ARIEL UNIVERSITY

אוניברסיטת אריאל בשומרון

Foundations of Modern Cryptography

Edward A. Hirsch

https://edwardahirsch.github.io/edwardahirsch

Neapolis University Pafos

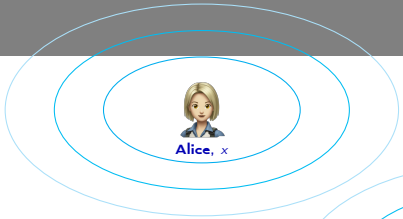
Lecture 10: December 19, 2024

Lecture Plan

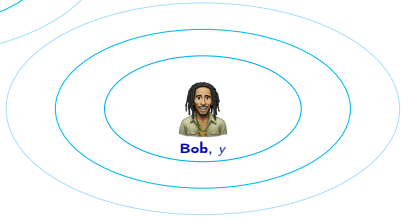
- Blockchains.

If the screen seems frozen and I do not respond,
please call me in Telegram.

Task: Blockchain

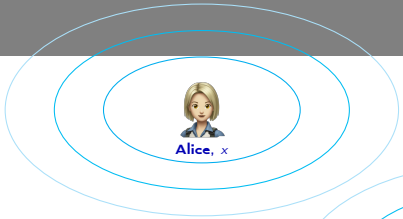


$f_{\mathbf{1}}(x, y, z, \ldots)$

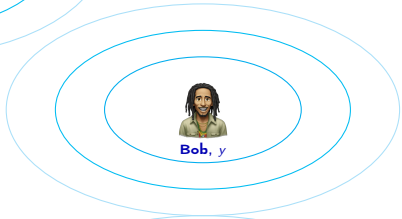


Pictures courtesy AI Emojis Generator

Task: Blockchain



$f_1(x, y, z, \ldots)$



Pictures courtesy AI Emojis Generator

Task: Blockchain

$f_1(x, y, z, \ldots)$

Alice, x

Anon4, w

Bob, y

Anon3, v

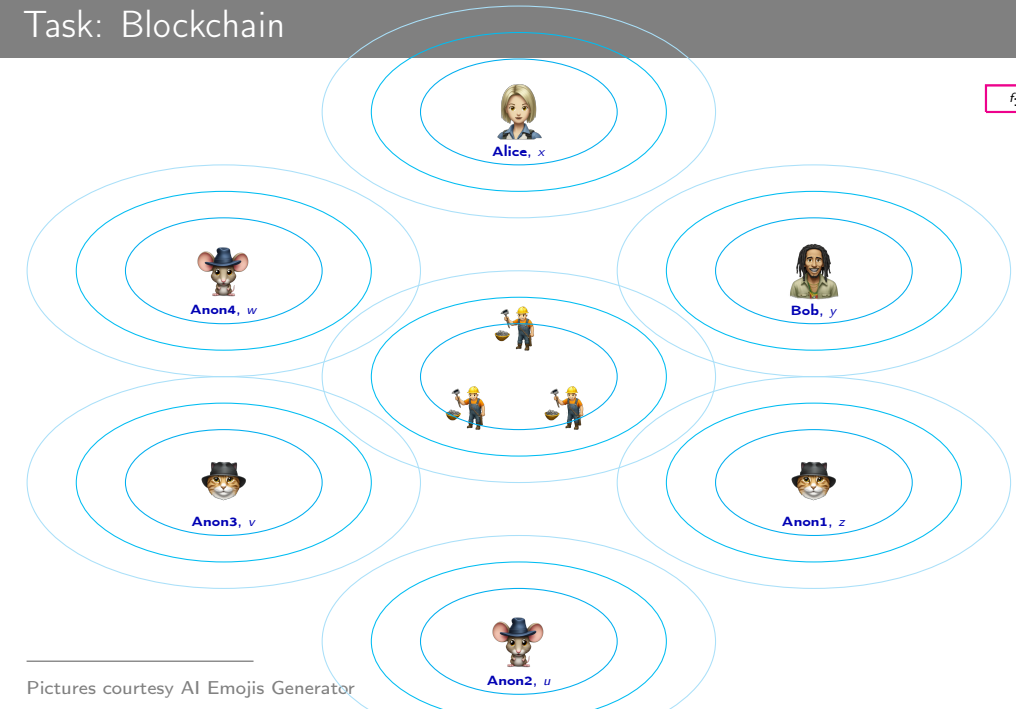
Anon1, z

Anon2, u

Pictures courtesy AI Emojis Generator

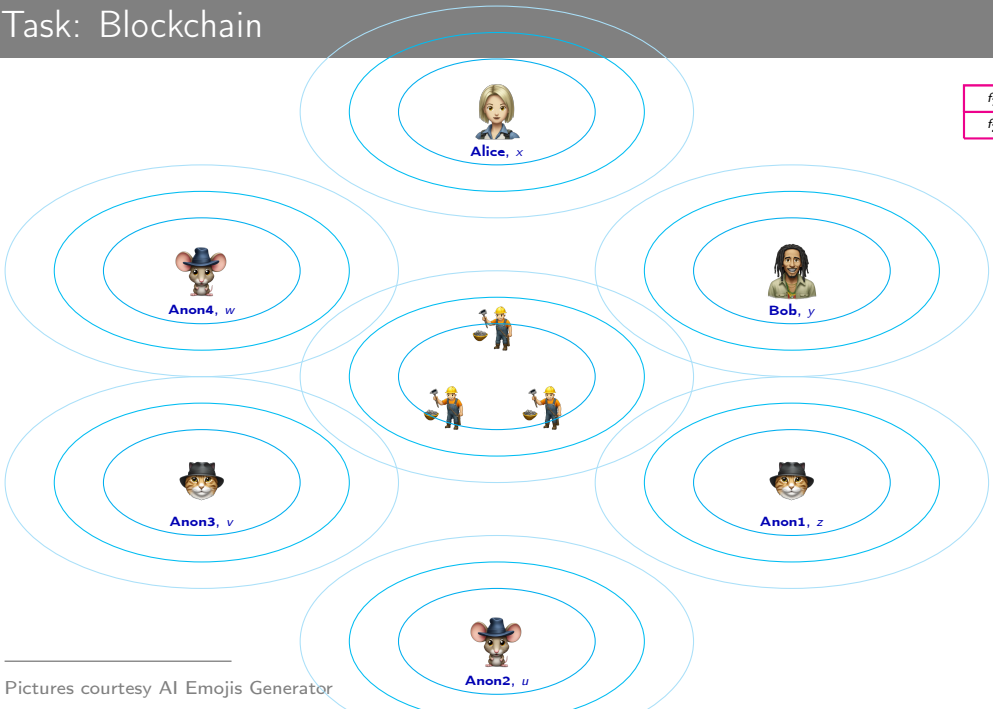
Task: Blockchain

$f_1(x, y, z, \ldots)$



Pictures courtesy AI Emojis Generator

Task: Blockchain

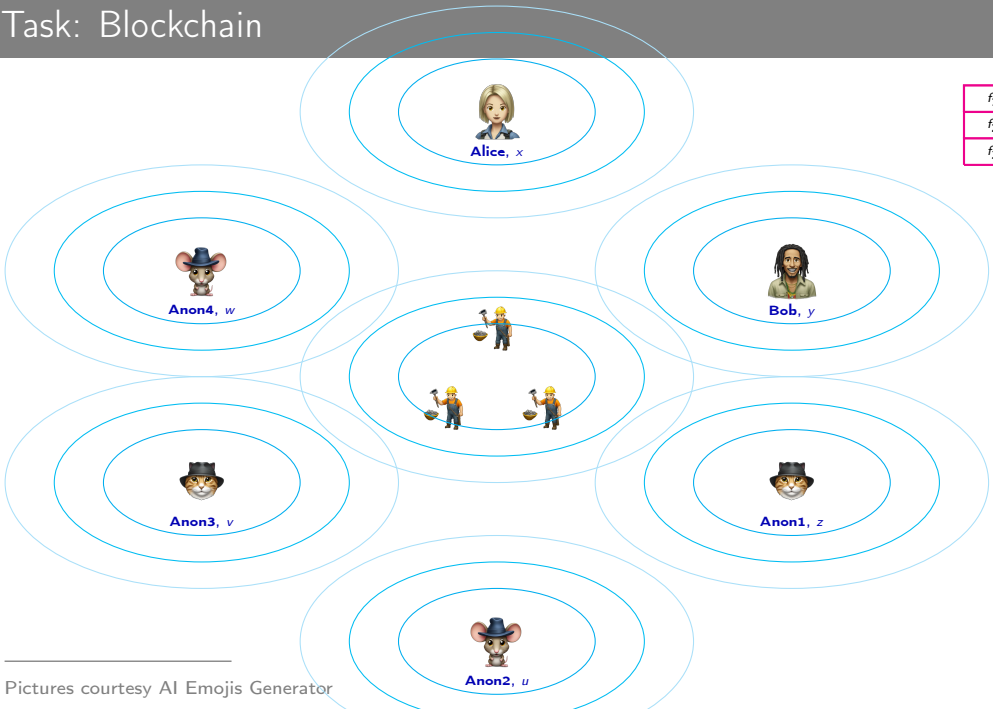


$f_1(x, y, z, \ldots)$

$f_2(x, y, z, \ldots)$

Pictures courtesy AI Emojis Generator

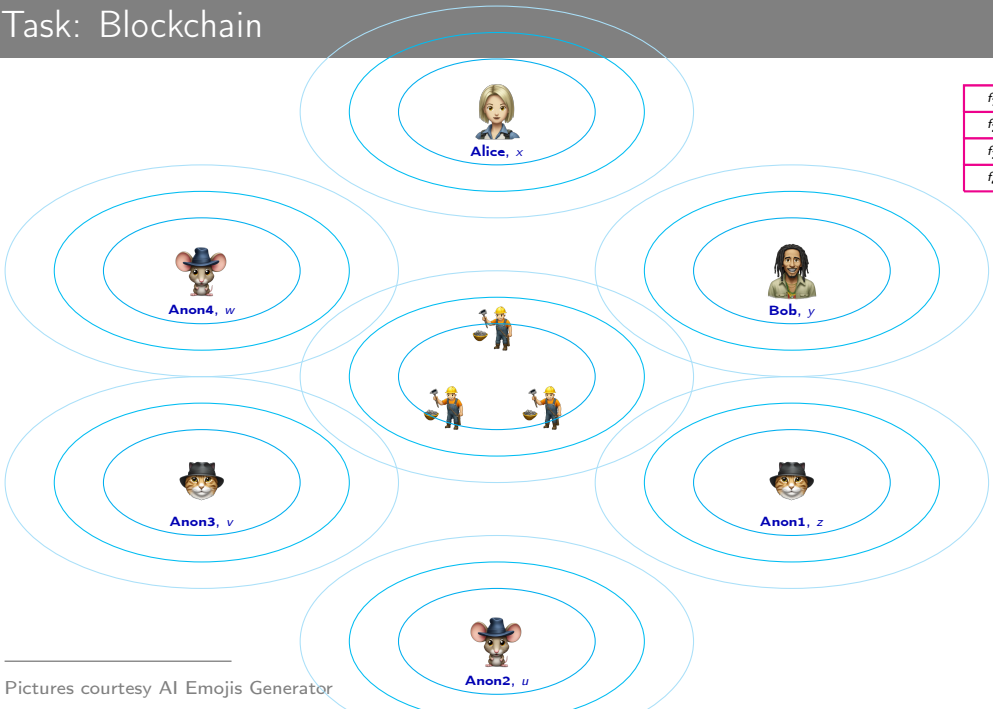
Task: Blockchain



$f_1(x, y, z, \ldots)$
$f_2(x, y, z, \ldots)$
$f_3(x, y, z, \ldots)$

Pictures courtesy AI Emojis Generator

Task: Blockchain



$f_1(x, y, z, \ldots)$

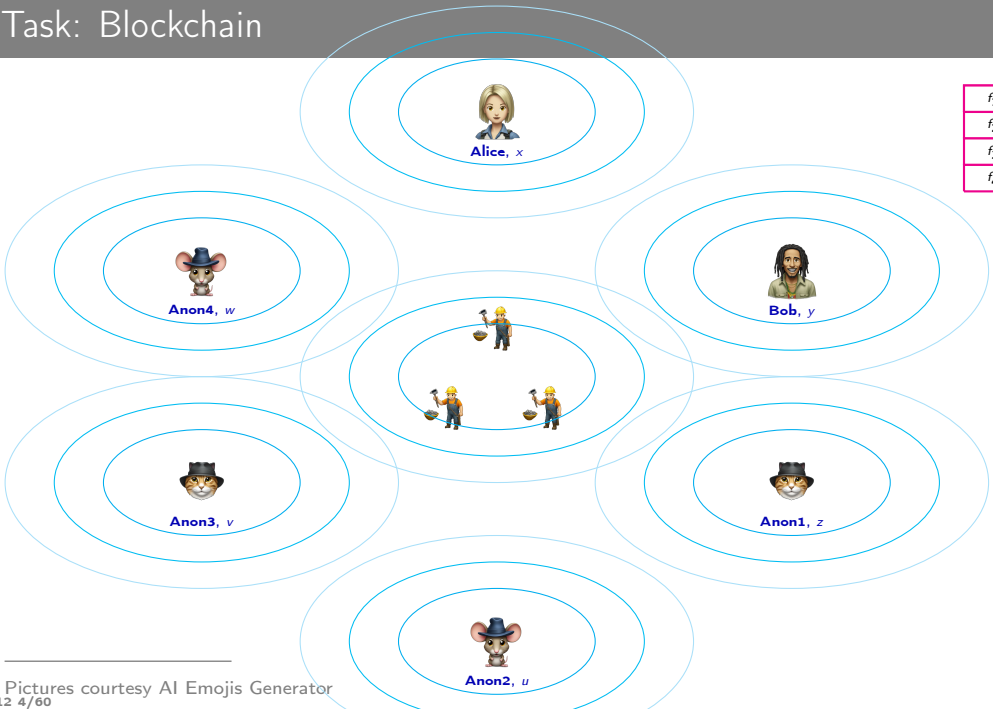
$f_2(x, y, z, \ldots)$

$f_3(x, y, z, \ldots)$

$f_4(x, y, z, \ldots)$

Pictures courtesy AI Emojis Generator

Task: Blockchain



$f_1(x, y, z, \ldots)$

$f_2(x, y, z, \ldots)$

$f_3(x, y, z, \ldots)$

$f_4(x, y, z, \ldots)$

Pictures courtesy AI Emojis Generator

What is a blockchain?

This is a global database and a global computer on top of it

Layers

0. Reliable communication between the participants (internet, peer-to-peer).
1. **Consensus layer.**

What is a blockchain?

This is a global database and a global computer on top of it

Layers

0. Reliable communication between the participants (internet, peer-to-peer).
1. **Consensus layer.**
   - How to keep the things in sync (and yet update the database).
   - How to deal with new non-authenticated dishonest participants.
   - How to compute.

What is a blockchain?

This is a global database and a global computer on top of it

Layers

0. Reliable communication between the participants (internet, peer-to-peer).
1. **Consensus layer.**
   - How to keep the things in sync (and yet update the database).
   - How to deal with new non-authenticated dishonest participants.
   - How to compute.
