The PQ Transition at Signal

Rolfe Schmidt
Signal Messenger
SPIQE - 24 Jun 2025

About Signal

```
> echo $STD_SGNL_RLVNT_SPIEL
```

Signal’s Mission and Vision

- Protect free expression and enable secure global communication through open source privacy technology.
- Make conversations with anyone in the world effortless, private, even joyful.
- Privacy isn’t an optional mode — it’s how Signal works. Every message, every call, every time.
- We are an independent 501c3 nonprofit and will never compromise the mission.

Signal’s Mission and Vision

Our mission is **NOT**:

- Research
- Deploying fancy cryptography
- Developing general purpose open source software libraries

But sometimes we need to do these things to get our work done.

also, they're fun

Signal's Mission ⇒ Usability is Key

"Effortless" + "anyone in the world" is hard.

- It constrains what we can deploy.
- It requires a large engineering effort.

But worth it.

Our focus on **usability** without compromising privacy is one key thing made Signal successful **from the start**.

Global usability is part of our mission.

PQ Transition ⊊ Ongoing Operations

To fulfill our mission we are **always scanning for threats** to our users' privacy and prioritizing our **limited resources** to address them.

Some threats are immediate, others distant.

Some threats are large, others small.

Some are easy to address, others are complex, and for some we don't have solutions.

And some happen to be threats from quantum computing.

Thinking About a Project

Impact	Development Effort and Risk
 Targeted	 Lower
 Widespread	 Moderate
 Anticipated	 High
	 Major cross-team deployment

We have to consider the potential impact of a threat vs the risk and effort of mitigating it.

Now let's see how we are applying these principles to our post-quantum transition.

Transitioning the Signal Protocol

A Case Study

- PQXDH (2023)
- Triple Ratchet (soon!)
- Full Hybrid Security (?)

The Signal Protocol (2013)

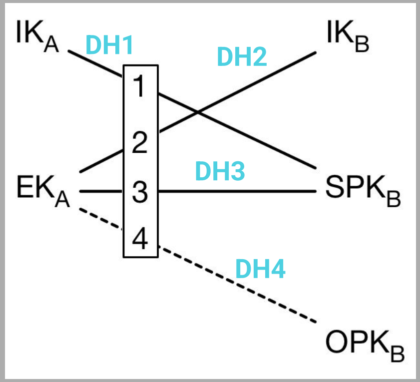
Two parts:

- X3DH handshake
- Double Ratchet for continuous key agreement

Important security guarantees:

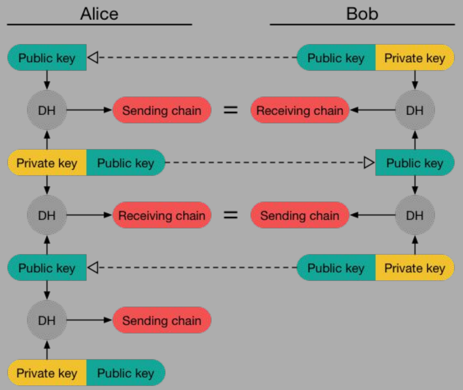
- Confidentiality
- Mutual authentication
- Post-compromise security
- Forward secrecy
- Deniability

X3DH



SK = KDF(DH1 || DH2 || DH3 || DH4)

Double Ratchet



The Signal Protocol (2013)

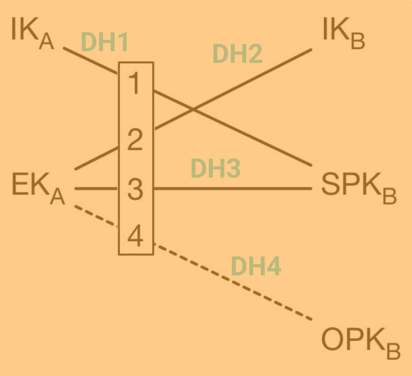
Two parts:

- X3DH handshake
- Double Ratchet for continuous key agreement

Important security guarantees:

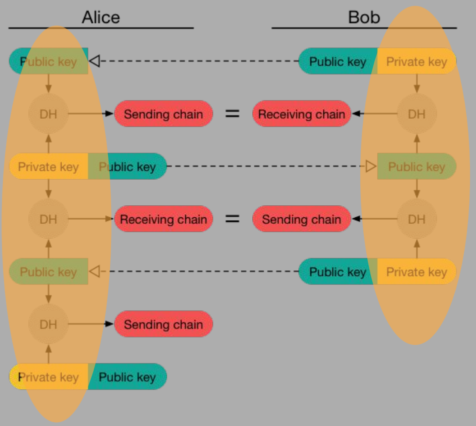
- Confidentiality
- Mutual authentication
- Post-compromise security
- Forward secrecy
- Deniability

X3DH



SK = KDF(DH1 || DH2 || DH3 || DH4)

Double Ratchet



These need post-quantum protection!

The Signal Protocol (2013)

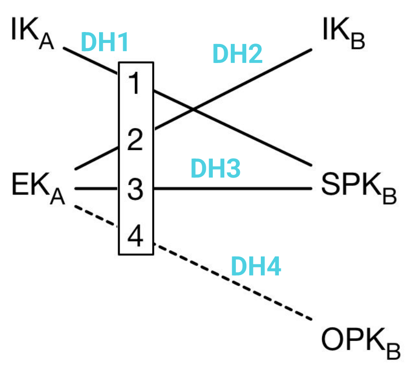
Two parts:

- X3DH handshake
- Double Ratchet for continuous key agreement

Important security guarantees:

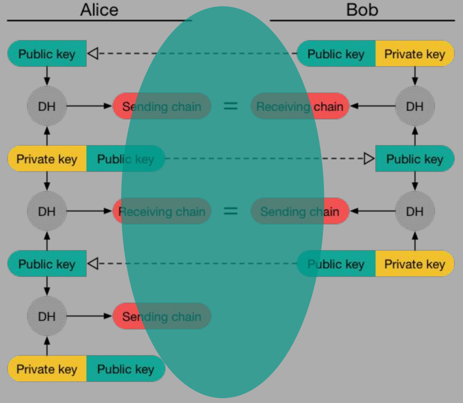
- Confidentiality
- Mutual authentication
- Post-compromise security
- Forward secrecy
- Deniability

X3DH



SK = KDF(DH1 || DH2 || DH3 || DH4)

Double Ratchet



Symmetric Key crypto is already quantum safe.

The PQXDH Handshake

K. Bhargavan, C. Jacomme, F. Kiefer and the Signal Team

X3DH and PQXDH: The Problem

Flashback Spring 2023

Problem: An attacker that can compute curve25519 logarithms could compute X3DH session secrets and Double Ratchet updates, learning all session secrets.

Scope: All user messages and media were at risk to a HNDL attack.

But it wasn't 2013: NIST post-quantum standardization was pretty far along.

