Advanced Topics in Cryptography

Lecture 10
Secure Multi-Party Computation

Benny Pinkas

Secure Computation in the Multi-Party Setting

The setting

- Parties $P_1,\dots,P_n$
- Inputs $x_1,\dots,x_n$ (bits, but can be easily generalized)
- Outputs $y_1,\dots,y_n$

- The functionality is described as a Boolean circuit.
  - Wlog, uses only XOR (+) and AND gates
  - NOT(x) is computed as a x+1
  - Wires are **ordered** so that if wire k is a function of wires i and j, then i<k and j<k.

The GMW protocol for semi-honest setting

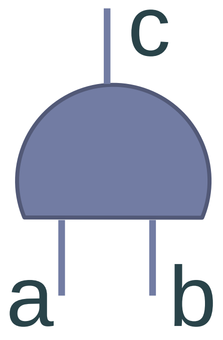
- The protocol:
  - Each party shares its input bit
  - Scan the circuit gate by gate
    - Input values of gate are shared by the parties
    - Run a protocol computing a sharing of the output value of the gate
    - Repeat
  - Publish outputs

Protocol for semi-honest setting

- The protocol:
  - Each party shares its input bit
  - The sharing procedure:
    - P_i has input bit x_i
    - It chooses random bits $r_{i,j}$ for all $i \neq j$.
    - Sends bit $r_{i,j}$ to P_j.
    - Sets its own share to $r_{i,i} = x_i + (\Sigma_{j \neq i}\, r_{i,j}) \bmod 2$
    - Therefore $\Sigma_{j=1 \ldots n}\, r_{i,j} = x_i \bmod 2$.
  - Now every P_j has n shares, one for each input x_i of each P_i.

Evaluating the circuit

- Scan circuit by the order of wires
- Wire c is a function of wires a,b
  - P_i has shares a_i, b_i. Must get share of c_i .



  - Addition gate:
    - P_i computes $c_i=a_i+b_i$ mod 2.
  - Indeed, $c = a+b \pmod 2) = (a_1+\ldots+a_n) + (b_1+\ldots+b_n) = (a_1+b_1)+\ldots+(a_n+b_n) = c_1+\ldots+c_n$

Evaluating multiplication gates

- $c = a \cdot b = (a_1+\ldots+a_n) \cdot (b_1+\ldots+b_n) = \Sigma_{i=1\ldots n}\, a_i b_i + \Sigma_{i \neq j}\, a_i b_j = \Sigma_{i=1\ldots n}\, a_i b_i + \Sigma_{1 \leq i < j \leq n}\, (a_i b_j + a_j b_i)$

- P_i will receive a share of $a_i b_i + \Sigma_{i<j\leq n}\, (a_i b_j + a_j b_i)$

- Computing $a_i b_i$ by P_i is easy
- What about $a_i b_j + a_j b_i$?
- P_i and P_j run the following protocol for every $i<j$.

Evaluating multiplication gates

- Input: P_i has a_i,b_i, P_j has a_j,b_j.
- P_i outputs $a_ib_j+a_jb_i+s_{i,j}$. P_j outputs $s_{i,j}$.
- P_j:
  - Chooses a random $s_{i,j}$
  - Computes the four possible outcomes of $a_ib_j+a_jb_i+s_{i,j}$, depending on the four options for P_i's inputs.
  - Sets these values to be its input to a 1-out-of-4 OT

- P_i is the receiver, with input $2a_i+b_i$.

Recovering the output bits

- The protocol computes shares of the output wires.
- Each party sends its share of an output wire to the party P_i that should learn that output.
- P_i can then sum the shares, obtain the value and output it.

Proof of Security

- Recall definition of security for semi-honest setting:
  - Simulation - Given input and output, can generate the adversary’s view of a protocol execution.

- Suppose that adversary controls the set J of all parties but P_i.
- The simulator is given (x_j,y_j) for all $P_j \in J$.

The simulator

- Shares of input wires: $\forall j \in J$ choose
  - a random share $r_{j,i}$ to be sent from P_j to P_i,
  - and a random share $r_{i,j}$ to be sent from P_i to P_j.
- Shares of multiplication gate wires:
  - $\forall j<i$, choose a random bit as the value learned in the 1-out-of-4 OT.
  - $\forall j>i$, choose a random $s_{i,j}$, and set the four inputs of the OT with P_i accordingly.

- Output wire y_j of $j \in J$: set the message received from P_i as the XOR of y_j and the shares of that wire held by $P_j \in J$.

Security proof

- The output of the simulation is distributed identically to the view in the real protocol
  - Certainly true for the random shares $r_{i,j}$, $r_{j,i}$ sent from and to P_i.
  - OT for $j<i$: output is random, as in the real protocol.
  - OT for $j<i$: input to the OT defined as in the real protocol.
  - Output wires: message from P_i distributed as in the real protocol.

