Mix-Nets

Lecture 19
Some tools for electronic-voting (and other things)
Mix-Nets
Mix-Nets

Originally proposed by Chaum (1981) for anonymous communication
Mix-Nets

- Originally proposed by Chaum (1981) for anonymous communication
- Input: a vector of ciphertexts under a “threshold encryption scheme”
Mix-Nets

- Originally proposed by Chaum (1981) for anonymous communication
- Input: a vector of ciphertexts under a “threshold encryption scheme”
- Mix-servers take turns to perform “verifiable shuffles”
Mix-Nets

- Originally proposed by Chaum (1981) for anonymous communication
- Input: a vector of ciphertexts under a “threshold encryption scheme”
- Mix-servers take turns to perform “verifiable shuffles”
- Final shuffled vector decrypted by decryption-servers
Mix-Nets

Originally proposed by Chaum (1981) for anonymous communication

Input: a vector of ciphertexts under a "threshold encryption scheme"

Mix-servers take turns to perform "verifiable shuffles"

Final shuffled vector decrypted by decryption-servers

(Omitted: Decryption mix-nets, which combine shuffling and decryption. Here: Re-encryption mix-nets)
Mix-Nets

- Originally proposed by Chaum (1981) for anonymous communication
- Input: a vector of ciphertexts under a “threshold encryption scheme”
- Mix-servers take turns to perform “verifiable shuffles”
- Final shuffled vector decrypted by decryption-servers
  - (Omitted: Decryption mix-nets, which combine shuffling and decryption. Here: Re-encryption mix-nets)
- Ideal functionality: input a vector of private messages from senders, and a permutation from each mix server; output the messages permuted using the composed permutation
Mix-Nets

Originally proposed by Chaum (1981) for anonymous communication

Input: a vector of ciphertexts under a “threshold encryption scheme”

Mix-servers take turns to perform “verifiable shuffles”

Final shuffled vector decrypted by decryption-servers

(Omitted: Decryption mix-nets, which combine shuffling and decryption. Here: Re-encryption mix-nets)

Ideal functionality: input a vector of private messages from senders, and a permutation from each mix server; output the messages permuted using the composed permutation

Corruption model: Active adversary can corrupt a limited number of servers
Threshold Decryption
Threshold Decryption

Key pairs \((SK_i, PK_i)\) generated by a set of servers (separate from sender/receiver). (Receiver may set up parameters.)
Threshold Decryption

- Key pairs \((SK_i,PK_i)\) generated by a set of servers (separate from sender/receiver). (Receiver may set up parameters.)

- Ciphertexts generated by honest player (not CCA security)
Threshold Decryption

- Key pairs \((SK_i, PK_i)\) generated by a set of servers (separate from sender/receiver). (Receiver may set up parameters.)
- Ciphertexts generated by honest player (not CCA security)
- Decryption by public discussion among servers and receiver (all the servers and the receiver see all the messages)
Threshold Decryption

- Key pairs \((SK_i, PK_i)\) generated by a set of servers (separate from sender/receiver). (Receiver may set up parameters.)
- Ciphertexts generated by honest player (not CCA security)
- Decryption by public discussion among servers and receiver (all the servers and the receiver see all the messages)
- Active adversary can corrupt a limited number of servers
Threshold Decryption

- Key pairs \((SK_i, PK_i)\) generated by a set of servers (separate from sender/receiver). (Receiver may set up parameters.)
- Ciphertexts generated by honest player (not CCA security)
- Decryption by public discussion among servers and receiver (all the servers and the receiver see all the messages)
- Active adversary can corrupt a limited number of servers
- Ideal: Same as for SIM-CPA, but with servers also getting the message (if the receiver decides to get it); if number of corrupted servers above threshold, adversary can block (but not substitute) output to others
Threshold Decryption
Threshold Decryption

E.g. Threshold El Gamal for threshold $n$ out of $n$
Threshold Decryption

*E.g. Threshold El Gamal for threshold n out of n*

*KeyGen: (SKᵢ,PKᵢ) = (yᵢ, Yᵢ := g^{yᵢ})* (group, g are system parameters)
Threshold Decryption

- E.g. Threshold El Gamal for threshold n out of n

  **KeyGen:** $(SK_i, PK_i) = (y_i, Y_i := g^{yi})$ (group, g are system parameters)

  **Encryption:** El Gamal, with PK $(g, Y)$ where $Y = \prod_i g^{yi}$
Threshold Decryption

E.g. Threshold El Gamal for threshold n out of n

KeyGen: \((SK_i, PK_i) = (y_i, Y_i := g^{yi})\) (group, g are system parameters)

Encryption: El Gamal, with PK \((g, Y)\) where \(Y = \prod_i g^{yi}\)

Decryption: Given \((A, B) := (g^r, mY^r)\), \(i^{th}\) server outputs \(A_i := (g^r)^{yi}\)
and proves (to the receiver) equality of discrete log for \((g, Y_i)\)
and \((A, A_i)\). Receiver recovers \(m\) as \(B / \prod_i A_i\). 
Threshold Decryption

