CMPT 980 – Information Privacy

Module 3: Privacy-Preserving Cryptography

Repudiability

- PKE + PKI allows authentication
- But having our identities provably attached to our message isn’t always desirable
  - Connecting identity with behavior compromises privacy
- Repudiability/Deniability: Messages sent in this channel cannot be proven by any other party to have originated from the sender
- Can we design a cryptographic protocol to allow authentication, but also allow repudiability?
  - That is to say, Bob believes Alice is Alice, but Bob cannot prove to anyone else that Alice is Alice

Repudiability

- Consider a SKE setup:

Alice $\qquad\qquad\longrightarrow\qquad\qquad$ Bob

K $\qquad$ $Enc_K(M)$, $Hash(Enc_K(M), K)$ $\qquad$ K

- Bob can check the MAC to ensure that whomever sent this must have the secret key
- Bob knows he himself did not write *M*, so Alice did
- But Bob cannot prove Alice wrote *M* to anyone else, since Bob could've written *M*

Forgeability

- A forgeable ciphertext is a ciphertext that *anyone*, not just Alice or Bob, could have written
  - Even the intercepting attacker could have created this message
- This can be achieved with malleable encryption
  - Recall: Ciphertexts encrypted with malleable encryption can be edited to produce predictable changes in the plaintext
- This can also be achieved by revealing the key

Forward secrecy

- We want to limit damage if keys are exposed
- A (long-term) key in a cryptosystem has **forward secrecy** if leaking that key does not expose *past* conversations
- To achieve this, we ensure that:
  - Long-term keys are only used for signing
  - Encryption is done only with short-term (session) keys

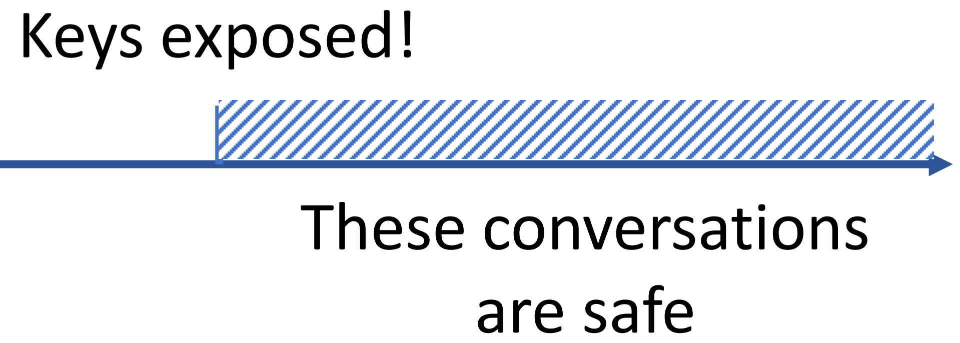
Keys exposed!

These conversations are safe

These are not safe
(Eve could’ve started them)

Break-in Recovery

- A cryptosystem has break-in recovery if future conversations after the point of compromise are safe
  - Also known as future secrecy
  - If we only use short-term keys, we have break-in recovery; but we need long-term keys to bootstrap trust