X3DH →PQXDH: Adding Post-Quantum Security

Classic X3DH

Alex **Blake**

$IK_A \leftrightarrow SPK_B$
$EK_A \leftrightarrow IK_B$
$EK_A \leftrightarrow SPK_B$
(* $EK_A \leftrightarrow OPK_B$ *)

$SK = KDF(DH_1 || DH_2 || DH_3 || DH_4)$

100s of bytes
Handshake overhead

X3DH →PQXDH: Adding Post-Quantum Security

Classic X3DH

Alex **Blake**

$IK_A \leftrightarrow SPK_B$
$EK_A \leftrightarrow IK_B$
$EK_A \leftrightarrow SPK_B$
(* $EK_A \leftrightarrow OPK_B$ *)

$SK = KDF(DH_1 \,||\, DH_2 \,||\, DH_3 \,||\, DH_4)$

100s of bytes
Handshake overhead

PQXDH

Alex **Blake**

$IK_A \leftrightarrow SPK_B$
$EK_A \leftrightarrow IK_B$
$EK_A \leftrightarrow SPK_B$
(* $EK_A \leftrightarrow OPK_B$ *)

$KEM.Encaps(PQPK_B) \rightarrow (SS, CT)$

$SK = KDF(DH_1 \,||\, DH_2 \,||\, DH_3 \,||\, DH_4 \,||\, SS)$

1000s of bytes
Handshake overhead

X3DH →PQXDH: Adding Post-Quantum Security

Classic X3DH

Alex | Blake

$IK_A \leftrightarrow SPK_B$
$EK_A \leftrightarrow IK_B$
$EK_A \leftrightarrow SPK_B$
(* $EK_A \leftrightarrow OPK_B$ *)

$SK = KDF(DH_1 || DH_2 || DH_3 || DH_4)$

100s of bytes
Handshake overhead

PQXDH

Alex | Blake

$IK_A \leftrightarrow SPK_B$
$EK_A \leftrightarrow IK_B$
$EK_A \leftrightarrow SPK_B$
(* $EK_A \leftrightarrow OPK_B$ *)

$KEM.Encaps(PQPK_B) \rightarrow (SS, CT)$

$SK = KDF(DH_1 || DH_2 || DH_3 || DH_4 || SS)$

1000s of bytes
Handshake overhead

Key Design Principles

- **Minimal change:** Add ONE post-quantum key/ciphertext
- **PQ HNDL protection:** A MAJOR gain
- **No loss of ECDH security:** Don't remove effective security!
- **Acceptable cost:** Storage costs are significant, but worth it.

Classical
Quantum vulnerable

HNDL-Hybrid
Partially quantum resistant

Protocol Details Matter

There is a lot more to specifying a protocol than a nice picture.

Details matter, and formal verification - with ProVerif and CryptoVerif [BJKS24] - was an important part of getting it right.

Protocol description: https://signal.org/docs/specifications/pqxdh/

[BJKS24]: https://www.usenix.org/conference/usenixsecurity24/presentation/bhargavan

PQXDH Impact, Risk, and Effort

Impact	Development Effort and Risk
 Widespread.	 Doesn’t get much easier.

Few challenges.
Acceptable cost.
Huge impact.

This was an easy choice.

What this means for our users

All Signal Protocol sessions started in our app today are just as secure as ever, but also enjoy post-quantum HNDL protection.

Even better - once the session is established, even an attacker with a quantum computer won’t be able to read the messages.

Unless they compromise one of the devices...

The Ratchets

B. Auerbach, Y. Dodis, D. Jost, S. Katsumata, T. Prest, K. Bhargavan, F. Kiefer and the Signal Team

Double Ratchet: The Problem

Problem: Signal messages do not have post-quantum PCS. This matters today: a device compromise creates an HNDL opportunity.

Scope: Targeted.

What needs to change: The “Public Ratchet” of the Double Ratchet protocol needs post quantum security.

Public Ratchet Basic Idea

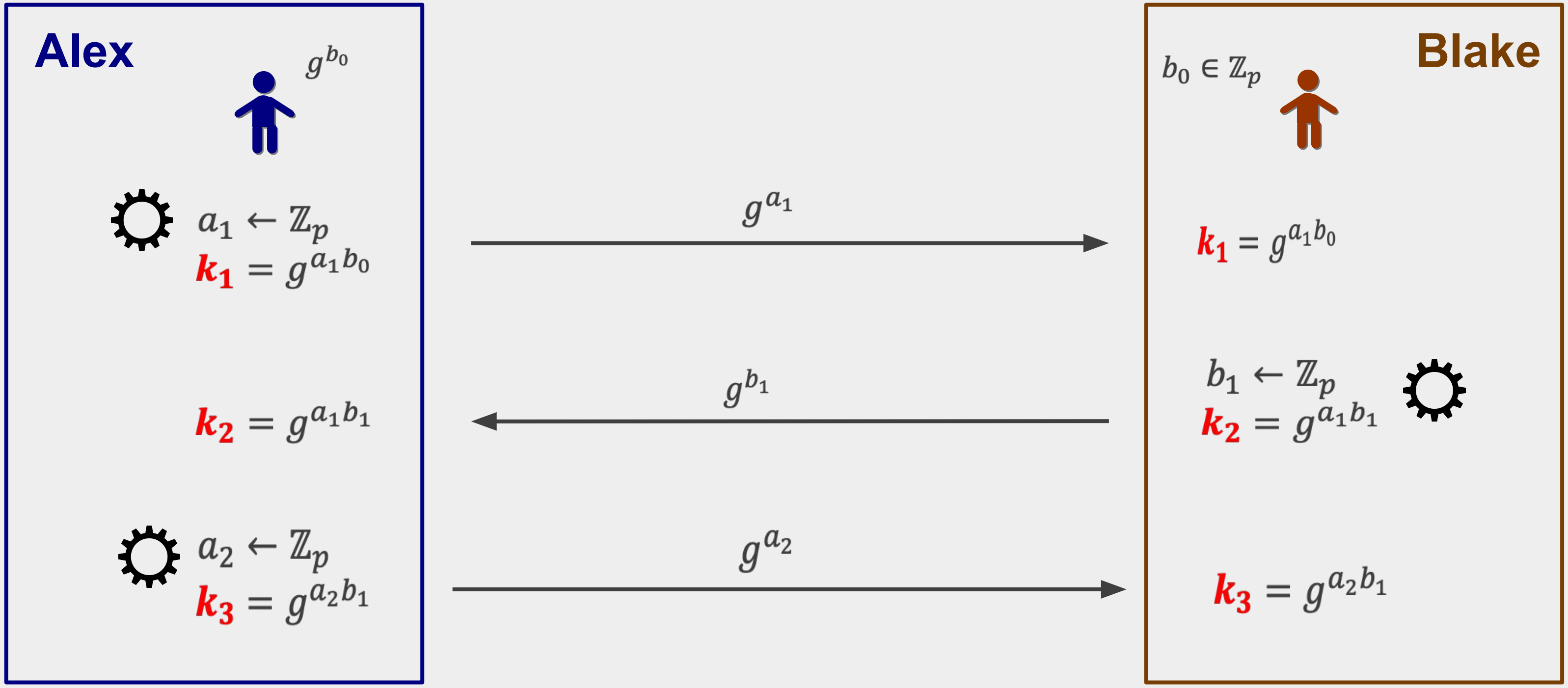
Perform fresh key exchanges as you send and receive messages.