E.g. Threshold El Gamal for threshold n out of n

\textbf{KeyGen}: \((SK_i,PK_i) = (y_i,Y_i:=g^{yi})\) (group, g are system parameters)

\textbf{Encryption}: El Gamal, with PK \((g,Y)\) where \(Y = \prod_i g^{yi}\)

\textbf{Decryption}: Given \((A,B) := (g^r,mY^r)\), \(i^{th}\) server outputs \(A_i := (g^r)^{yi}\) and proves (to the receiver) equality of discrete log for \((g,Y_i)\) and \((A,A_i)\). Receiver recovers \(m\) as \(B/\prod_i A_i\)

\textbf{Proof using an Honest-Verifier ZK proof}
Threshold Decryption

E.g. Threshold El Gamal for threshold \( n \) out of \( n \)

**KeyGen:** \((SK_i,PK_i) = (y_i,Y_i:=g^{y_i})\) (group, \( g \) are system parameters)

**Encryption:** El Gamal, with PK \((g,Y)\) where \( Y = \prod_i g^{y_i} \)

**Decryption:** Given \((A,B) := (g^r,mY^r)\), \( i \)th server outputs \( A_i := (g^r)^{y_i} \) and proves (to the receiver) equality of discrete log for \((g,Y_i)\) and \((A,A_i)\). Receiver recovers \( m \) as \( B/\prod_i A_i \)

Proof using an Honest-Verifier ZK proof

Using a special purpose proof (Chaum-Pederson), rather than ZK for general NP statements
Honest-Verifier ZK Proofs
Honest-Verifier ZK Proofs

ZK Proof of knowledge of \textit{discrete log} of $A = g^r$
Honest-Verifier ZK Proofs

- ZK Proof of knowledge of **discrete log** of $A=g^r$

- This can be used to prove knowledge of the message in an El Gamal encryption $(A,B) = (g^r, m Y^r)$
Honest-Verifier ZK Proofs

- ZK Proof of knowledge of discrete log of $A = g^r$
  - This can be used to prove knowledge of the message in an El Gamal encryption $(A, B) = (g^r, mY^r)$

- $P \rightarrow V$: $U := g^u$; $V \rightarrow P$: $v$; $P \rightarrow V$: $w := rv + u$
- $V$ checks: $g^w = A^vU$
Honest-Verifier ZK Proofs

- ZK Proof of knowledge of \textit{discrete log} of $A = g^r$
  
  This can be used to prove knowledge of the message in an El Gamal encryption $(A, B) = (g^r, m \cdot Y^r)$

- $P \rightarrow V$: $U := g^u$; $V \rightarrow P$: $v$; $P \rightarrow V$: $w := rv + u$
  
  $V$ checks: $g^w = AyU$

- Proof of Knowledge:
Honest-Verifier ZK Proofs

- ZK Proof of knowledge of discrete log of $A=g^r$
  
  This can be used to prove knowledge of the message in an El Gamal encryption $(A,B)=(g^r, m^r Y^r)$

- $P\rightarrow V$: $U := g^u$ ; $V\rightarrow P$: $v$ ; $P\rightarrow V$: $w := rv + u$ ;
  
  $V$ checks: $g^w = A^v U$

- Proof of Knowledge:
  
  Firstly, $g^w = A^v U \Rightarrow w = rv + u$, where $U = g^u$
Honest-Verifier ZK Proofs

- ZK Proof of knowledge of discrete log of $A = g^r$

  This can be used to prove knowledge of the message in an El Gamal encryption $(A, B) = (g^r, mY^r)$

  $P \rightarrow V: U := g^u; V \rightarrow P: v; P \rightarrow V: w := rv + u$

  $V$ checks: $g^w = A^vU$

Proof of Knowledge:

- Firstly, $g^w = A^vU \Rightarrow w = rv + u$, where $U = g^u$

- If after sending $U$, $P$ could respond to two different values of $v$: $w_1 = rv_1 + u$ and $w_2 = rv_2 + u$, then can solve for $r$
Honest-Verifier ZK Proofs

- ZK Proof of knowledge of discrete log of $A = g^r$

This can be used to prove knowledge of the message in an El Gamal encryption $(A, B) = (g^r, m Y^r)$

**Proof**:

- **P $\rightarrow$ V**: $U := g^u$; $V \rightarrow P$: $v$; $P \rightarrow V$: $w := rv + u$;
  - $V$ checks: $g^w = A^v U$

**Proof of Knowledge**:

- Firstly, $g^w = A^v U \Rightarrow w = rv + u$, where $U = g^u$
- If after sending $U$, $P$ could respond to two different values of $v$: $w_1 = rv_1 + u$ and $w_2 = rv_2 + u$, then can solve for $r$
- **ZK**: simulation picks $w, v$ first and sets $U = g^w / A^v$
HVZK and Special Soundness
HVZK and Special Soundness

**HVZK**: Simulation for honest (passively corrupt) verifier
HVZK and Special Soundness

- **HVZK**: Simulation for honest (passively corrupt) verifier
  
  e.g. in PoK of discrete log, simulator picks \((v, w)\) first and computes \(U\) (without knowing \(u\)). Relies on verifier to pick \(v\) independent of \(U\).
HVZK and Special Soundness

**HVZK**: Simulation for honest (passively corrupt) verifier

e.g. in PoK of discrete log, simulator picks \((v,w)\) first and computes \(U\) (without knowing \(u\)). Relies on verifier to pick \(v\) independent of \(U\).

