Secure Messaging Protocols

Sebastian R. Verschoor

Offensive Technologies – Guest Lecture

10 April 2025

University of Amsterdam
Informatics Institute

QuSoft

Disclaimer

These slides are optimized for didactic purposes. Primitives and protocols have been simplified, sometimes to the point where technically they are incorrect (and most likely insecure).

Outline

History (1/3)

- 1991: Phil Zimmermann creates Pretty Good Privacy (PGP)
  - February 1993: US starts criminal investigation for “munitions export without a license”
  - 1995: PGP source code published as a physical book
    - US first amendment protects export of books
  - 1996: Criminal investigation was dropped, no charges were filed
- 2004: Nikita Borisov, Ian Goldberg and Eric Brewer create OTR
  - “Off-the-Record Communication, or, Why Not To Use PGP”
  - Has forward secrecy
  - Has deniability
  - Requires both parties online for setting up

History (2/3)

- 2011: Gary Belvin introduces SecureSMS; "OTR for SMS"
- 2012: SCIMP (Silent Circle instant messaging protocol)
  - By Vinnie Moscaritolo, Gary Belvin and Phil Zimmermann
  - SecureSMS for XMPP
  - I formally verified its security with ProVerif
- February 2014: Open Whisper Systems releases TextSecure v2
  - Asynchronous: allows offline initial user message
  - Later renamed to Signal
- May 2014: SC updates to SCIMP v2
  - Asynchronous: allows offline initial user message
- August 2015: SC releases code for SCIMP v2
  - Adds more inconsistencies between code and documentation
  - I find and report many security bugs in the code
- September 2015: SC discontinues SCIMP, switches to Signal based protocol

History (3/3)

- February 2014: Facebook (now Meta) acquires WhatsApp
- April 2014: Signal announces partnership with WhatsApp
- April 2016: WhatsApp completes integration of Signal protocol
- November 2016: Trevor Perrin and Moxie Marlinspike release official specification for the Signal protocol
- September 2023: Signal gets post-quantum confidentiality

Security and Privacy features

- Confidentiality
- Integrity
- Availability
- (Key) Authentication
- Forward Secrecy
- Post-Compromise Security (PCS)
- Deniability vs. Non-repudiation
- Transport Privacy

Trust Establishment

- Opportunistic encryption
- Public Key Infrastructure (PKI)
- Web-of-Trust (WoT)
- Trust-On-First-Use (TOFU)
- Fingerprint verification
    - Socialist Millionaire Protocol (SMP)
    - Short Authentication String (SAS)
    - Safety Numbers
- Key directory
- Key transparency
- Blockchain(?)

Miscellaneous

- User experience
- Multi-device
- Group chat
- File transfer
- Video-chat
- Backups
- (Formal) verification
- Implementation security
- Audits

Apps and protocols

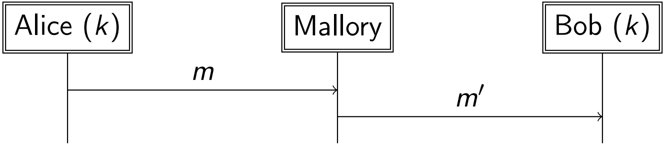
- Briar
- Discord (Dave)
- Dust
- Facebook Messenger
- Google Allo
- Google Chat
- Google Messages
- iMessage
- irc
- LINE
- Matrix (Olm/Megolm)
- Mattermost
- Pond
- QQ Mobile
- Rocket.Chat
- Session
- SimpleX
- Skype
- Slack
- SnapChat
- Teams
- Telegram (MTProto)
- Threema
- Viber
- WeChat
- WhatsApp
- Wickr
- Wire
- X
- XMPP (OMEMO)
- Zoom
- Zulip
- ...

many use the Signal protocol or a variant

Attacker model

End-to-End security (E2E, sometimes E2EE for encryption)

- all messages are handed to the adversary for delivery.[1]



Mallory has full control over all messages

- she may learn, change, inject, drop, reorder, and replay all messages

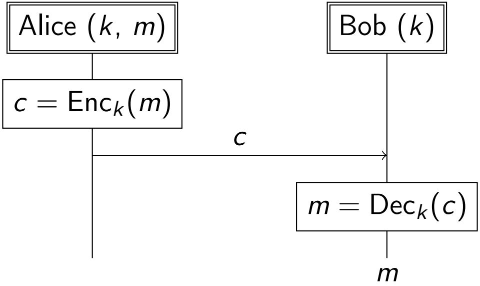
Kerckhoffs principle

- Mallory knows everything except the key

---

[1] For simplicity, I will omit Mallory from most diagrams for now.

Encryption

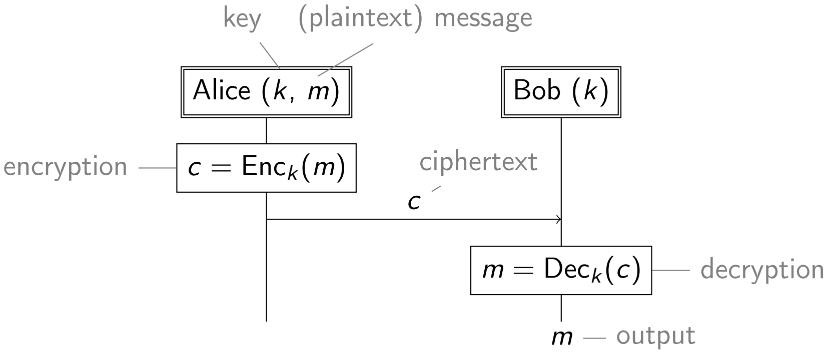


Provides confidentiality: Mallory learns nothing[2] about m

---

[2] almost nothing: I will not formalize this today

Encryption

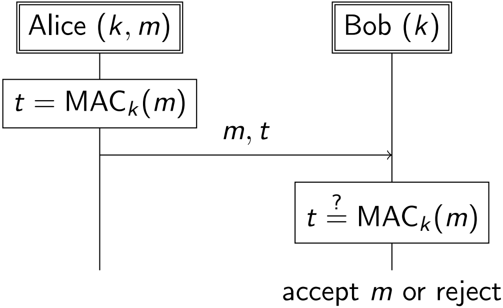


Provides confidentiality: Mallory learns nothing[2] about m

---

[2] almost nothing: I will not formalize this today

Message Authentication Code (MAC)



Alice (k, m)

Bob (k)

$t = \mathrm{MAC}_k(m)$

m, t

$t \overset{?}{=} \mathrm{MAC}_k(m)$

accept m or reject

Provides integrity: Mallory cannot[3] change m

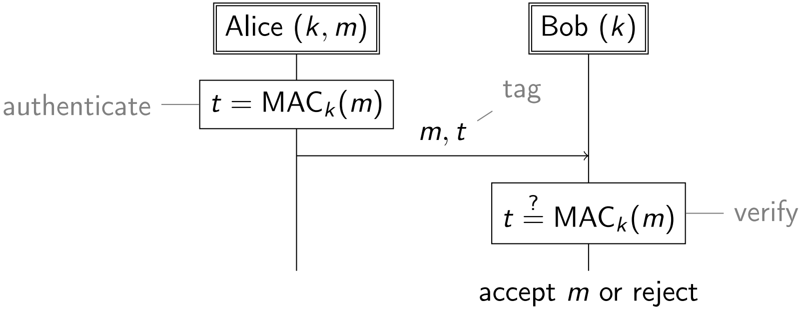
Provides authenticity: Bob knows m was sent by Alice

- *implicit*: Bob assumes only Alice knows k

---

[3] almost cannot: again we omit the details

Message Authentication Code (MAC)



Alice (k, m)

Bob (k)

authenticate

$t = \mathrm{MAC}_k(m)$

tag

m, t

$t \overset{?}{=} \mathrm{MAC}_k(m)$

verify

accept m or reject

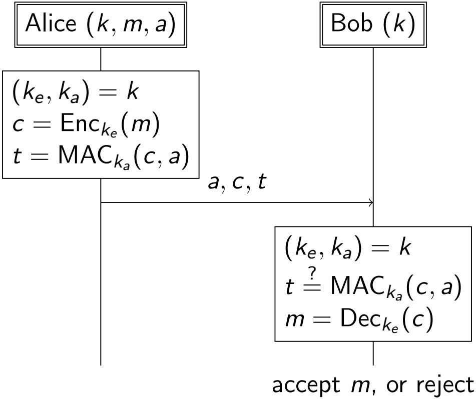
Provides integrity: Mallory cannot[3] change m
Provides authenticity: Bob knows m was sent by Alice

- *implicit*: Bob assumes only Alice knows k

---

[3] almost cannot: again we omit the details

Authenticated Encryption



Alice (k, m, a)

Bob (k)

$(k_e, k_a) = k$
$c = \mathrm{Enc}_{k_e}(m)$
$t = \mathrm{MAC}_{k_a}(c, a)$

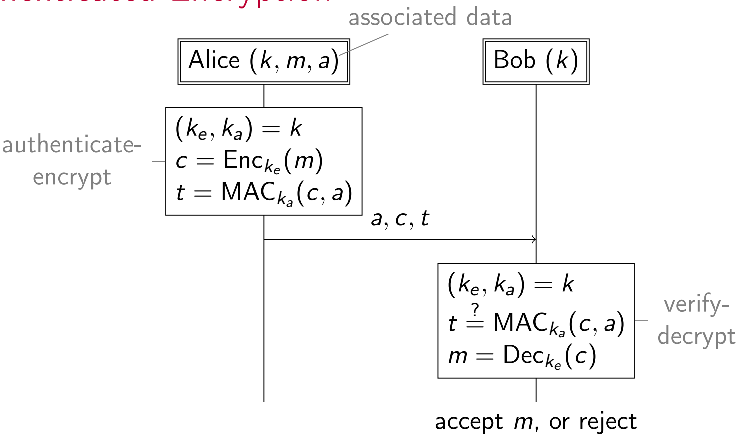
a, c, t

$(k_e, k_a) = k$
$t \overset{?}{=} \mathrm{MAC}_{k_a}(c, a)$
$m = \mathrm{Dec}_{k_e}(c)$

accept m, or reject

- Real-world AE often differs from encrypt-then-MAC
- Simplified notation:
  - $c = \mathrm{AEAD}_k(m, a)$ c includes the tag
  - $c = \mathrm{AEnc}_k(m)$ no a
  - $m = \mathrm{VDec}_k(c, a)$ $m = \perp$ on rejection

Authenticated Encryption



associated data

Alice (k, m, a)

Bob (k)

authenticate-encrypt

$(k_e, k_a) = k$
$c = \mathrm{Enc}_{k_e}(m)$
$t = \mathrm{MAC}_{k_a}(c, a)$

a, c, t

$(k_e, k_a) = k$
$t \overset{?}{=} \mathrm{MAC}_{k_a}(c, a)$
$m = \mathrm{Dec}_{k_e}(c)$

verify-decrypt

accept m, or reject

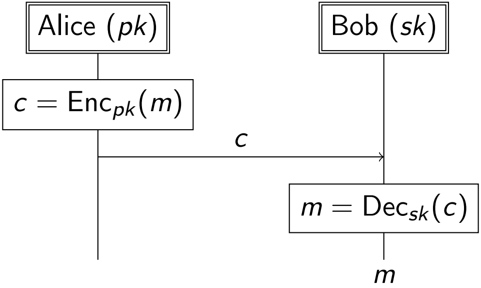
- Real-world AE often differs from encrypt-then-MAC
- Simplified notation:
  - $c = \mathrm{AEAD}_k(m, a)$ c includes the tag
  - $c = \mathrm{AEnc}_k(m)$ no a
  - $m = \mathrm{VDec}_k(c, a)$ $m = \perp$ on rejection

Symmetric Cryptography

Strong assumption:

- Alice and Bob have *the same* secret key k

Public Key Encryption



Alice (pk)

Bob (sk)

$c = \mathsf{Enc}_{pk}(m)$

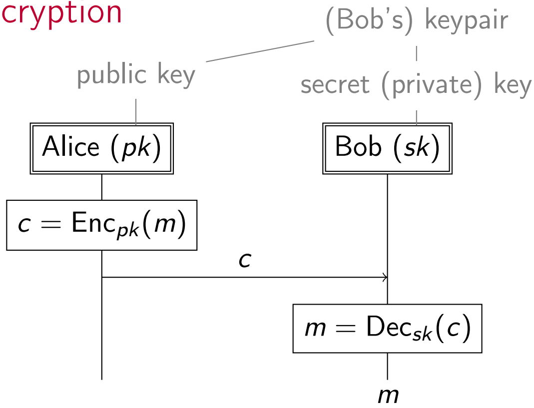
c

$m = \mathsf{Dec}_{sk}(c)$

m

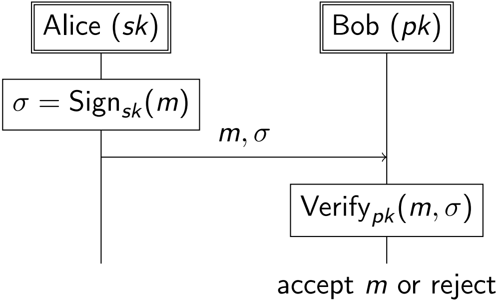
- Provides confidentiality
- Bob publishes pk, keeps sk secret
  - Anyone can encrypt, but only Bob can decrypt

Public Key Encryption



- Provides confidentiality
- Bob publishes *pk*, keeps *sk* secret
    - Anyone can encrypt, but only Bob can decrypt

Digital Signatures



Alice (sk)

Bob (pk)

$\sigma = \mathsf{Sign}_{sk}(m)$

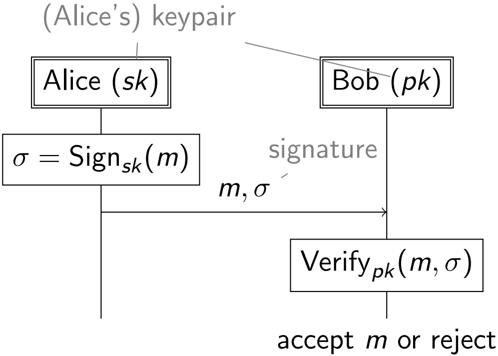
m, σ

$\mathsf{Verify}_{pk}(m, \sigma)$

accept m or reject

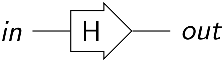
- Provides integrity and authentication
- Alice publishes pk, keeps sk secret
  - Only Alice can sign, but anyone can verify
  - Provides non-repudiation: (m, σ) is proof *for anyone* that Alice sent m

Digital Signatures



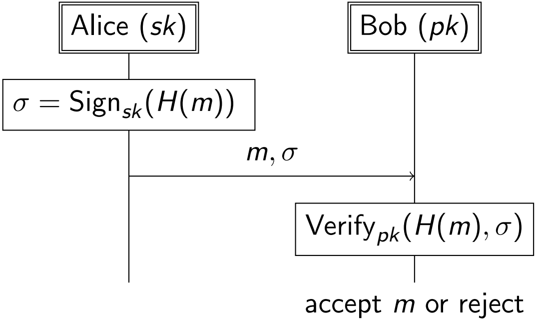
- Provides integrity and authentication
- Alice publishes *pk*, keeps *sk* secret
  - Only Alice can sign, but anyone can verify
  - Provides non-repudiation: (m, σ) is proof *for anyone* that Alice sent m

Cryptographic Hash



- Let H be a hash function: $out = H(in)$
  - large input
  - small output (digest)
- Security
  - the output "seems random" different ways to formalize this
  - Mallory cannot compute *out* without knowing *in*
  - Mallory cannot compute *in* when given *out*
- Note there is no key involved
  - Mallory can compute H on inputs of her choice

Digital signatures with hashes



Alice (sk)

Bob (pk)

$\sigma = \mathsf{Sign}_{sk}(H(m))$

m, σ

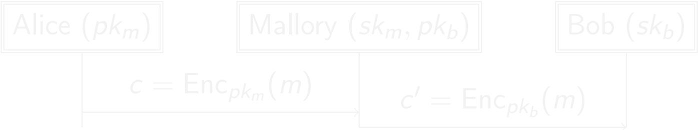
$\mathsf{Verify}_{pk}(H(m), \sigma)$

accept m or reject

Public Key Cryptography

- No large messages
  - PKE has limited size of messages
  - PKE is relatively slow
- How does Alice know that *pk* belongs to Bob?

Mallory-in-the-Middle (MitM) attack:



Alice (pk_m)

Mallory (sk_m, pk_b)

Bob (sk_b)

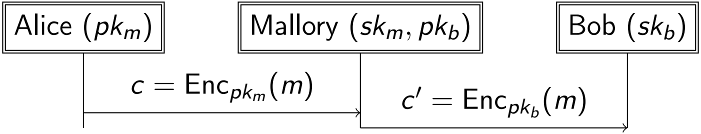
$c = \mathrm{Enc}_{pk_m}(m)$

$c' = \mathrm{Enc}_{pk_b}(m)$

Public Key Cryptography

- No large messages
  - PKE has limited size of messages
  - PKE is relatively slow
- How does Alice know that *pk* belongs to Bob?