Double Ratchet Algorithm

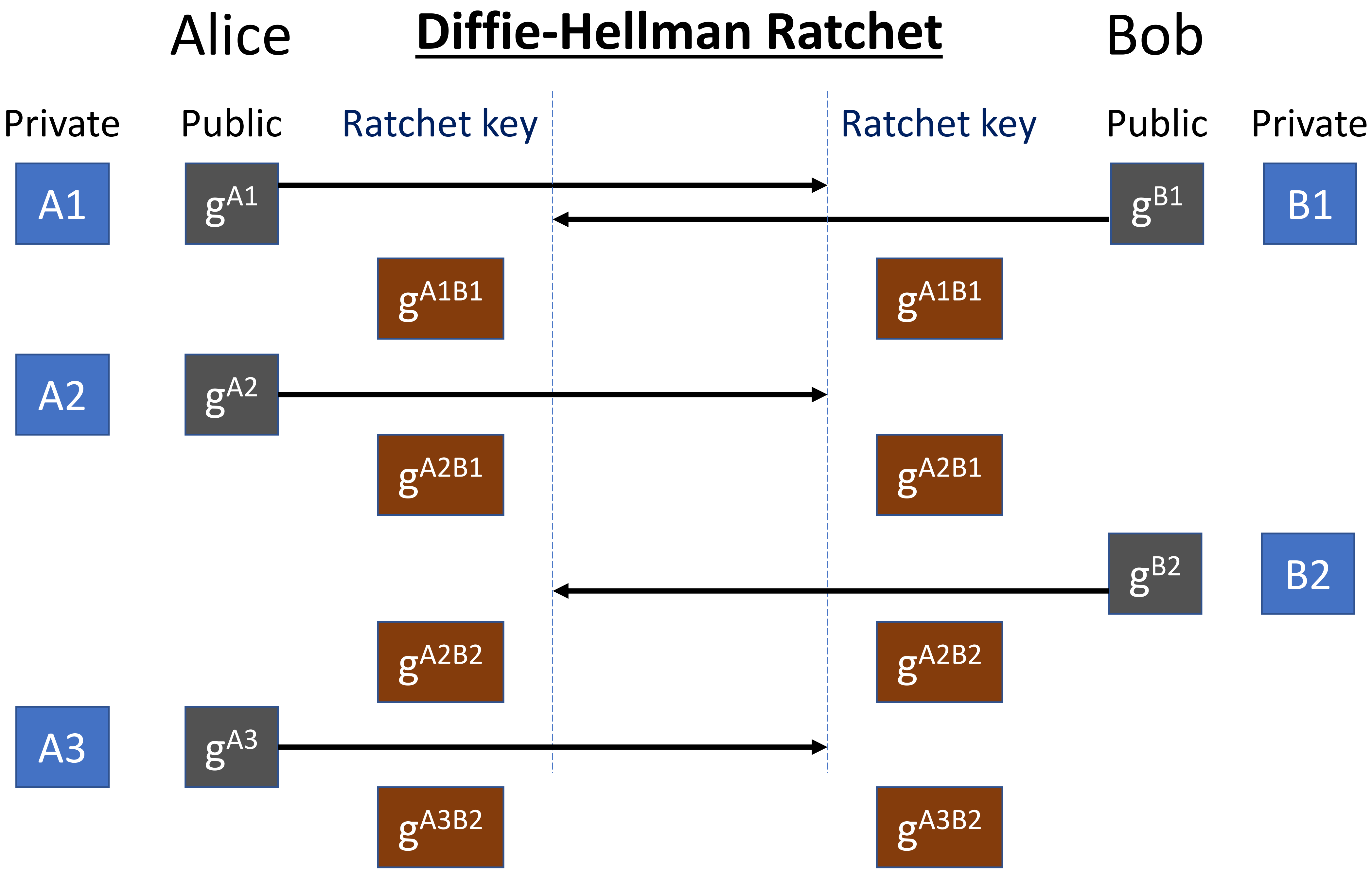
- Used in the Signal Protocol
  - WhatsApp, possibly Facebook Messenger and Skype
- Based on the Off-the-Record Messaging algorithm
- Achieves repudiation, forward secrecy, and break-in recovery
- Based on two sets of ratchets:
  - The **Diffie-Hellman ratchet** generates ratchet keys
  - The **symmetric key ratchet** generates message keys based on ratchet keys
  - A ratchet key can be used to generate several message keys from the same sender

Double Ratchet Algorithm

Diffie-Hellman Ratchet

- Consider DH:
  - Generator g
  - Alice's private key is x, public key is g^x
  - Bob's private key is y, public key is g^y
  - Shared secret becomes g^{xy}
- In the Diffie-Hellman Ratchet, a sequence of shared secrets is generated
- A new shared secret is generated whenever someone who has just received a message wants to send a message
- Ratchet keys will be generated from those shared secrets