**Special soundness**: given \((U,v,w)\) and \((U,v',w')\) s.t. \(v \neq v'\) and both accepted by verifier, can derive a witness (in stand-alone setting)
HVZK and Special Soundness

- **HVZK**: Simulation for honest (passively corrupt) verifier
  
  e.g. in PoK of discrete log, simulator picks \((v,w)\) first and computes \(U\) (without knowing \(u\)). Relies on verifier to pick \(v\) independent of \(U\).

- **Special soundness**: given \((U,v,w)\) and \((U,v',w')\) s.t. \(v \neq v'\) and both accepted by verifier, can derive a witness (in stand-alone setting)
  
  e.g. solve \(r\) from \(w = rv + u\) and \(w' = rv' + u\) (given \(v,w,v',w'\))
HVZK and Special Soundness

**HVZK**: Simulation for honest (passively corrupt) verifier

- e.g. in PoK of discrete log, simulator picks \((v,w)\) first and computes \(U\) (without knowing \(u\)). Relies on verifier to pick \(v\) independent of \(U\).

**Special soundness**: given \((U,v,w)\) and \((U,v',w')\) s.t. \(v \neq v'\) and both accepted by verifier, can derive a witness (in stand-alone setting)

- e.g. solve \(r\) from \(w = rv + u\) and \(w' = rv' + u\) (given \(v,w,v',w')\)

**Implies soundness**: for each \(U\) s.t. prover has significant probability of being able to convince, can extract \(r\) from the prover with comparable probability (using “rewinding”)
HVZK and Special Soundness

**HVZK**: Simulation for honest (passively corrupt) verifier

- e.g. in PoK of discrete log, simulator picks \((v, w)\) first and computes \(U\) (without knowing \(u\)). Relies on verifier to pick \(v\) independent of \(U\).

**Special soundness**: given \((U, v, w)\) and \((U, v', w')\) s.t. \(v \neq v'\) and both accepted by verifier, can derive a witness (in stand-alone setting)

- e.g. solve \(r\) from \(w = rv + u\) and \(w' = rv' + u\) (given \(v, w, v', w'\))

**Implies soundness**: for each \(U\) s.t. prover has significant probability of being able to convince, can extract \(r\) from the prover with comparable probability (using “rewinding”)

Can amplify soundness using parallel repetition: still 3 rounds
Honest-Verifier ZK Proofs
Honest-Verifier ZK Proofs

ZK PoK to prove equality of discrete logs for \(((g,Y),(C,D))\), i.e., \(Y = g^r\) and \(D = C^r\) [Chaum-Pederson]
Honest-Verifier ZK Proofs

- ZK PoK to prove equality of discrete logs for \(((g,Y),(C,D))\), i.e., \(Y = g^r\) and \(D = C^r\) [Chaum-Pedersen]

- Can be used to prove equality of two El Gamal encryptions \((A,B)\) & \((A',B')\) w.r.t public-key \((g,Y)\): set \((C,D) := (A/A',B/B')\)
Honest-Verifier ZK Proofs

ZK PoK to prove equality of discrete logs for ((g,Y),(C,D)), i.e., Y = g^r and D = C^r [Chaum-Pederson]

Can be used to prove equality of two El Gamal encryptions (A,B) & (A',B') w.r.t public-key (g,Y): set (C,D) := (A/A',B/B')

P→V: (U,M) := (g^u,C^u); V→P: v ; P→V: w := rv+u ;

V checks: g^w = Y^vU and C^w = D^vM
Honest-Verifier ZK Proofs

ZK PoK to prove equality of discrete logs for \((g,Y),(C,D)\), i.e., \(Y = g^r\) and \(D = C^r\) [Chaum-Pederson]

Can be used to prove equality of two El Gamal encryptions \((A,B)\) & \((A',B')\) w.r.t public-key \((g,Y)\): set \((C,D) := (A/A',B/B')\)

\[ P \rightarrow V: (U,M) := (g^u,C^u); \quad V \rightarrow P: v; \quad P \rightarrow V: w := rv+u; \]

\(V\) checks: \(g^w = Y^vU\) and \(C^w = D^vM\)

Proof of Knowledge:
Honest-Verifier ZK Proofs

ZK PoK to prove equality of discrete logs for \((g,Y),(C,D)\), i.e., \(Y = g^r\) and \(D = C^r\) [Chaum-Pederson]

Can be used to prove equality of two El Gamal encryptions \((A,B)\) & \((A',B')\) w.r.t public-key \((g,Y)\): set \((C,D) := (A/A',B/B')\)

\[\begin{align*}
P &\rightarrow V: (U,M) := (g^u,C^u); \\
V &\rightarrow P: v; \\
P &\rightarrow V: w := rv+u; \\
V \text{ checks: } g^w &= Y^vU \text{ and } C^w = D^vM
\end{align*}\]