Use the fresh keys to update your session state.

We call this
Continuous Key Agreement (CKA)
[EC:ACD19]

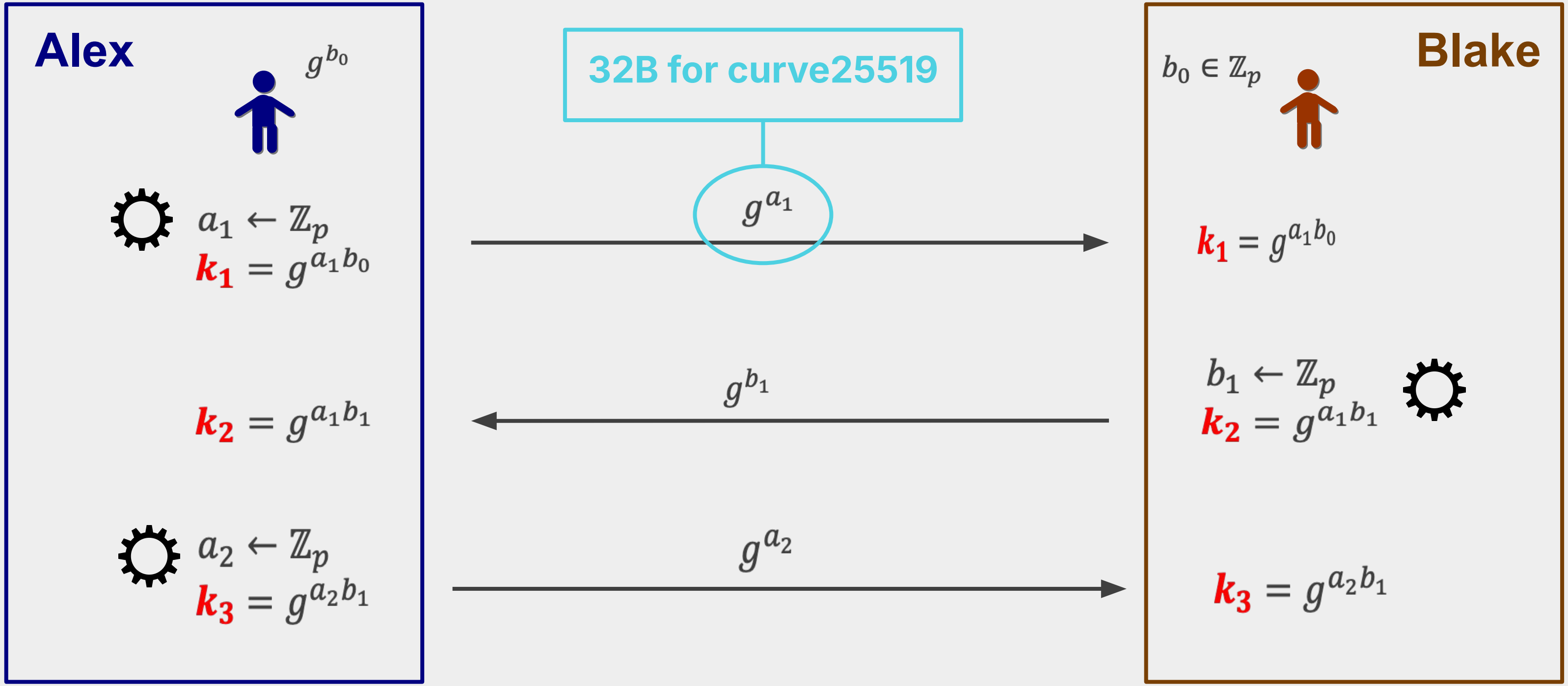
The Diffie-Hellman Ratchet

CKA is easy with Diffie-Hellman key agreement:



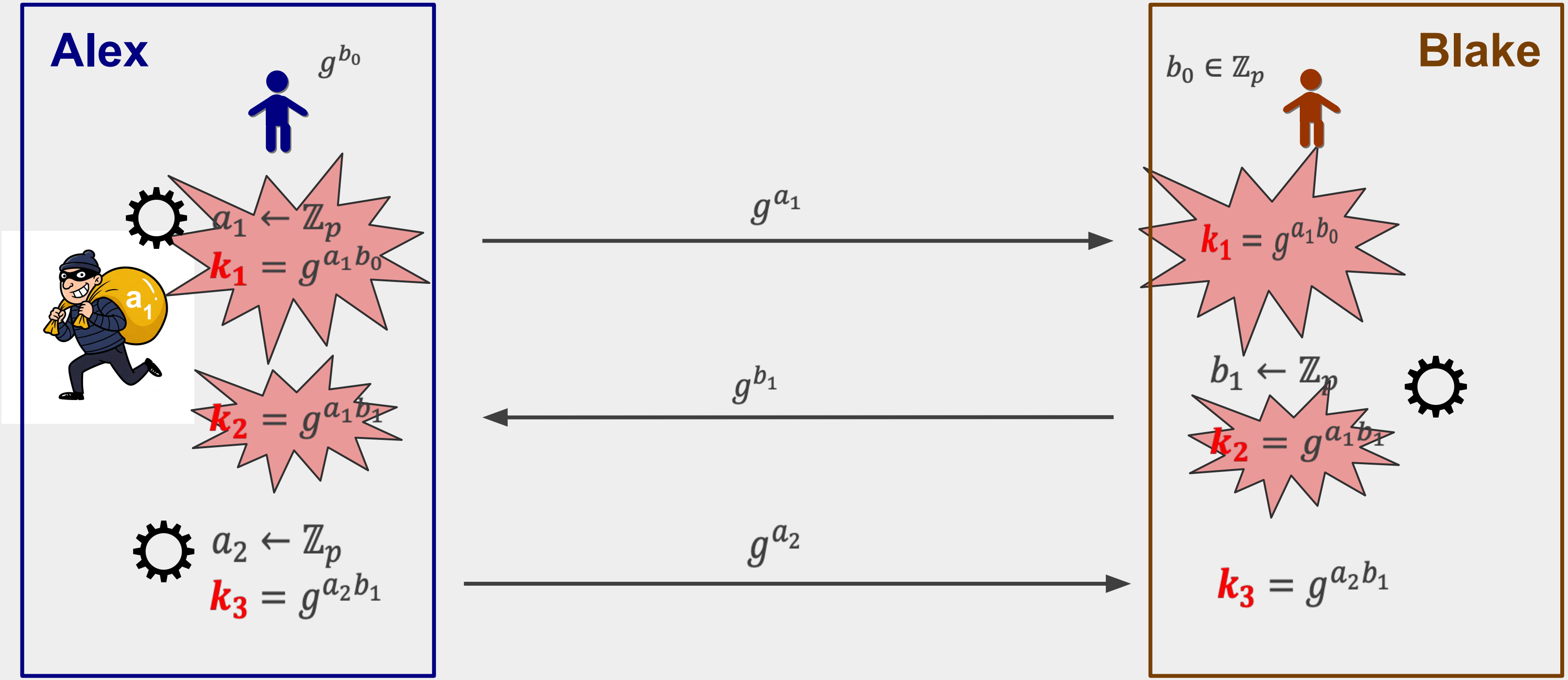
The Diffie-Hellman Ratchet

CKA is easy with Diffie-Hellman key agreement:



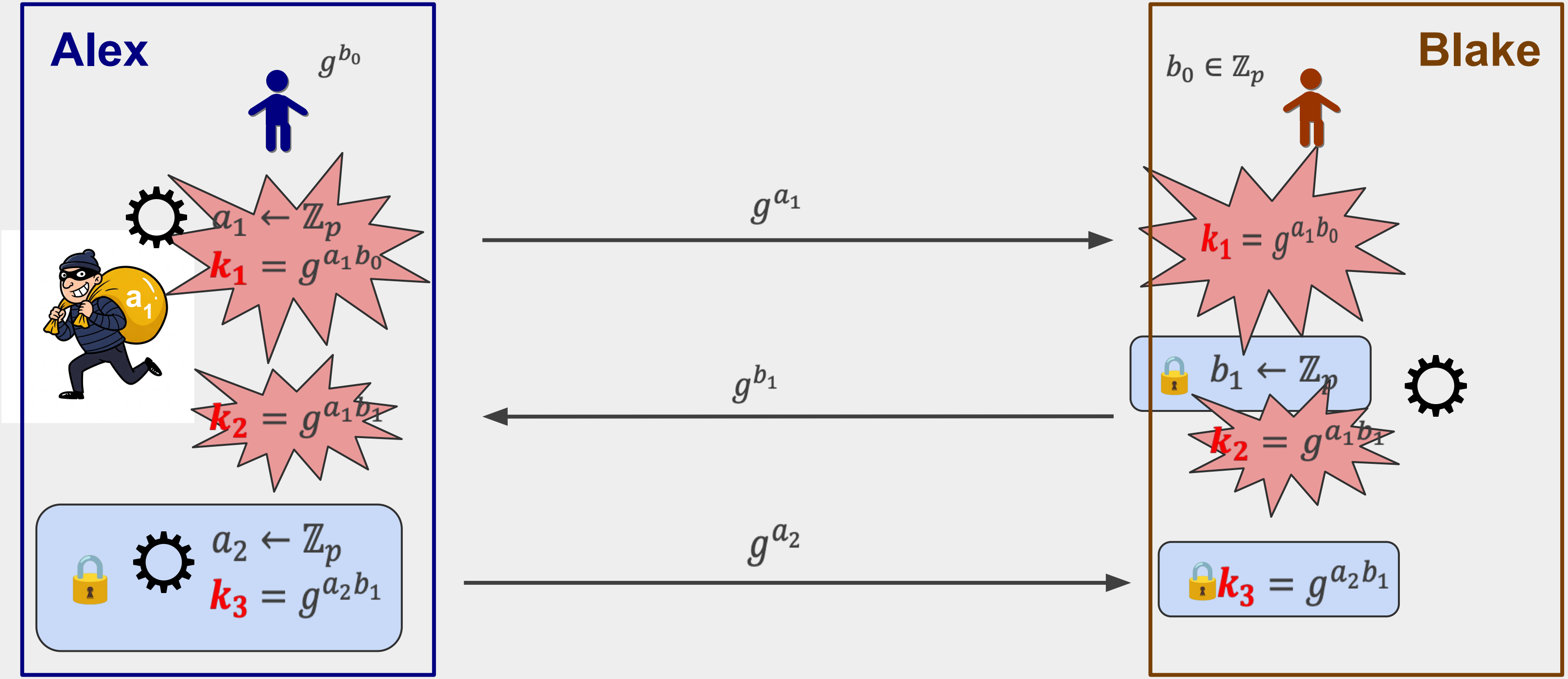
The Diffie-Hellman Ratchet

CKA is easy with Diffie-Hellman key agreement:



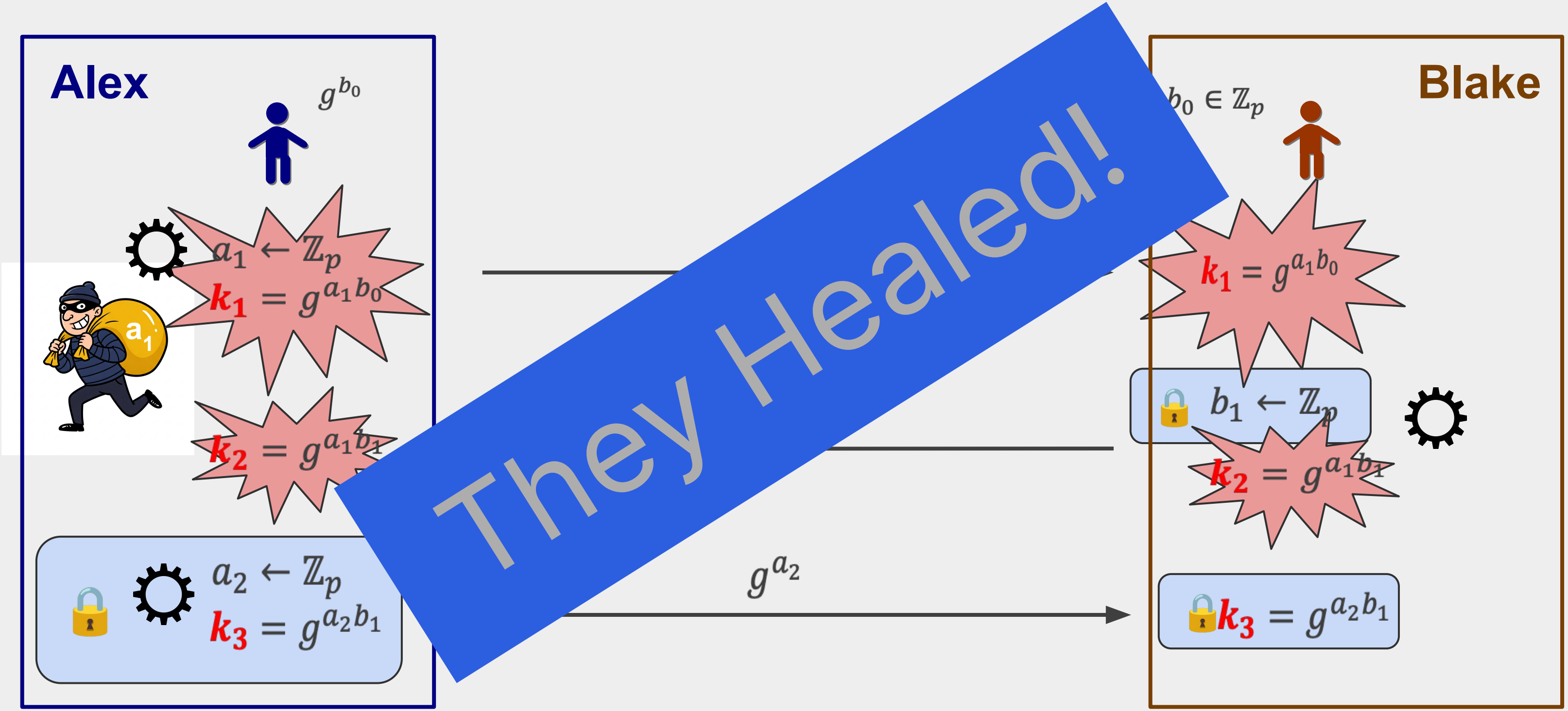
The Diffie-Hellman Ratchet

CKA is easy with Diffie-Hellman key agreement:



The Diffie-Hellman Ratchet

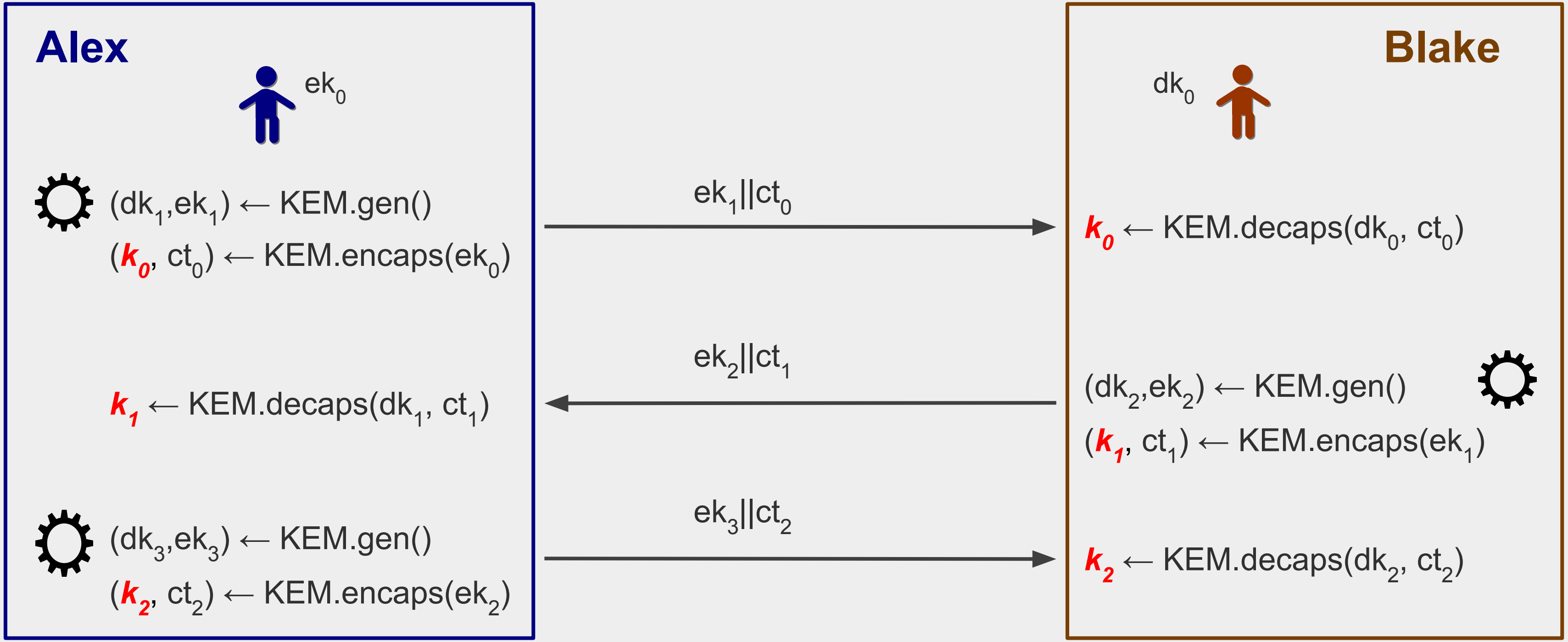
CKA is easy with Diffie-Hellman key agreement:



That’s Post Compromise Security (PCS).

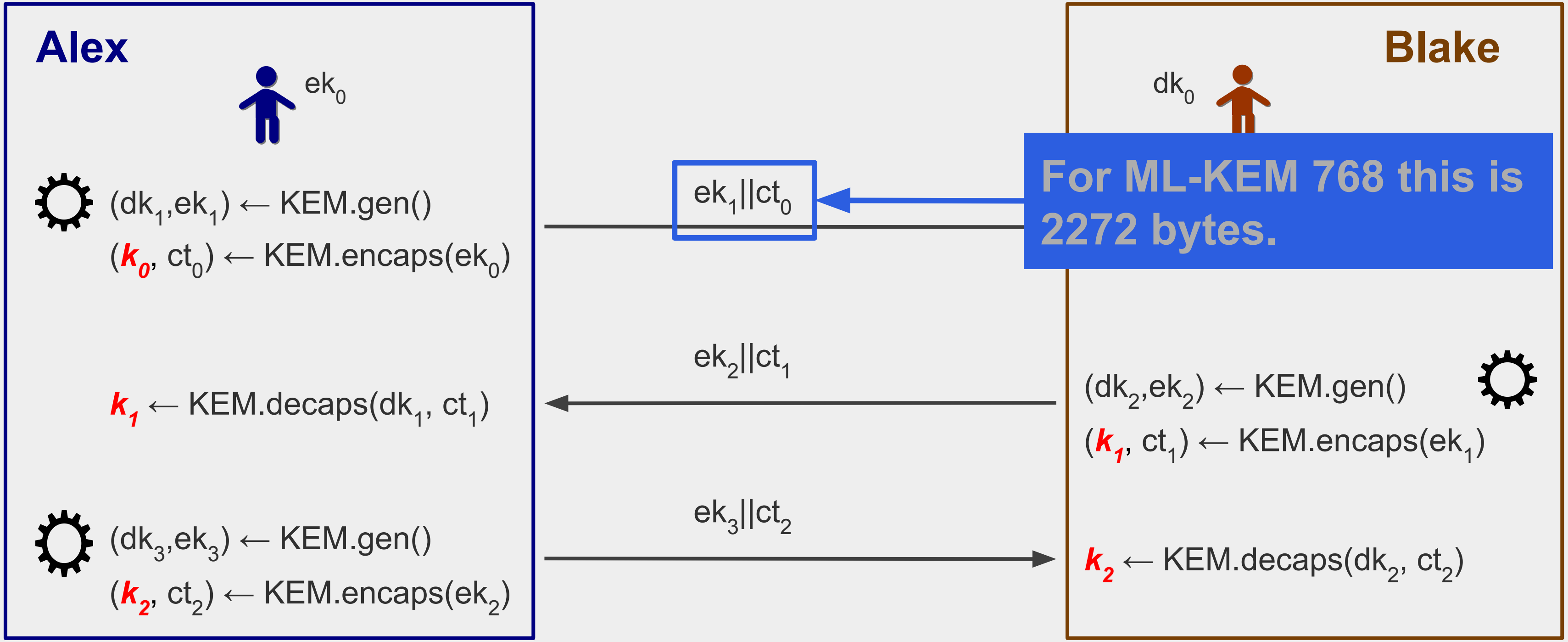
A Post Quantum Ratchet

CKA looks almost the same with a PQ KEM [EC:ACD19]



A Post Quantum Ratchet

CKA looks almost the same with a PQ KEM [EC:ACD19]



35x

Using ML-KEM 768 like this would increase the size of a typical small message by a factor of 35.