Diffie-Hellman Ratchet



Double Ratchet Algorithm

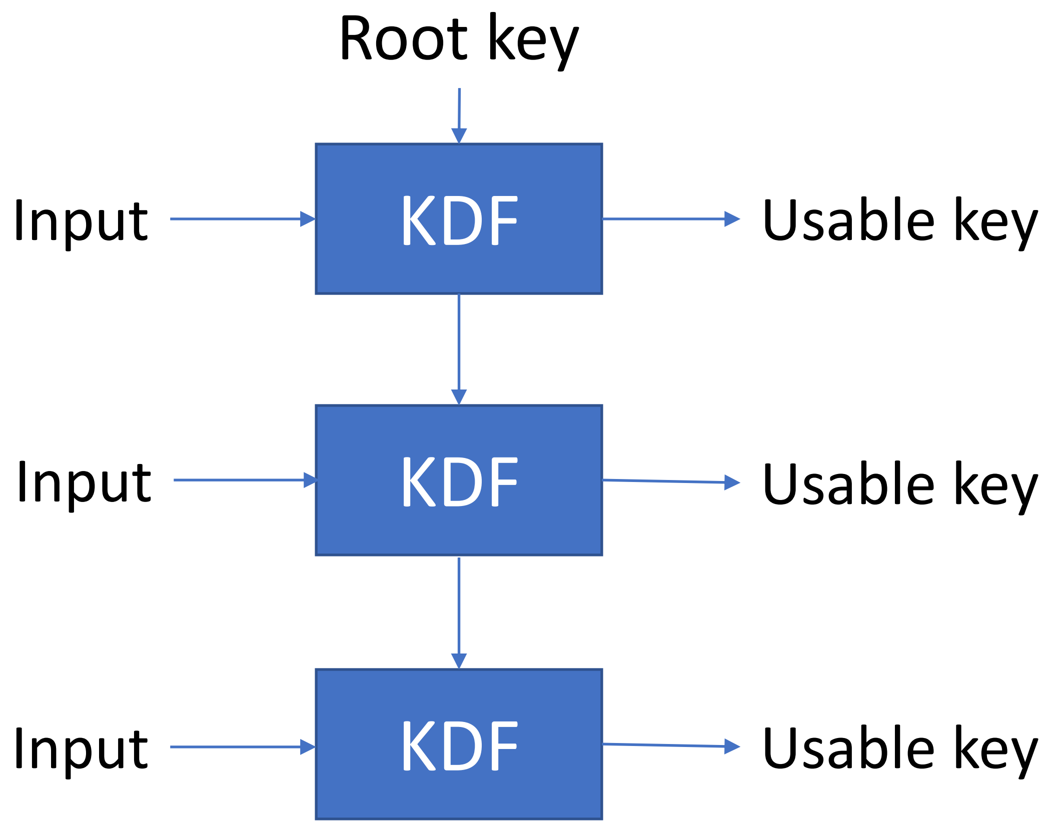
Diffie-Hellman Ratchet

- What happens if a private key is compromised later?
  - Then exactly 2 ratchet keys are compromised
  - If it is B5, then they would be g^{A5B5}, g^{A6B5} (if Alice talks first)
  - No other past or future ratchet keys, or messages depending on those keys, are compromised: **forward secrecy** and **future secrecy**
- The ability to encrypt and create HMACs using the ratchet key would also provide repudiability, as long as we avoid signatures
  - In reality, we refrain from using the ratchet key directly to further reduce the attacker’s attack surface

Double Ratchet Algorithm

Symmetric Key Ratchet

Based on Key Derivation Function Chains:

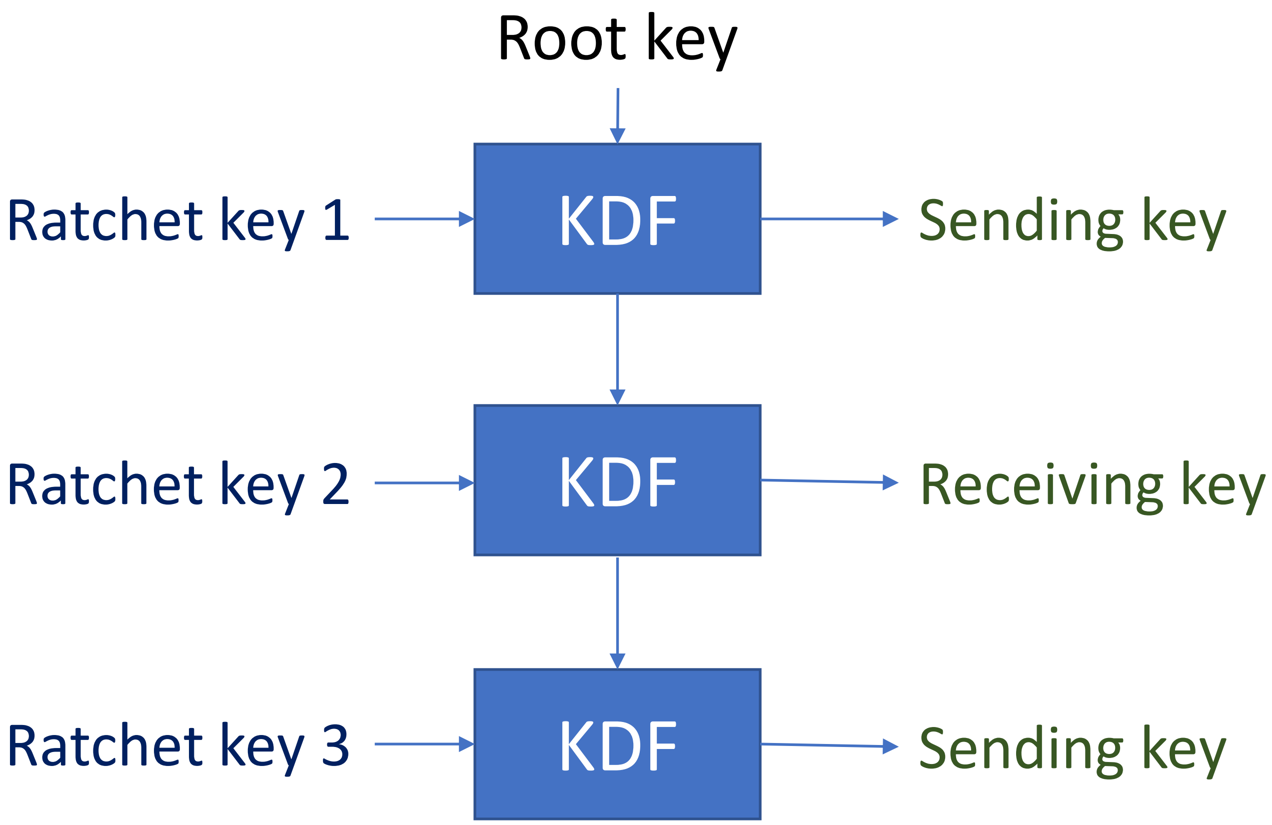


The point is to create usable temporary keys that can be leaked without compromising other keys.

Double Ratchet Algorithm

Symmetric Key Ratchet

First, the ratchet keys produces sending/receiving keys:

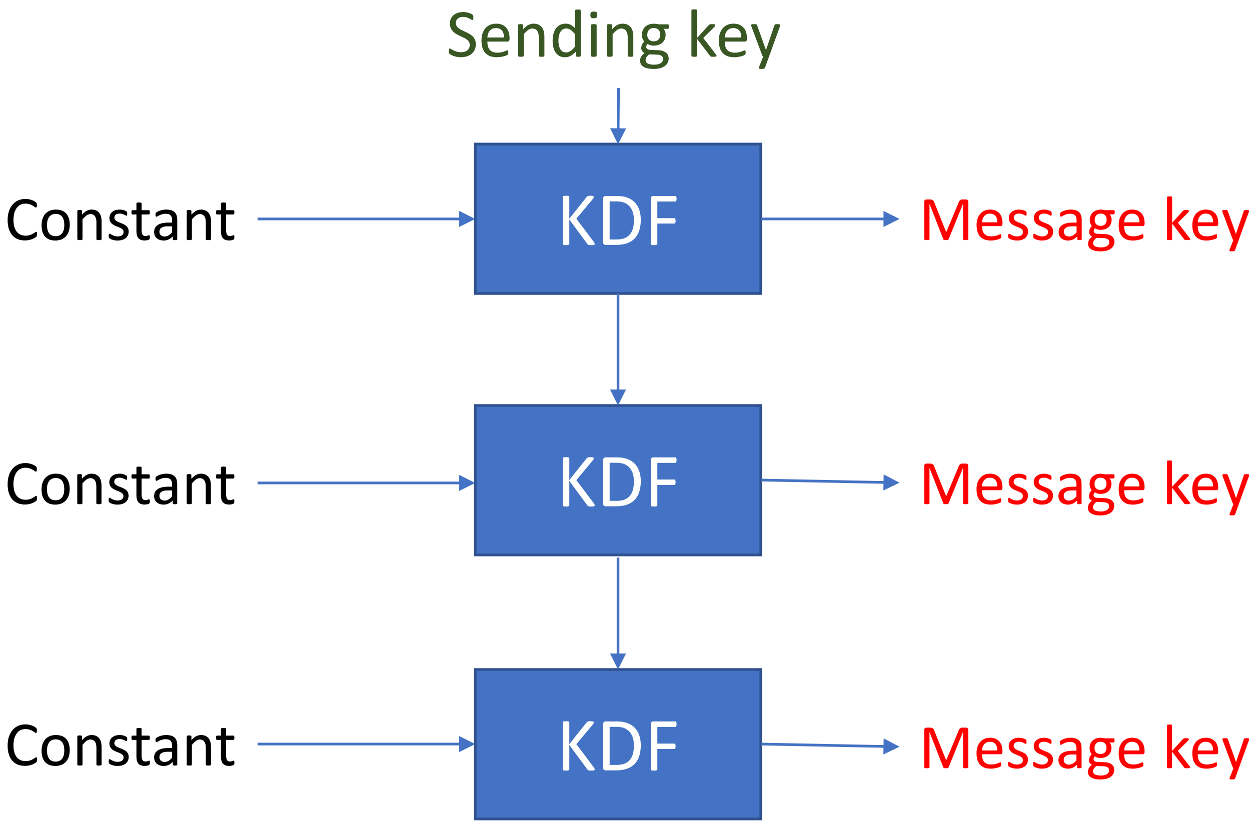


(Alice’s side) First ratchet key is Alice’s first sending key; Bob’s would start with a receiving key

Double Ratchet Algorithm

Symmetric Key Ratchet

Each sending/receiving key starts its own symmetric key KDF chain:



Each message key is used for only one message.