Proof of Knowledge:

\[\begin{align*}
g^w &= Y^vU, \\
C^w &= D^vM \\
\Rightarrow \quad w &= rv+u = r'v+u' \quad \text{where } U = g^u, M = g^{u'}, \text{ and } Y = g^r, D = C^r'
\]
Honest-Verifier ZK Proofs

- ZK PoK to prove equality of discrete logs for \(((g,Y),(C,D))\), i.e., \(Y = g^r\) and \(D = C^r\) [Chaum-Pederson]

- Can be used to prove equality of two El Gamal encryptions \((A,B)\) & \((A',B')\) w.r.t public-key \((g,Y)\): set \((C,D) := (A/A',B/B')\)

- Proof of Knowledge:
  \[ g^w = Y^v U \quad \text{and} \quad C^w = D^v M \]

  - \(g^w = Y^v U, \ C^w = D^v M \Rightarrow w = rv+u = r'v+u'\)
  - where \(U = g^u, M = g^{u'}\) and \(Y = g^r, D = C^{r'}\)
  - If after sending \((U,M)\) \(P\) could respond to two different values of \(v\): \(rv_1 + u = r'v_1 + u'\) and \(rv_2 + u = r'v_2 + u'\), then \(r = r'\)
Honest-Verifier ZK Proofs

ZK PoK to prove equality of discrete logs for \((g,Y),(C,D))\), i.e., \(Y = g^r\) and \(D = C^r\) [Chaum-Pederson]

Can be used to prove equality of two El Gamal encryptions \((A,B)\) & \((A',B')\) w.r.t public-key \((g,Y)\): set \((C,D) := (A/A', B/B')\)

\[ \text{P} \rightarrow \text{V}: (U,M) := (g^u, C^u); \text{V} \rightarrow \text{P}: v; \text{P} \rightarrow \text{V}: w := rv+u; \]

\(V\) checks: \(g^w = Y^v U\) and \(C^w = D^v M\)

Proof of Knowledge:

\(g^w = Y^v U, C^w = D^v M \Rightarrow w = rv+u = r'v+u'\)

where \(U=g^u, M=g^{u'}\) and \(Y=g^r, D=C^{r'}\)

If after sending \((U,M)\) \(P\) could respond to two different values of \(v\): \(rv_1 + u = r'v_1 + u'\) and \(rv_2 + u = r'v_2 + u'\), then \(r = r'\)

ZK: simulation picks \(w, v\) first and sets \(U = g^w/A^v, M = C^w/D^v\)
Fiat-Shamir Heuristic
Fiat–Shamir Heuristic

Limitation: Honest-Verifier ZK does not guarantee ZK when verifier is actively corrupt
Fiat-Shamir Heuristic

Limitation: Honest-Verifier ZK does not guarantee ZK when verifier is actively corrupt

Can be fixed by implementing the verifier using MPC
Fiat–Shamir Heuristic

Limitation: Honest-Verifier ZK does not guarantee ZK when verifier is actively corrupt

Can be fixed by implementing the verifier using MPC

If verifier is a public-coin protocol -- i.e., only picks random elements publicly -- then MPC only to generate random coins
Fiat–Shamir Heuristic

- Limitation: Honest-Verifier ZK does not guarantee ZK when verifier is actively corrupt
  - Can be fixed by implementing the verifier using MPC
    - If verifier is a public-coin protocol -- i.e., only picks random elements publicly -- then MPC only to generate random coins
    - Fiat–Shamir Heuristic: random coins from verifier defined as $R(\text{trans})$, where $R$ is a random oracle and trans is the transcript of the proof so far
Fiat-Shamir Heuristic

Limitation: Honest-Verifier ZK does not guarantee ZK when verifier is actively corrupt

Can be fixed by implementing the verifier using MPC

If verifier is a public-coin protocol -- i.e., only picks random elements publicly -- then MPC only to generate random coins

Fiat-Shamir Heuristic: random coins from verifier defined as $R(\text{trans})$, where $R$ is a random oracle and trans is the transcript of the proof so far

Removes need for interaction!
Verifiable Shuffle
Verifiable Shuffle

(Not so) ideal functionality: takes as input encrypted messages from a sender, and a permutation and randomness from a mixer; outputs rerandomized encryptions of permuted messages to a receiver. (Mixer gets encryptions, then picks its inputs.)
Verifiable Shuffle

(Not so) ideal functionality: takes as input encrypted messages from a sender, and a permutation and randomness from a mixer; outputs rerandomized encryptions of permuted messages to a receiver. (Mixer gets encryptions, then picks its inputs.)

Will settle for stand-alone security, and restrict to active corruption of mixer and passive corruption of sender/receiver.
Verifiable Shuffle

(Not so) ideal functionality: takes as input encrypted messages from a sender, and a permutation and randomness from a mixer; outputs rerandomized encryptions of permuted messages to a receiver. (Mixer gets encryptions, then picks its inputs.)

