Introduction to Secure Multi-Party Computation

Computer Systems Security
ENEE 457/CMSC 498E

Based on notes from:
Vitaly Shmatikov
Motivation

◆ General framework for describing computation between parties who do not trust each other

◆ Example: elections
  • N parties, each one has a “Yes” or “No” vote
  • Goal: determine whether the majority voted “Yes”, but no voter should learn how other people voted

◆ Example: auctions
  • Each bidder makes an offer
    – Offer should be committing! (can’t change it later)
  • Goal: determine whose offer won without revealing losing offers
More Examples

◆ Example: distributed data mining
  • Two companies want to compare their datasets without revealing them
    – For example, compute the intersection of two lists of names

◆ Example: database privacy
  • Evaluate a query on the database without revealing the query to the database owner
  • Evaluate a statistical query on the database without revealing the values of individual entries
  • Many variations
A Couple of Observations

◆ In all cases, we are dealing with **distributed multi-party protocols**
  - A protocol describes how parties are supposed to exchange messages on the network
◆ All of these tasks can be easily computed by a trusted third party
  - The goal of secure multi-party computation is to achieve the same result without involving a trusted third party
How to Define Security?

◆ Must be mathematically rigorous
◆ Must capture all realistic attacks that a malicious participant may try to stage
◆ Should be “abstract”
  • Based on the desired “functionality” of the protocol, not a specific protocol
  • Goal: define security for an entire class of protocols
Functionality

- K mutually distrustful parties want to jointly carry out some task
- Model this task as a function

\[ f: (\{0,1\}^*)^K \rightarrow (\{0,1\}^*)^K \]

- K inputs (one per party); each input is a bitstring
- Assume that this functionality is computable in probabilistic polynomial time
**Ideal Model**

- Intuitively, we want the protocol to behave “as if” a trusted third party collected the parties’ inputs and computed the desired functionality.
  - **Computation in the ideal model is secure by definition!**

\[ f_1(x_1, x_2) \]

\[ f_2(x_1, x_2) \]
A protocol is secure if it *emulates* an ideal setting where the parties hand their inputs to a "trusted party," who locally computes the desired outputs and hands them back to the parties.

[Goldreich-Micali-Wigderson 1987]
Adversary Models

- Some of protocol participants may be corrupt
  - If all were honest, would not need secure multi-party computation

- Semi-honest (aka passive; honest-but-curious)
  - Follows protocol, but tries to learn more from received messages than he would learn in the ideal model

- Malicious
  - Deviates from the protocol in arbitrary ways, lies about his inputs, may quit at any point

- For now, we will focus on two-party protocols
Correctness and Security

How do we argue that the real protocol “emulates” the ideal protocol?

Correctness

- All honest participants should receive the correct result of evaluating function f
  - Because a trusted third party would compute f correctly

Security

- All corrupt participants should learn no more from the protocol than what they would learn in ideal model
- What does corrupt participant learn in ideal model?
  - His input (obviously) and the result of evaluating f
Simulation

- Corrupt participant’s view of the protocol = record of messages sent and received
  - In the ideal world, view consists simply of his input and the result of evaluating f
- How to argue that real protocol does not leak more useful information than ideal-world view?
- Key idea: simulation
  - If real-world view (i.e., messages received in the real protocol) can be simulated with access only to the ideal-world view, then real-world protocol is secure
  - Simulation must be indistinguishable from real view
SMC Definition (First Attempt)

- Protocol for computing $f(*,*)$ between A and B is secure if there exist efficient simulator algorithms $S_A$ and $S_B$ such that

- Correctness: for all input pairs, prot. output is correct.

- Intuition: outputs received by honest parties are indistinguishable from the correct result of evaluating $f$

- Security: $\text{view}_A(\text{real protocol}) \approx S_A$ (gets to query ideal functionality)
  
  $\text{view}_B(\text{real protocol}) \approx S_B$ (gets to query ideal functionality)

  Intuition: a corrupt party’s view of the protocol can be simulated from its input and output

- This definition does not work! Why?
Randomized Ideal Functionality

- Consider a coin flipping functionality $f() = (-, b)$ where $b$ is a random bit.
  - $f()$ flips a coin and tells $B$ the result; $A$ gets no output.

- The following protocol "implements" $f()$:
  1. $A$ chooses bit $b$ randomly.
  2. $A$ sends $b$ to $B$.

- It is obviously insecure (why?)

- Yet it is correct and simulatable according to our attempted definition (why?)
SMC Definition

◆ Protocol for computing $f(*,*)$ betw. A and B is secure if there exist efficient simulator algorithms $S_A$ and $S_B$ such that:

◆ Correctness: for all input pairs, prot. output is correct.

◆ Security:

$$(\text{view}_A(\text{real prot}), \text{output}_B(\text{real prot})) \approx (S_A, y_B)$$

$$(\text{view}_B(\text{real prot}), \text{output}_A(\text{real prot})) \approx (S_B, y_A)$$

◆ Intuition: if a corrupt party’s view of the protocol is correlated with the honest party’s output, the simulator must be able to capture this correlation

◆ Does this fix the problem with coin-flipping $f$?
Oblivious Transfer (OT)

- Fundamental SMC primitive

- A inputs two bits, B inputs the index of one of A’s bits
- B learns his chosen bit, A learns nothing
  - A does not learn which bit B has chosen; B does not learn the value of the bit that he did not choose
- Generalizes to bitstrings, M instead of 2, etc.