Double Ratchet Algorithm

Review

- KDF chains generates a series of keys, each key based on the previous root key and an input
- The DH ratchet generates and procedurally updates ratchet keys
  - A new chain is started whenever one side switches from receiving to sending
- The ratchet keys are used as input to the DH KDF chain to generate sending and receiving chain keys
- Chain keys are used as the bootstrapping root key for symmetric key DF chains to generate message keys

Double Ratchet Algorithm

- Benefits of using two ratchets:
  - Each message key can be deleted after one use; sending/receiving keys can be deleted after all relevant messages are sent/received
  - Handling of out-of-order/dropped messages is possible
  - Limits compromise of messages from key leakage
    - Sending/receiving keys can compromise multiple messages
    - Ratchet key plus a previous root key for the same
    - Each message key can leak one message

Double Ratchet Algorithm

- Can we also achieve forgeability?
  - Possibly, by releasing MAC keys (not decryption keys)
- A message participant can still *collude* with an outsider to prove messages sent by the other participant are real
  - It is possible to resolve this problem ("strong deniability")
- This does not work for group messaging
  - The property that an HMAC indirectly proves identity does not follow for group messaging

Zero-Knowledge Proofs

I want to login as Alice.

Okay, what's your password?

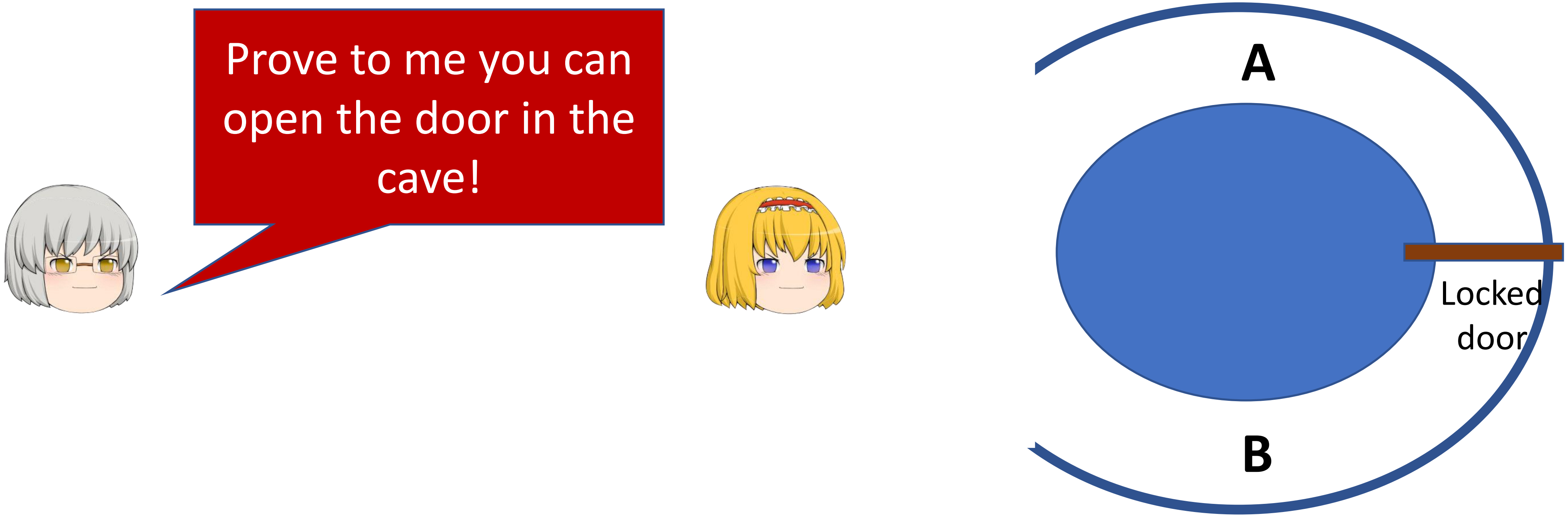
But I don't want to tell you my password!

??

What is zero knowledge?