Will settle for stand-alone security, and restrict to active corruption of mixer and passive corruption of sender/receiver

Security against active corruption will be enforced separately (say using the Fiat-Shamir heuristic for receivers; audits/physical means for senders in voting)
Verifiable Shuffle

(Note so) ideal functionality: takes as input encrypted messages from a sender, and a permutation and randomness from a mixer; outputs rerandomized encryptions of permuted messages to a receiver. (Mixer gets encryptions, then picks its inputs.)

Will settle for stand-alone security, and restrict to active corruption of mixer and passive corruption of sender/receiver

Security against active corruption will be enforced separately (say using the Fiat-Shamir heuristic for receivers; audits/physical means for senders in voting)

We shall consider El Gamal encryption
Verifiable Shuffle

(Not so) ideal functionality: takes as input encrypted messages from a sender, and a permutation and randomness from a mixer; outputs rerandomized encryptions of permuted messages to a receiver. (Mixer gets encryptions, then picks its inputs.)

Will settle for stand-alone security, and restrict to active corruption of mixer and passive corruption of sender/receiver.

Security against active corruption will be enforced separately (say using the Fiat-Shamir heuristic for receivers; audits/physical means for senders in voting).

We shall consider El Gamal encryption.

Mixer will be given encrypted messages and it will perform the permutation and reencryptions.
Verifiable Shuffle for 2 inputs
Verifiable Shuffle for 2 inputs

On input \((C_1, C_2)\), produce \((D_1, D_2)\) by shuffling and rerandomizing.
Verifiable Shuffle for 2 inputs

On input $(C_1, C_2)$, produce $(D_1, D_2)$ by shuffling and rerandomizing HVZK proofs that $[(C_1 \rightarrow D_1) \text{ or } (C_1 \rightarrow D_2)]$ and $[(C_2 \rightarrow D_1) \text{ or } (C_2 \rightarrow D_2)]$
Verifiable Shuffle for 2 inputs

On input \((C_1, C_2)\), produce \((D_1, D_2)\) by shuffling and rerandomizing HVZK proofs that \([(C_1 \rightarrow D_1) \text{ or } (C_1 \rightarrow D_2)]\) and \([(C_2 \rightarrow D_1) \text{ or } (C_2 \rightarrow D_2)]\)

To prove \([\text{stmt}_1 \text{ or } \text{stmt}_2]\), given an HVZK/SS proof system for a single statement (here: equality of El Gamal encryptions)
Verifiable Shuffle for 2 inputs

- On input \((C_1, C_2)\), produce \((D_1, D_2)\) by shuffling and rerandomizing
- HVZK proofs that \( [(C_1 \rightarrow D_1) \text{ or } (C_1 \rightarrow D_2)] \) and \( [(C_2 \rightarrow D_1) \text{ or } (C_2 \rightarrow D_2)] \)

To prove \([ \text{stmt}_1 \text{ or } \text{stmt}_2 ]\), given an HVZK/SS proof system for a single statement (here: equality of El Gamal encryptions)

- Denote the messages in the original system by \((U,v,w)\)
Verifiable Shuffle for 2 inputs

- On input \((C_1, C_2)\), produce \((D_1, D_2)\) by shuffling and rerandomizing.
- HVZK proofs that \([(C_1 \rightarrow D_1) \text{ or } (C_1 \rightarrow D_2)]\) and \([(C_2 \rightarrow D_1) \text{ or } (C_2 \rightarrow D_2)]\).

To prove \([\text{stmt}_1 \text{ or } \text{stmt}_2]\), given an HVZK/SS proof system for a single statement (here: equality of El Gamal encryptions).

Denote the messages in the original system by \((U, v, w)\).

- \(P\): Run simulator to get \((U_{3-i}, v_{3-i}, w_{3-i})\) when \(\text{stmt}_i\) true.
- \(\overrightarrow{P} V\): \((U_1, U_2); V \overrightarrow{P}: v; P \overrightarrow{V}: (v_1, v_2, w_1, w_2)\) where \(v_i = v - v_{3-i}\).

Verifier checks: \(v_1 + v_2 = v\) and verifies \((U_1, v_1, w_1)\) and \((U_2, v_2, w_2)\).
Verifiable Shuffle for 2 inputs

- On input \((C_1, C_2)\), produce \((D_1, D_2)\) by shuffling and rerandomizing HVZK proofs that \[ ((C_1 \rightarrow D_1) \text{ or } (C_1 \rightarrow D_2)) \text{ and } ((C_2 \rightarrow D_1) \text{ or } (C_2 \rightarrow D_2)) \]

- To prove \([\text{stmt}_1 \text{ or } \text{stmt}_2]\), given an HVZK/SS proof system for a single statement (here: equality of El Gamal encryptions)

- Denote the messages in the original system by \((U,v,w)\)

- \(P\): Run simulator to get \((U_{3-i}, v_{3-i}, w_{3-i})\) when \(\text{stmt}_i\) true