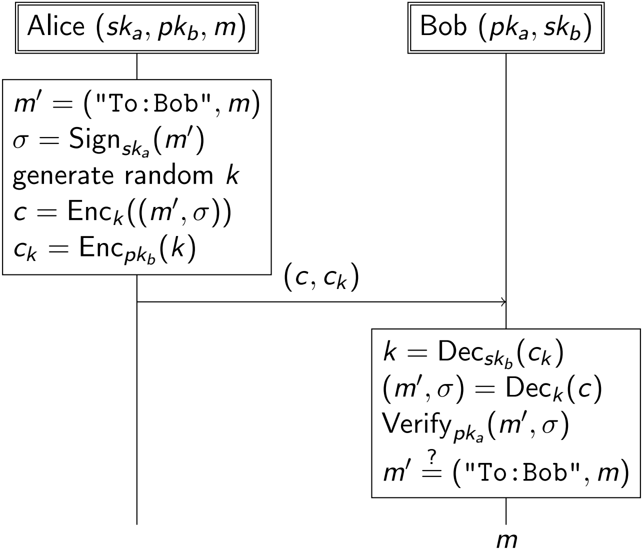
Mallory-in-the-Middle (MitM) attack:



PKC: key/user authentication

- Not just an issue for secure messaging protocols
- SSH uses Trust On First Use (TOFU)
  - asks user to verify public key fingerprint (its hash) on first login
    - how to verify this, the protocol does not say
  - once accepted, it will silently keep accepting until the key changes
- TLS uses certificates and a PKI
  - you connect to a server
  - server presents a certificate: “this public key belongs to this website”
  - the certificate is signed by an authority
  - you as a user trust the authority (right?)
- PGP, OTR, SCIMP and Signal all use different methods

Pretty Good Privacy



Alice (sk_a, pk_b, m)

$m' = (\texttt{"To:Bob"}, m)$
$\sigma = \mathsf{Sign}_{sk_a}(m')$
generate random k
$c = \mathsf{Enc}_k((m', \sigma))$
$c_k = \mathsf{Enc}_{pk_b}(k)$

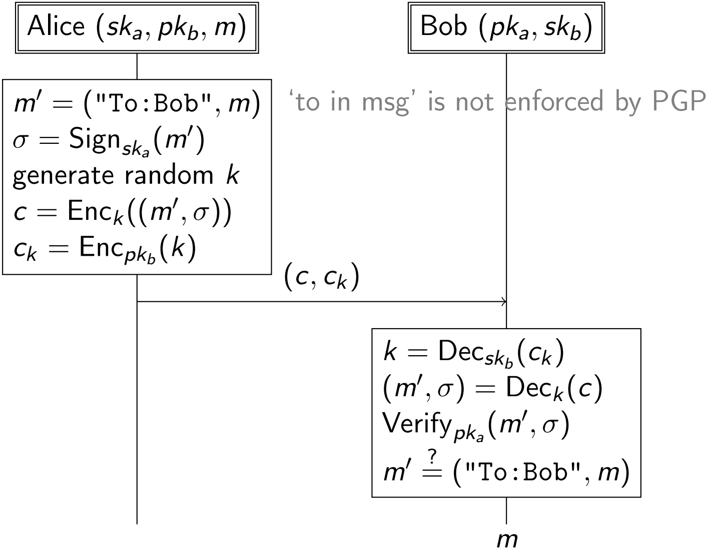
(c, c_k)

Bob (pk_a, sk_b)

$k = \mathsf{Dec}_{sk_b}(c_k)$
$(m', \sigma) = \mathsf{Dec}_k(c)$
$\mathsf{Verify}_{pk_a}(m', \sigma)$
$m' \stackrel{?}{=} (\texttt{"To:Bob"}, m)$

m

Pretty Good Privacy



Alice (sk_a, pk_b, m)

Bob (pk_a, sk_b)

$m' = (\texttt{"To:Bob"}, m)$
$\sigma = \mathsf{Sign}_{sk_a}(m')$
generate random k
$c = \mathsf{Enc}_k((m', \sigma))$
$c_k = \mathsf{Enc}_{pk_b}(k)$

‘to in msg’ is not enforced by PGP

(c, c_k)

$k = \mathsf{Dec}_{sk_b}(c_k)$
$(m', \sigma) = \mathsf{Dec}_k(c)$
$\mathsf{Verify}_{pk_a}(m', \sigma)$
$m' \overset{?}{=} (\texttt{"To:Bob"}, m)$

m

PGP: key authentication

- PGP uses the Web of Trust (WoT)
- You meet in person, then sign each others key
  - yes there are (were?) key signing parties!
- Everyone publishes the signed keys
- If you get a key you don't know, you check if it's signed by someone you trust

Everyone loves this system and it scales great in practice!

PGP

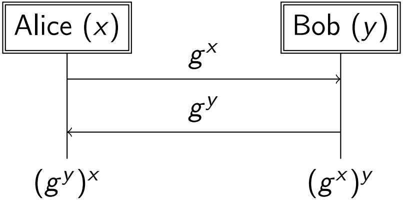
- PGP uses hybrid encryption
    - faster + larger messages
- PGP has many options
    - I am 80% sure the above is the sign-and-encrypt option
    - Without recipient ID, Bob could re-encrypt to others
        - not part of the PGP specification
    - Complexity leads to bad user experience, which leads to loss of security
- Two major issues
    - If a private key leaks, all messages leak (past and future)
    - Bob can publish (m, σ) as evidence that Alice said m

Off-the-Record (OTR)

Design philosophy: A secure online conversation should be more like a private in-person conversation

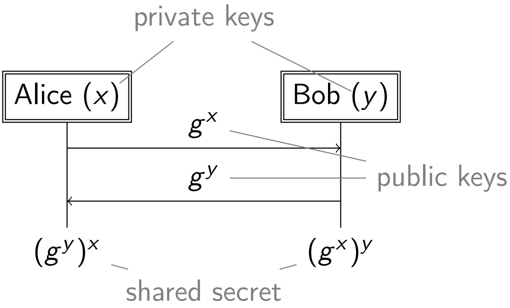
- Forward secrecy
    - leaking long-term keys should not reveal information about old messages
    - "key erasure property"
- Deniability (informal)
    - leaking a secure online conversation should not leak any more information than leaking a plain-text conversation would
        - has many subtly different *mathematical* formalizations
        - unclear if this affects *legal* deniability

Diffie-Hellman Key Exchange



- correct: $g^{xy} = g^{yx}$
- security: follows from the Diffie-Hellman assumption:
  - Given g^x and g^y, it's hard to compute g^{xy}

Diffie-Hellman Key Exchange



private keys

Alice (x)

Bob (y)

g^x

g^y

public keys

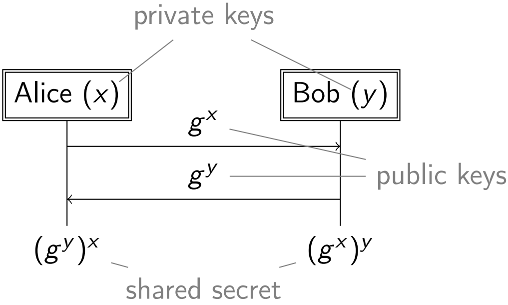
$(g^y)^x$

$(g^x)^y$

shared secret

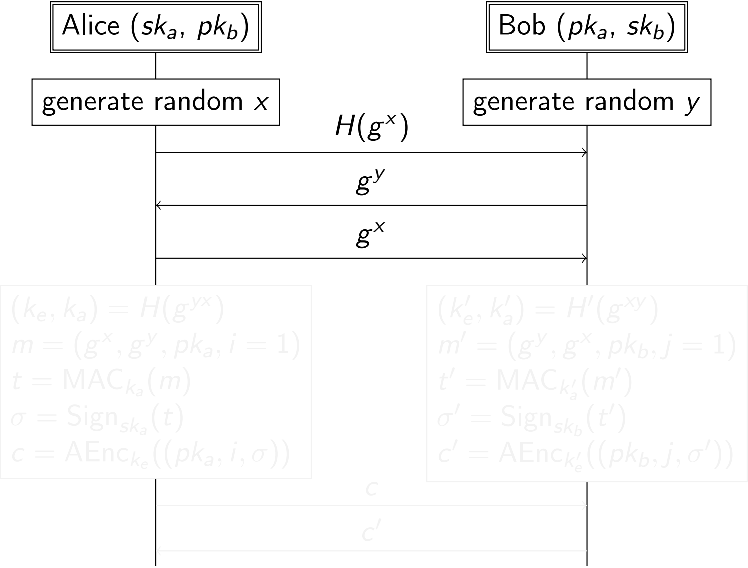
- correct: $g^{xy} = g^{yx}$
- security: follows from the Diffie-Hellman assumption:
  - Given g^x and g^y, it's hard to compute g^{xy}

Diffie-Hellman Key Exchange



- correct: $g^{xy} = g^{yx}$
- security: follows from the Diffie-Hellman assumption:
  - Given g^x and g^y, it's hard to compute g^{xy}

OTR: Authenticated Key Exchange



Alice (sk_a, pk_b)

Bob (pk_a, sk_b)

generate random x

generate random y

$H(g^x)$

g^y

g^x

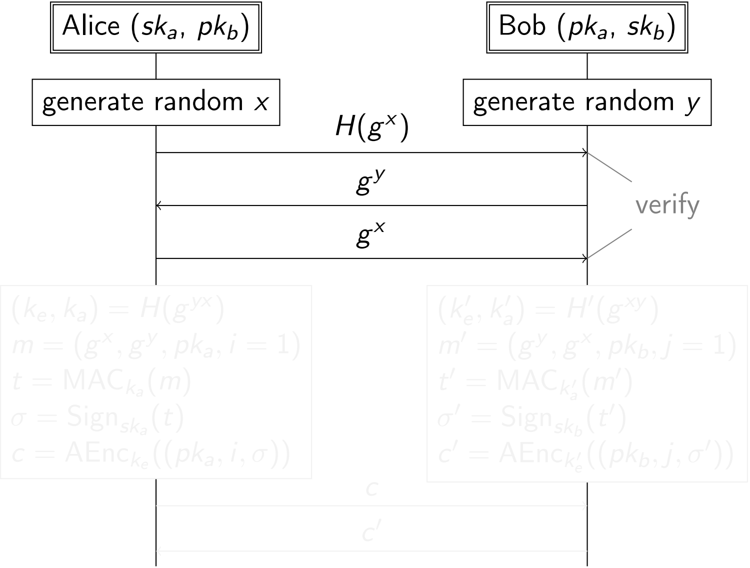
$(k_e, k_a) = H(g^{yx})$
$m = (g^x, g^y, pk_a, i = 1)$
$t = \mathrm{MAC}_{k_a}(m)$
$\sigma = \mathrm{Sign}_{sk_a}(t)$
$c = \mathrm{AEnc}_{k_e}((pk_a, i, \sigma))$

$(k'_e, k'_a) = H'(g^{xy})$
$m' = (g^y, g^x, pk_b, j = 1)$
$t' = \mathrm{MAC}_{k'_a}(m')$
$\sigma' = \mathrm{Sign}_{sk_b}(t')$
$c' = \mathrm{AEnc}_{k'_e}((pk_b, j, \sigma'))$

c

c'

OTR: Authenticated Key Exchange



Alice (sk_a, pk_b)

Bob (pk_a, sk_b)

generate random x

generate random y

$H(g^x)$

g^y

g^x

verify

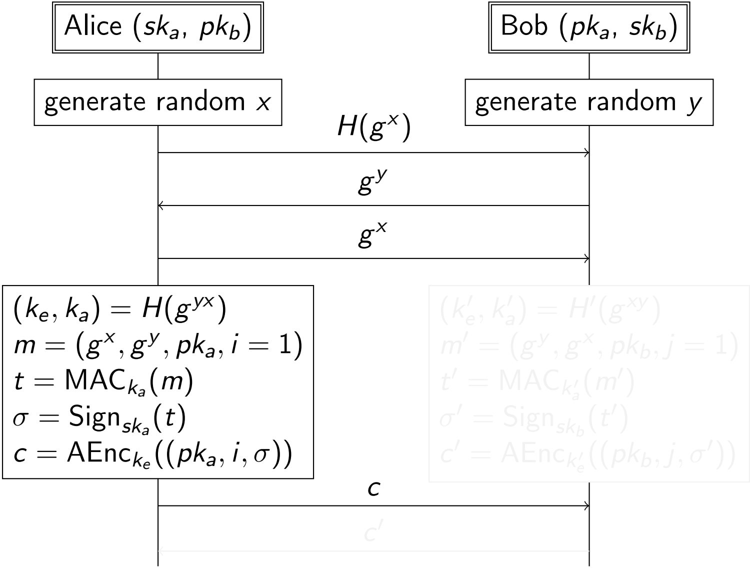
$(k_e, k_a) = H(g^{yx})$
$m = (g^x, g^y, pk_a, i = 1)$
$t = \mathrm{MAC}_{k_a}(m)$
$\sigma = \mathrm{Sign}_{sk_a}(t)$
$c = \mathrm{AEnc}_{k_e}((pk_a, i, \sigma))$

$(k'_e, k'_a) = H'(g^{xy})$
$m' = (g^y, g^x, pk_b, j = 1)$
$t' = \mathrm{MAC}_{k'_a}(m')$
$\sigma' = \mathrm{Sign}_{sk_b}(t')$
$c' = \mathrm{AEnc}_{k'_e}((pk_b, j, \sigma'))$

c

c'

OTR: Authenticated Key Exchange



Alice (sk_a, pk_b)

Bob (pk_a, sk_b)

generate random x

generate random y

$H(g^x)$

g^y

g^x

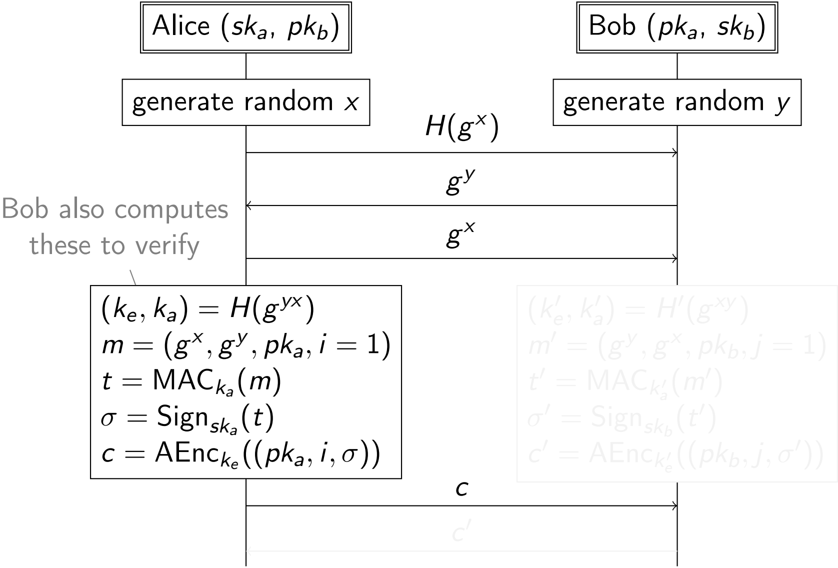
$(k_e, k_a) = H(g^{yx})$
$m = (g^x, g^y, pk_a, i = 1)$
$t = \mathrm{MAC}_{k_a}(m)$
$\sigma = \mathrm{Sign}_{sk_a}(t)$
$c = \mathrm{AEnc}_{k_e}((pk_a, i, \sigma))$

$(k'_e, k'_a) = H'(g^{xy})$
$m' = (g^y, g^x, pk_b, j = 1)$
$t' = \mathrm{MAC}_{k'_a}(m')$
$\sigma' = \mathrm{Sign}_{sk_b}(t')$
$c' = \mathrm{AEnc}_{k'_e}((pk_b, j, \sigma'))$

c

c'

OTR: Authenticated Key Exchange



Alice $(sk_a,\ pk_b)$

Bob $(pk_a,\ sk_b)$

generate random x

generate random y

$H(g^x)$

g^y

g^x

Bob also computes these to verify

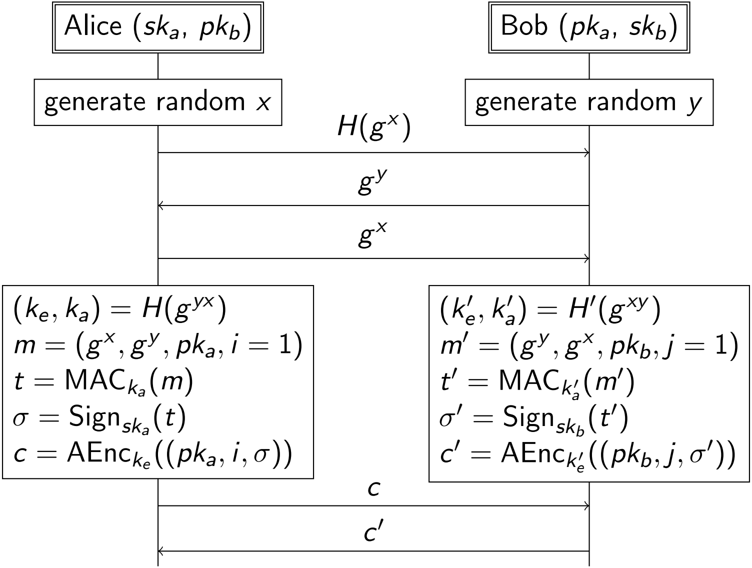
$(k_e, k_a) = H(g^{yx})$
$m = (g^x, g^y, pk_a, i = 1)$
$t = \mathrm{MAC}_{k_a}(m)$
$\sigma = \mathrm{Sign}_{sk_a}(t)$
$c = \mathrm{AEnc}_{k_e}((pk_a, i, \sigma))$

$(k_e', k_a') = H'(g^{xy})$
$m' = (g^y, g^x, pk_b, j = 1)$
$t' = \mathrm{MAC}_{k_a'}(m')$
$\sigma' = \mathrm{Sign}_{sk_b}(t')$
$c' = \mathrm{AEnc}_{k_e'}((pk_b, j, \sigma'))$

c

c'

OTR: Authenticated Key Exchange



Alice (sk_a, pk_b)