2. Scaling layer (how to deal with huge numbers of participants and transactions).

What is a blockchain?

This is a global database and a global computer on top of it

Layers

0. Reliable communication between the participants (internet, peer-to-peer).
1. **Consensus layer.**
   - How to keep the things in sync (and yet update the database).
   - How to deal with new non-authenticated dishonest participants.
   - How to compute.
2. Scaling layer (how to deal with huge numbers of participants and transactions).
3. Application layer.

What is a blockchain?

This is a global database and a global computer on top of it

Layers

0. Reliable communication between the participants (internet, peer-to-peer).
1. **Consensus layer.**
   - How to keep the things in sync (and yet update the database).
   - How to deal with new non-authenticated dishonest participants.
   - How to compute.
2. Scaling layer (how to deal with huge numbers of participants and transactions).
3. Application layer.

Consensus layer

- The communication is digitally signed.

What is a blockchain?

This is a global database and a global computer on top of it

Layers

0. Reliable communication between the participants (internet, peer-to-peer).
1. **Consensus layer.**
   - How to keep the things in sync (and yet update the database).
   - How to deal with new non-authenticated dishonest participants.
   - How to compute.
2. Scaling layer (how to deal with huge numbers of participants and transactions).
3. Application layer.

Consensus layer

- The communication is digitally signed.
- Each node keeps an ordered copy of blockchain database (history, ledger):
  - prefix-consistent (u and ux, but no ux and uy with $x \neq y$),

What is a blockchain?

This is a global database and a global computer on top of it

Layers

0. Reliable communication between the participants (internet, peer-to-peer).
1. **Consensus layer.**
   - How to keep the things in sync (and yet update the database).
   - How to deal with new non-authenticated dishonest participants.
   - How to compute.
2. Scaling layer (how to deal with huge numbers of participants and transactions).
3. Application layer.

Consensus layer

- The communication is digitally signed.
- Each node keeps an ordered copy of blockchain database (history, ledger):
  - prefix-consistent (u and ux, but no ux and uy with $x \neq y$),
  - eventually every new block is propagated to every node,

What is a blockchain?

This is a global database and a global computer on top of it