- \(P \rightarrow V\): \((U_1, U_2)\); \(V \rightarrow P\): \(v\); \(P \rightarrow V\): \((v_1, v_2, w_1, w_2)\) where \(v_i = v - v_{3-i}\)

- **Verifier checks**: \(v_1 + v_2 = v\) and verifies \((U_1, v_1, w_1)\) and \((U_2, v_2, w_2)\)

- Special soundness: given answers for \(v \neq v'\) either \(v_1 \neq v'_1\) or \(v_2 \neq v'_2\). By special soundness, extract witness for \(\text{stmt}_1\) or \(\text{stmt}_2\).
From 2 inputs to many
From 2 inputs to many

- Using a sorting network
From 2 inputs to many

- Using a sorting network

- A circuit with “comparison gates” such that for inputs in any order the output is sorted
From 2 inputs to many

- Using a **sorting network**
- A circuit with “comparison gates” such that for inputs in any order the output is sorted

(Bitonic sort: from Wikipedia)
From 2 inputs to many

- Using a sorting network

- A circuit with “comparison gates” such that for inputs in any order the output is sorted

- Simple $O(n \log^2 n)$ size networks known

(Bitonic sort: from Wikipedia)
From 2 inputs to many

- Using a sorting network
  - A circuit with “comparison gates” such that for inputs in any order the output is sorted
  - Simple $O(n \log^2 n)$ size networks known
- Fix a sorting network, and use a 2x2 verifiable shuffle at each comparison gate

(Bitonic sort: from Wikipedia)
From 2 inputs to many

- Using a sorting network
  - A circuit with “comparison gates” such that for inputs in any order the output is sorted
  - Simple $O(n \log^2 n)$ size networks known
- Fix a sorting network, and use a 2x2 verifiable shuffle at each comparison gate
  - Permutations at the comparison gates chosen so as to implement the overall permutation
From 2 inputs to many

- Using a sorting network
- A circuit with “comparison gates” such that for inputs in any order the output is sorted
- Simple $O(n \log^2 n)$ size networks known
- Fix a sorting network, and use a 2x2 verifiable shuffle at each comparison gate
- Permutations at the comparison gates chosen so as to implement the overall permutation
- 3 rounds: Parallel composition of HVZK proofs

(Bitonic sort: from Wikipedia)
Alternate Verifiable-Shuffles
Alternate Verifiable-Shuffles

More efficient (w.r.t. communication/computation) protocols known:
Alternate Verifiable-Shuffles

More efficient (w.r.t. communication/computation) protocols known:

- 3 rounds, using “permutation matrices”
Alternate Verifiable-Shuffles

More efficient (w.r.t. communication/computation) protocols known:

- 3 rounds, using "permutation matrices"
- With linear communication
Alternate Verifiable-Shuffles

More efficient (w.r.t. communication/computation) protocols known:

- 3 rounds, using "permutation matrices"
- With linear communication
- 7 rounds, using homomorphic commitments
Alternate Verifiable-Shuffles

More efficient (w.r.t. communication/computation) protocols known:

- 3 rounds, using “permutation matrices”
- With linear communication
- 7 rounds, using homomorphic commitments
- Possible with sub-linear communication for the proof
Homomorphic Commitment
Homomorphic Commitment

A commitment scheme over a group
Homomorphic Commitment

- A commitment scheme over a group

- \( \text{com}(x; r) = c \), where \( x, r, c \) are from their respective groups
Homomorphic Commitment

- A commitment scheme over a group
  - \( \text{com}(x;r) = c \), where \( x, r, c \) are from their respective groups
- Hiding and binding
Homomorphic Commitment

- A commitment scheme over a group
  - $\text{com}(x;r) = c$, where $x$, $r$, $c$ are from their respective groups
- Hiding and binding
- Homomorphism: $\text{com}(x;r) \times \text{com}(x';r') = \text{com}(x+x';r+r')$
Homomorphic Commitment

- A commitment scheme over a group
  - \( \text{com}(x; r) = c \), where \( x, r, c \) are from their respective groups
- Hiding and binding
- Homomorphism: \( \text{com}(x; r) \times \text{com}(x'; r') = \text{com}(x+x'; r+r') \)
  - (Operations in respective groups)
Commitment from CRHF
Commitment from CRHF

Let $H$ be a CRHF s.t. $H_K(x,r)$ is uniformly random for a random $r$, for any $x$ and any $K$. 
Commitment from CRHF

Let $H$ be a CRHF s.t. $H_K(x,r)$ is uniformly random for a random $r$, for any $x$ and any $K$

Commitment: Receiver sends a random key $K$ for $H$, and sender sends $\text{Com}_K(x;r) := H_K(x,r)$
Commitment from CRHF

Let $H$ be a CRHF s.t. $H_K(x,r)$ is uniformly random for a random $r$, for any $x$ and any $K$

Commitment: Receiver sends a random key $K$ for $H$, and sender sends $\text{Com}_K(x;r) := H_K(x,r)$

Perfectly hiding, because $r$ will be chosen at random by the committer
Commitment from CRHF

Let $H$ be a CRHF s.t. $H_K(x,r)$ is uniformly random for a random $r$, for any $x$ and any $K$.