This costs us and our users.

This affects usability for users with poor connections.

Two ways to reduce bandwidth

Amortize (like PQ3)

- Don’t send any messages for a long time.
- Then send a big message and repeat it until you get a response.
- Great in some situations, less great in others.

Two ways to reduce bandwidth

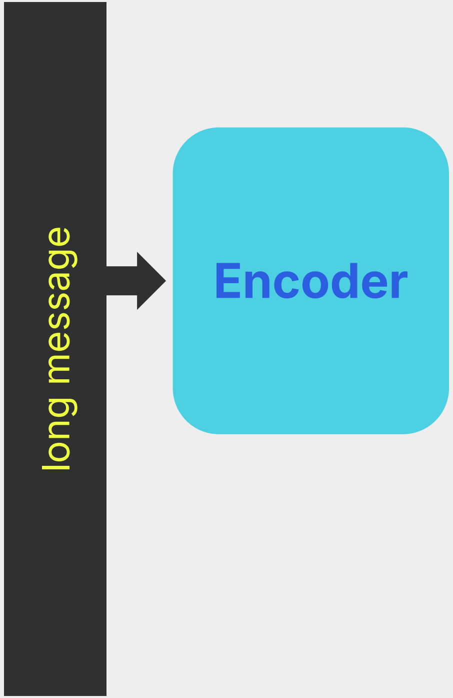
Amortize (like PQ3)

- Don't send any messages for a long time.
- Then send a big message and repeat it until you get a response.
- Great in some situations, less great in others.

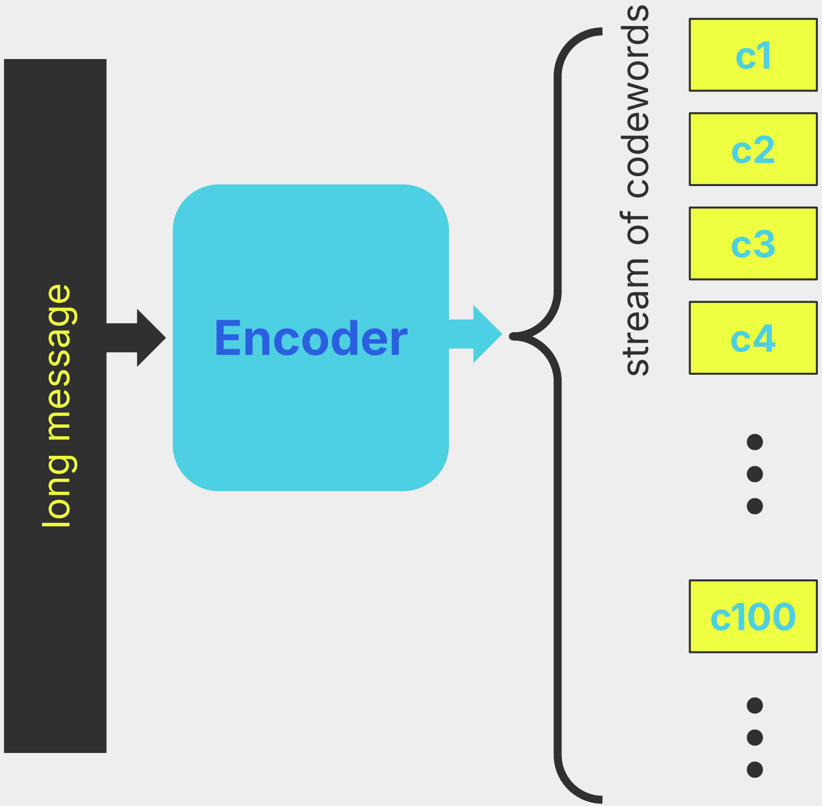
Transmit in pieces

- Break a long message into smaller chunks.
- Send one chunk per message.
- **Careful!**
  - Messages must get transmitted even if the chunks can be adversarially dropped!
  - Can't just send each chunk once.
  - Can't even send round-robin.

Chunking with (Systematic) Erasure Codes



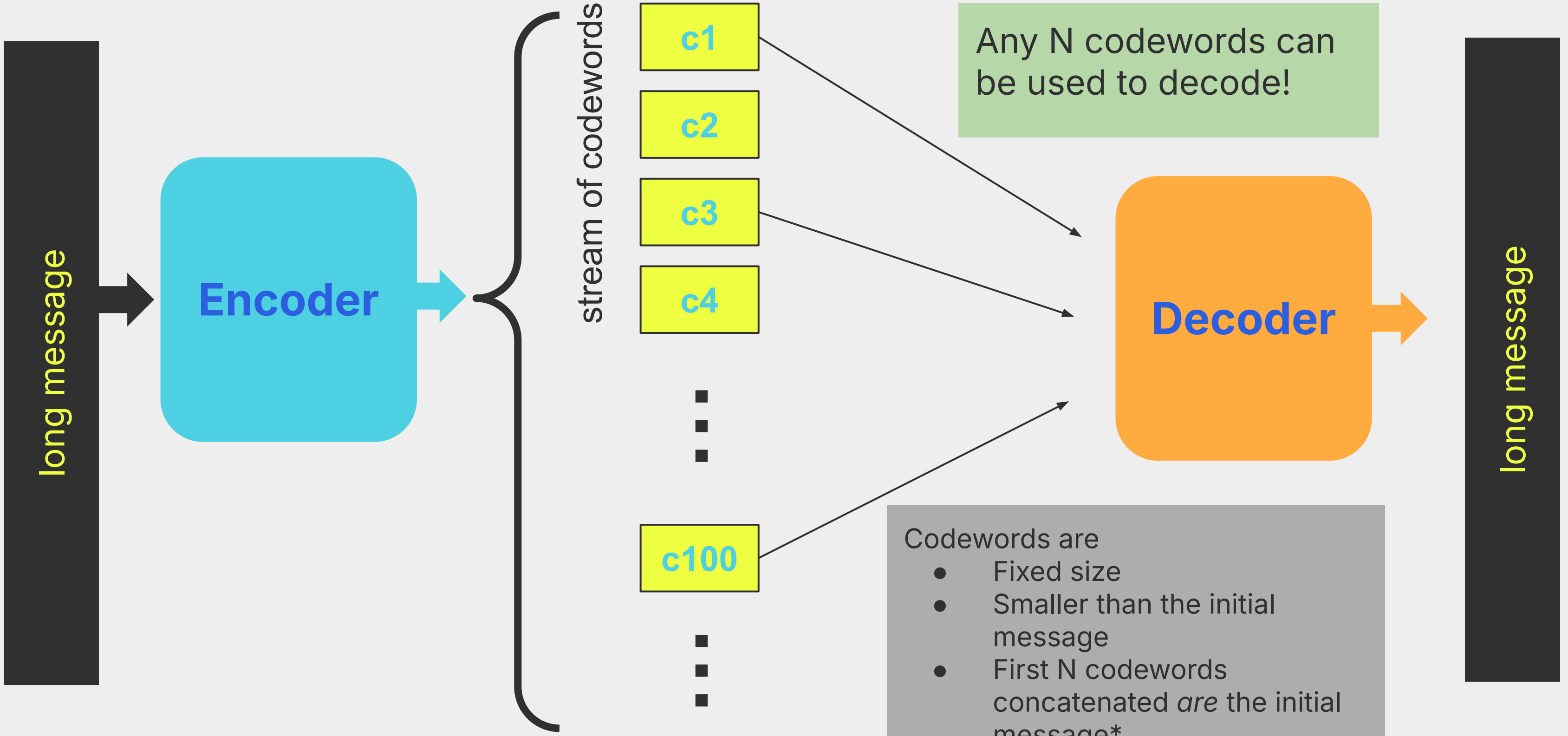
Chunking with (Systematic) Erasure Codes



Codewords are

- Fixed size
- Smaller than the initial message
- First N codewords concatenated *are* the initial message*

Chunking with (Systematic) Erasure Codes



Any N codewords can be used to decode!