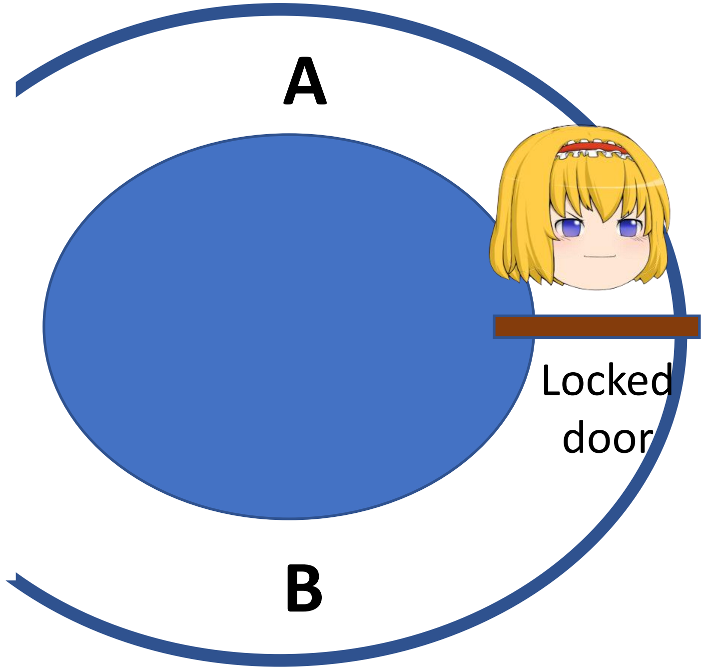
- Can Bob verify Alice without gaining knowledge of her password?
  - This is possible if Bob trusts Alice’s public key – they can use a signature protocol, but this is not zero-knowledge
- Generally, we want Alice to be able to prove her knowledge of her password while giving no knowledge to any observer
- This means that:
  - Any observer of the interactive proof gains no knowledge (how do we formalize this?)
  - The proof itself must also be unconvincing to an observer
  - The proof is only convincing to Bob

Example: ZKP Cave



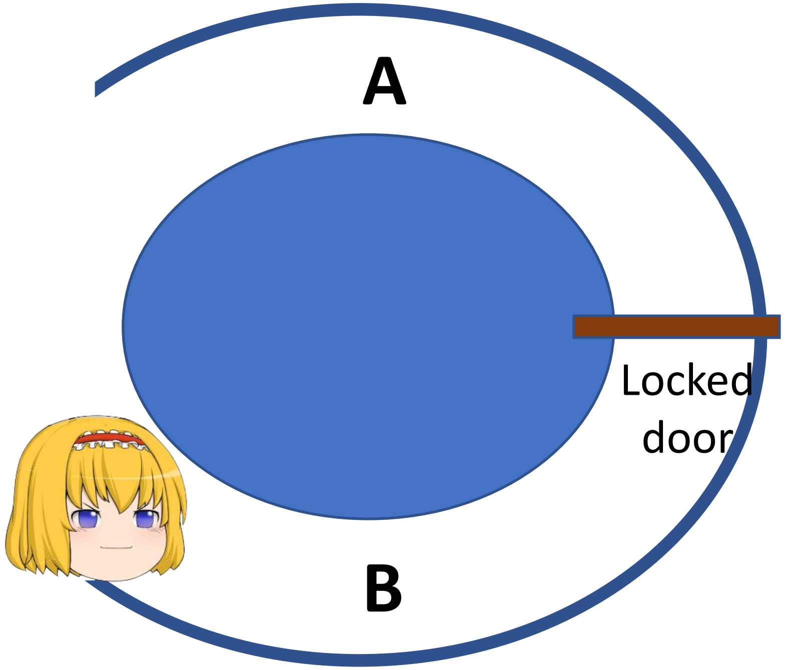
Alice wants to prove she knows the passcode for the door. She enters the cave (either from A or B) and waits there.

Example: ZKP Cave



Bob cannot see where Alice went. He secretly flips a coin and tells Alice to come out of the cave that way.

Example: ZKP Cave



Repeat the experiment enough times that Alice’s ability to come out of the cave every time is not due to random chance.

ZKP Transcripts must be Unconvincing

- The experiment cannot convince anyone but Bob that Alice can open the door
  - An observer notes that Bob and Alice can be colluding
  - If Bob’s coin flips are publicly recorded, then the proof is not zero-knowledge (it is convincing to an observer)
    - In practice, Bob could also pre-share his PRNG seed with Alice, so even non-recorded coin flips are not convincing
  - If Alice’s entry into the cave is recorded, it is also not zero-knowledge

Defining ZKP

- An interactive proof system is zero-knowledge if for any statement it is able to prove, there exists a **simulator** that can create a transcript of the interactive proof
  - The transcript must match what the verifier sees
  - The simulator is given any coin flips that the verifier may perform, and any pre-knowledge the verifier can use
  - Otherwise the simulator knows nothing
- It is easy for a simulator to create a transcript of the ZKP cave: The verifier yells a letter and Alice comes out of the cave that way

ZKP of discrete log

- Given *y* and a prime group modulo *p*, Alice proves that she knows *A* such that $g^A = y \bmod p$
  - This means proving possession of the private key under ElGamal for a given public key

Protocol

1. Alice generates a random number *r* and sends $g^r \bmod p$ to Bob.
2. Bob flips a coin.
   - Heads: Ask Alice to send *r*.
   - Tails: Ask Alice to send $(A+r) \bmod p\text{-}1$.
3. Bob verifies Alice sent the right message.