Commitment: Receiver sends a random key $K$ for $H$, and sender sends $\text{Com}_K(x;r) := H_K(x,r)$.

- Perfectly hiding, because $r$ will be chosen at random by the committer.

Reveal: send $(x,r)$.
Commitment from CRHF

Let H be a CRHF s.t. $H_K(x,r)$ is uniformly random for a random r, for any x and any K

Commitment: Receiver sends a random key K for H, and sender sends $\text{Com}_K(x;r) := H_K(x,r)$

- Perfectly hiding, because r will be chosen at random by the committer

Reveal: send (x,r)

- Binding, because of collision resistance when K picked at random
Pedersen Commitment
Pedersen Commitment

Recall CRHF $H_{g,h}(x,r) = g^xh^r$ (collision resistant under Discrete Log assumption)
Pedersen Commitment

Recall CRHF $H_{g,h}(x,r) = g^x h^r$ (collision resistant under Discrete Log assumption)

Binding by collision-resistance: receiver picks $(g,h)$
Pedersen Commitment

Recall CRHF $H_{g,h}(x,r) = g^x h^r$ (collision resistant under Discrete Log assumption)

- **Binding** by collision-resistance: receiver picks $(g,h)$
- **Perfectly Hiding** in a prime order group
Pedersen Commitment

Recall CRHF $H_{g,h}(x,r) = g^x h^r$ (collision resistant under Discrete Log assumption)

- Binding by collision-resistance: receiver picks $(g,h)$
- Perfectly Hiding in a prime order group
  - If group is prime order, then all $h$ are generators
Pedersen Commitment

Recall CRHF $H_{g,h}(x,r) = g^x h^r$ (collision resistant under Discrete Log assumption)

- **Binding** by collision-resistance: receiver picks $(g,h)$
- **Perfectly Hiding** in a prime order group
  - If group is prime order, then all $h$ are generators
  - Then for all $x$, $H_{g,h}(x,r)$ is random if $r$ random
Pedersen Commitment

Recall CRHF $H_{g,h}(x,r) = g^x h^r$ (collision resistant under Discrete Log assumption)

- **Binding** by collision-resistance: receiver picks $(g,h)$
- **Perfectly Hiding** in a prime order group
  - If group is prime order, then all $h$ are generators
  - Then for all $x$, $H_{g,h}(x,r)$ is random if $r$ random

Homomorphism: $\text{Com}_{g,h}(x;r) \times \text{Com}_{g,h}(x';r') = \text{Com}_{g,h}(x+x';r+r')$
Pedersen Commitment

Recall CRHF $H_{g,h}(x,r) = g^x h^r$ (collision resistant under Discrete Log assumption)

- **Binding** by collision-resistance: receiver picks $(g,h)$
- **Perfectly Hiding** in a prime order group
  - If group is prime order, then all $h$ are generators
  - Then for all $x$, $H_{g,h}(x,r)$ is random if $r$ random

Homomorphism: $\text{Com}_{g,h}(x;r) \ast \text{Com}_{g,h}(x';r') = \text{Com}_{g,h}(x+x';r+r')$

HVZK PoK of $(x,r)$: Send $\text{Com}_{g,h}(u_1;u_2)$, and on challenge $v$, send $(xv+u_1)$ and $(rv+u_2)$
Pedersen Commitment

Recall CRHF $H_{g,h}(x,r) = g^x h^r$ (collision resistant under Discrete Log assumption)

- Binding by collision-resistance: receiver picks $(g,h)$
- Perfectly Hiding in a prime order group
  - If group is prime order, then all $h$ are generators
  - Then for all $x$, $H_{g,h}(x,r)$ is random if $r$ random

Homomorphism: $\text{Com}_{g,h}(x;r) \times \text{Com}_{g,h}(x';r') = \text{Com}_{g,h}(x+x';r+r')$

HVZK PoK of $(x,r)$: Send $\text{Com}_{g,h}(u_1;u_2)$, and on challenge $v$, send $(xv+u_1)$ and $(rv+u_2)$

Improved efficiency: $H_{g_1,\ldots,g_n,h}(x_1,\ldots,x_n,r) = g_1^{x_1} \cdots g_n^{x_n} h^r$
Using Homomorphic Commitments
Using Homomorphic Commitments

Sub-problem: given a plaintext vector \((m_1, \ldots, m_n)\), verifiably commit to a permutation of it (using a vector commitment)
Using Homomorphic Commitments

Sub-problem: given a plaintext vector \((m_1,\ldots,m_n)\), verifiably commit to a permutation of it (using a vector commitment).