- QED

Performance

- Must run an OT for every multiplication gate
  - Namely, public key operations per multiplication gate
  - Need a communication round between all parties per every multiplication gate

  - Can process together a set of multiplication gates if all their input wires are already shared
  - Therefore number of rounds is O(d), where d is the depth of the circuit (counting only multiplication gates).

The BMR protocol

- Beaver-Micali-Rogaway
- A multi-party version of Yao’s protocol
- Works in O(1) communication rounds, regardless of the depth of the Boolean circuit.
  - D. Beaver, S. Micali and P. Rogaway, “The round complexity of secure protocols”, 1990.
  - A. Ben-David, N. Nisan and B. Pinkas, “FairplayMP – A System for Secure Multi-Party Computation”, 2010.

The BMR protocol

- Two random seeds (garbled values) are set for every wire of the Boolean circuit:
  - Each seed is a concatenation of seeds generated by all players and secretly shared among them.
- The parties securely compute together a 4x1 table for every gate (in parallel):
  - Given 0/1 seeds of the input wires, the table reveals the seed of the resulting value of the output wire.

The BMR protocol

- The parties securely compute together a 4x1 table for every gate (in parallel):
  - This is essentially a secure computation of the table
  - But all tables can be computed in parallel. Therefore O(1) rounds.
  - This is the main bottleneck of the BMR protocol.

- Given the tables, and seeds of the input values, it is easy to compute the circuit output.

The malicious case

- What can go wrong with malicious behavior?
  - Using shares other than those defined by the protocol, using arbitrary inputs to the OT protocol and sending wrong shares of output wires…
- We will show a compiler which forces the parties to operate as in the semi-honest model. (For both GMW and BMR.)
- The basic idea:
  - In every step, each P_i proves in zero knowledge that its messages were computed according to the protocol

Zero knowledge proofs
(we studied this already)

- Prover P, verifier V, language L
- P proves that $x \in L$ without revealing anything
  - Completeness: V always accepts when $x \in L$, and an honest P and V interact.
  - Soundness: V accepts with negligible probability when $x \notin L$, for any P^*.
    - Computational soundness: only holds when P^* is polynomial-time
- Zero-knowledge:
  - There exists a simulator S such that S(x) is indistinguishable from a real proof execution.

A warm-up

- Assume that each party P_i runs a deterministic program Π_i. The compiler is the following:
  - Each P_i commits to its input x_i by sending $C_i(r_i,x_i)$, where r_i is a random string used for the commitment.
  - Let T_i^s be the transcript of P_i at step s, i.e. all messages received and sent by P_i until that step.
  - Define the language $L_i = \{T_i^s$ s.t. $\exists x_i,r_i$ so that all messages sent by P_i until step s are the output of Π_i applied to x_i,r_i and to all messages received by P_i up to that step$\}$
  - When sending a message in step s prove in zero-knowledge that $T_i^s \in L_i$.

Handling randomized protocols

- The previous construction assumes that P_i's program, Π_i, is deterministic.

- This is not true in the semi-honest protocol we have seen.
  - In particular, the choice of shares, and the sender's input to the OT, must be random.
  - The compiler must ensure that P_i chooses its random coins independently of the messages received from other parties.
  - This is not ensured by the previous construction.

The compiler

- We will describe the basic issues of a protocol secure against any number of malicious parties, but with no aborts allowed.
- Communication model:
  - Messages are published on a bulletin board, and can be read by all parties.
  - This implements a broadcast, ensuring that all parties receive the same message.
  - Broadcast can be easily implemented if a public key infrastructure exists.
  - We assume that a PKI does exist.

The compiler

- Input commitment phase:
  - Each party commits to its input.
- Coin generation phase:
  - The parties generate random tapes for each other (this ensures that the randomness is independent of the messages.)
  - Initial idea: random tape of P_i is defined as $s_{1,i} \oplus s_{2,i} \oplus \ldots \oplus s_{n,i}$, where $s_{j,i}$ is chosen by P_j.
  - But this lets P_n control the outcome ☹
- Protocol emulation phase:
  - Run the protocol while proving that the operations of the parties comply with their inputs and random tapes.

The protocol: Input commitment phase

- The required functionality for P_1 is $(x,1^{|x|},\ldots 1^{|x|}) \rightarrow (r,C_r(x),\ldots C_r(x))$, and similarly for each P_i.
  - (This is required in order to choose the randomness, but we must ensure that P_1 actually knows r, and that r was chosen at random.)
- It is not sufficient to ask P_1 to just broadcast a commitment of its input
  - This does not ensure that this is a random commitment for which P_i knows a decommitment.
- The protocol is more complex…
- It is useful to first design tools that can help in constructing the compiler.

Tool 1: image transmission

- The required functionality is $(a,1^{|a|},\ldots 1^{|a|}) \to (\lambda, f(a), \ldots, f(a))$ (<u>all</u> receive the same function of a)
- Protocol
  - P_1 broadcasts an encryption of f(a) (f() is a public function)
  - For j=2…n, P_1 proves to P_j a zero-knowledge proof of knowledge of a value a corresponding to f(a).
  - If P_j rejects, it broadcasts the coins it used in the proof.
- Output: For j=2…n, if P_j sees a justifiable rejection it aborts, otherwise it outputs f(a).
  - Agreement to whether P_1 misbehaved is reduced to the decision on whether some verifier has justifiably rejected the proof.