Bob (pk_a, sk_b)

generate random x

generate random y

$H(g^x)$

g^y

g^x

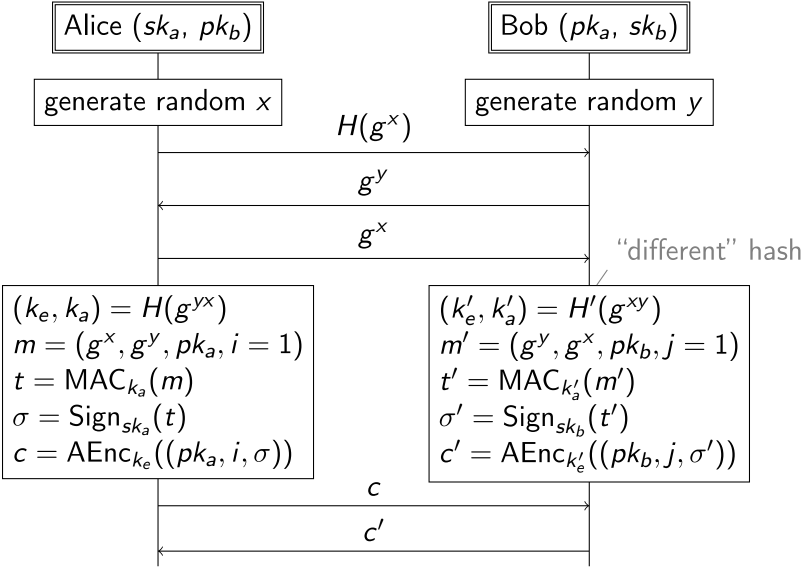
$(k_e, k_a) = H(g^{yx})$
$m = (g^x, g^y, pk_a, i = 1)$
$t = \mathrm{MAC}_{k_a}(m)$
$\sigma = \mathrm{Sign}_{sk_a}(t)$
$c = \mathrm{AEnc}_{k_e}((pk_a, i, \sigma))$

$(k'_e, k'_a) = H'(g^{xy})$
$m' = (g^y, g^x, pk_b, j = 1)$
$t' = \mathrm{MAC}_{k'_a}(m')$
$\sigma' = \mathrm{Sign}_{sk_b}(t')$
$c' = \mathrm{AEnc}_{k'_e}((pk_b, j, \sigma'))$

c

c'

OTR: Authenticated Key Exchange



Alice $(sk_a,\ pk_b)$

Bob $(pk_a,\ sk_b)$

generate random x

generate random y

$H(g^x)$

g^y

g^x

“different” hash

$$(k_e, k_a) = H(g^{yx})$$
$$m = (g^x, g^y, pk_a, i = 1)$$
$$t = \mathrm{MAC}_{k_a}(m)$$
$$\sigma = \mathrm{Sign}_{sk_a}(t)$$
$$c = \mathrm{AEnc}_{k_e}((pk_a, i, \sigma))$$

$$(k'_e, k'_a) = H'(g^{xy})$$
$$m' = (g^y, g^x, pk_b, j = 1)$$
$$t' = \mathrm{MAC}_{k'_a}(m')$$
$$\sigma' = \mathrm{Sign}_{sk_b}(t')$$
$$c' = \mathrm{AEnc}_{k'_e}((pk_b, j, \sigma'))$$

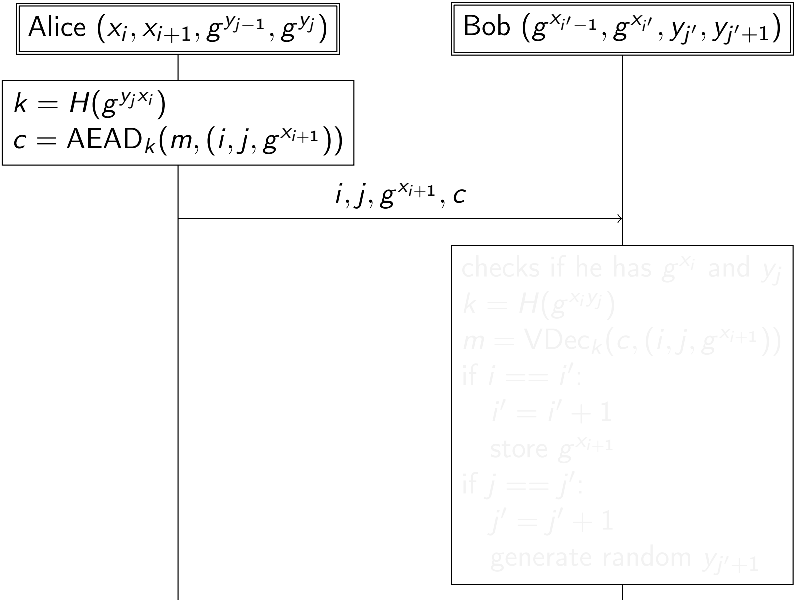
c

c'

OTR

- Deniability:
  - Parties only ever sign public values (g^x, g^y, pk)
    - No proof of conversation contents
  - Parties only ever sign *their own* public key
    - No proof of intent to communicate with other party
- Forward secrecy:
  - Securely delete x once we are done with it
    - (x, g^x) is called an ephemeral keypair
  - No information left on the device to recompute k_e
  - Discard k_e once we are done with it
    - . . . but when is that?

OTR: Sending Data Message



Alice $(x_i, x_{i+1}, g^{y_{j-1}}, g^{y_j})$

Bob $(g^{x_{i'-1}}, g^{x_{i'}}, y_{j'}, y_{j'+1})$

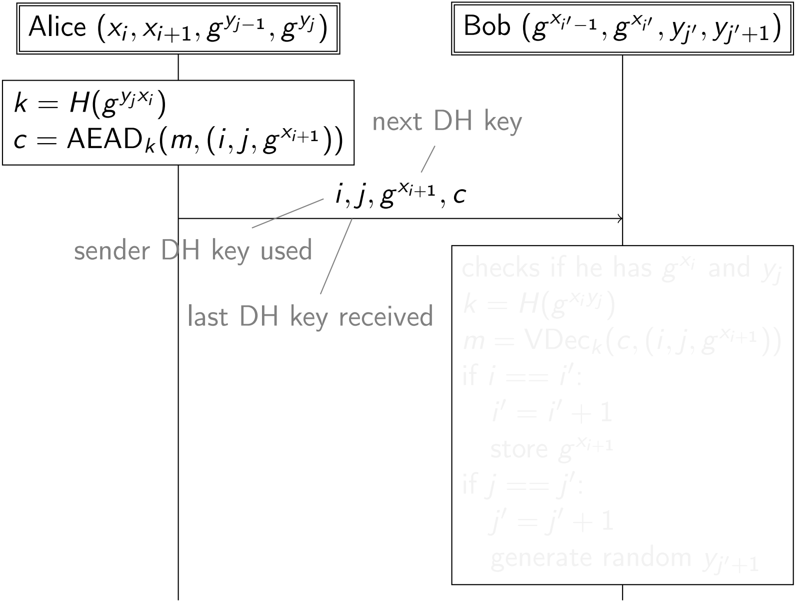
$k = H(g^{y_j x_i})$

$c = \text{AEAD}_k(m, (i, j, g^{x_{i+1}}))$

$i, j, g^{x_{i+1}}, c$

```
checks if he has $g^{x_i}$ and $y_j$
$k = H(g^{x_i y_j})$
$m = \text{VDec}_k(c, (i, j, g^{x_{i+1}}))$
if $i == i'$:
    $i' = i' + 1$
    store $g^{x_{i+1}}$
if $j == j'$:
    $j' = j' + 1$
    generate random $y_{j'+1}$
```

OTR: Sending Data Message



Alice $(x_i, x_{i+1}, g^{y_{j-1}}, g^{y_j})$

Bob $(g^{x_{i'-1}}, g^{x_{i'}}, y_{j'}, y_{j'+1})$

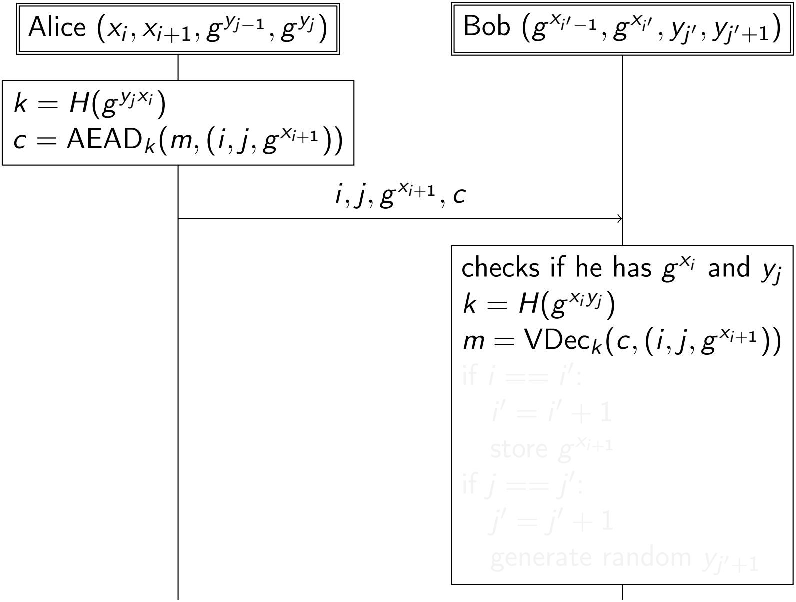
$k = H(g^{y_j x_i})$

$c = \text{AEAD}_k(m, (i, j, g^{x_{i+1}}))$

$i, j, g^{x_{i+1}}, c$

```
checks if he has $g^{x_i}$ and $y_j$
$k = H(g^{x_i y_j})$
$m = \text{VDec}_k(c, (i, j, g^{x_{i+1}}))$
if $i == i'$:
    $i' = i' + 1$
    store $g^{x_{i+1}}$
if $j == j'$:
    $j' = j' + 1$
    generate random $y_{j'+1}$
```

OTR: Sending Data Message



Alice $(x_i, x_{i+1}, g^{y_{j-1}}, g^{y_j})$

Bob $(g^{x_{i'-1}}, g^{x_{i'}}, y_{j'}, y_{j'+1})$

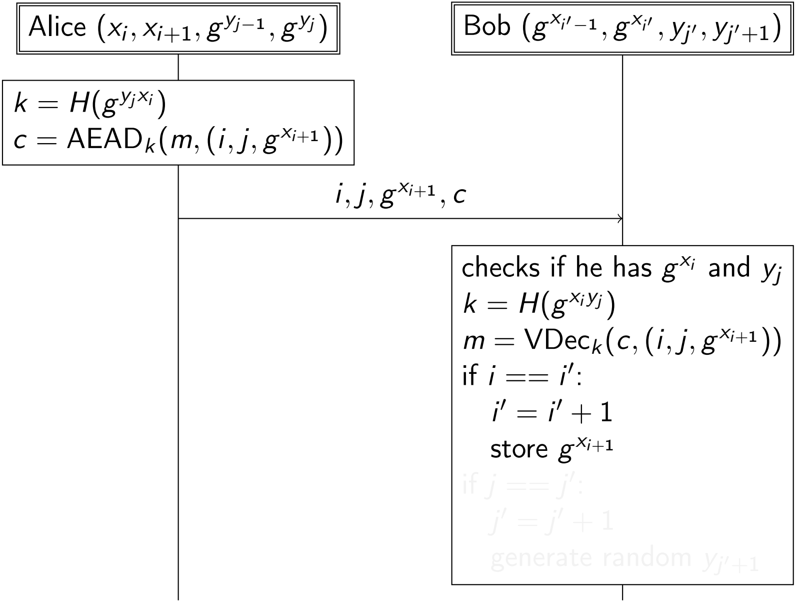
$k = H(g^{y_j x_i})$
$c = \mathrm{AEAD}_k(m, (i, j, g^{x_{i+1}}))$

$i, j, g^{x_{i+1}}, c$

checks if he has g^{x_i} and y_j
$k = H(g^{x_i y_j})$
$m = \mathrm{VDec}_k(c, (i, j, g^{x_{i+1}}))$

```
if $i == i'$:
    $i' = i' + 1$
    store $g^{x_{i+1}}$
if $j == j'$:
    $j' = j' + 1$
    generate random $y_{j'+1}$
```

OTR: Sending Data Message



Alice $(x_i, x_{i+1}, g^{y_{j-1}}, g^{y_j})$

Bob $(g^{x_{i'-1}}, g^{x_{i'}}, y_{j'}, y_{j'+1})$

$k = H(g^{y_j x_i})$

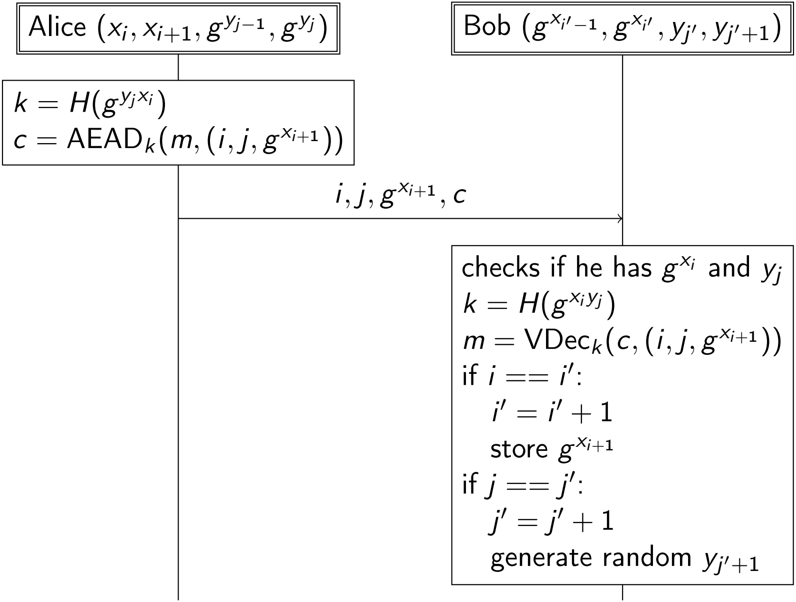
$c = \mathrm{AEAD}_k(m, (i, j, g^{x_{i+1}}))$

$i, j, g^{x_{i+1}}, c$

checks if he has g^{x_i} and y_j

```
$k = H(g^{x_i y_j})$
$m = \mathrm{VDec}_k(c, (i, j, g^{x_{i+1}}))$
if $i == i'$:
    $i' = i' + 1$
    store $g^{x_{i+1}}$
if $j == j'$:
    $j' = j' + 1$
    generate random $y_{j'+1}$
```

OTR: Sending Data Message



Alice $(x_i, x_{i+1}, g^{y_{j-1}}, g^{y_j})$

Bob $(g^{x_{i'-1}}, g^{x_{i'}}, y_{j'}, y_{j'+1})$

```
$k = H(g^{y_j x_i})$
$c = \mathrm{AEAD}_k(m, (i, j, g^{x_{i+1}}))$
```

$i, j, g^{x_{i+1}}, c$

```
checks if he has $g^{x_i}$ and $y_j$
$k = H(g^{x_i y_j})$
$m = \mathrm{VDec}_k(c, (i, j, g^{x_{i+1}}))$
if $i == i'$:
    $i' = i' + 1$
    store $g^{x_{i+1}}$
if $j == j'$:
    $j' = j' + 1$
    generate random $y_{j'+1}$
```

OTR

- Forward Secrecy:
  - with every *reply* we remove an old key
  - old keys cannot be derived from previous ones
  - one-sided conversations don't move forward
    - could be fixed with heartbeat messages
- Post Compromise Security:
  - if Mallory steals your keys she can read your messages
  - once you generated a new DH key, she no longer has access
  - (this is not true if she actively maintains a MitM attack)
- Can handle missing a message, however
  - cannot handle out of order messages
    - storing old keys would compromise FS

OTR: key authentication

Option 1:

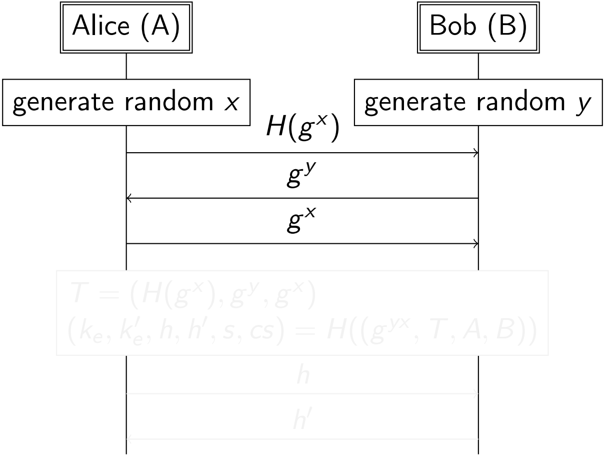
- Users can see the used public key fingerprints
- Verify these out-of-band

Option 2:

- Users are assumed to share some secret that Mallory doesn't know
- Hashes used public keys and the secret together
- Compare if they are the same using a zero-knowledge protocol
- this all happens in-band!