ZKP of discrete log – convinces Bob

- Bob's verification
  - Heads: Bob can compute $g^r \ mod \ p$.
    - This doesn't prove Alice knows A, though; only that she did not otherwise cheat in the protocol.
  - Tails: Bob can compute $g^{(A+r) \ mod \ p-1} \ mod \ p = g^A * g^r \ mod \ p$
    - Unless Alice knows A, this is highly unlikely
- Overall, for one run of the protocol, there is a marginally less than ½ chance that Alice can cheat Bob
- Repeat enough times for Bob to be convinced

ZKP of discrete log – does not convince anyone else

- How is the proof ZK? (Is it unconvincing to any observer?)
  - Equivalently, how can Alice and Bob "cheat" if Bob pre-shared his coin flips with Alice?
- If Alice knew ahead of time that Bob would flip tails…
  - Instead of generating a random number *r* and sending $g^r \bmod p$, she sends $g^{r'} * (g^x)^{-1}$ to Bob for some random number *r'*
  - In the second step, she simply sends *r'* for tails
- The above is how a simulator would create a correct transcript for any *A* (as the simulator is also given Bob's coin flips)

ZKP of 3-Coloring

- Given a graph, Alice proves that she knows how to assign vertices to up to 3 colors such that no edge connects two vertices of the same color
  - This is NP-complete => All NP-complete proofs can be ZKP

<u>Protocol</u>

1. Alice randomly chooses 1 of 6 possible colorings
2. Alice encrypts all edges and sends encryptions to Bob, but not the keys
3. Bob requires Alice to send keys for a random edge (two vertices), verifies them

This is the step that allows “cheating” to achieve zero-knowledge

Commitment schemes

- We can use commitment schemes to force Alice to commit
- Alice commits to a value *x*, *COM(x)*, such that:
  - Binding: Alice cannot find another *y* such that

    COM(y) = COM(x)
  - Hiding: An observer cannot find *x*
- Later, Alice can "open" the commitment to reveal *x*
- Pedersen commitment: COM(x) = $g^x h^r$ for public g, h, random r; open this commitment by revealing r
- Correct version of previous slide: Replace encryption with commitments (why?)

Reducing round-trips

- Both of the previous proofs require many rounds to complete, but this can be reduced to 1. Recall discrete log ZKP:

2. Bob flips a coin.
    - Heads (0): Ask Alice to send $(0*A+r) \bmod p-1$.
    - Tails (1): Ask Alice to send $(1*A+r) \bmod p-1$.

2. Bob generates a random challenge c.
    - Ask Alice to send $(c*A+r) \bmod p-1$

Reducing round-trips

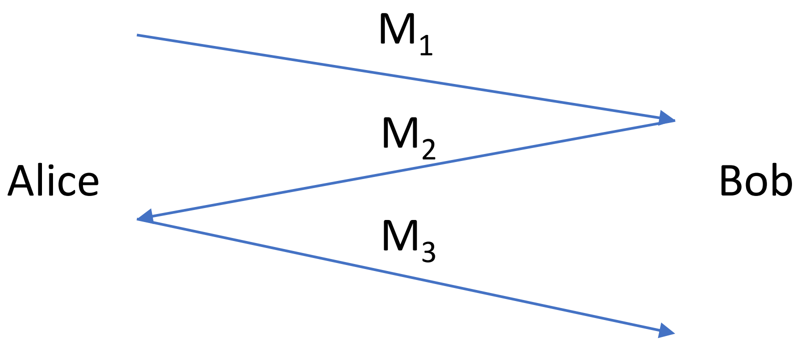
- Verification: Check that

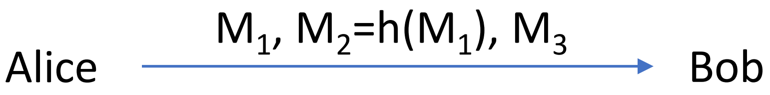
$$(g^A)^c g^r \equiv g^{(cA+r)\, mod\, p-1}\ (mod\ p)$$

- Simulator's "cheat": Knowing *c* ahead of time, Alice sends $g^{r'} * (g^{-Ac})$ in step one and $g^{r'}$ in step three
- p is a security parameter for the soundness of this proof

Non-Interactive ZKP

- We can further reduce one round to zero rounds using the *Fiat-Shamir heuristic* if Bob's only messages are random coin flips





- Key requirement: existence of a cryptographic hash that produces truly random output

Non-Interactive ZKP

- The hash function is used as a random oracle: It has a consistent but unpredictable mapping between inputs and outputs
- Open question: Is a hash truly a random oracle?
- Open question: What is truly *zero-knowledge* in the random oracle model?
    - The simulator needs to be able to “choose” the output of a random oracle – so have we lost deniability in ZKPs?

