Session 4: Security against Malicious Adversaries

Yehuda Lindell
Bar-Ilan University
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...
  – In the OT protocol we saw, the receiver can easily and undetectably learn both of the sender’s inputs
    • Just chooses $h_0, h_1$ so that it knows both DLOGs
    • This completely breaks the protocol!
Proving Security

• Recall the definition
  – Simulator interacts with a trusted party
    • Simulator sends corrupted parties’ inputs
    • Simulator receives corrupted parties’ outputs
  – Output distribution of simulator and the honest parties is like in a real execution

• Input extraction
  – In order for the honest parties to output the same in a real and ideal execution, the simulator must extract the input used by the adversary
  – A by-product of the definition is that the parties’ inputs in the protocol are “explicit”
Malicious Adversaries

• We will show a generic compiler which forces the parties to operate as in the semi-honest model
  – It can be applied to any protocol
  – Called the GMW compiler

• The basic idea:
  – In every step, each $P_i$ proves in zero knowledge that its messages were computed according to the protocol specification
Zero knowledge – Reminder

• 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 the verifier’s output after a real proof execution.
Zero-Knowledge for NP

• A fundamental theorem:
  – Any language in NP can be proven in zero knowledge

• \textbf{NP} = the class of all languages that can be verified efficiently
  – There exists a polytime \( V \) such that
    • For every \( x \in L \) there exists a \( w \) such that \( V(x, w) = 1 \)
    • For every \( x \notin L \) and every \( w \) it holds that \( V(x, w) = 0 \)
A Warmup

• Assume that each $P_i$ runs a deterministic program $\Pi_i$. 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 $s$ of the protocol, i.e. all messages received and sent by $P_i$ until that step
A Warmup

• Assume that each $P_i$ runs a deterministic program $\Pi_i$. The compiler is the following:

  – Define the language $L_i = \{T_i^s \text{ s.t. } \exists x_i, r_i \text{ so that all messages sent by } P_i \text{ until step } s \text{ are the output of } \Pi_i \text{ applied to } x_i, r_i \text{ and to all messages received by } P_i \text{ up to that step}\}$

  – When sending a message in step $s$ prove in zero-knowledge that $T_i^s \in L_i$

    • (The overhead is polynomial, but might not be very efficient)
Two Subtle Issues

• The language has to be in NP
  – The input commitment must be perfectly binding
    • Actually not a must, but makes it easier
  – Verifying requires knowing all of the incoming messages to $P_i$
    • This is fine for two-party protocols
    • For multiparty protocols, it means that a type of secure broadcast must be used

• The simulator must extract the inputs
  – $P_i$ must run a ZK proof of knowledge that it knows the committed value
Handling Randomized Protocols

• The previous construction assumes that Pi’s program $\Pi_i$ is \textit{deterministic}
  – But secure protocols \textit{cannot be deterministic}
  – Concretely, in GMW: 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 \textit{independently} of the messages received from other parties
Handling Randomized Protocols

• We need to formalize an NP statement

• If we say “there exists randomness such that…” then:
  – Consider the ElGamal based oblivious transfer
    • The receiver chooses $h_0, h_1$ so that it only knows one of the DLOGs
  – How is it possible to guarantee this?
    • There always exists randomness so that one is chosen at random in the group and one is chosen knowing the DLOG
GMW Compiler Components

• **Input commitment**
  – A secure protocol for computing the functionality
    \(((x, r), \lambda, \ldots, \lambda) \rightarrow (\lambda, \text{Com}(x; r), \ldots, \text{Com}(x; r))\)
  – Note that this already contains input extraction

• **Coin tossing**
  – A secure protocol for “committed” coin tossing
    \((\lambda, \ldots, \lambda) \rightarrow ((b, r), \text{Com}(b; r), \ldots, \text{Com}(b; r))\)
    where \(b \in \{0,1\}\) and \(r \in \{0,1\}^n\) are random
  – Observe: no party can control the coins it receives

• **Protocol emulation**
  – Prove correctness of each message relative to committed
    in put and committed coins in **zero knowledge**
GMW Compiler

• For “simplicity”, we will consider two parties from here on
Input Commitment

• **Functionality** \((x, r, \lambda) \rightarrow (\lambda, \text{Com}(x; r))\)

• **Protocol**
  - \(P_1\) computes \(c = \text{Com}(x; r)\) and sends \(c\) to \(P_2\)
  - \(P_1\) proves a **zero-knowledge proof of knowledge** that it knows \((x, r)\) such that \(c = \text{Com}(x; r)\)

• **Proof of security**
  - \(P_1\) is corrupted: verify proof and extract “witness”; send \((x, r)\) to the trusted party
  - \(P_2\) is corrupted: commit to garbage and run zero knowledge simulator
Coin Tossing

• Functionality \((\lambda, \lambda) \rightarrow ((b, r), \text{Com}(b; r))\)

• Use “truncated” Blum coin tossing:
  – Repeat for \(i = 0, \ldots, n:\)
    • \(P_1\) chooses random \((b_i, r_i)\) and sends \(c_i = \text{Com}(b_i; r_i)\) to \(P_2\)
    • \(P_2\) sends a random \(\beta_i \in \{0,1\}\) to \(P_1\)
  – \(P_1\) sets \(b = b_0 \oplus \beta_0\) and \(r = (b_1 \oplus \beta_1, \ldots, b_n \oplus \beta_n)\) and sends \(c = \text{Com}(b; r)\) to \(P_2\)
  – \(P_1\) proves a zero-knowledge proof of knowledge that this is correct
    • It is an NP statement
Security

• $P_1$ is corrupted
  – Simulator receives $(b, r)$ from trusted party
  – Simulator rewinds in each iteration to make each bit correct
    • Note that the simulator does not get the decommitment of $b_i$ like in Blum
    • However, it can run all the way to the end and run the extractor for the proof
  – Quite complex

• $P_2$ is corrupted
  – Simulator receives $c$ from trusted party
  – Simulator runs first part honestly with adversary
  – Simulator gives $c$ at end and simulates the zero knowledge
Better Coin Tossing

• This is very expensive
  – It actually suffices to toss only one coin per bit
  – This still requires many rounds

• It is possible to toss many coins in a constant number of rounds efficiently
Protocol Emulation

• The input and randomness of each party is fixed
  – This is run by each party (in each direction)

• Parties send each message and prove in zero knowledge that it is correct according to the protocol
  – Reduce security to semi-honest
    – A subtlety: need augmented semi-honest where the corrupted party may replace its input

• The full proof of security is very complex (see Goldreich04)
Demonstration on Yao

• Parties run input commitment phase
• Parties run coin tossing phase
• Parties run oblivious transfer
  – Use zero knowledge to ensure that receiver chooses $h_0, h_1$ correctly
  – Use zero knowledge to ensure that sender provides correct garbled values (relative to randomness)

• $P_1$ constructs garbled circuit
  – Proves in zero knowledge that it is correct relative to randomness

• $P_1$ sends garbled values
  – Use zero knowledge to ensure that sender provides correct garbled values
Complexity

• Amount of randomness needed is huge
  – Can use a PRG but then this must be proven inside ZK as well

• Need to prove a very complex NP statement
  – Entire garbled circuit is constructed correctly
  – Each gate uses PRF computations (e.g., AES)
Summary

• It is possible to convert protocol secure for semi-honest into one secure for malicious
  – This is very surprising!

• Observe that the compiler can all be achieved with one-way functions
  – This is even more surprising: from a complexity perspective getting semi-honest is “harder” than transforming semi-honest to malicious

• Obtaining security against malicious adversaries is hard