Layers

0. Reliable communication between the participants (internet, peer-to-peer).
1. **Consensus layer.**
   - How to keep the things in sync (and yet update the database).
   - How to deal with new non-authenticated dishonest participants.
   - How to compute.
2. Scaling layer (how to deal with huge numbers of participants and transactions).
3. Application layer.

Consensus layer

- The communication is digitally signed.
- Each node keeps an ordered copy of blockchain database (history, ledger):
  - prefix-consistent (u and ux, but no ux and uy with $x \neq y$),
  - eventually every new block is propagated to every node,
- Each node can:
  - perform local computations,
  - receive messages from other nodes and from clients,
  - send messages to other nodes (e.g., from local computations).

What is a blockchain?

This is a global database and a global computer on top of it

Layers

0. Reliable communication between the participants (internet, peer-to-peer).
1. **Consensus layer.**
   - How to keep the things in sync (and yet update the database).
   - How to deal with new non-authenticated dishonest participants.
   - How to compute.
2. Scaling layer (how to deal with huge numbers of participants and transactions).
3. Application layer.

Consensus layer

- The communication is digitally signed.
- Each node keeps an ordered copy of blockchain database (history, ledger):
  - prefix-consistent (u and ux, but no ux and uy with $x \neq y$),
  - eventually every new block is propagated to every node,
- Each node can:
  - perform local computations,
  - receive messages from other nodes and from clients,
  - send messages to other nodes (e.g., from local computations).
- Nothing comes "out of the blue", there is a reason for every record,
- There are additional checks of validity (who sent a command, was it a permitted sender, etc).

Byzantine agreement

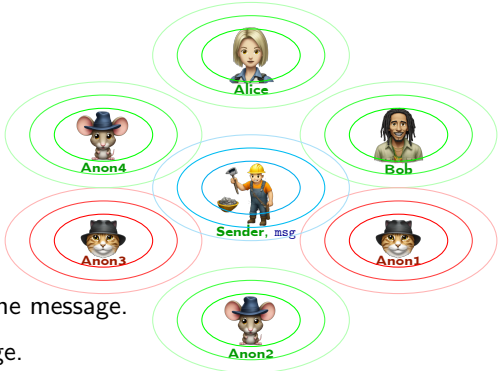
Initial setting

- Fixed set of participants, at most f of them are "Byzantine".
- Each participant has a $(\mathtt{pk}_i, \mathtt{sk}_i)$ key pair for signing, everyone knows $\mathtt{pk}_i$'s.
- Synchronous (time is broadcast by the authorities, each step takes fixed time).

Byzantine agreement

Initial setting

- Fixed set of participants, at most f of them are "Byzantine".
- Each participant has a $(\mathtt{pk}_i, \mathtt{sk}_i)$ key pair for signing, everyone knows $\mathtt{pk}_i$'s.
- Synchronous (time is broadcast by the authorities, each step takes fixed time).