Tool 1: image transmission

- The required functionality is $(a,1^{|a|},\dots 1^{|a|}) \longrightarrow (\lambda, f(a),\dots,f(a))$
- Agreement as to whether P_1 misbehaved is reduced to the decision on whether some verifier has justifiably rejected the proof.
- If P_1 is honest, then no malicious party can claim that it cheated.

Tool 2: authenticated computation

- The required functionality is

  $(a,b_2,\ldots,b_n)\longrightarrow(\lambda,v_2,\ \ldots,v_n)$,

  where $v_j=f(a)$ if $b_j=h(a)$ and $v_j=\lambda$ otherwise.
- Namely, P_j learns a function f(a) of a, if it already has h(a) (e.g., if h(a) is a commitment to a and P_j already has it)

- Protocol:
  - Use the image transmission tool to broadcast (f(a),h(a)) to all P_j, j=2…n.
  - P_j outputs f(a) if $b_j=h(a)$, and λ otherwise.

Tool 3: multi-party augmented coin-tossing

- The required functionality is
  $(1^n,,\ldots,1^n) \rightarrow (r,g(r),\ldots, g(r))$.

- Typically we will use it for computing
  $(1^n,,\ldots,1^n) \rightarrow ((r,s), C_s(r),\ldots, C_s(r))$, where r is random.

- The challenge: Ensuring that P_1's output is random. We cannot trust P_1 to choose a random output.

Tool 3: multi-party augmented coin-tossing

- $(1^n,,\ldots,1^n)\rightarrow((r,s), C_s(r),\ldots, C_s(r))$.
  - **Toss** and commit: $\forall i$, P_i chooses r_i,s_i and uses the image transmission tool to send $c_i=C_{Si}(r_i)$ to all P_j.
  - **Open** commits: $\forall i\geq 2$, P_i uses the authenticated computation tool to send s_i,r_i to all parties that already have c_i.
  - If P_j obtains r_i agreeing with c_i, it sets $r_i^j=r_i$ (also, $r_j^j=r_j$).

  Otherwise it aborts.
  - If P_1 did not abort, it sets $r=\bigoplus_{i=1\ldots n} r_i$, sends $C_s(r)$ to all other parties (to be used for the main protocol), and proves that $C_s(r)$ was constructed correctly.
    - (details in the next slide)

Tool 3: multi-party augmented coin-tossing (contd.)

- P_1 sends $C_s(r)$ to all other parties, and proves that it was constructed correctly.

- Namely, P_1 runs the authenticated computation functionality
  - P_1 chooses a random s. Its input to the protocol is $(r_1, s_1, s, \oplus_{j=2\ldots n} r_i^1)$
  - P_j's input is c_1, $\oplus_{j=2\ldots n} r_i^j$.
  - If $c_1 = C_{S1}(r_1)$ and $\oplus_{j=2\ldots n} r_i^j = \oplus_{j=2\ldots n} r_i^1$, then P_j outputs $C_s(\oplus_{j=1\ldots n} r_i) = C_s(r)$. Otherwise it aborts.
  - P_1 outputs r.

The main protocol:
Input commitment phase

- Protocol:
  - P_i chooses random r'_i and uses the image transmission functionality to send $c'=C_{r'_i}(x_i)$ to all parties.
  - Run augmented coin-tossing protocol s.t. P_i learns (r_i,r''_i) and others learn $c''=C_{r''_i}(r_i)$.
  - Run authenticated computation where P_i has input (x_i,r_i,r'_i,r''_i) and others input (c',c''), and others learn $C_{r_i}(x_i)$ if (c',c'') are the required functions of P_i's input.

The main protocol: coin generation phase

- Each P_i runs the augmented coin tossing protocol where
  - P_i learns (r^i,s^i)
  - The other parties learn $C_{si}(r^i)$.

The main protocol:
Protocol emulation phase

- The parties use the authenticated computation functionality
  - $(a,b_2,\ldots,b_n)\rightarrow(\lambda,v_2,\ldots,v_n)$, where $v_j=f(a)$ if $b_j=h(a)$ and $v_j=\lambda$ otherwise.
- Suppose that it is P_i's turn to send a message
  - Its input is (x_i,r^i,T_t), as well as the coins used for commitments, where T_t is the sequence of messages exchanged so far.
  - Every other party has input $(C(x_i),C(r^i),T_t)$
  - $f(x_i,r^i,T_t)$ is the message P_i must send
  - It is accepted if $(C(x_i),C(r_i),T)$ agree with x_i,r_i,T and the program that is run

Summary

- Can compute any functionality securely in presence of semi-honest adversaries
- Protocol is efficient enough for use, for circuits that are not too large
- The full proof is in Goldreich’s book.