Usability studies show issues with both

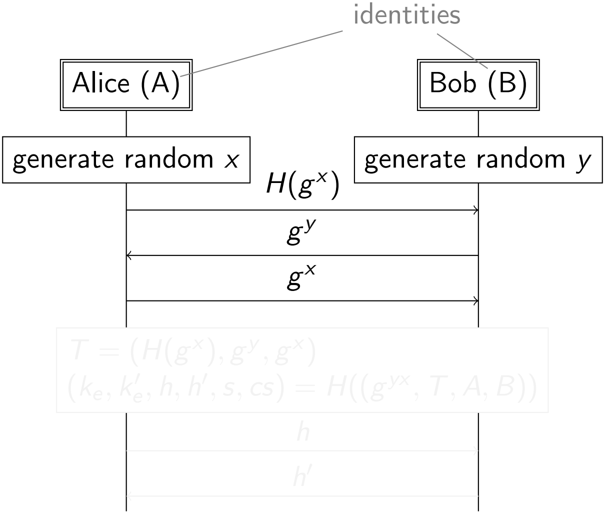
SCIMP: Key Exchange



$T = (H(g^x), g^y, g^x)$

$(k_e, k'_e, h, h', s, cs) = H((g^{yx}, T, A, B))$

SCIMP: Key Exchange



identities

Alice (A)

Bob (B)

generate random x

generate random y

$H(g^x)$

g^y

g^x

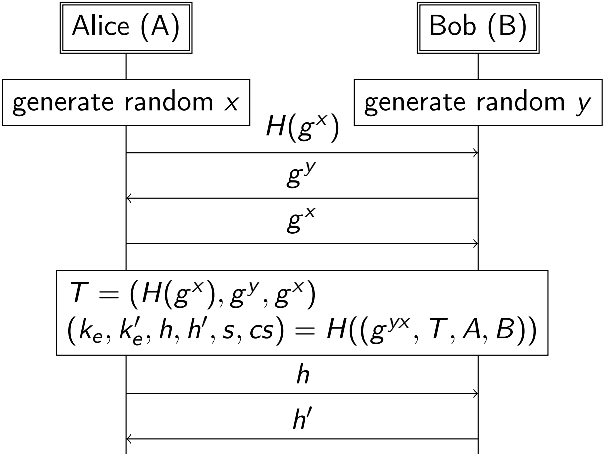
$T = (H(g^x), g^y, g^x)$

$(k_e, k'_e, h, h', s, cs) = H((g^{yx}, T, A, B))$

h

h'

SCIMP: Key Exchange



Alice (A)

Bob (B)

generate random x

generate random y

$H(g^x)$

g^y

g^x

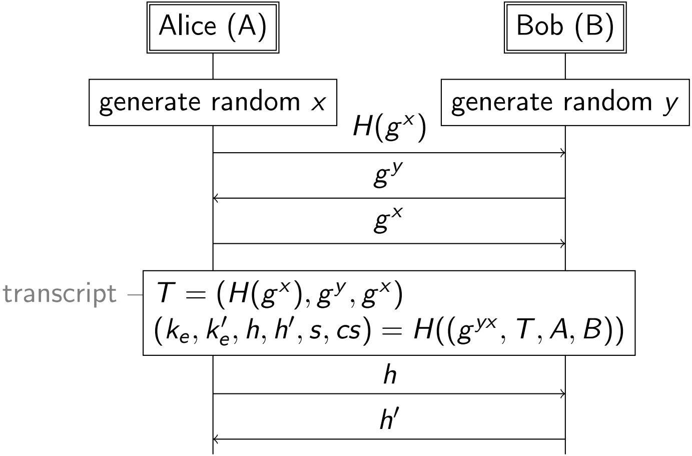
$T = (H(g^x), g^y, g^x)$

$(k_e, k_e', h, h', s, cs) = H((g^{yx}, T, A, B))$

h

h'

SCIMP: Key Exchange



Alice (A)

Bob (B)

generate random x

generate random y

$H(g^x)$

g^y

g^x

transcript

$T = (H(g^x), g^y, g^x)$

$(k_e, k'_e, h, h', s, cs) = H((g^{yx}, T, A, B))$

h

h'

SCIMP: (key) authentication

- There are no public keys, so key authentication is impossible
- In fact, the key exchange is **unauthenticated**
- To authenticate the (already established) session:
  - s is the "short authentication string"
  - Alice and Bob **must** compare s out-of-band
- But it also means we have good deniability

SCIMP Symmetric Ratchet

- symmetric key ratchet: $k_{e,i+1} = H(k_{e,i})$
- Send index i alongside ciphertext
- For example, Bob
  - has $k_{e,1}$, gets $i = 1$ (in order)
  - decrypts with $k_{e,1}$
  - computes $k_{e,2} = H(k_{e,1})$
  - deletes $k_{e,1}$
  - gets $i = 4$
  - computes $k_{e,5} = H(H(H(k_{e,2})))$
  - decrypts with $k_{e,4}$, then deletes $k_{e,4}$
  - stores $k_{e,2}$, $k_{e,3}$, and $k_{e,5}$
  - gets $i = 3$
  - decrypts with $k_{e,3}$
  - deletes $k_{e,3}$
  - $k_{e,2}$ threatens forward secrecy of msgs ≥ 2

SCIMP Symmetric Ratchet

$$k_{e,1} \rightarrow \boxed{H} \rightarrow k_{e,2} \rightarrow \boxed{H} \rightarrow k_{e,3} \rightarrow \boxed{H} \rightarrow k_{e,4} \rightarrow \boxed{H} \rightarrow k_{e,5}$$

- symmetric key ratchet: $k_{e,i+1} = H(k_{e,i})$
- Send index i alongside ciphertext
- For example, Bob
  - has $k_{e,1}$, gets $i = 1$ (in order)
  - decrypts with $k_{e,1}$
  - computes $k_{e,2} = H(k_{e,1})$
  - deletes $k_{e,1}$
  - gets $i = 4$ (messages skipped)
  - computes $k_{e,5} = H(H(H(k_{e,2})))$
  - decrypts with $k_{e,4}$, then deletes $k_{e,4}$
  - stores $k_{e,2}$, $k_{e,3}$, and $k_{e,5}$
  - gets $i = 3$ (out-of-order message)
  - decrypts with $k_{e,3}$
  - deletes $k_{e,3}$
  - $k_{e,2}$ threatens forward secrecy of msgs ≥ 2

SCIMP Symmetric Ratchet

$$k_{e,1} \xrightarrow{H} k_{e,2} \xrightarrow{H} k_{e,3} \xrightarrow{H} k_{e,4} \xrightarrow{H} k_{e,5}$$

- symmetric key ratchet: $k_{e,i+1} = H(k_{e,i})$
- Send index i alongside ciphertext
- For example, Bob
  - has $k_{e,1}$, gets $i = 1$ (in order)
  - decrypts with $k_{e,1}$
  - computes $k_{e,2} = H(k_{e,1})$
  - deletes $k_{e,1}$
  - gets $i = 4$ (messages skipped)
  - computes $k_{e,5} = H(H(H(k_{e,2})))$
  - decrypts with $k_{e,4}$, then deletes $k_{e,4}$
  - stores $k_{e,2}$, $k_{e,3}$, and $k_{e,5}$
  - gets $i = 3$ (out-of-order message)
  - decrypts with $k_{e,3}$
  - deletes $k_{e,3}$
  - $k_{e,2}$ threatens forward secrecy of msgs ≥ 2

SCIMP Symmetric Ratchet

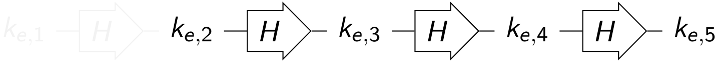
$k_{e,1} \rightarrow \boxed{H} \rightarrow k_{e,2} \rightarrow \boxed{H} \rightarrow k_{e,3} \rightarrow \boxed{H} \rightarrow k_{e,4} \rightarrow \boxed{H} \rightarrow k_{e,5}$

- symmetric key ratchet: $k_{e,i+1} = H(k_{e,i})$
- Send index i alongside ciphertext
- For example, Bob
  - has $k_{e,1}$, gets $i = 1$ (in order)
  - decrypts with $k_{e,1}$
  - computes $k_{e,2} = H(k_{e,1})$
  - deletes $k_{e,1}$
  - gets $i = 4$ (messages skipped)
  - computes $k_{e,5} = H(H(H(k_{e,2})))$
  - decrypts with $k_{e,4}$, then deletes $k_{e,4}$
  - stores $k_{e,2}$, $k_{e,3}$, and $k_{e,5}$
  - gets $i = 3$ (out-of-order message)
  - decrypts with $k_{e,3}$
  - deletes $k_{e,3}$
  - $k_{e,2}$ threatens forward secrecy of msgs ≥ 2

SCIMP Symmetric Ratchet

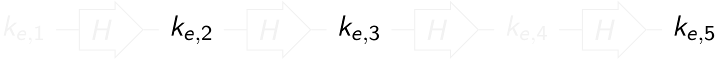
- symmetric key ratchet: $k_{e,i+1} = H(k_{e,i})$
- Send index i alongside ciphertext
- For example, Bob
    - has $k_{e,1}$, gets $i = 1$ (in order)
    - decrypts with $k_{e,1}$
    - computes $k_{e,2} = H(k_{e,1})$
    - deletes $k_{e,1}$
    - gets $i = 4$ (messages skipped)
    - computes $k_{e,5} = H(H(H(k_{e,2})))$
    - decrypts with $k_{e,4}$, then deletes $k_{e,4}$
    - stores $k_{e,2}$, $k_{e,3}$, and $k_{e,5}$
    - gets $i = 3$ (out-of-order message)
    - decrypts with $k_{e,3}$
    - deletes $k_{e,3}$
    - $k_{e,2}$ threatens forward secrecy of msgs ≥ 2

SCIMP Symmetric Ratchet



- symmetric key ratchet: $k_{e,i+1} = H(k_{e,i})$
- Send index i alongside ciphertext
- For example, Bob
  - has $k_{e,1}$, gets $i = 1$ (in order)
  - decrypts with $k_{e,1}$
  - computes $k_{e,2} = H(k_{e,1})$
  - deletes $k_{e,1}$
  - gets $i = 4$ (messages skipped)
  - computes $k_{e,5} = H(H(H(k_{e,2})))$
  - decrypts with $k_{e,4}$, then deletes $k_{e,4}$
  - stores $k_{e,2}$, $k_{e,3}$, and $k_{e,5}$
  - gets $i = 3$ (out-of-order message)
  - decrypts with $k_{e,3}$
  - deletes $k_{e,3}$
  - $k_{e,2}$ threatens forward secrecy of msgs ≥ 2

SCIMP Symmetric Ratchet



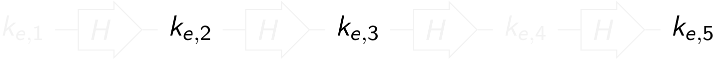
- symmetric key ratchet: $k_{e,i+1} = H(k_{e,i})$
- Send index i alongside ciphertext
- For example, Bob
  - has $k_{e,1}$, gets $i = 1$ (in order)
  - decrypts with $k_{e,1}$
  - computes $k_{e,2} = H(k_{e,1})$
  - deletes $k_{e,1}$
  - gets $i = 4$ (messages skipped)
  - computes $k_{e,5} = H(H(H(k_{e,2})))$
  - decrypts with $k_{e,4}$, then deletes $k_{e,4}$
  - stores $k_{e,2}$, $k_{e,3}$, and $k_{e,5}$
  - gets $i = 3$ (out-of-order message)
  - decrypts with $k_{e,3}$
  - deletes $k_{e,3}$
  - $k_{e,2}$ threatens forward secrecy of msgs ≥ 2

SCIMP Symmetric Ratchet

$$k_{e,1} \xrightarrow{H} k_{e,2} \xrightarrow{H} k_{e,3} \xrightarrow{H} k_{e,4} \xrightarrow{H} k_{e,5}$$

- symmetric key ratchet: $k_{e,i+1} = H(k_{e,i})$
- Send index i alongside ciphertext
- For example, Bob
  - has $k_{e,1}$, gets $i = 1$ (in order)
  - decrypts with $k_{e,1}$
  - computes $k_{e,2} = H(k_{e,1})$
  - deletes $k_{e,1}$
  - gets $i = 4$ (messages skipped)
  - computes $k_{e,5} = H(H(H(k_{e,2})))$
  - decrypts with $k_{e,4}$, then deletes $k_{e,4}$
  - stores $k_{e,2}$, $k_{e,3}$, and $k_{e,5}$
  - gets $i = 3$ (out-of-order message)
  - decrypts with $k_{e,3}$
  - deletes $k_{e,3}$
  - $k_{e,2}$ threatens forward secrecy of msgs ≥ 2

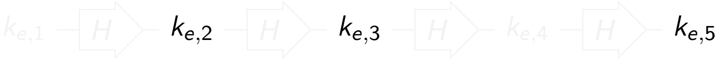
SCIMP Symmetric Ratchet



$k_{e,1} \xrightarrow{H} k_{e,2} \xrightarrow{H} k_{e,3} \xrightarrow{H} k_{e,4} \xrightarrow{H} k_{e,5}$

- symmetric key ratchet: $k_{e,i+1} = H(k_{e,i})$
- Send index i alongside ciphertext
- For example, Bob
  - has $k_{e,1}$, gets $i = 1$ (in order)
  - decrypts with $k_{e,1}$
  - computes $k_{e,2} = H(k_{e,1})$
  - deletes $k_{e,1}$
  - gets $i = 4$ (messages skipped)
  - computes $k_{e,5} = H(H(H(k_{e,2})))$
  - decrypts with $k_{e,4}$, then deletes $k_{e,4}$
  - stores $k_{e,2}$, $k_{e,3}$, and $k_{e,5}$
  - gets $i = 3$ (out-of-order message)
  - decrypts with $k_{e,3}$
  - deletes $k_{e,3}$
  - $k_{e,2}$ threatens forward secrecy of msgs ≥ 2

SCIMP Symmetric Ratchet

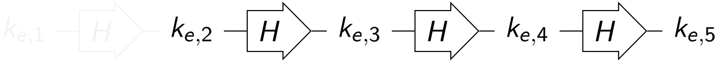


- symmetric key ratchet: $k_{e,i+1} = H(k_{e,i})$
- Send index i alongside ciphertext
- For example, Bob
  - has $k_{e,1}$, gets $i = 1$ (in order)
  - decrypts with $k_{e,1}$
  - computes $k_{e,2} = H(k_{e,1})$
  - deletes $k_{e,1}$
  - gets $i = 4$ (messages skipped)
  - computes $k_{e,5} = H(H(H(k_{e,2})))$
  - decrypts with $k_{e,4}$, then deletes $k_{e,4}$
  - stores $k_{e,2}$, $k_{e,3}$, and $k_{e,5}$
  - gets $i = 3$ (out-of-order message)
  - decrypts with $k_{e,3}$
  - deletes $k_{e,3}$
  - $k_{e,2}$ threatens forward secrecy of msgs ≥ 2

SCIMP Symmetric Ratchet

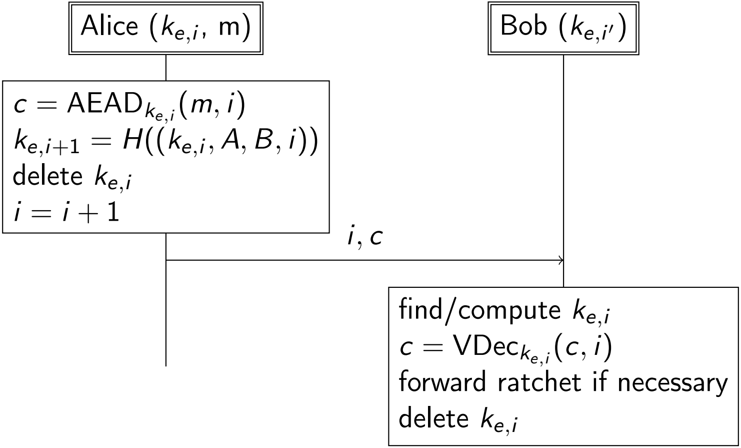
- symmetric key ratchet: $k_{e,i+1} = H(k_{e,i})$
- Send index i alongside ciphertext
- For example, Bob
  - has $k_{e,1}$, gets $i = 1$ (in order)
  - decrypts with $k_{e,1}$
  - computes $k_{e,2} = H(k_{e,1})$
  - deletes $k_{e,1}$
  - gets $i = 4$ (messages skipped)
  - computes $k_{e,5} = H(H(H(k_{e,2})))$
  - decrypts with $k_{e,4}$, then deletes $k_{e,4}$
  - stores $k_{e,2}$, $k_{e,3}$, and $k_{e,5}$
  - gets $i = 3$ (out-of-order message)
  - decrypts with $k_{e,3}$
  - deletes $k_{e,3}$
  - $k_{e,2}$ threatens forward secrecy of msgs ≥ 2

SCIMP Symmetric Ratchet



- symmetric key ratchet: $k_{e,i+1} = H(k_{e,i})$
- Send index i alongside ciphertext
- For example, Bob
  - has $k_{e,1}$, gets $i = 1$ (in order)
  - decrypts with $k_{e,1}$
  - computes $k_{e,2} = H(k_{e,1})$
  - deletes $k_{e,1}$
  - gets $i = 4$ (messages skipped)
  - computes $k_{e,5} = H(H(H(k_{e,2})))$
  - decrypts with $k_{e,4}$, then deletes $k_{e,4}$
  - stores $k_{e,2}$, $k_{e,3}$, and $k_{e,5}$
  - gets $i = 3$ (out-of-order message)
  - decrypts with $k_{e,3}$
  - deletes $k_{e,3}$
  - $k_{e,2}$ threatens forward secrecy of msgs ≥ 2

SCIMP: Sending Data Messages



Alice ($k_{e,i}$, m)

Bob ($k_{e,i'}$)

$c = \text{AEAD}_{k_{e,i}}(m, i)$
$k_{e,i+1} = H((k_{e,i}, A, B, i))$
delete $k_{e,i}$
$i = i + 1$

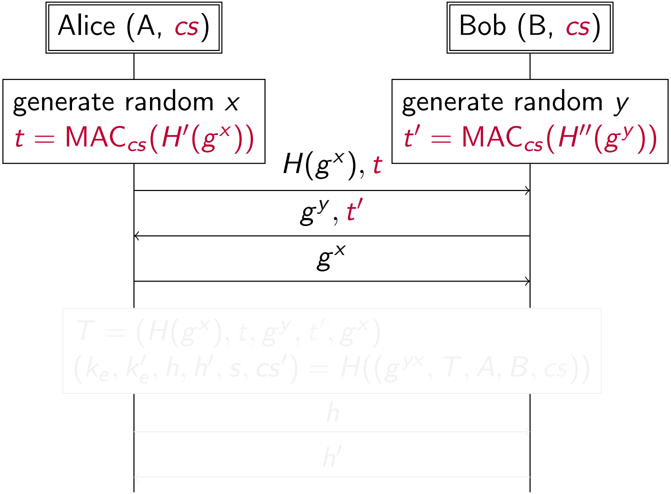
i, c

find/compute $k_{e,i}$
$c = \text{VDec}_{k_{e,i}}(c, i)$
forward ratchet if necessary
delete $k_{e,i}$