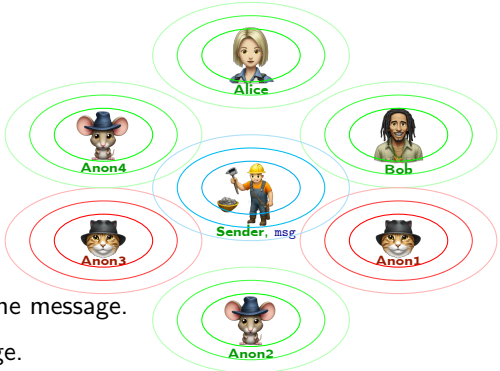
Main problem (Byzantine broadcast)

- There is a designated Sender (honest or not).
- All honest participants end up agreeing on the same message.
- If Sender is honest, this is exactly Sender's message.

Byzantine agreement

Initial setting

- Fixed set of participants, at most f of them are "Byzantine".
  - To be extended to the dynamic setting later.
- Each participant has a $(\mathtt{pk}_i, \mathtt{sk}_i)$ key pair for signing, everyone knows $\mathtt{pk}_i$'s.
  - Fair enough at present, but we could live without it.
- Synchronous (time is broadcast by the authorities, each step takes fixed time).
  - In a real network we need to account for unexpected delays.



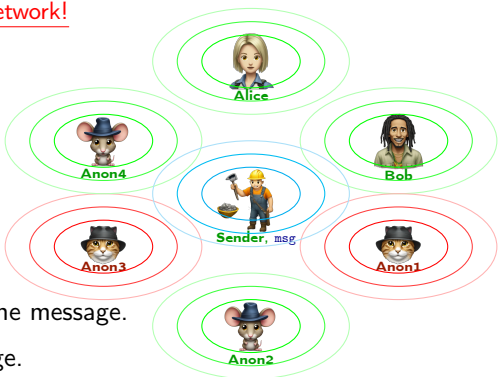
Main problem (Byzantine broadcast)

- There is a designated Sender (honest or not).
- All honest participants end up agreeing on the same message.
- If Sender is honest, this is exactly Sender's message.

Byzantine agreement

Initial setting

- Fixed set of participants, at most f of them are "Byzantine".
  - To be extended to the dynamic setting later.
- Each participant has a $(\mathtt{pk}_i, \mathtt{sk}_i)$ key pair for signing, everyone knows $\mathtt{pk}_i$'s.
  - Fair enough at present, but we could live without it.
- Synchronous (time is broadcast by the authorities, each step takes fixed time).
  - In a real network we need to account for unexpected delays.
  - Broadcasting is implemented as a peer-to-peer network!



Main problem (Byzantine broadcast)

- There is a designated Sender (honest or not).
- All honest participants end up agreeing on the same message.
- If Sender is honest, this is exactly Sender's message.

Byzantine broadcast: Dolev-Strong protocol

The protocol proceeds in "rounds".
The format is (`bit`,

Byzantine broadcast: Dolev-Strong protocol

The protocol proceeds in "rounds".
The format is (`bit`, `nr`, `sign`,

Byzantine broadcast: Dolev-Strong protocol

The protocol proceeds in "rounds".
The format is $(\texttt{bit}, \texttt{nr}, \texttt{sign}, \texttt{nr}', \texttt{sign}',$

Byzantine broadcast: Dolev-Strong protocol

The protocol proceeds in "rounds".
The format is $(\texttt{bit}, \texttt{nr}, \texttt{sign}, \texttt{nr}', \texttt{sign}', \ldots)$, where $\texttt{sign}^{(i)} = S_{\texttt{nr}^{(i)}}(\texttt{bit}, \ldots, \texttt{sign}^{(i-1)})$.
The number of signatures corresponds to the number of rounds passed.

Byzantine broadcast: Dolev-Strong protocol

The protocol proceeds in "rounds".
The format is $(\texttt{bit}, \texttt{nr}, \texttt{sign}, \texttt{nr}', \texttt{sign}', \ldots)$, where $\texttt{sign}^{(i)} = S_{\texttt{nr}^{(i)}}(\texttt{bit}, \ldots, \texttt{sign}^{(i-1)})$.
The number of signatures corresponds to the number of rounds passed.

1. Sender sends his signed message $(\texttt{msg}, 1, S_1(\texttt{msg}))$.

Byzantine broadcast: Dolev-Strong protocol

The protocol proceeds in "rounds".
The format is $(\texttt{bit}, \texttt{nr}, \texttt{sign}, \texttt{nr}', \texttt{sign}', \ldots)$, where $\texttt{sign}^{(i)} = S_{\texttt{nr}^{(i)}}(\texttt{bit}, \ldots, \texttt{sign}^{(i-1)})$.
The number of signatures corresponds to the number of rounds passed.

1. Sender sends his signed message $(\texttt{msg}, 1, S_1(\texttt{msg}))$.
2. $\ldots$ $n-1$. Each participant j:

   For each $b \in \{0, 1\}$, if a valid $s = (b, \ldots)$ has been received in the previous round, then sign it (any of them) and broadcast $s \circ (j, S_j(s))$.

Byzantine broadcast: Dolev-Strong protocol

The protocol proceeds in "rounds".
The format is $(\texttt{bit}, \texttt{nr}, \texttt{sign}, \texttt{nr}', \texttt{sign}', \ldots)$, where $\texttt{sign}^{(i)} = S_{\texttt{nr}^{(i)}}(\texttt{bit}, \ldots, \texttt{sign}^{(i-1)})$.
The number of signatures corresponds to the number of rounds passed.

1. Sender sends his signed message $(\texttt{msg}, 1, S_1(\texttt{msg}))$.
2. $\ldots$ $n-1$. Each participant j:

   For each $b \in \{0, 1\}$, if a valid $s = (b, \ldots)$ has been received in the previous round, then sign it (any of them) and broadcast $s \circ (j, S_j(s))$.

n. Each participant that received only specific bit outputs it (and otherwise $\perp$).

Byzantine broadcast: Dolev-Strong protocol

The protocol proceeds in "rounds".
The format is $(\texttt{bit}, \texttt{nr}, \texttt{sign}, \texttt{nr}', \texttt{sign}', \ldots)$, where $\texttt{sign}^{(i)} = S_{\texttt{nr}^{(i)}}(\texttt{bit}, \ldots, \texttt{sign}^{(i-1)})$.
The number of signatures corresponds to the number of rounds passed.

1. Sender sends his signed message $(\texttt{msg}, 1, S_1(\texttt{msg}))$.
2. $\ldots$ $n-1$. Each participant j:

   For each $b \in \{0, 1\}$, if a valid $s = (b, \ldots)$ has been received in the previous round, then sign it (any of them) and broadcast $s \circ (j, S_j(s))$.
3. n. Each participant that received only specific bit outputs it (and otherwise $\perp$).