Applications of ZKP

- Authentication
  - Proof of ownership of private key for signatures
  - Password systems, access control
- Voting, auctions
- Electronic cash – ‘I have enough money for this transaction’
  - Adding privacy to Bitcoin (Zerocoin)
- Implementing private group chat
- Theoretical interest: language of statements that can be proven in ZKP

Blind Signatures

- Sometimes we want the signing party to gain no information on what they are signing
- e.g. electronic voting, e-cash
- Chaum's blind signatures:
  - Recall that in RSA, to sign a message *m* with private key d:

    $$\boxed{\text{Sign}(m, d) \equiv m^d \pmod N} \text{ for some } N = pq$$

  - Instead of signing m, use the signer's public key to blind the message with a random *r*:

    $$\boxed{m' \equiv mr^e \pmod N}$$

  - Then r^{-1} Sign(m', d) = Sign(m, d), i.e. Alice can obtain a valid signature for m without ever revealing it

Chaum’s e-cash

- Application of blind signatures:
  - Each signature is 1 coin (e.g. worth $1)
  - When a payer asks the bank to blindly sign a number “*c*”, the bank takes $1 from their account; “*c*” is now an e-coin
  - When a payer pays the payee, the payee checks the signature
  - The payee presents the signature to the coin, who will credit $1 to the payee’s account
- Signatures are recorded to prevent double spending
- Blind signatures are used so that the bank cannot use *c* themselves

Blind Signatures

- In the previous scheme, a seller needs to contact the bank before accepting a transaction
- For an off-line scheme, we need a way to detect cheating:
  - Alice has a public identity number *u*
  - Replace *c* in the previous slide with a coin of a specific format:

$$c = (c_1, c_2, \ldots, c_n),\ c_i = (h(a_i, b_i), h(a_i \oplus u, d_i))$$

  - When spending a coin, the seller randomly asks Alice to reveal either (a_i, b_i) or $(a_i \oplus u, d_i)$ for each *i*
  - If Alice double spends a coin, there is a high chance she will reveal both (a_i, b_i) and $(a_i \oplus u, d_i)$ for the same *i*

Secret Sharing

- We may want to “divide” a secret so that it can only be recovered only if *k* out of *n* people agree to recover it
  - e.g. nuclear codes, top-level secrets, data loss
- If fewer than *k* people agree to recover it, **no information** about the secret is discoverable
- Simple scheme for $k = n$:
  - $n - 1$ people get a random bit string
  - The last person gets the XOR of the secret with the $n-1$ random bit strings
  - Fewer than *k* people would essentially obtain a random string

Shamir’s Secret Sharing

- Intuition: *k* points define a polynomial of degree *k-1*
  - Any number of degree *k-1* polynomials can pass through *k-1* points
- Randomly generate a polynomial of degree *k-1*
- Find *n* points on the polynomial
  - e.g. (1, f(1)), (2, f(2)), …, (n, f(n))
- The points are the secret shares
- The constant is the secret

Verifiable Secret Sharing

- If secret generation is offloaded to a *dealer*, how can we know that the dealer has given us correct shares?
  - Risk: Dealer can distribute shares that are *inconsistent*, that is, some set of shares will reveal a different secret than some other set
  - In this case the polynomial has degree more than *k-1*
- (Benaloh's) Intuition: If two polynomials add up to degree at most k-1, then either they are both polynomials of degree at most k-1 or both polynomials of degree more than k-1

Verifiable Secret Sharing

Verify:

- Besides P, dealer also shares many "verification polynomials" P_1, P_2, … with degree at most *k-1*
- Verifier chooses a random subset *S* of verification polynomials, and asks the dealer to reveal all shares of *S*
- Everyone can recover the polynomials of S and see that they are degree at most *k-1*
- Dealer also reveals all shares of

$$\sum_{P_j \notin S} P_j + P$$

- Verifier also checks this has degree at most *k-1*
- It is very unlikely that *P* has degree more than *k-1* in this case