SCIMP

- Forward secrecy
  - We forward the ratchet on each *sent* message
  - But stealing old keys also leaks newer keys
  - If a key is to old ($>$ 32 hashes old) it is removed

SCIMP: Rekeying



Alice (A, cs)

Bob (B, cs)

generate random x

$t = \mathrm{MAC}_{cs}(H'(g^x))$

generate random y

$t' = \mathrm{MAC}_{cs}(H''(g^y))$

$H(g^x), t$

g^y, t'

g^x

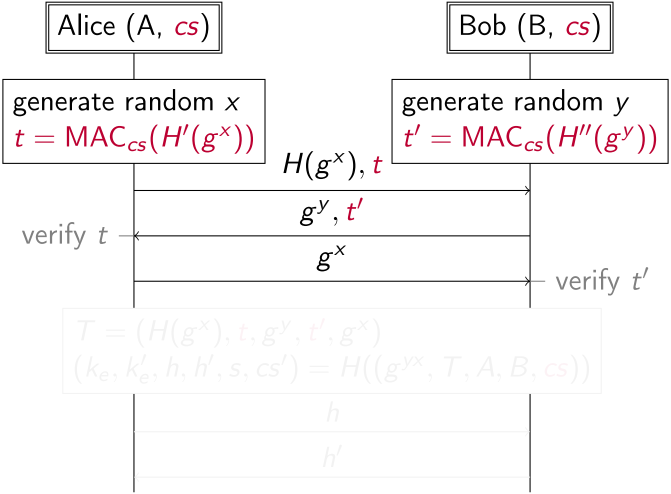
$T = (H(g^x), t, g^y, t', g^x)$

$(k_e, k_e', h, h', s, cs') = H((g^{yx}, T, A, B, cs))$

h

h'

SCIMP: Rekeying



Alice (A, cs)

Bob (B, cs)

generate random x
$t = \mathrm{MAC}_{cs}(H'(g^x))$

generate random y
$t' = \mathrm{MAC}_{cs}(H''(g^y))$

$H(g^x), t$

g^y, t'

verify t

g^x

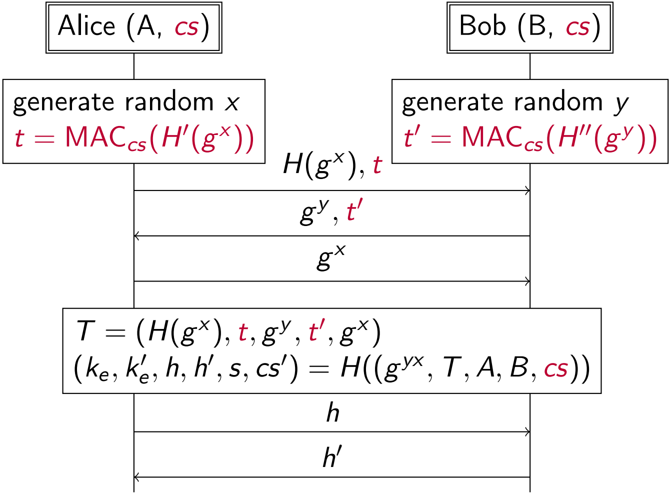
verify t'

$T = (H(g^x), t, g^y, t', g^x)$
$(k_e, k_e', h, h', s, cs') = H((g^{yx}, T, A, B, cs))$

h

h'

SCIMP: Rekeying



Alice (A, cs)

Bob (B, cs)

generate random x

$t = \mathrm{MAC}_{cs}(H'(g^x))$

generate random y

$t' = \mathrm{MAC}_{cs}(H''(g^y))$

$H(g^x), t$

g^y, t'

g^x

$T = (H(g^x), t, g^y, t', g^x)$

$(k_e, k'_e, h, h', s, cs') = H((g^{yx}, T, A, B, cs))$

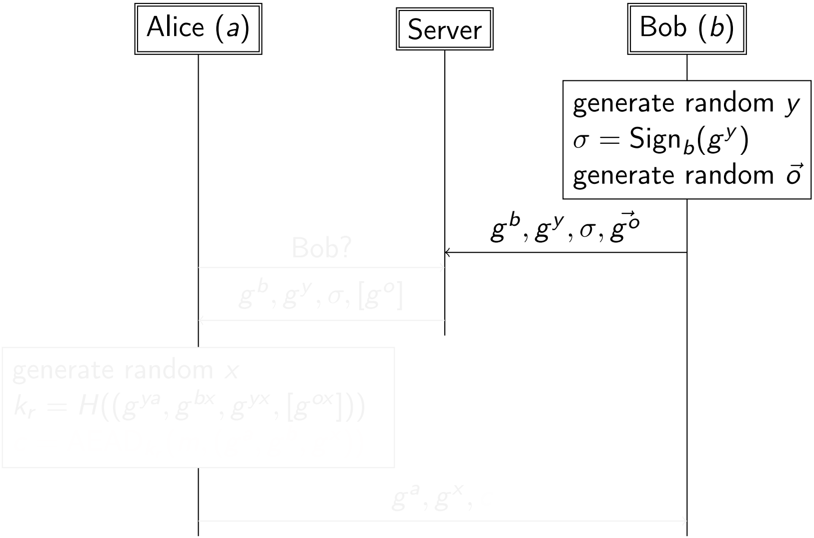
h

h'

SCIMP

- Future secrecy
  - rekeying mixes in new ephemeral keys
- Unspecified when rekeying should happen
- Store oldest unused receive key
  - in case out-of-order messages arrive
  - compromise between usability and forward secrecy
- On invalid t: all state is deleted
  - no longer authenticated!
  - Mallory can easily desynchronize

Signal: X3DH



Alice (a)

Server

Bob (b)

generate random y
$\sigma = \text{Sign}_b(g^y)$
generate random $\vec{o}$

$g^b, g^y, \sigma, g^{\vec{o}}$

Bob?

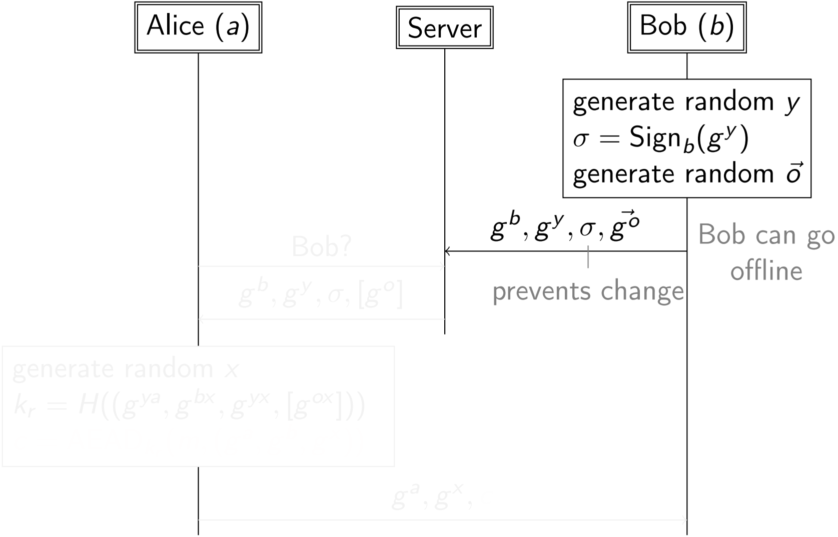
$g^b, g^y, \sigma, [g^o]$

generate random x
$k_r = H((g^{ya}, g^{bx}, g^{yx}, [g^{ox}]))$
$c = \text{AEAD}_{k_r}(m, (g^a, g^b, g^x))$

g^a, g^x, c

Signal: X3DH

b is a DH *and* a sign key
generally **not** recommended



Alice (a)

Server

Bob (b)

generate random y
$\sigma = \mathrm{Sign}_b(g^y)$
generate random $\vec{o}$

$g^b, g^y, \sigma, g^{\vec{o}}$

prevents change

Bob can go offline

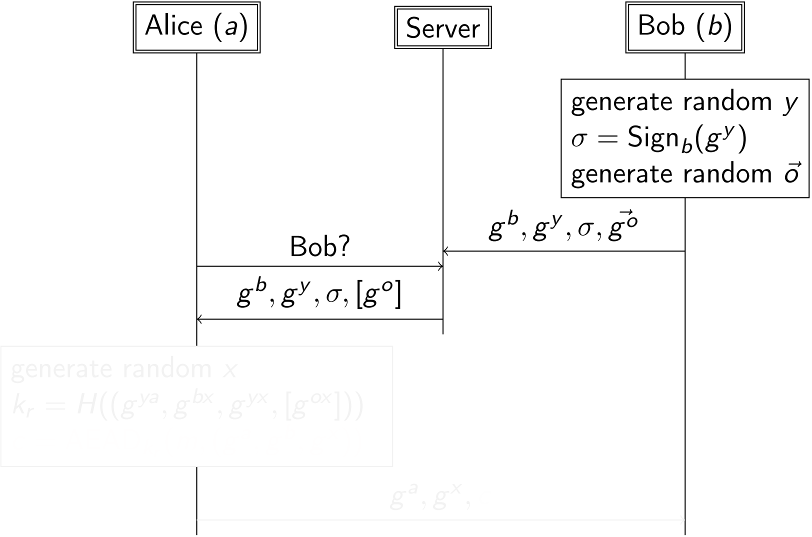
Bob?

$g^b, g^y, \sigma, [g^o]$

generate random x
$k_r = H((g^{ya}, g^{bx}, g^{yx}, [g^{ox}]))$
$c = \mathrm{AEAD}_{k_r}(m, (g^a, g^b, g^x))$

g^a, g^x, c

Signal: X3DH



Alice (a)

Server

Bob (b)

generate random y
$\sigma = \text{Sign}_b(g^y)$
generate random $\vec{o}$

$g^b, g^y, \sigma, g^{\vec{o}}$

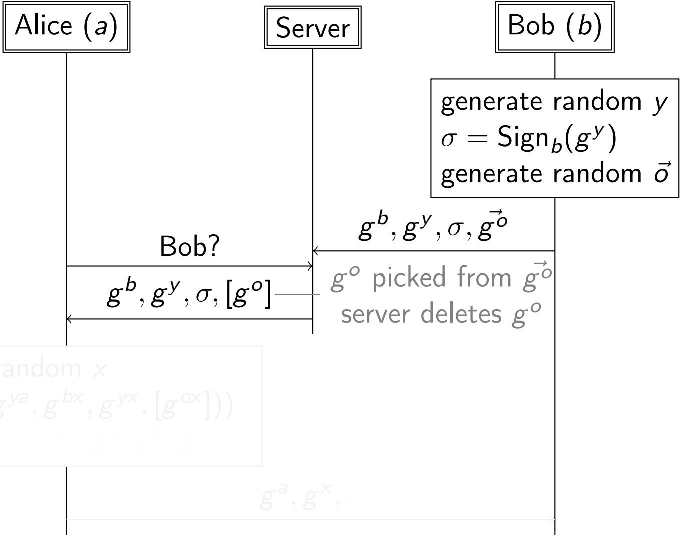
Bob?

$g^b, g^y, \sigma, [g^o]$

generate random x
$k_r = H((g^{ya}, g^{bx}, g^{yx}, [g^{ox}]))$
$c = \text{AEAD}_{k_r}(m, (g^a, g^b, g^x))$

g^a, g^x, c

Signal: X3DH



Alice (a)

Server

Bob (b)

generate random y
$\sigma = \mathsf{Sign}_b(g^y)$
generate random $\vec{o}$

$g^b, g^y, \sigma, g^{\vec{o}}$

Bob?

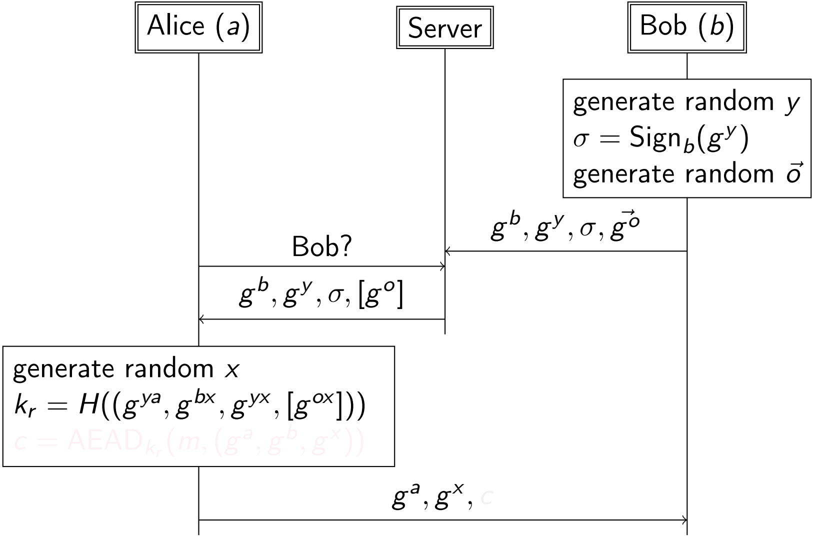
$g^b, g^y, \sigma, [g^o]$

g^o picked from $g^{\vec{o}}$
server deletes g^o

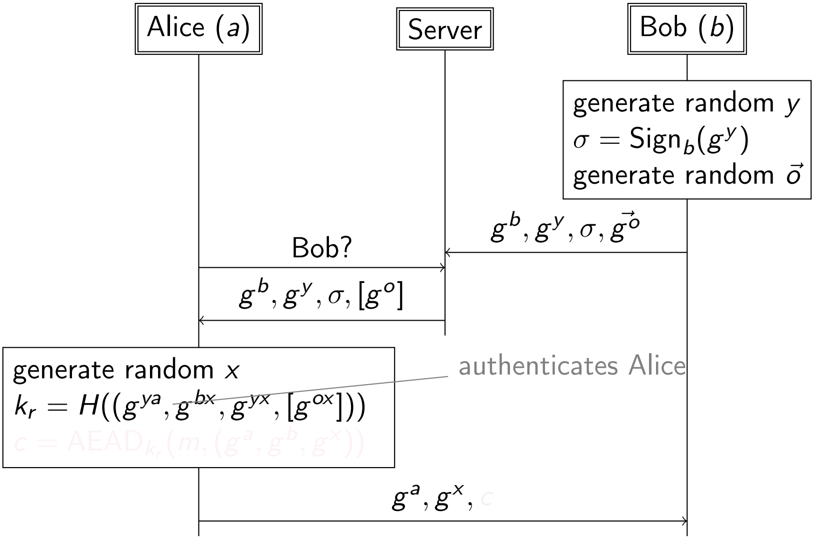
generate random x
$k_r = H((g^{ya}, g^{bx}, g^{yx}, [g^{ox}]))$
$c = \mathsf{AEAD}_{k_r}(m, (g^a, g^b, g^x))$

g^a, g^x, c

Signal: X3DH


Alice (a)
Server
Bob (b)
generate random y
$\sigma = \mathrm{Sign}_b(g^y)$
generate random $\vec{o}$
$g^b, g^y, \sigma, g^{\vec{o}}$
Bob?
$g^b, g^y, \sigma, [g^o]$
generate random x
$k_r = H((g^{ya}, g^{bx}, g^{yx}, [g^{ox}]))$
$c = \mathrm{AEAD}_{k_r}(m, (g^a, g^b, g^x))$
g^a, g^x, c

Signal: X3DH



Alice (a)

Server

Bob (b)

generate random y
$\sigma = \mathrm{Sign}_b(g^y)$
generate random $\vec{o}$

$g^b, g^y, \sigma, g^{\vec{o}}$

Bob?

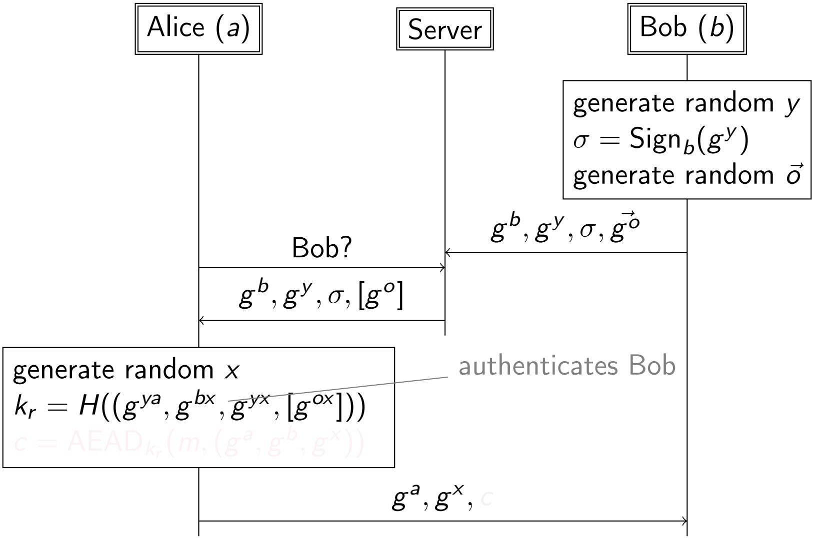
$g^b, g^y, \sigma, [g^o]$

generate random x
$k_r = H((g^{ya}, g^{bx}, g^{yx}, [g^{ox}]))$
$c = \mathrm{AEAD}_{k_r}(m, (g^a, g^b, g^x))$

authenticates Alice

g^a, g^x, c

Signal: X3DH



Alice (a)

Server

Bob (b)

generate random y
$\sigma = \mathrm{Sign}_b(g^y)$
generate random $\vec{o}$

$g^b, g^y, \sigma, g^{\vec{o}}$

Bob?