Proof:

- If Sender is honest, then all honest participants agree to his bit $\texttt{msg}$.
  (Noone can forge the bit, because Sender's signature is included.)

Byzantine broadcast: Dolev-Strong protocol

The protocol proceeds in "rounds".
The format is $(\texttt{bit}, \texttt{nr}, \texttt{sign}, \texttt{nr}', \texttt{sign}', \ldots)$, where $\texttt{sign}^{(i)} = S_{\texttt{nr}^{(i)}}(\texttt{bit}, \ldots, \texttt{sign}^{(i-1)})$.
The number of signatures corresponds to the number of rounds passed.

1. Sender sends his signed message $(\texttt{msg}, 1, S_1(\texttt{msg}))$.
2. $\ldots$ $n-1$. Each participant j:

   For each $b \in \{0, 1\}$, if a valid $s = (b, \ldots)$ has been received in the previous round, then sign it (any of them) and broadcast $s \circ (j, S_j(s))$.
3. n. Each participant that received only specific bit outputs it (and otherwise $\perp$).

Proof:

- If Sender is honest, then all honest participants agree to his bit msg.
  (Noone can forge the bit, because Sender's signature is included.)
- If Sender is dishonest, dishonest parties can forge their signatures and create a mess.

Byzantine broadcast: Dolev-Strong protocol

The protocol proceeds in "rounds".
The format is $(\texttt{bit}, \texttt{nr}, \texttt{sign}, \texttt{nr}', \texttt{sign}', \ldots)$, where $\texttt{sign}^{(i)} = S_{\texttt{nr}^{(i)}}(\texttt{bit}, \ldots, \texttt{sign}^{(i-1)})$.
The number of signatures corresponds to the number of rounds passed.

1. Sender sends his signed message $(\texttt{msg}, 1, S_1(\texttt{msg}))$.
2. $\ldots$ $n-1$. Each participant j:

   For each $b \in \{0, 1\}$, if a valid $s = (b, \ldots)$ has been received in the previous round, then sign it (any of them) and broadcast $s \circ (j, S_j(s))$.
3. n. Each participant that received only specific bit outputs it (and otherwise $\perp$).

Proof:

- If Sender is honest, then all honest participants agree to his bit msg.
  (Noone can forge the bit, because Sender's signature is included.)
- If Sender is dishonest, dishonest parties can forge their signatures and create a mess.
- Still once one honest player receives a valid $(b_*, \ldots)$ in Rounds $1 \ldots n-2$, then they all will receive a valid $(b_*, \ldots)$ by the end (and agree).

Byzantine broadcast: Dolev-Strong protocol

The protocol proceeds in "rounds".
The format is $(\texttt{bit}, \texttt{nr}, \texttt{sign}, \texttt{nr}', \texttt{sign}', \ldots)$, where $\texttt{sign}^{(i)} = S_{\texttt{nr}^{(i)}}(\texttt{bit}, \ldots, \texttt{sign}^{(i-1)})$.
The number of signatures corresponds to the number of rounds passed.

1. Sender sends his signed message $(\texttt{msg}, 1, S_1(\texttt{msg}))$.
2. $\ldots$ $n-1$. Each participant j:

   For each $b \in \{0, 1\}$, if a valid $s = (b, \ldots)$ has been received in the previous round, then sign it (any of them) and broadcast $s \circ (j, S_j(s))$.
3. n. Each participant that received only specific bit outputs it (and otherwise $\perp$).

Proof:

- If Sender is honest, then all honest participants agree to his bit msg.
  (Noone can forge the bit, because Sender's signature is included.)
- If Sender is dishonest, dishonest parties can forge their signatures and create a mess.
- Still once one honest player receives a valid $(b_*, \ldots)$ in Rounds $1 \ldots n-2$, then they all will receive a valid $(b_*, \ldots)$ by the end (and agree).
  - If j receives it in round i, then k either has already received it or will receive it next round.

Byzantine broadcast: Dolev-Strong protocol

The protocol proceeds in "rounds".
The format is $(\texttt{bit}, \texttt{nr}, \texttt{sign}, \texttt{nr}', \texttt{sign}', \ldots)$, where $\texttt{sign}^{(i)} = S_{\texttt{nr}^{(i)}}(\texttt{bit}, \ldots, \texttt{sign}^{(i-1)})$.
The number of signatures corresponds to the number of rounds passed.

1. Sender sends his signed message $(\texttt{msg}, 1, S_1(\texttt{msg}))$.
2. $\ldots$ $n-1$. Each participant j:

   For each $b \in \{0,1\}$, if a valid $s = (b, \ldots)$ has been received in the previous round, then sign it (any of them) and broadcast $s \circ (j, S_j(s))$.
3. n. Each participant that received only specific bit outputs it (and otherwise $\perp$).

Proof:

- If Sender is honest, then all honest participants agree to his bit msg.
  (Noone can forge the bit, because Sender's signature is included.)
- If Sender is dishonest, dishonest parties can forge their signatures and create a mess.
- Still once one honest player receives a valid $(b_*, \ldots)$ in Rounds $1 \ldots n-2$, then they all will receive a valid $(b_*, \ldots)$ by the end (and agree).
  - If j receives it in round i, then k either has already received it or will receive it next round.
  - If j receives it in round $n-1$, then it is signed by all honest parties.

Byzantine broadcast: Dolev-Strong protocol

The protocol proceeds in "rounds".
The format is $(\texttt{bit}, \texttt{nr}, \texttt{sign}, \texttt{nr}', \texttt{sign}', \ldots)$, where $\texttt{sign}^{(i)} = S_{\texttt{nr}^{(i)}}(\texttt{bit}, \ldots, \texttt{sign}^{(i-1)})$.
The number of signatures corresponds to the number of rounds passed.

1. Sender sends his signed message $(\texttt{msg}, 1, S_1(\texttt{msg}))$.
2. $\ldots$ $n-1$. Each participant j:

   For each $b \in \{0,1\}$, if a valid $s = (b, \ldots)$ has been received in the previous round, then sign it (any of them) and broadcast $s \circ (j, S_j(s))$.
3. n. Each participant that received only specific bit outputs it (and otherwise $\perp$).

Proof:

- If Sender is honest, then all honest participants agree to his bit msg.
  (Noone can forge the bit, because Sender's signature is included.)
- If Sender is dishonest, dishonest parties can forge their signatures and create a mess.
- Still once one honest player receives a valid $(b_*, \ldots)$ in Rounds $1 \ldots n-2$, then they all will receive a valid $(b_*, \ldots)$ by the end (and agree).
  - If j receives it in round i, then k either has already received it or will receive it next round.
  - If j receives it in round $n-1$, then it is signed by all honest parties.

Exercise: check that the more honest parties, the fewer rounds required.