Idea: \((z_1,\ldots,z_n)\) is a permutation of \((m_1,\ldots,m_n)\) iff the polynomials
\[ f(X) := \prod_i (X-m_i) \] and
\[ h(X) := \prod_i (X-z_i) \] are the same.
Using Homomorphic Commitments

Sub-problem: given a plaintext vector \((m_1, \ldots, m_n)\), verifiably commit to a permutation of it (using a vector commitment)

Idea: \((z_1, \ldots, z_n)\) is a permutation of \((m_1, \ldots, m_n)\) iff the polynomials \(f(X) := \prod_i (X-m_i)\) and \(h(X) := \prod_i (X-z_i)\) are the same

 probabilistically verified by assigning a random value \(x\) to \(X\)
Using Homomorphic Commitments

Sub-problem: given a plaintext vector \((m_1,\ldots,m_n)\), verifiably commit to a permutation of it (using a vector commitment)

Idea: \((z_1,\ldots,z_n)\) is a permutation of \((m_1,\ldots,m_n)\) iff the polynomials 
\[
f(X) := \prod_i (X-m_i) \quad \text{and} \quad h(X) := \prod_i (X-z_i)
\]
are the same

Probabilistically verified by assigning a random value \(x\) to \(X\)

If the field is large (super-polynomial), soundness error is negligible: if not identically 0, \(f(X)-h(X)\) has at most \(n\) roots
Using Homomorphic Commitments

Sub-problem: given a plaintext vector \((m_1,\ldots,m_n)\), verifiably commit to a permutation of it (using a vector commitment)

Idea: \((z_1,\ldots,z_n)\) is a permutation of \((m_1,\ldots,m_n)\) iff the polynomials
\[
f(X) := \Pi_i (X-m_i) \quad \text{and} \quad h(X) := \Pi_i (X-z_i)
\]
are the same

Probabilistically verified by assigning a random value \(x\) to \(X\)

If the field is large (super-polynomial), soundness error is negligible: if not identically 0, \(f(X)-h(X)\) has at most \(n\) roots

Use homomorphic commitments to carry out the polynomial evaluation and check equality (details omitted)
Using Homomorphic Commitments

Sub-problem: given a plaintext vector \((m_1, ..., m_n)\), verifiably commit to a permutation of it (using a vector commitment)
Using Homomorphic Commitments

Sub-problem: given a plaintext vector \((m_1, ..., m_n)\), verifiably commit to a permutation of it (using a vector commitment)

For shuffling ciphertexts:
Using Homomorphic Commitments

Sub-problem: given a plaintext vector \((m_1, ..., m_n)\), verifiably commit to a permutation of it (using a vector commitment)

For shuffling ciphertexts:

Suppose verifier knew the permutation. Then task reduces to proving equality of messages in ciphertext pairs.
Using Homomorphic Commitments

- Sub-problem: given a plaintext vector \((m_1, ..., m_n)\), verifiably commit to a permutation of it (using a vector commitment)

- For shuffling ciphertexts:
  - Suppose verifier knew the permutation. Then task reduces to proving equality of messages in ciphertext pairs
  - Can’t reveal the permutation: instead commit to a permutation of \((1, 2, ..., n)\)
Using Homomorphic Commitments

Sub-problem: given a plaintext vector \((m_1,\ldots,m_n)\), verifiably commit to a permutation of it (using a vector commitment)

For shuffling ciphertexts:

- Suppose verifier knew the permutation. Then task reduces to proving equality of messages in ciphertext pairs
- Can’t reveal the permutation: instead commit to a permutation of \((1,2,\ldots,n)\)
  
  Use the sub-protocol to do this verifiably
Using Homomorphic Commitments

Sub-problem: given a plaintext vector \((m_1, \ldots, m_n)\), verifiably commit to a permutation of it (using a vector commitment)

For shuffling ciphertexts:

- Suppose verifier knew the permutation. Then task reduces to proving equality of messages in ciphertext pairs
- Can't reveal the permutation: instead commit to a permutation of \((1,2,\ldots,n)\)
  - Use the sub-protocol to do this verifiably
  - Use homomorphic properties of the commitments to carry out equality proofs w.r.t committed permutation (omitted)
Today
Today

Mix-Nets
Today

- Mix-Nets
- Verifiable shuffles for El Gamal encryption
Today

- Mix-Nets
- Verifiable shuffles for El Gamal encryption
- Also known for Paillier encryption
Today

- Mix-Nets
- Verifiable shuffles for El Gamal encryption
- Also known for Paillier encryption
- Useful in the “back-end” of voting schemes
Today

- Mix-Nets
- Verifiable shuffles for El Gamal encryption
  - Also known for Paillier encryption
- Useful in the “back-end” of voting schemes
- In principle, general MPC would work
Today

- Mix-Nets

- Verifiable shuffles for El Gamal encryption
  - Also known for Paillier encryption

- Useful in the “back-end” of voting schemes

- In principle, general MPC would work

- Special constructions with better efficiency
Today

- Mix-Nets
- Verifiable shuffles for El Gamal encryption
  - Also known for Paillier encryption
- Useful in the “back-end” of voting schemes
  - In principle, general MPC would work
  - Special constructions with better efficiency
- Next: Voting
Today

- Mix-Nets
  - Verifiable shuffles for El Gamal encryption
  - Also known for Paillier encryption
  - Useful in the “back-end” of voting schemes
  - In principle, general MPC would work
  - Special constructions with better efficiency

Next: Voting

- Several subtleties (especially in the “front-end”)