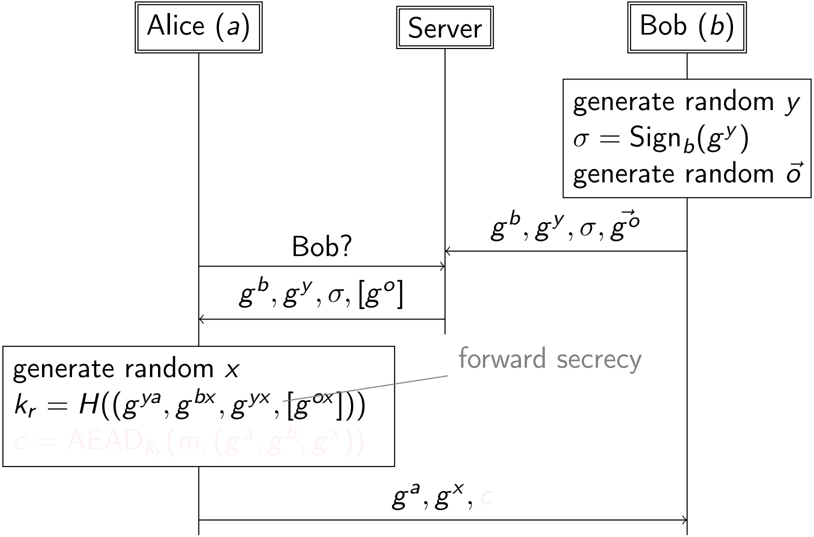
$g^b, g^y, \sigma, [g^o]$

generate random x
$k_r = H((g^{ya}, g^{bx}, g^{yx}, [g^{ox}]))$
$c = \mathrm{AEAD}_{k_r}(m, (g^a, g^b, g^x))$

authenticates Bob

g^a, g^x, c

Signal: X3DH



Alice (a)

Server

Bob (b)

generate random y
$\sigma = \text{Sign}_b(g^y)$
generate random $\vec{o}$

$g^b, g^y, \sigma, g^{\vec{o}}$

Bob?

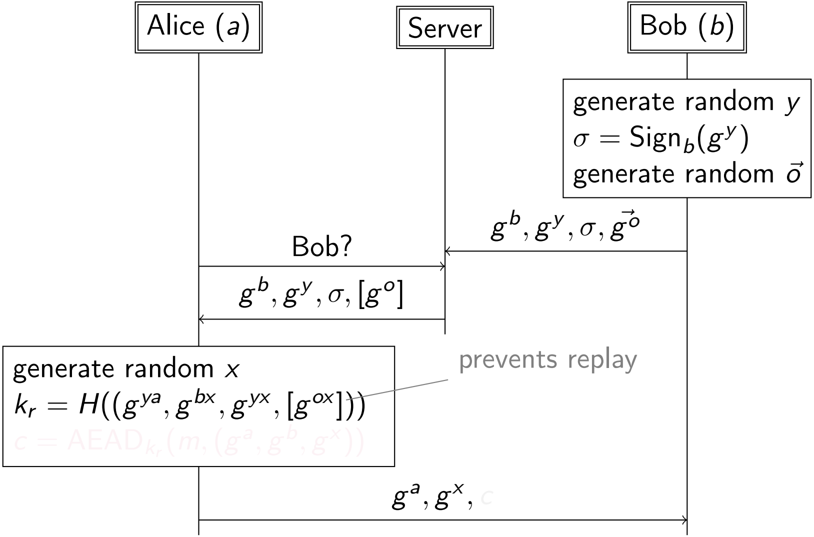
$g^b, g^y, \sigma, [g^o]$

generate random x
$k_r = H((g^{ya}, g^{bx}, g^{yx}, [g^{ox}]))$
$c = \text{AEAD}_{k_r}(m, (g^a, g^b, g^x))$

forward secrecy

g^a, g^x, c

Signal: X3DH



Alice (a)

Server

Bob (b)

generate random y
$\sigma = \mathrm{Sign}_b(g^y)$
generate random $\vec{o}$

$g^b, g^y, \sigma, g^{\vec{o}}$

Bob?

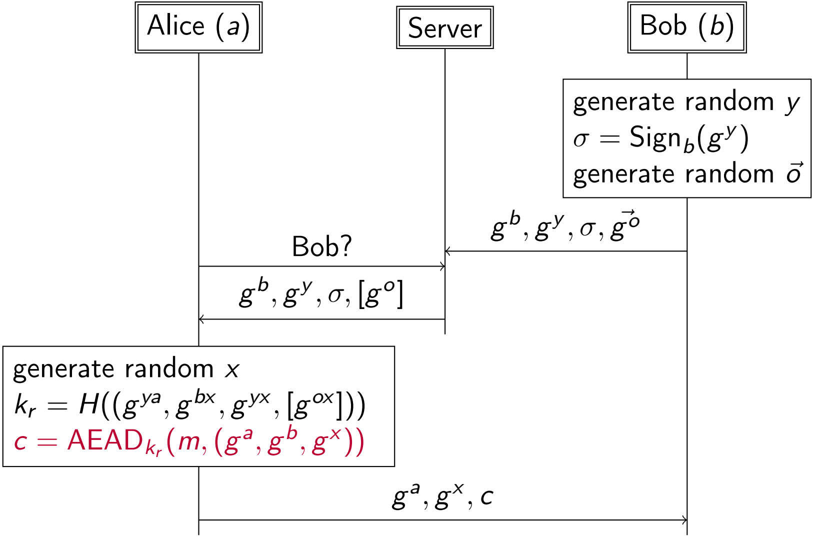
$g^b, g^y, \sigma, [g^o]$

generate random x
$k_r = H((g^{ya}, g^{bx}, g^{yx}, [g^{ox}]))$
$c = \mathrm{AEAD}_{k_r}(m, (g^a, g^b, g^x))$

prevents replay

g^a, g^x, c

Signal: X3DH



Alice (a)

Server

Bob (b)

generate random y

$\sigma = \mathrm{Sign}_b(g^y)$

generate random $\vec{o}$

$g^b, g^y, \sigma, g^{\vec{o}}$

Bob?

$g^b, g^y, \sigma, [g^o]$

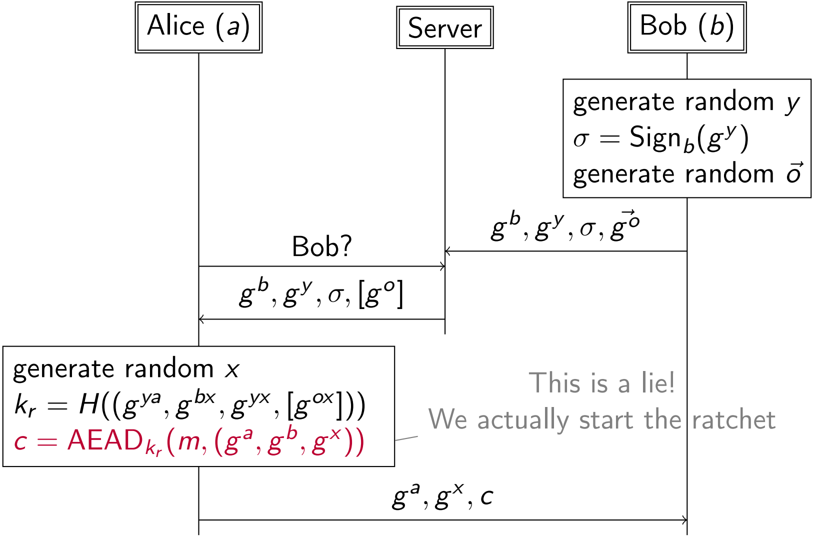
generate random x

$k_r = H((g^{ya}, g^{bx}, g^{yx}, [g^{ox}]))$

$c = \mathrm{AEAD}_{k_r}(m, (g^a, g^b, g^x))$

g^a, g^x, c

Signal: X3DH



Alice (a)

Server

Bob (b)

generate random y
$\sigma = \mathsf{Sign}_b(g^y)$
generate random $\vec{o}$

$g^b, g^y, \sigma, g^{\vec{o}}$

Bob?

$g^b, g^y, \sigma, [g^o]$

generate random x
$k_r = H((g^{ya}, g^{bx}, g^{yx}, [g^{ox}]))$
$c = \mathsf{AEAD}_{k_r}(m, (g^a, g^b, g^x))$

This is a lie!
We actually start the ratchet

g^a, g^x, c

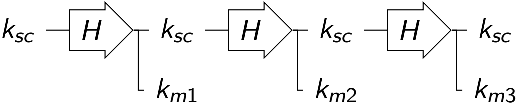
Signal: X3DH

- Asynchronous
  - Bob's handshake is independent of who wants to contact him
  - Bob does not have to be online
- All messages are delivered via the server
  - Mallory may control the server
- Alice encrypts a data message with her first message(!)

Signal: Double Ratchet

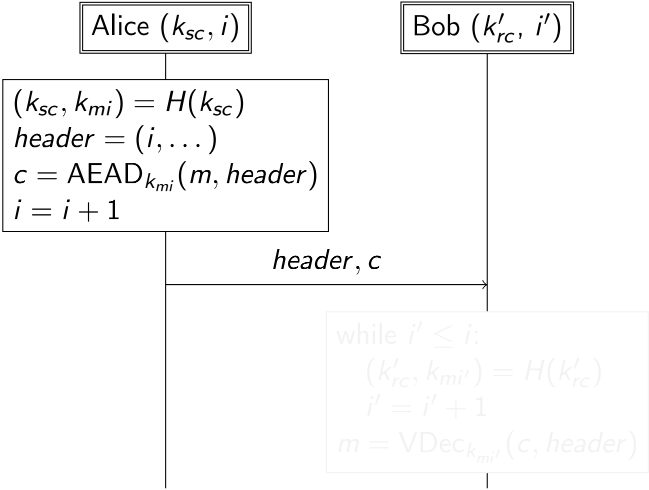
- Combine
  - the symmetrical ratchet (from SCIMP)
    - but split the chain key from the message key
  - the Diffie-Hellman ratchet (from OTR)
    - but require storage of fewer DH keys

Signal: Symmetric Ratchet



- Parties only ever store a single k_{sc}
- $(k_{sc}, k_{mi}) = H(k_{sc})$
- If Bob has $i = 0$ and receives $i = 3$
  - he iterates H three times
    - he overwrites k_{sc} with the new value
    - he stores k_{m1}, k_{m2}
    - he uses k_{m3} to decrypt
  - old k_m does not impact other keys
    - thus it does not threaten forward secrecy of other keys

Signal: Symmetric Ratchet



Alice (k_{sc}, i)

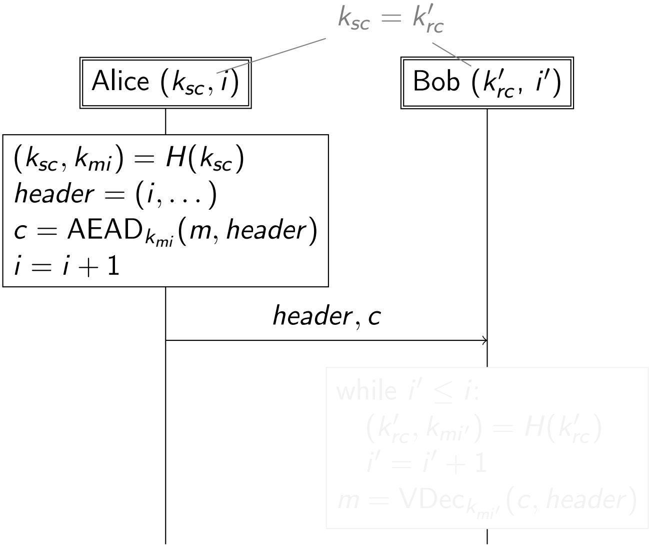
Bob $(k'_{rc},\ i')$

$(k_{sc}, k_{mi}) = H(k_{sc})$
$header = (i, \dots)$
$c = \mathrm{AEAD}_{k_{mi}}(m, header)$
$i = i + 1$

$header, c$

```
while $i' \leq i$:
    $(k'_{rc}, k_{mi'}) = H(k'_{rc})$
    $i' = i' + 1$
$m = \mathrm{VDec}_{k_{mi'}}(c, header)$
```

Signal: Symmetric Ratchet



$k_{sc} = k'_{rc}$

Alice (k_{sc}, i)

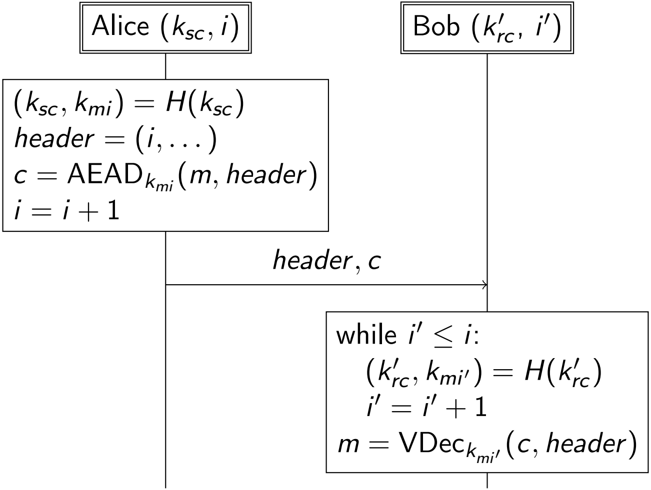
Bob $(k'_{rc},\ i')$

```
$(k_{sc}, k_{mi}) = H(k_{sc})$
$header = (i, \dots)$
$c = \mathrm{AEAD}_{k_{mi}}(m, header)$
$i = i + 1$
```

$header, c$

```
while $i' \leq i$:
    $(k'_{rc}, k_{mi'}) = H(k'_{rc})$
    $i' = i' + 1$
$m = \mathrm{VDec}_{k_{mi'}}(c, header)$
```

Signal: Symmetric Ratchet



Alice (k_{sc}, i)

Bob $(k'_{rc},\ i')$

$(k_{sc}, k_{mi}) = H(k_{sc})$

$header = (i, \dots)$

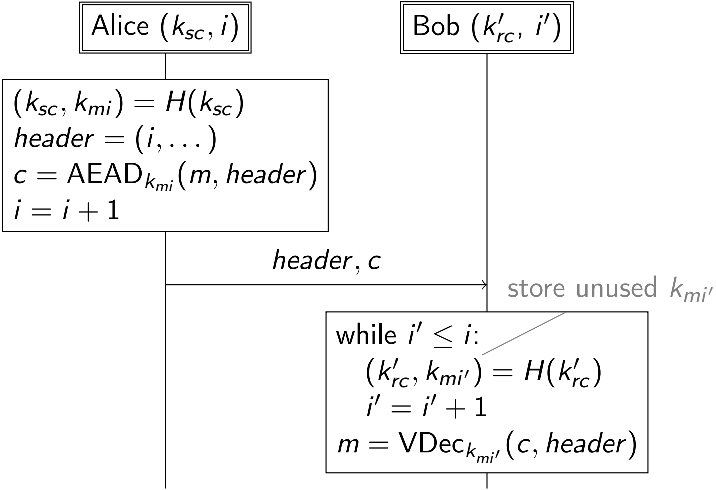
$c = \mathrm{AEAD}_{k_{mi}}(m, header)$

$i = i + 1$

$\xrightarrow{header, c}$

```
while $i' \leq i$:
    $(k'_{rc}, k_{mi'}) = H(k'_{rc})$
    $i' = i' + 1$
$m = \mathrm{VDec}_{k_{mi'}}(c, header)$
```

Signal: Symmetric Ratchet



Alice (k_{sc}, i)

Bob $(k'_{rc},\ i')$

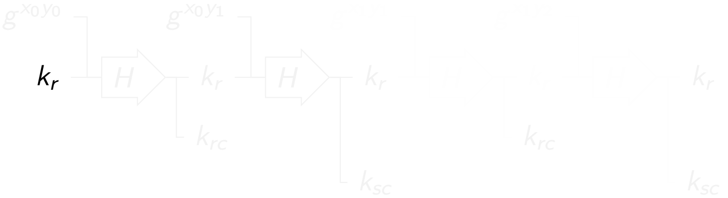
$(k_{sc}, k_{mi}) = H(k_{sc})$
$header = (i, \ldots)$
$c = \text{AEAD}_{k_{mi}}(m, header)$
$i = i + 1$

$header, c$

store unused $k_{mi'}$

while $i' \leq i$:
  $(k'_{rc}, k_{mi'}) = H(k'_{rc})$
  $i' = i' + 1$
$m = \text{VDec}_{k_{mi'}}(c, header)$

Signal: Diffie-Hellman Ratchet

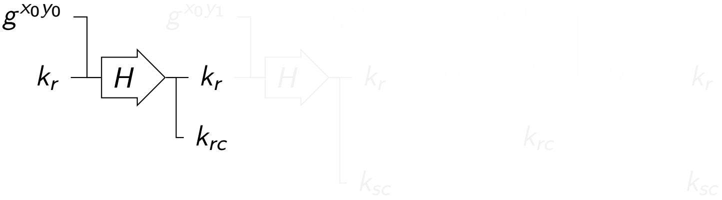


- Bob has k_r, y_0
- Alice sends a new g^{x_0}:
  - Bob computes $(g^{x_0})^{y_0}$
  - Bob ratchets: next receiving chain key k_{rc}
  - Bob generates new random y_1 and computes $g^{x_0 y_1}$
  - Bob ratchets: next sending chain key k_{sc}
  - Bob sends g^{y_1} to Alice
- When Alice sends g^{x_1}, Bob ratchets again (twice)
- Old values are deleted

Signal: Diffie-Hellman Ratchet

- Bob has k_r, y_0
- Alice sends a new g^{x_0}:
  - Bob computes $(g^{x_0})^{y_0}$
  - Bob ratchets: next receiving chain key k_{rc}
  - Bob generates new random y_1 and computes $g^{x_0 y_1}$
  - Bob ratchets: next sending chain key k_{sc}
  - Bob sends g^{y_1} to Alice
- When Alice sends g^{x_1}, Bob ratchets again (twice)
- Old values are deleted