Byzantine agreement (based on broadcast)

. . . and a more realistic setting

Reduction of agreement to broadcast Each of N participants takes the lead in their turn.

Epoch ("Superround") ℓ: ($\ell \mod N$)-th player is Sender.

Sender broadcasts all transactions he is aware of.
Other players add them to their history.

Byzantine agreement (based on broadcast)

. . . and a more realistic setting

Reduction of agreement to broadcast Each of N participants takes the lead in their turn.

Epoch ("Superround") ℓ: ($\ell \mod N$)-th player is Sender.

Sender broadcasts all transactions he is aware of.
Other players add them to their history.

- Dishonest senders do not affect data integrity (consistency).
- Dishonest senders do not affect data sent between the honest players.

Byzantine agreement (based on broadcast)

. . . and a more realistic setting

Reduction of agreement to broadcast Each of N participants takes the lead in their turn.

Epoch ("Superround") ℓ: (ℓ mod N)-th player is Sender.

Sender broadcasts all transactions he is aware of.
Other players add them to their history.

- Dishonest senders do not affect data integrity (consistency).
- Dishonest senders do not affect data sent between the honest players.

Tendermint protocol: (Partially) asynchronous setting with $\leqslant f$ dishonest players

- Timing becomes important: some messages arrive later.

Byzantine agreement (based on broadcast)

. . . and a more realistic setting

Reduction of agreement to broadcast Each of N participants takes the lead in their turn.

Epoch ("Superround") ℓ: (ℓ mod N)-th player is Sender.

Sender broadcasts all transactions he is aware of.
Other players add them to their history.

- Dishonest senders do not affect data integrity (consistency).
- Dishonest senders do not affect data sent between the honest players.

Tendermint protocol: (Partially) asynchronous setting with $\leqslant f$ dishonest players

- Timing becomes important: some messages arrive later.
- Sender extends the latest confirmed epoch (number ℓ) data by transactions received before Δ.

Byzantine agreement (based on broadcast)

. . . and a more realistic setting

Reduction of agreement to broadcast Each of N participants takes the lead in their turn.

Epoch ("Superround") ℓ: (ℓ mod N)-th player is Sender.

Sender broadcasts all transactions he is aware of.
Other players add them to their history.

- Dishonest senders do not affect data integrity (consistency).
- Dishonest senders do not affect data sent between the honest players.

Tendermint protocol: (Partially) asynchronous setting with $\leqslant f$ dishonest players

- Timing becomes important: some messages arrive later.
- Sender extends the latest confirmed epoch (number ℓ) data by transactions received before Δ.
- After deadline Δ, other players vote for the received block if ℓ is also their latest.
- $n - f$ votes (received before 2Δ) are required for the (local) decision.
- Now consistent?

Byzantine agreement (based on broadcast)

. . . and a more realistic setting

Reduction of agreement to broadcast Each of N participants takes the lead in their turn.

Epoch ("Superround") ℓ: (ℓ mod N)-th player is Sender.

Sender broadcasts all transactions he is aware of.
Other players add them to their history.

- Dishonest senders do not affect data integrity (consistency).
- Dishonest senders do not affect data sent between the honest players.

Tendermint protocol: (Partially) asynchronous setting with $\leqslant f$ dishonest players

- Timing becomes important: some messages arrive later.
- Sender extends the latest confirmed epoch (number ℓ) data by transactions received before Δ.
- After deadline Δ, other players vote for the received block if ℓ is also their latest.
- $n - f$ votes (received before 2Δ) are required for the (local) decision.
- Now consistent? No, as some honest players may miss the voting results (quorum).

Byzantine agreement (based on broadcast)

. . . and a more realistic setting

Reduction of agreement to broadcast Each of N participants takes the lead in their turn.

Epoch ("Superround") ℓ: (ℓ mod N)-th player is Sender.

Sender broadcasts all transactions he is aware of.
Other players add them to their history.

- Dishonest senders do not affect data integrity (consistency).
- Dishonest senders do not affect data sent between the honest players.

Tendermint protocol: (Partially) asynchronous setting with $\leqslant f$ dishonest players

- Timing becomes important: some messages arrive later.
- Sender extends the latest confirmed epoch (number ℓ) data by transactions received before Δ.
- After deadline Δ, other players vote for the received block if ℓ is also their latest.
- $n - f$ votes (received before 2Δ) are required for the (local) decision.
- Now consistent? No, as some honest players may miss the voting results (quorum).
- Those received the (positive) results, "lock" the block and vote again, $n - 2f$ votes required before 3Δ.

Byzantine agreement (based on broadcast)

. . . and a more realistic setting

Reduction of agreement to broadcast Each of N participants takes the lead in their turn.

Epoch ("Superround") ℓ: ($\ell \mod N$)-th player is Sender.

Sender broadcasts all transactions he is aware of.
Other players add them to their history.

- Dishonest senders do not affect data integrity (consistency).
- Dishonest senders do not affect data sent between the honest players.

Tendermint protocol: (Partially) asynchronous setting with $\leqslant f$ dishonest players

- Timing becomes important: some messages arrive later.
- Sender extends the latest confirmed epoch (number ℓ) data by transactions received before Δ.
- After deadline Δ, other players vote for the received block if ℓ is also their latest.
- $n - f$ votes (received before 2Δ) are required for the (local) decision.
- Now consistent? No, as some honest players may miss the voting results (quorum).
- Those received the (positive) results, "lock" the block and vote again, $n - 2f$ votes required before 3Δ.
- "Lock" means unapproval of any data inconsistent with the locked block, so we are waiting for its approval or extension.

Byzantine agreement (based on broadcast)

. . . and a more realistic setting

Reduction of agreement to broadcast Each of N participants takes the lead in their turn.

Epoch ("Superround") ℓ: ($\ell \mod N$)-th player is Sender.

Sender broadcasts all transactions he is aware of.
Other players add them to their history.

- Dishonest senders do not affect data integrity (consistency).
- Dishonest senders do not affect data sent between the honest players.

Tendermint protocol: (Partially) asynchronous setting with $\leqslant f$ dishonest players

- Timing becomes important: some messages arrive later.
- Sender extends the latest confirmed epoch (number ℓ) data by transactions received before Δ.
- After deadline Δ, other players vote for the received block if ℓ is also their latest.
- $n - f$ votes (received before 2Δ) are required for the (local) decision.
- Now consistent? No, as some honest players may miss the voting results (quorum).
- Those received the (positive) results, "lock" the block and vote again, $n - 2f$ votes required before 3Δ.
- "Lock" means unapproval of any data inconsistent with the locked block, so we are waiting for its approval or extension. At most f honest players can be locked on an incompatible block.

Byzantine agreement (based on broadcast)

. . . and a more realistic setting

REDUCTION OF AGREEMENT TO BROADCAST Each of N participants takes the lead in their turn.

Epoch ("Superround") ℓ: ($\ell \mod N$)-th player is Sender.