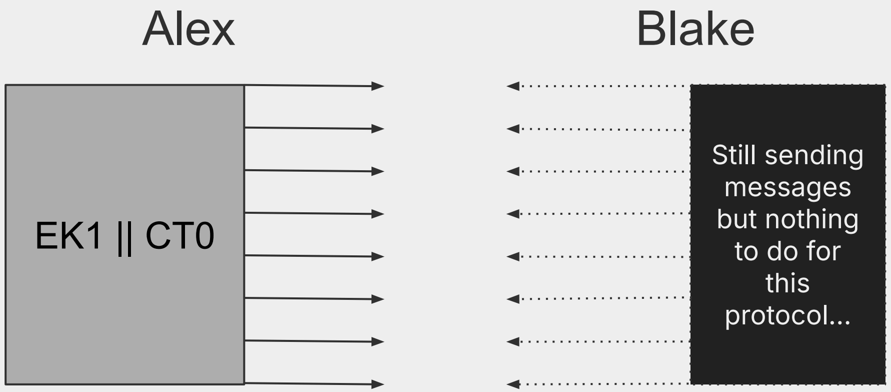
Codewords are
- Fixed size
- Smaller than the initial message
- First N codewords concatenated *are* the initial message*

Secure Messaging with Sparse CKA

Now we can take any CKA and turn it into a “chunked” protocol.

Note: It isn’t a CKA anymore syntactically because it doesn’t emit a new key every time it sends or receives a message.

So we define a “Sparse CKA” (SCKA) and show how to construct secure Messaging from a Sparse CKA.

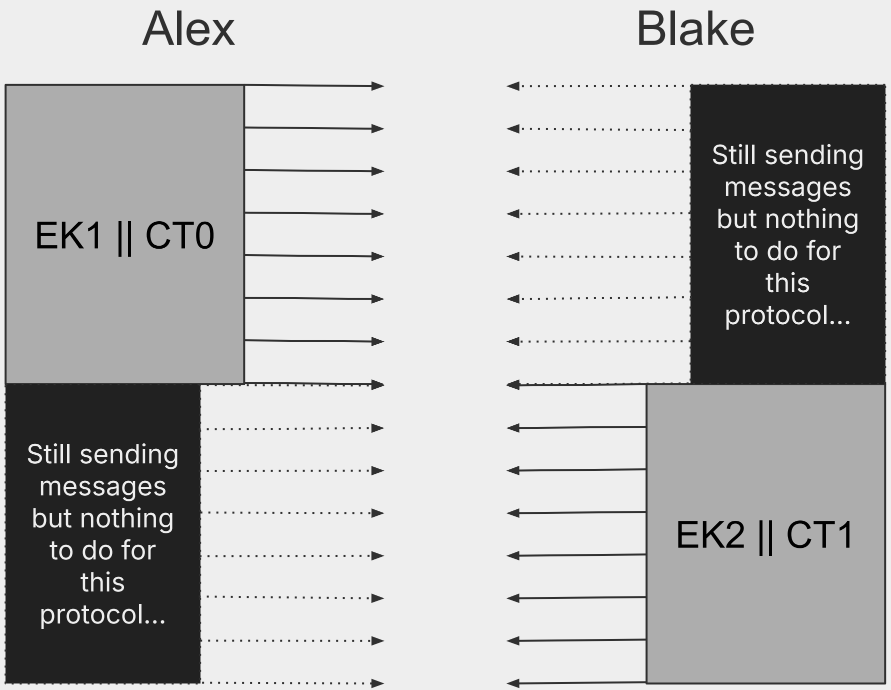


Secure Messaging with Sparse CKA

Now we can take any CKA and turn it into a “chunked” protocol.

Note: It isn’t a CKA anymore syntactically because it doesn’t emit a new key every time it sends or receives a message.

So we define a “Sparse CKA” (SCKA) and show how to construct secure Messaging from a Sparse CKA.

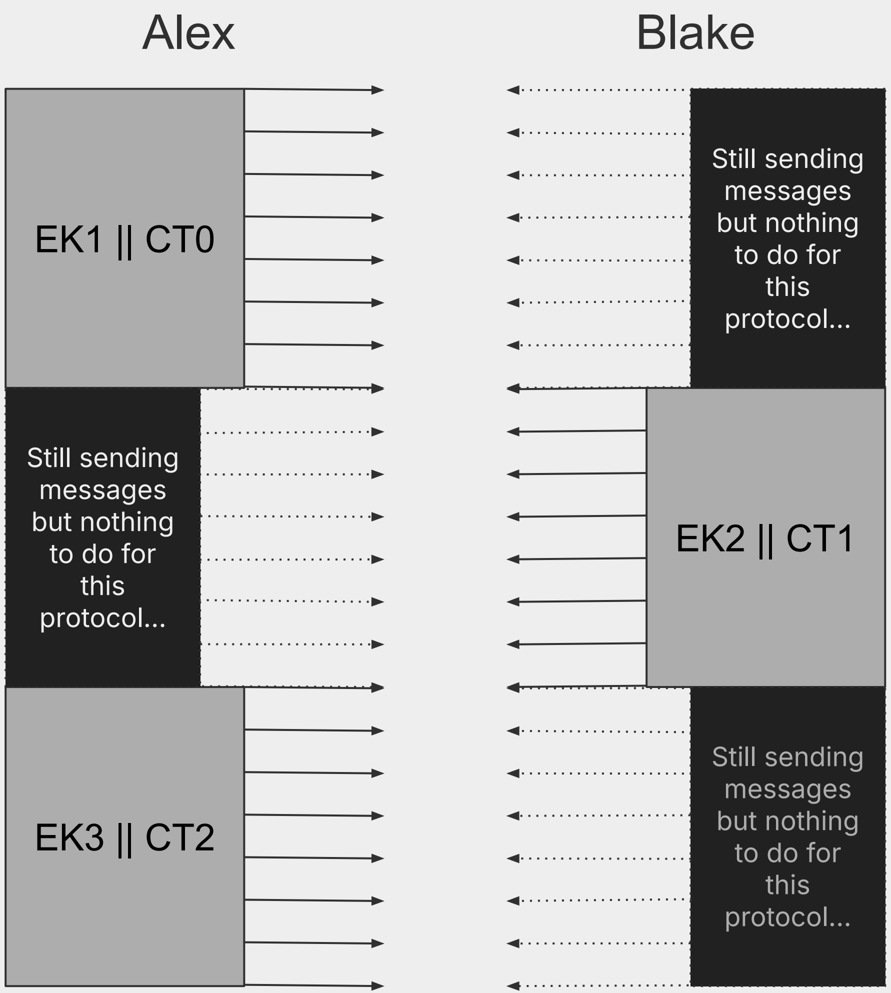


Secure Messaging with Sparse CKA

Now we can take any CKA and turn it into a “chunked” protocol.