Signal: Diffie-Hellman Ratchet

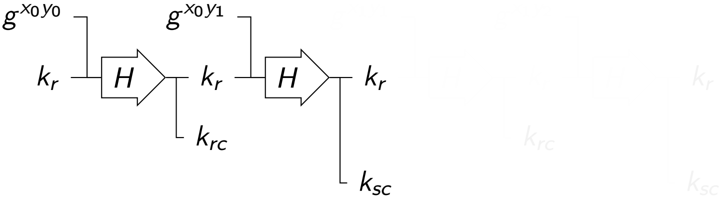


- Bob has k_r, y_0
- Alice sends a new g^{x_0}:
  - Bob computes $(g^{x_0})^{y_0}$
  - Bob ratchets: next receiving chain key k_{rc}
  - Bob generates new random y_1 and computes $g^{x_0 y_1}$
  - Bob ratchets: next sending chain key k_{sc}
  - Bob sends g^{y_1} to Alice
- When Alice sends g^{x_1}, Bob ratchets again (twice)
- Old values are deleted

Signal: Diffie-Hellman Ratchet

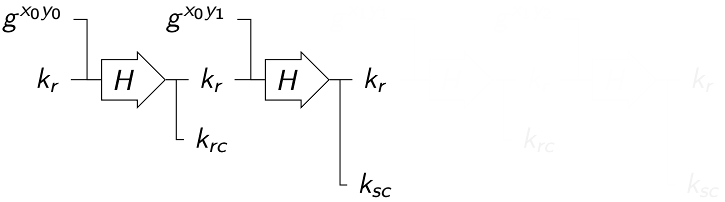
- Bob has k_r, y_0
- Alice sends a new g^{x_0}:
  - Bob computes $(g^{x_0})^{y_0}$
  - Bob ratchets: next receiving chain key k_{rc}
  - Bob generates new random y_1 and computes $g^{x_0 y_1}$
  - Bob ratchets: next sending chain key k_{sc}
  - Bob sends g^{y_1} to Alice
- When Alice sends g^{x_1}, Bob ratchets again (twice)
- Old values are deleted

Signal: Diffie-Hellman Ratchet



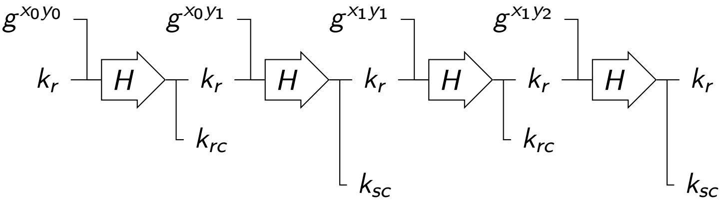
- Bob has k_r, y_0
- Alice sends a new g^{x_0}:
  - Bob computes $(g^{x_0})^{y_0}$
  - Bob ratchets: next receiving chain key k_{rc}
  - Bob generates new random y_1 and computes $g^{x_0 y_1}$
  - Bob ratchets: next sending chain key k_{sc}
  - Bob sends g^{y_1} to Alice
- When Alice sends g^{x_1}, Bob ratchets again (twice)
- Old values are deleted

Signal: Diffie-Hellman Ratchet



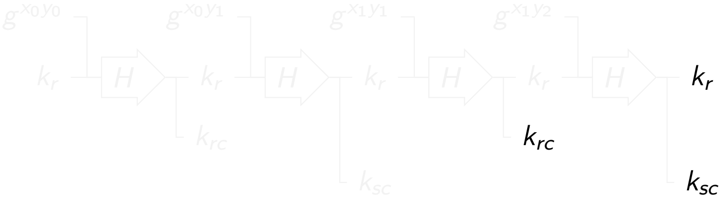
- Bob has k_r, y_0
- Alice sends a new g^{x_0}:
  - Bob computes $(g^{x_0})^{y_0}$
  - Bob ratchets: next receiving chain key k_{rc}
  - Bob generates new random y_1 and computes $g^{x_0 y_1}$
  - Bob ratchets: next sending chain key k_{sc}
  - Bob sends g^{y_1} to Alice
- When Alice sends g^{x_1}, Bob ratchets again (twice)
- Old values are deleted

Signal: Diffie-Hellman Ratchet



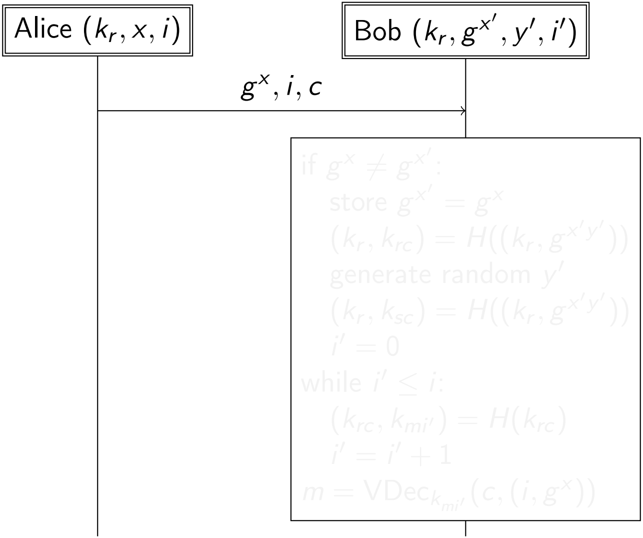
- Bob has k_r, y_0
- Alice sends a new g^{x_0}:
  - Bob computes $(g^{x_0})^{y_0}$
  - Bob ratchets: next receiving chain key k_{rc}
  - Bob generates new random y_1 and computes $g^{x_0 y_1}$
  - Bob ratchets: next sending chain key k_{sc}
  - Bob sends g^{y_1} to Alice
- When Alice sends g^{x_1}, Bob ratchets again (twice)
- Old values are deleted

Signal: Diffie-Hellman Ratchet



- Bob has k_r, y_0
- Alice sends a new g^{x_0}:
  - Bob computes $(g^{x_0})^{y_0}$
  - Bob ratchets: next receiving chain key k_{rc}
  - Bob generates new random y_1 and computes $g^{x_0y_1}$
  - Bob ratchets: next sending chain key k_{sc}
  - Bob sends g^{y_1} to Alice
- When Alice sends g^{x_1}, Bob ratchets again (twice)
- Old values are deleted

Signal: Diffie-Hellman ratchet



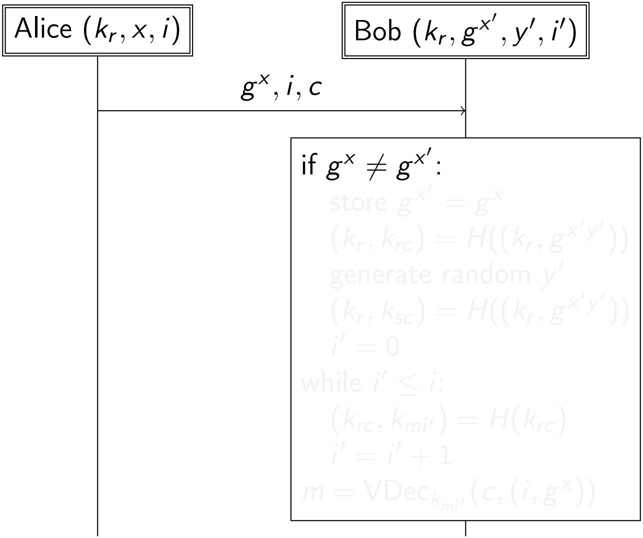
Alice (k_r, x, i)

Bob $(k_r, g^{x'}, y', i')$

g^x, i, c

```
if $g^x \neq g^{x'}$:
    store $g^{x'} = g^x$
    $(k_r, k_{rc}) = H((k_r, g^{x'y'}))$
    generate random $y'$
    $(k_r, k_{sc}) = H((k_r, g^{x'y'}))$
    $i' = 0$
while $i' \leq i$:
    $(k_{rc}, k_{mi'}) = H(k_{rc})$
    $i' = i' + 1$
$m = \mathrm{VDec}_{k_{mi'}}(c, (i, g^x))$
```

Signal: Diffie-Hellman ratchet



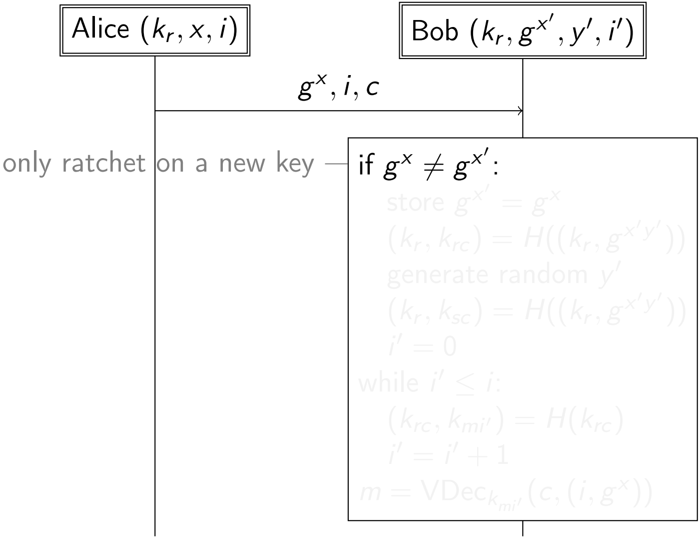
Alice (k_r, x, i)

Bob $(k_r, g^{x'}, y', i')$

g^x, i, c

```
if $g^x \neq g^{x'}$:
    store $g^{x'} = g^x$
    $(k_r, k_{rc}) = H((k_r, g^{x'y'}))$
    generate random $y'$
    $(k_r, k_{sc}) = H((k_r, g^{x'y'}))$
    $i' = 0$
while $i' \leq i$:
    $(k_{rc}, k_{mi'}) = H(k_{rc})$
    $i' = i' + 1$
$m = \mathrm{VDec}_{k_{mi'}}(c, (i, g^x))$
```

Signal: Diffie-Hellman ratchet



Alice (k_r, x, i)

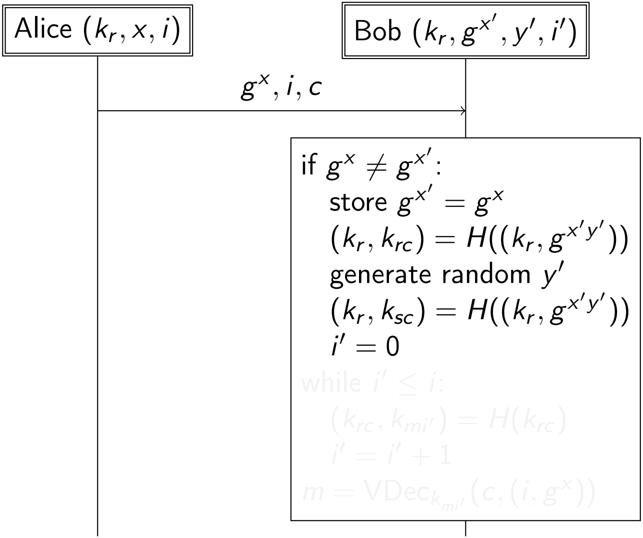
Bob $(k_r, g^{x'}, y', i')$

g^x, i, c

only ratchet on a new key

```
if $g^x \neq g^{x'}$:
    store $g^{x'} = g^x$
    $(k_r, k_{rc}) = H((k_r, g^{x'y'}))$
    generate random $y'$
    $(k_r, k_{sc}) = H((k_r, g^{x'y'}))$
    $i' = 0$
while $i' \leq i$:
    $(k_{rc}, k_{mi'}) = H(k_{rc})$
    $i' = i' + 1$
$m = \text{VDec}_{k_{mi'}}(c, (i, g^x))$
```

Signal: Diffie-Hellman ratchet



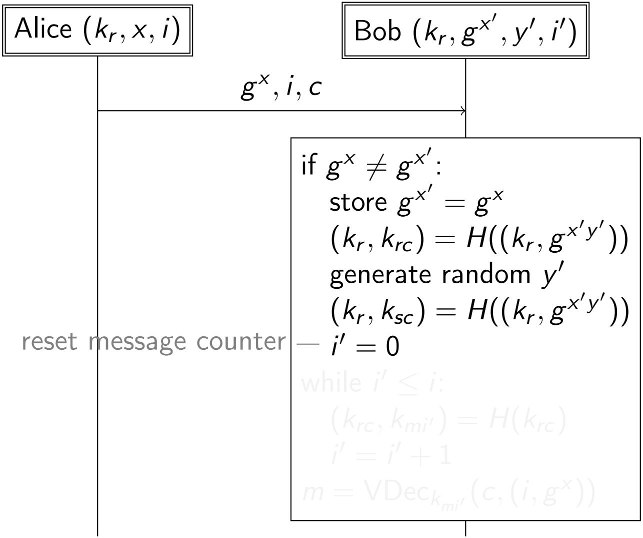
Alice (k_r, x, i)

Bob $(k_r, g^{x'}, y', i')$

g^x, i, c

```
if $g^x \neq g^{x'}$:
    store $g^{x'} = g^x$
    $(k_r, k_{rc}) = H((k_r, g^{x'y'}))$
    generate random $y'$
    $(k_r, k_{sc}) = H((k_r, g^{x'y'}))$
    $i' = 0$
while $i' \leq i$:
    $(k_{rc}, k_{mi'}) = H(k_{rc})$
    $i' = i' + 1$
$m = \mathrm{VDec}_{k_{mi'}}(c, (i, g^x))$
```

Signal: Diffie-Hellman ratchet



Alice (k_r, x, i)

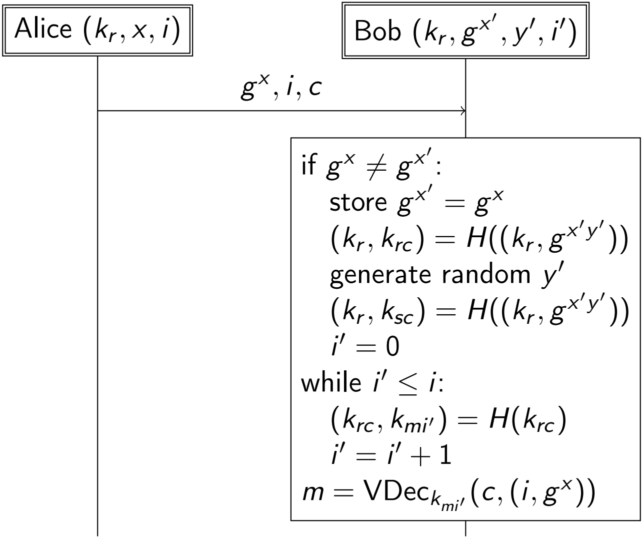
Bob $(k_r, g^{x'}, y', i')$

g^x, i, c

```
if $g^x \neq g^{x'}$:
    store $g^{x'} = g^x$
    $(k_r, k_{rc}) = H((k_r, g^{x'y'}))$
    generate random $y'$
    $(k_r, k_{sc}) = H((k_r, g^{x'y'}))$
    $i' = 0$
while $i' \leq i$:
    $(k_{rc}, k_{mi'}) = H(k_{rc})$
    $i' = i' + 1$
$m = \mathrm{VDec}_{k_{mi'}}(c, (i, g^x))$
```

reset message counter —

Signal: Diffie-Hellman ratchet



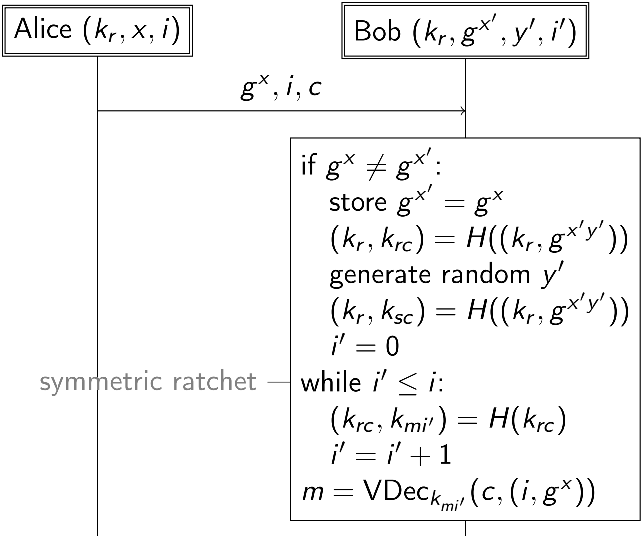
Alice (k_r, x, i)

Bob $(k_r, g^{x'}, y', i')$

g^x, i, c

```
if $g^x \neq g^{x'}$:
    store $g^{x'} = g^x$
    $(k_r, k_{rc}) = H((k_r, g^{x'y'}))$
    generate random $y'$
    $(k_r, k_{sc}) = H((k_r, g^{x'y'}))$
    $i' = 0$
while $i' \leq i$:
    $(k_{rc}, k_{mi'}) = H(k_{rc})$
    $i' = i' + 1$
$m = \mathrm{VDec}_{k_{mi'}}(c, (i, g^x))$
```

Signal: Diffie-Hellman ratchet



Alice (k_r, x, i)