Sender broadcasts all transactions he is aware of.
Other players add them to their history.

- Dishonest senders do not affect data integrity (consistency).
- Dishonest senders do not affect data sent between the honest players.

TENDERMINT PROTOCOL: (Partially) asynchronous setting with $\leqslant f$ dishonest players

- Timing becomes important: some messages arrive later.
- Sender extends the latest confirmed epoch (number ℓ) data by transactions received before Δ.
- After deadline Δ, other players vote for the received block if ℓ is also their latest.
- $n - f$ votes (received before 2Δ) are required for the (local) decision.
- Now consistent? No, as some honest players may miss the voting results (quorum).
- Those received the (positive) results, "lock" the block and vote again, $n - 2f$ votes required before 3Δ.
- "Lock" means unapproval of any data inconsistent with the locked block, so we are waiting for its approval or extension. At most f honest players can be locked on an incompatible block.
- If a quorum for a later epoch eventually arrives, the player becomes unlocked.
  We assume that every message is delivered within "stabilization time" Σ.

Tendermint simulation

Alice

b_1

Anon4

b_1

Bob

Sender

b_1

Anon3

Anon1

Anon2

b_1

Tendermint simulation

Once it is confirmed by $n - 2f$, no incompatible block will ever get a quorum

Tendermint simulation

b_1

b'_2

Alice

Once it is confirmed by $n - 2f$, no incompatible block will ever get a quorum

b_1

b_2

Anon4

b_1

b''_2

Bob

Sender

Anon3

Anon1

Sender

b_1

b'''_2

Anon2

Tendermint simulation

Alice

b_1

b_2'

Once it is confirmed by $n - 2f$, no incompatible block will ever get a quorum

Anon4

b_1

b_2

Bob

b_1

b_2''

Sender

Anon3

Anon1

Sender

Anon2

b_1

b_2'''

Tendermint simulation

b_1
b_2'
b_3

Alice

Sender

Once it is confirmed by $n - 2f$, no incompatible block will ever get a quorum

b_1
b_2
b_3

Anon4

b_1
b_2''
b_3

Bob

Sender

Anon3

Anon1

Sender

b_1
b_2'''
b_3

Anon2

Tendermint simulation

Alice

Sender

b_1

b_3

Once it is confirmed by $n - 2f$, no incompatible block will ever get a quorum

Anon4

b_1

b_3

Bob

Sender

b_1

b_2''

b_3

Only f can stay locked

Anon3

Anon1

Sender

Anon2

b_1

b_3

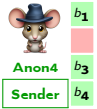
Tendermint simulation

Once it is confirmed by $n - 2f$, no incompatible block will ever get a quorum

Only f can stay locked

Tendermint simulation

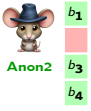
Once it is confirmed by $n - 2f$, no incompatible block will ever get a quorum



Only f can stay locked

Eventually quorum is received

One more voting allows for very late communication
as long as enough honest parties are communicating



Bitcoin and others: protocols allowing new users to come

New non-authenticated users come, billions of them!

- One can pretend to be a billion of user and have a billion of votes.

Bitcoin and others: protocols allowing new users to come

New non-authenticated users come, billions of them!

- One can pretend to be a billion of user and have a billion of votes.
- Idea: making a new user is costly.

Bitcoin and others: protocols allowing new users to come

New non-authenticated users come, billions of them!

- One can pretend to be a billion of user and have a billion of votes.
- Idea: making a new user is costly. **Proof of Work.**
- Include a solved hard problem instance in every processed block of transactions.

Bitcoin and others: protocols allowing new users to come

New non-authenticated users come, billions of them!

- One can pretend to be a billion of user and have a billion of votes.
- Idea: making a new user is costly. **Proof of Work.**
- Include a solved hard problem instance in every processed block of transactions.

A different criterion to keep the data consistent

- The longest chain is considered correct.

Bitcoin and others: protocols allowing new users to come

New non-authenticated users come, billions of them!

- One can pretend to be a billion of user and have a billion of votes.
- Idea: making a new user is costly. **Proof of Work.**
- Include a solved hard problem instance in every processed block of transactions.

A different criterion to keep the data consistent

- The longest chain is considered correct.
- To rely on the data, one needs to wait for more blocks.

Bitcoin and others: protocols allowing new users to come

New non-authenticated users come, billions of them!

- One can pretend to be a billion of user and have a billion of votes.
- Idea: making a new user is costly. **Proof of Work.**
- Include a solved hard problem instance in every processed block of transactions.

A different criterion to keep the data consistent

- The longest chain is considered correct.
- To rely on the data, one needs to wait for more blocks.
- The probability of misuse becomes exponentially smaller:
  Someone altering the chain needs to overperform the others again and again.

Bitcoin and others: protocols allowing new users to come

New non-authenticated users come, billions of them!

- One can pretend to be a billion of user and have a billion of votes.
- Idea: making a new user is costly. **Proof of Work.**
- Include a solved hard problem instance in every processed block of transactions.

A different criterion to keep the data consistent

- The longest chain is considered correct.
- To rely on the data, one needs to wait for more blocks.
- The probability of misuse becomes exponentially smaller:
  Someone altering the chain needs to overperform the others again and again.

Going green: **Proof of Stake**

- Voting power is proportional to the money in posession.
- Then one can use protocol for authenticated users.

Layer 2: How can we optimize a blockchain?

. . . and a reminder of what is going on there

- We don't just put the transactions on the blockchain, we verify them:

  EXAMPLE: Alice wants to pay Bob 5 eth

  - Is this transaction correctly signed by Alice?
  - Does Alice has that much money?
  - In general, is the code executed correctly?
    (*oversimplified!*)

```
if money[Alice] >= 5
then
  money[Alice] -= 5;
  money[Bob] += 5
fi
```

Layer 2: How can we optimize a blockchain?

. . . and a reminder of what is going on there

- We don't just put the transactions on the blockchain, we verify them:

  EXAMPLE: Alice wants to pay Bob 5 eth

  - Is this transaction correctly signed by Alice?
  - Does Alice has that much money?
  - In general, is the code executed correctly?
    (*oversimplified!*)

```
if money[Alice] >= 5
then
  money[Alice] -= 5;
  money[Bob] += 5
fi
```

- Everyone does it, it's costly!
- Let the handling node do it for us and publish a proof!

Layer 2: How can we optimize a blockchain?

. . . and a reminder of what is going on there

- We don't just put the transactions on the blockchain, we verify them:

  EXAMPLE: Alice wants to pay Bob 5 eth

  - Is this transaction correctly signed by Alice?
  - Does Alice has that much money?
  - In general, is the code executed correctly?
    (*oversimplified!*)

```
if money[Alice] >= 5
then
  money[Alice] -= 5;
  money[Bob] += 5
fi
```

- Everyone does it, it's costly!
- Let the handling node do it for us and publish a proof!
- For example, Succinct Non-interactive (Zero Knowledge) Argument of Knowledge (zk-SNARK)

Layer 2: How can we optimize a blockchain?