[Rabin 1981]
Yao’s Protocol
Yao’s Protocol

- Compute any function securely
  - ... in the semi-honest model
- First, convert the function into a boolean circuit

<table>
<thead>
<tr>
<th>x</th>
<th>y</th>
<th>z</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>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>
1: Pick Random Keys For Each Wire

- Next, evaluate **one gate** securely
  - Later, generalize to the entire circuit

- Alice picks two **random keys** for each wire
  - One key corresponds to “0”, the other to “1”
  - 6 keys in total for a gate with 2 input wires
Alice encrypts each row of the truth table by encrypting the output-wire key with the corresponding pair of input-wire keys.

Original truth table:

<table>
<thead>
<tr>
<th>x</th>
<th>y</th>
<th>z</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>

Encrypted truth table:

\[
\begin{align*}
E_{k_{0x}}(E_{k_{0y}}(k_{0z})) \\
E_{k_{0x}}(E_{k_{1y}}(k_{0z})) \\
E_{k_{1x}}(E_{k_{0y}}(k_{0z})) \\
E_{k_{1x}}(E_{k_{1y}}(k_{1z}))
\end{align*}
\]
3: Send Garbled Truth Table

Alice randomly permutes ("garbles") encrypted truth table and sends it to Bob.

Alice

\[ \begin{align*}
E_{k_0x}(E_{k_0y}(k_{0z})) \\
E_{k_0x}(E_{k_1y}(k_{0z})) \\
E_{k_1x}(E_{k_0y}(k_{0z})) \\
E_{k_1x}(E_{k_1y}(k_{1z}))
\end{align*} \]

Bob

\[ \begin{align*}
E_{k_1x}(E_{k_0y}(k_{0z})) \\
E_{k_0x}(E_{k_1y}(k_{0z})) \\
E_{k_1x}(E_{k_1y}(k_{1z})) \\
E_{k_0x}(E_{k_0y}(k_{0z}))
\end{align*} \]

Garbled truth table:
4: Send Keys For Alice’s Inputs

Alice sends the key corresponding to her input bit

- Keys are random, so Bob does not learn what this bit is

Garbled truth table:

\[
\begin{align*}
E_{k_0x}(E_{k_0y}(k_{0z})) \\
E_{k_0x}(E_{k_1y}(k_{0z})) \\
E_{k_1x}(E_{k_0y}(k_{1z})) \\
E_{k_1x}(E_{k_1y}(k_{1z})) \\
E_{k_0x}(E_{k_0y}(k_{0z})) \\
\end{align*}
\]
5: Use OT on Keys for Bob’s Input

Alice and Bob run oblivious transfer protocol
  - Alice’s input is the two keys corresponding to Bob’s wire
  - Bob’s input into OT is simply his 1-bit input on that wire

Alice
\[ k_{0z}, k_{1z} \]

Bob
\[ k_{0x}, k_{1x} \]

Garbled truth table:
\[
\begin{align*}
E_{k_{1x}}(E_{k_{0y}}(k_{0z})) \\
E_{k_{0x}}(E_{k_{1y}}(k_{0z})) \\
E_{k_{1x}}(E_{k_{1y}}(k_{1z})) \\
E_{k_{0x}}(E_{k_{0y}}(k_{0z}))
\end{align*}
\]

Knows \( K_{b'x} \) where \( b' \) is Alice’s input bit and \( K_{by} \) where \( b \) is his own input bit

Run oblivious transfer
- Alice’s input: \( k_{0y}, k_{1y} \)
- Bob’s input: his bit \( b \)
- Bob learns \( K_{by} \)

What does Alice learn?
6: Evaluate Garbled Gate

◆ Using the two keys that he learned, Bob decrypts exactly one of the output-wire keys
  • Bob does not learn if this key corresponds to 0 or 1
    – Why is this important?

Garbled truth table:

- Suppose $b'=0$, $b=1$
- This is the only row Bob can decrypt.
- He learns $K_{0z}$

Knows $K_{b'x}$ where $b'$ is Alice’s input bit and $K_{by}$ where $b$ is his own input bit
In this way, Bob evaluates entire garbled circuit

- For each wire in the circuit, Bob learns only one key
- It corresponds to 0 or 1 (Bob does not know which)
  - Therefore, Bob does not learn intermediate values (why?)

Bob tells Alice the key for the final output wire and she tells him if it corresponds to 0 or 1

- Bob does not tell her intermediate wire keys (why?)
Brief Discussion of Yao’s Protocol

- Function must be converted into a circuit
  - For many functions, circuit will be huge
- If $m$ gates in the circuit and $n$ inputs, then need $4m$ encryptions and $n$ oblivious transfers
  - Oblivious transfers for all inputs can be done in parallel
- Yao’s construction gives a constant-round protocol for secure computation of any function in the semi-honest model
  - Number of rounds does not depend on the number of inputs or the size of the circuit!