table
Arithmetic circuits

- *Ben-Or, Goldwasser and Wigderson, 1988*
- *Chaum, Crépeau and Damgård, 1988*

**Idea**

- $P_i$ has input $x_i$

- $P_i$ “shares” $x_i$ between all parties $\Rightarrow [x_i]$

- All parties jointly evaluate the circuit
  
  $[y] = F([x_1], [x_2], \ldots, [x_n])$

- They reconstruct $[y] \Rightarrow y$
Secret sharing

- To share \( x \in \{0, 1, \ldots, p-1\} \)
  - Select a random t-degree polynomial \( g() \) such that \( f(0)=x \)
  - Sends \( f(i) \) to \( P_i \)
  - \( [x] = (f(1), f(2), \ldots, f(n)) \)

- Lagrange interpolation polynomial
  - t points: allow you to reconstruct the polynomial
  - t-1 points: don’t give you any information about the polynomial
  - (There are \( p \) polynomials that passes for t-1 points)
Computing on secret sharing

• Addition (offline)
  – Compute \([x+y]\) from \([x]\) and \([y]\)
  – \(f()\) such that \(f(0) = x\)
  – \(g()\) such that \(g(0) = y\)
  – \((f+g)()\) such that \((f+g)(0) = x+y\)

• Every party just add his shares
  \(\Rightarrow [x+y]=[x]+[y]\)
Computing on secret sharing

- Multiplication (online)
  - Compute $[xy]$ from $[x]$ and $[y]$  
  - $f()$ such that $f(0) = x$
  - $g()$ such that $g(0) = y$
  - $(fg)()$ such that $(fg)(0) = xy$
  - BUT: $(fg)$ has degree $2t$

- Interaction
  - is needed to compute $h$ such that $h(0) = xy$ and $h$ has degree $t$
Arithmetic circuit

• From addition and multiplication you can compute any circuit
  – NOT: 1-a
  – AND: ab
  – OR: a + b – ab
  – XOR: 1-(a-b)^2
Practical feasibility of general SMC

• Fairplay
  – implements the Yao’s technique
  – Malkhi et al. 2004

• SIMAP
  – implements secret sharing based SMC with applications to food market
  – national Danish Research Agency program
Fairplay

```plaintext
program Millionaires {
  type int = Int<4>;  // 4-bit integer
  type AliceInput = int;
  type BobInput = int;
  type AliceOutput = Boolean;
  type BobOutput = Boolean;
  type Output = struct {
    AliceOutput alice, BobOutput bob};
  type Input = struct {
    AliceInput alice, BobInput bob};

  function Output out(Input inp) {
    out.alice = inp.alice > inp.bob;
    out.bob = inp.bob > inp.alice;
  }
}
```
Fairplay

- **Execution time:**
  - Bit-wise AND between 8 bit register: 2.14s
  - Comparison between 32 bit integers: 4.03s
  - Median of two sorted 10-elements arrays of 16 bits integers: 40.55s
SIMAP

• Secret sharing *efficient* primitives (not just addition and multiplication)
  – Damgård et al. 2005 – now
  – Comparison, equality, exponentiation, bit-decomposition etc.

• Language, compiler:
  – Nielsen and Schwartzbach 2007
SIMAP

C1: declare client Millionaires:
C2:
C3:    tunnel of sint netWorth;
C4:
C5:    function void main(int[] args) {
C6:        ask();
C7:    }
C8:
C9:    function void ask() {
C10:       netWorth.put(readInt());
C11:    }
C12:
C13:    function void tell(bool b) {
C14:        if (b) {
C15:            display("You are the richest!");
C16:        }
C17:        else {
C18:            display("Make more money!");
C19:        }
C20:    }
**SIMAP**

```plaintext
S1: declare server Max:
S2: group of Millionaires mills;
S3: 
S4: function void main(int[] args) {
S5:  
S6:   `sint` max = 0;
S7:   sclient rich;
S8:  
S9:   for (client c in mills) {
S10:     `sint` netWorth = c.netWorth.get();
S11:     if (netWorth > max) {
S12:       max = netWorth;
S13:       rich = c;
S14:     }
S15:   }
S16: 
S17:   for (client c in mills) {
S18:     c.tell(open(c==rich|rich));
S19:   }
S20: }
```
### Timing, comparison

<table>
<thead>
<tr>
<th></th>
<th>SIMAP</th>
<th>Fairplay</th>
</tr>
</thead>
<tbody>
<tr>
<td>PIR</td>
<td>(3,1)</td>
<td>(5,2)</td>
</tr>
<tr>
<td></td>
<td>113</td>
<td>327</td>
</tr>
<tr>
<td></td>
<td>0.126</td>
<td>0.197</td>
</tr>
<tr>
<td>&lt;</td>
<td>3.8</td>
<td>8.9</td>
</tr>
</tbody>
</table>
SIMAP – application

• December 2007
  – for the first time SMC techniques will be used in a real-world application

• Secure auction
  – find the price at which to trade a certain item while keeping the individual bids private

• Danish sugarbeet market
  – producers will use the system to find a fair market price at which to trade contracts for production of beets.
Thank you!
Questions?