. . . and a reminder of what is going on there

- We don't just put the transactions on the blockchain, we verify them:

  EXAMPLE: Alice wants to pay Bob 5 eth

  - Is this transaction correctly signed by Alice?
  - Does Alice has that much money?
  - In general, is the code executed correctly?
    (*oversimplified!*)

```
if money[Alice] >= 5
then
  money[Alice] -= 5;
  money[Bob] += 5
fi
```

- Everyone does it, it's costly!
- Let the handling node do it for us and publish a proof!
- For example, Succinct Non-interactive (Zero Knowledge) Argument of Knowledge (zk-SNARK)
- Even better: Let the handler do it for many transactions and submit one such proof for a block.

Layer 2: How can we optimize a blockchain?

. . . and a reminder of what is going on there

- We don't just put the transactions on the blockchain, we verify them:

  EXAMPLE: Alice wants to pay Bob 5 eth

  - Is this transaction correctly signed by Alice?
  - Does Alice has that much money?
  - In general, is the code executed correctly?
    (*oversimplified!*)

```
if money[Alice] >= 5
then
  money[Alice] -= 5;
  money[Bob] += 5
fi
```

- Everyone does it, it's costly!
- Let the handling node do it for us and publish a proof!
- For example, Succinct Non-interactive (Zero Knowledge) Argument of Knowledge (zk-SNARK)
- Even better: Let the handler do it for many transactions and submit one such proof for a block.
- The community now spends much less time on verification.

Takeout

- Blockchains.

Takeout

- Blockchains.

Coming next:

- *(right now) Homework.*
- *(maybe) Exercises and Repeat.*
- *(maybe) Anything left about the blockchains?*
- *(maybe) The real grounds of the security in modern cryptography.*
- *(maybe) A complete (universal) public-key cryptosystem.*

Homework assignment #2

One can use an encryption scheme or a digital signature scheme for authentication:

- User creates (pk, sk) and sends pk to the server (using an authenticated channel).

Homework assignment #2

One can use an encryption scheme or a digital signature scheme for authentication:

- User creates (pk, sk) and sends pk to the server (using an authenticated channel).
- Next time: Server challenges the user (decrypt this? sign this?).
- User solves the challenge using sk and sends the answer.

Homework assignment #2

One can use an encryption scheme or a digital signature scheme for authentication:

- User creates (pk, sk) and sends pk to the server (using an authenticated channel).
- Next time: Server challenges the user (decrypt this? sign this?).
- User solves the challenge using sk and sends the answer.
- Server verifies the answer using pk.

Homework assignment #2

One can use an encryption scheme or a digital signature scheme for authentication:

- User creates (`pk`, `sk`) and sends `pk` to the server (using an authenticated channel).
- Next time: Server challenges the user (decrypt this? sign this?).
- User solves the challenge using `sk` and sends the answer.
- Server verifies the answer using `pk`.

Consider commonly used key types (type `ssh-keygen -h` and look for `-t`):

```
-t dsa | ecdsa | ecdsa-sk | ed25519 | ed25519-sk | rsa
        Specifies the type of key to create.  The possible values are “dsa”, “ecdsa”,
        “ecdsa-sk”, “ed25519”, “ed25519-sk”, or “rsa”.

        This flag may also be used to specify the desired signature type when signing cer-
        tificates using an RSA CA key.  The available RSA signature variants are “ssh-rsa”
        (SHA1 signatures, not recommended), “rsa-sha2-256”, and “rsa-sha2-512” (the default).
```

More specifically, type `ssh -Q sig` to list the protocols.

Also type `ssh -Q` ... (other interesting protocol types: `key` + `cipher`) if we run out of this.

Homework assignment #2

One can use an encryption scheme or a digital signature scheme for authentication:

- User creates (pk, sk) and sends pk to the server (using an authenticated channel).
- Next time: Server challenges the user (decrypt this? sign this?).
- User solves the challenge using sk and sends the answer.
- Server verifies the answer using pk.

Consider commonly used key types (type `ssh-keygen -h` and look for `-t`):

```
-t dsa | ecdsa | ecdsa-sk | ed25519 | ed25519-sk | rsa
        Specifies the type of key to create.  The possible values are “dsa”, “ecdsa”,
        “ecdsa-sk”, “ed25519”, “ed25519-sk”, or “rsa”.

        This flag may also be used to specify the desired signature type when signing cer-
        tificates using an RSA CA key.  The available RSA signature variants are “ssh-rsa”
        (SHA1 signatures, not recommended), “rsa-sha2-256”, and “rsa-sha2-512” (the default).
```

More specifically, type `ssh -Q sig` to list the protocols.

Also type `ssh -Q` ... (other interesting protocol types: `key` + `cipher`) if we run out of this.

Choose one protocol (typically found on nist.gov), identify and explain in our terms

- What is the cryptographic scheme (one of those we studied or maybe another one).
- What specific primitives are used (e.g., $f_{\mathrm{RSA}}(\ldots) = \ldots$ as a owff, not a tpff).
- On what specific assumption(s) the security is based.
- Based on that, how (and in which sense) the security can be proved and/or what are the caveats.

Homework assignment #2

One can use an encryption scheme or a digital signature scheme for authentication:

- User creates (pk, sk) and sends pk to the server (using an authenticated channel).
- Next time: Server challenges the user (decrypt this? sign this?).
- User solves the challenge using sk and sends the answer.
- Server verifies the answer using pk.

Consider commonly used key types (type `ssh-keygen -h` and look for `-t`):

```
-t dsa | ecdsa | ecdsa-sk | ed25519 | ed25519-sk | rsa
        Specifies the type of key to create.  The possible values are “dsa”, “ecdsa”,
        “ecdsa-sk”, “ed25519”, “ed25519-sk”, or “rsa”.

        This flag may also be used to specify the desired signature type when signing cer-
        tificates using an RSA CA key.  The available RSA signature variants are “ssh-rsa”
        (SHA1 signatures, not recommended), “rsa-sha2-256”, and “rsa-sha2-512” (the default).
```

More specifically, type `ssh -Q sig` to list the protocols.

Also type `ssh -Q` ... (other interesting protocol types: `key` + `cipher`) if we run out of this.

Telegram group: *Кто первый встал, того и тапки.*

Choose one protocol (typically found on nist.gov), identify and explain in our terms

- What is the cryptographic scheme (one of those we studied or maybe another one).
- What specific primitives are used (e.g., $f_{\mathrm{RSA}}(\ldots) = \ldots$ as a owff, not a tpff).
- On what specific assumption(s) the security is based.
- Based on that, how (and in which sense) the security can be proved and/or what are the caveats.