Bob $(k_r, g^{x'}, y', i')$

g^x, i, c

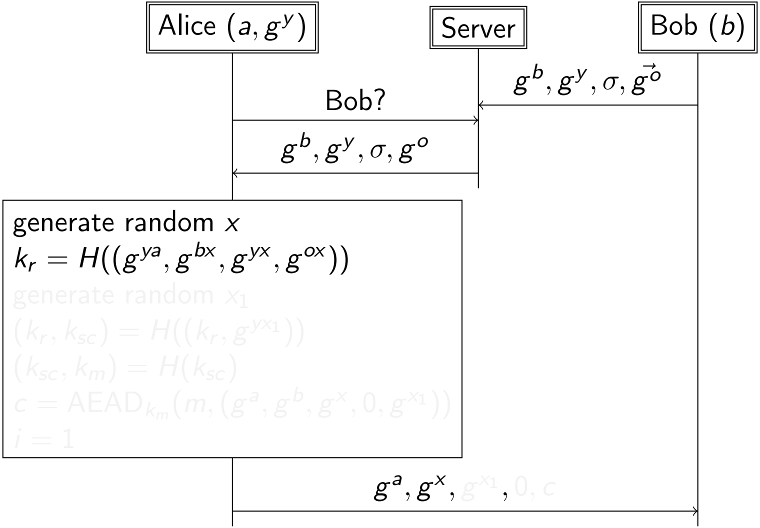
symmetric ratchet

```
if $g^x \neq g^{x'}$:
    store $g^{x'} = g^x$
    $(k_r, k_{rc}) = H((k_r, g^{x'y'}))$
    generate random $y'$
    $(k_r, k_{sc}) = H((k_r, g^{x'y'}))$
    $i' = 0$
while $i' \leq i$:
    $(k_{rc}, k_{mi'}) = H(k_{rc})$
    $i' = i' + 1$
$m = \mathrm{VDec}_{k_{mi'}}(c, (i, g^x))$
```

Signal: Diffie-Hellman ratchet

- Compare against OTR:
  - Bob uses g^x immediately
  - Bob verifies authenticity of g^x through associated data: $\mathsf{VDec}_{k_{mi}}(\cdot, (i, g^x, \dots))$
    - but k_{mi} is derived from g^x(!)
    - this turns out to be secure in this context, but this is not at all obvious (to me)
  - store only one g^x and one y per peer
- What if you missed a message before ratcheting?
  - header includes a value i_p: the total number of messages sent with the previous send chain key
  - compute all missed $k_{mi'}$ before starting the DH ratchet

Signal: X3DH + ratchet start



Alice (a, g^y)

Server

Bob (b)

$g^b, g^y, \sigma, g^{\vec{o}}$

Bob?

g^b, g^y, σ, g^o

generate random x

$k_r = H((g^{ya}, g^{bx}, g^{yx}, g^{ox}))$

generate random x_1

$(k_r, k_{sc}) = H((k_r, g^{yx_1}))$

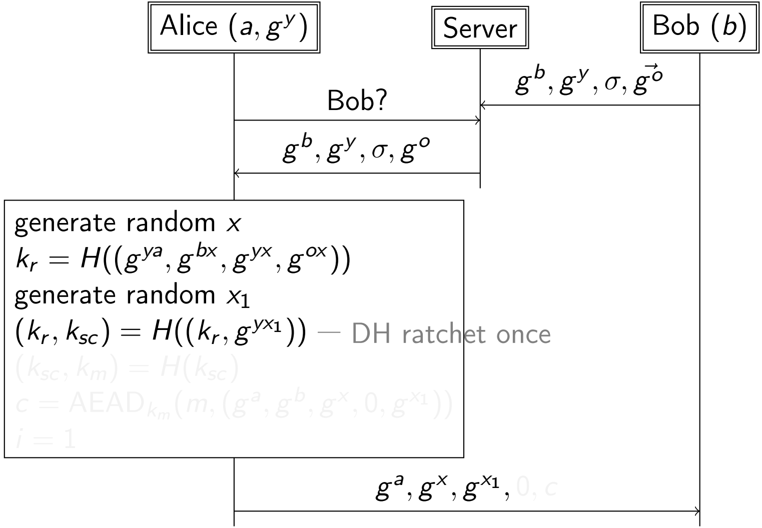
$(k_{sc}, k_m) = H(k_{sc})$

$c = \mathrm{AEAD}_{k_m}(m, (g^a, g^b, g^x, 0, g^{x_1}))$

$i = 1$

$g^a, g^x, g^{x_1}, 0, c$

Signal: X3DH + ratchet start



Alice (a, g^y)

Server

Bob (b)

$g^b, g^y, \sigma, g^{\vec{o}}$

Bob?

g^b, g^y, σ, g^o

generate random x

$k_r = H((g^{ya}, g^{bx}, g^{yx}, g^{ox}))$

generate random x_1

$(k_r, k_{sc}) = H((k_r, g^{yx_1}))$ — DH ratchet once

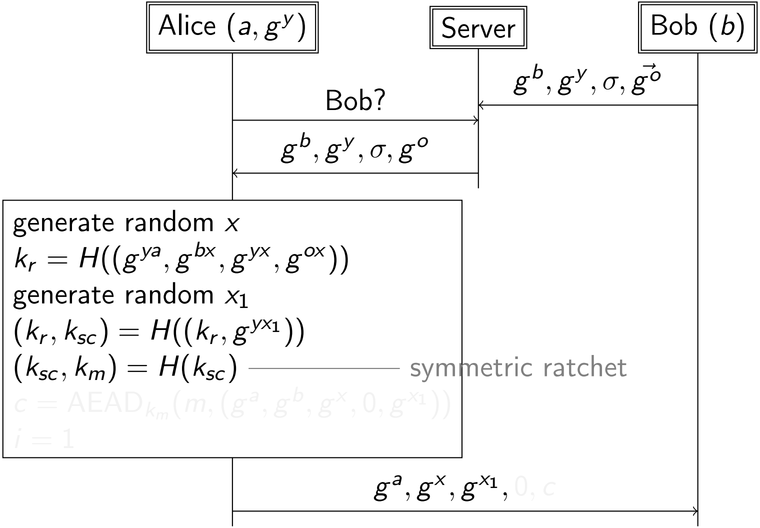
$(k_{sc}, k_m) = H(k_{sc})$

$c = \mathrm{AEAD}_{k_m}(m, (g^a, g^b, g^x, 0, g^{x_1}))$

$i = 1$

$g^a, g^x, g^{x_1}, 0, c$

Signal: X3DH + ratchet start



Alice (a, g^y)

Server

Bob (b)

$g^b, g^y, \sigma, g^{\vec{o}}$

Bob?

g^b, g^y, σ, g^o

generate random x

$k_r = H((g^{ya}, g^{bx}, g^{yx}, g^{ox}))$

generate random x_1

$(k_r, k_{sc}) = H((k_r, g^{yx_1}))$

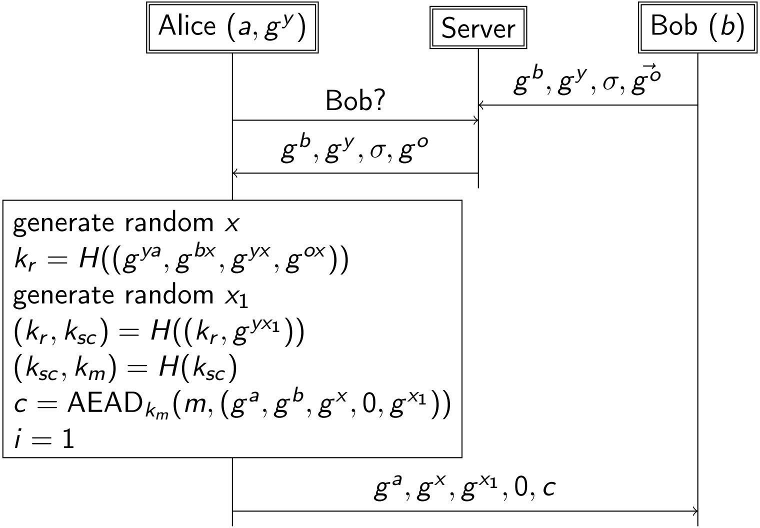
$(k_{sc}, k_m) = H(k_{sc})$ symmetric ratchet

$c = \mathrm{AEAD}_{k_m}(m, (g^a, g^b, g^x, 0, g^{x_1}))$

$i = 1$

$g^a, g^x, g^{x_1}, 0, c$

Signal: X3DH + ratchet start



Alice (a, g^y)

Server

Bob (b)

$g^b, g^y, \sigma, g^{\vec{o}}$

Bob?

g^b, g^y, σ, g^o

generate random x

$k_r = H((g^{ya}, g^{bx}, g^{yx}, g^{ox}))$

generate random x_1

$(k_r, k_{sc}) = H((k_r, g^{yx_1}))$

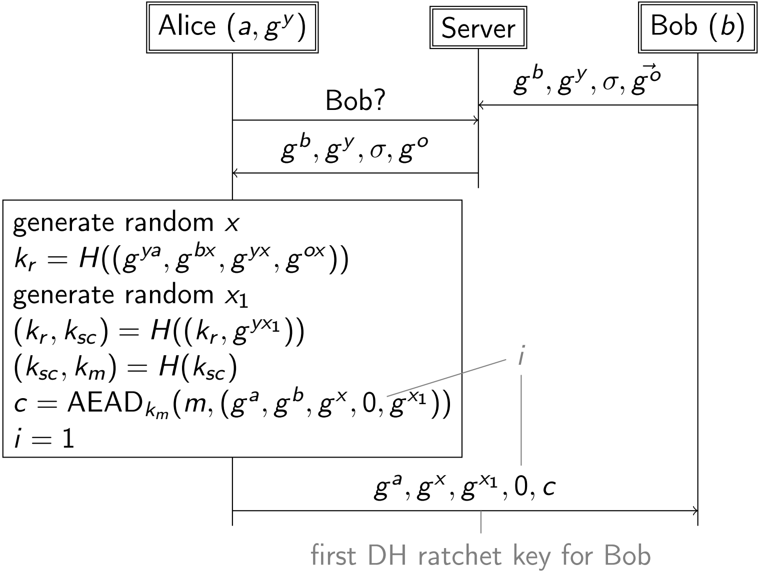
$(k_{sc}, k_m) = H(k_{sc})$

$c = \text{AEAD}_{k_m}(m, (g^a, g^b, g^x, 0, g^{x_1}))$

$i = 1$

$g^a, g^x, g^{x_1}, 0, c$

Signal: X3DH + ratchet start



Alice (a, g^y)

Server

Bob (b)

$g^b, g^y, \sigma, g^{\vec{o}}$

Bob?

g^b, g^y, σ, g^o

generate random x

$k_r = H((g^{ya}, g^{bx}, g^{yx}, g^{ox}))$

generate random x_1

$(k_r, k_{sc}) = H((k_r, g^{yx_1}))$

$(k_{sc}, k_m) = H(k_{sc})$

$c = \text{AEAD}_{k_m}(m, (g^a, g^b, g^x, 0, g^{x_1}))$

$i = 1$

i

$g^a, g^x, g^{x_1}, 0, c$

first DH ratchet key for Bob

Signal: key authentication

Safety numbers:

- these are hashes of the public key (+ some other values)
- users should compare these out-of-band

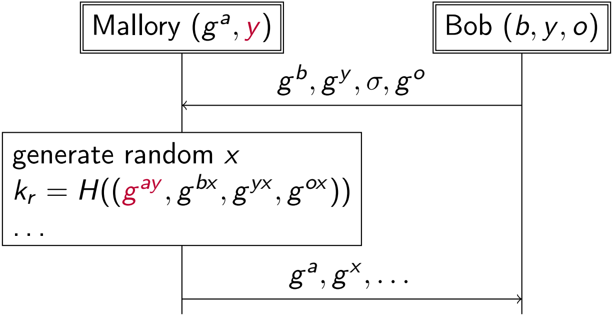
Signal Protocol

What security is not provided by this protocol?

- Mallory can block all messages
- Server may send the wrong g^b:
  - MitM, if Alice and Bob don't check the safety number
- If there's no g^o in original message (server ran out or is malicious)
  - Messages can be replayed to Bob
  - Reduced forward secrecy, until Bob refreshes g^y
- Key Compromise Impersonation
- Unknown Key Share

Signal: Key Compromise Impersonation

- Expected: If Mallory steals b, she can impersonate Bob to others
- KCI attack: If Mallory steals y (a key from Bob), she can impersonate others to Bob



Mallory (g^a, y)

Bob (b, y, o)

g^b, g^y, σ, g^o

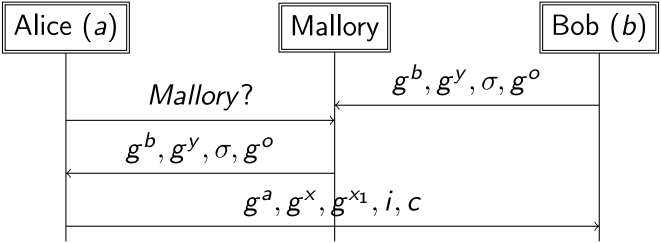
generate random x

$k_r = H((g^{ay}, g^{bx}, g^{yx}, g^{ox}))$

$\ldots$

$g^a, g^x, \ldots$

Signal: Unknown Key Share



- Alice thinks she is talking to Mallory
- Mallory substituted her own keys with those of Bob
- Alice is actually talking to Bob

Signal: multiple devices

Setup

- Desktop displays QR
    - address
    - ephemeral public key
- Phone scans QR, encrypts to device's ephemeral key
    - identity key pair
    - account info
    - linking token
- Desktop registers with server as new device

Sending messages

- Encrypt the message to each device of the user
- Encrypt the message to each other device of yourself

Phishing

- Mallory sends device link QR, disguised as group invite QR
- user scans and doesn't read the pop-up message
- Mallory can now read along and/or impersonate

Signal: multiple devices

Setup

- Desktop displays QR
    - address
    - ephemeral public key
- Phone scans QR, encrypts to device's ephemeral key
    - identity key pair
    - account info
    - linking token
- Desktop registers with server as new device

Sending messages

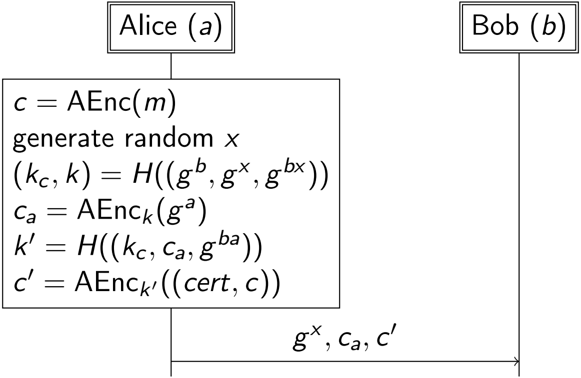
- Encrypt the message to each device of the user
- Encrypt the message to each other device of yourself

Phishing

- Mallory sends device link QR, disguised as group invite QR
- user scans and doesn't read the pop-up message
- Mallory can now read along and/or impersonate

Signal: sealed sender

Hide sender from metadata



Alice (a)

Bob (b)

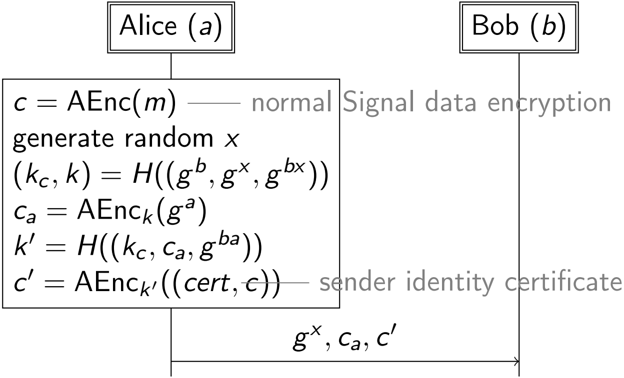
$c = \mathrm{AEnc}(m)$
generate random x
$(k_c, k) = H((g^b, g^x, g^{bx}))$
$c_a = \mathrm{AEnc}_k(g^a)$
$k' = H((k_c, c_a, g^{ba}))$
$c' = \mathrm{AEnc}_{k'}((cert, c))$

g^x, c_a, c'

- Delivered to server over one-way authenticated channel
- Cautious senders should use TOR/VPN to hide their ip address

Signal: sealed sender

Hide sender from metadata



Alice (a)

Bob (b)

$c = \text{AEnc}(m)$ — normal Signal data encryption

generate random x

$(k_c, k) = H((g^b, g^x, g^{bx}))$

$c_a = \text{AEnc}_k(g^a)$

$k' = H((k_c, c_a, g^{ba}))$

$c' = \text{AEnc}_{k'}((cert, c))$ — sender identity certificate

g^x, c_a, c'

- Delivered to server over one-way authenticated channel
- Cautious senders should use TOR/VPN to hide their ip address

Signal: attachments

- generate a random key k
- encrypt the file using k
- upload the encrypted file to the file server
- send k and address to recipient *over a pairwise Signal session*
- only need to re-encrypt k and address for other recipients

~~Signal~~ WhatsApp: groups

Each group member:

- generates a sending chain key k_{sc}
- generates an ephemeral signing key pair (sk, pk)
- sends k_{sc}, pk to each group member *over a pairwise Signal session*
- if anyone leaves the group: delete k_{sc} (and sk?)

For each message m

- ratchet forward: $(k_{sc}, k_m) = H(k_{sc})$
- encrypt: $c = \mathrm{Enc}_{k_m}(m)$
- authenticate: $\sigma = \mathrm{Sign}_{sk}(c)$
- send (c, σ) to the server
- server forward (c, σ) to all group members

References

- 2015, Unger et al. SoK: Secure Messaging https://doi.org/10.1109/SP.2015.22
- PGP: RFC 9580
- OTR: https://otr.cypherpunks.ca/Protocol-v3-4.0.0.html
- SCIMP: https://ia.cr/2016/703
- Signal: https://signal.org/docs/

Slides will be made available on my website zeroknowledge.me

Public Key Authenticated Encryption?

How not to do it:

- Sign-then-encrypt
  - $\sigma = \mathsf{Sign}_{sk_a}(m)$
  - $c = \mathsf{Enc}_{pk_b}((m, \sigma))$
  - Did Alice intend the message to be delivered to Bob?
- Encrypt-then-sign
  - $c = \mathsf{Enc}_{pk_b}(m)$
  - $\sigma = \mathsf{Sign}_{sk_a}(c)$
  - Did Alice generate the original ciphertext?
- PGP usability problems lead security problems
  - user responsible for key management
  - user can arbitrarily combine encryption and signing

How to do it:

- use `crypto_box` from NaCl
- use age `age-encryption.org`