Note: It isn’t a CKA anymore syntactically because it doesn’t emit a new key every time it sends or receives a message.

So we define a “Sparse CKA” (SCKA) and show how to construct secure Messaging from a Sparse CKA.

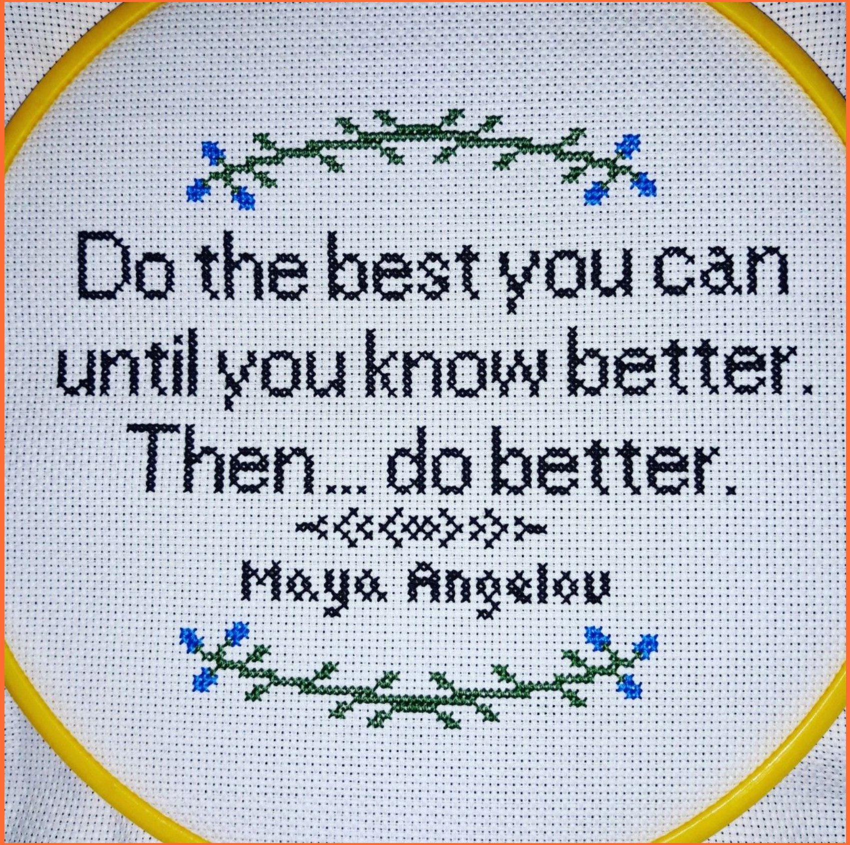


So we’re done?

- Use ML-KEM to instantiate the KEM-based CKA from [EC:ACD19].
- Use our “chunking compiler” to turn it into an SCKA.
- Drop this into our SCKA-based Secure Messaging protocol to get messaging with MLWE-based security.
- Hybridize it with the classic double ratchet.

No.

We can do better.



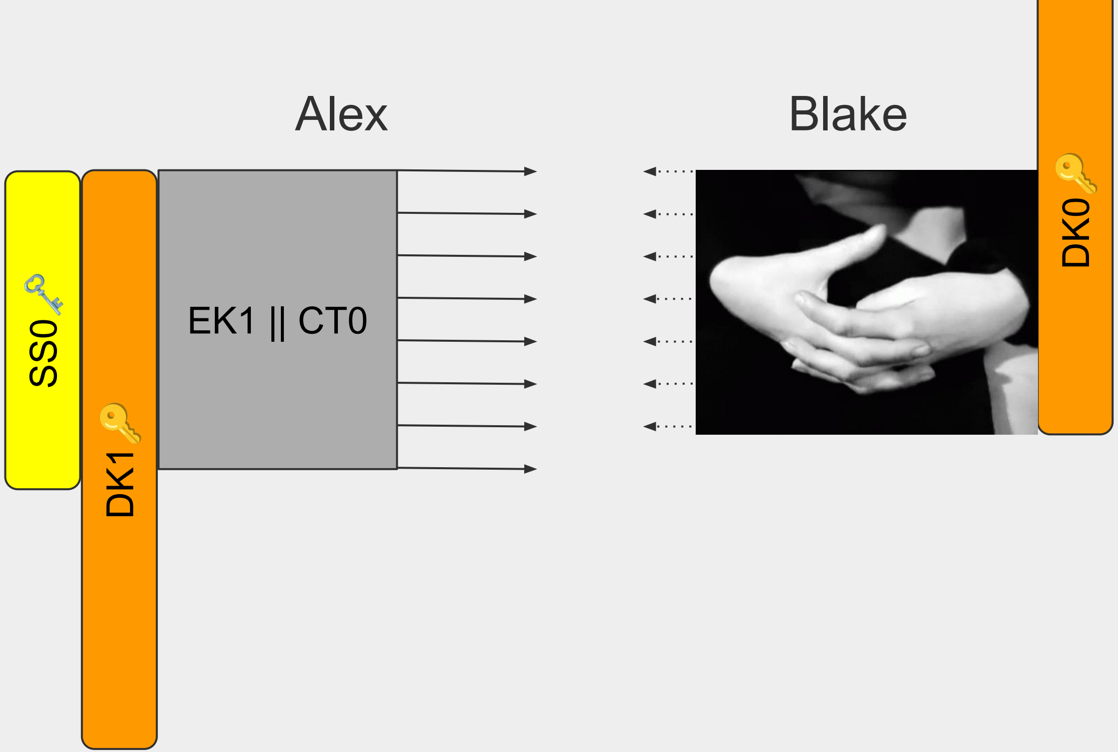
The Problems

When we “chunk” the Standard KEM CKA protocol, there is always someone sitting quiet.

And look how long Alex and Blake have to hold onto their secrets. 😬.

Big attack surface, slow key emission.

Can’t they do *something*?



KEM Shared Secret 🗝️

Decapsulation Key 🔑

The Problems

When we “chunk” the Standard KEM CKA protocol, there is always someone sitting quiet.

And look how long Alex and Blake have to hold onto their secrets. 😬.

Big attack surface, slow key emission.

Can’t they do *something*?

KEM Shared Secret 🗝️

Decapsulation Key 🔑

The Problems

When we “chunk” the Standard KEM CKA protocol, there is always someone sitting quiet.

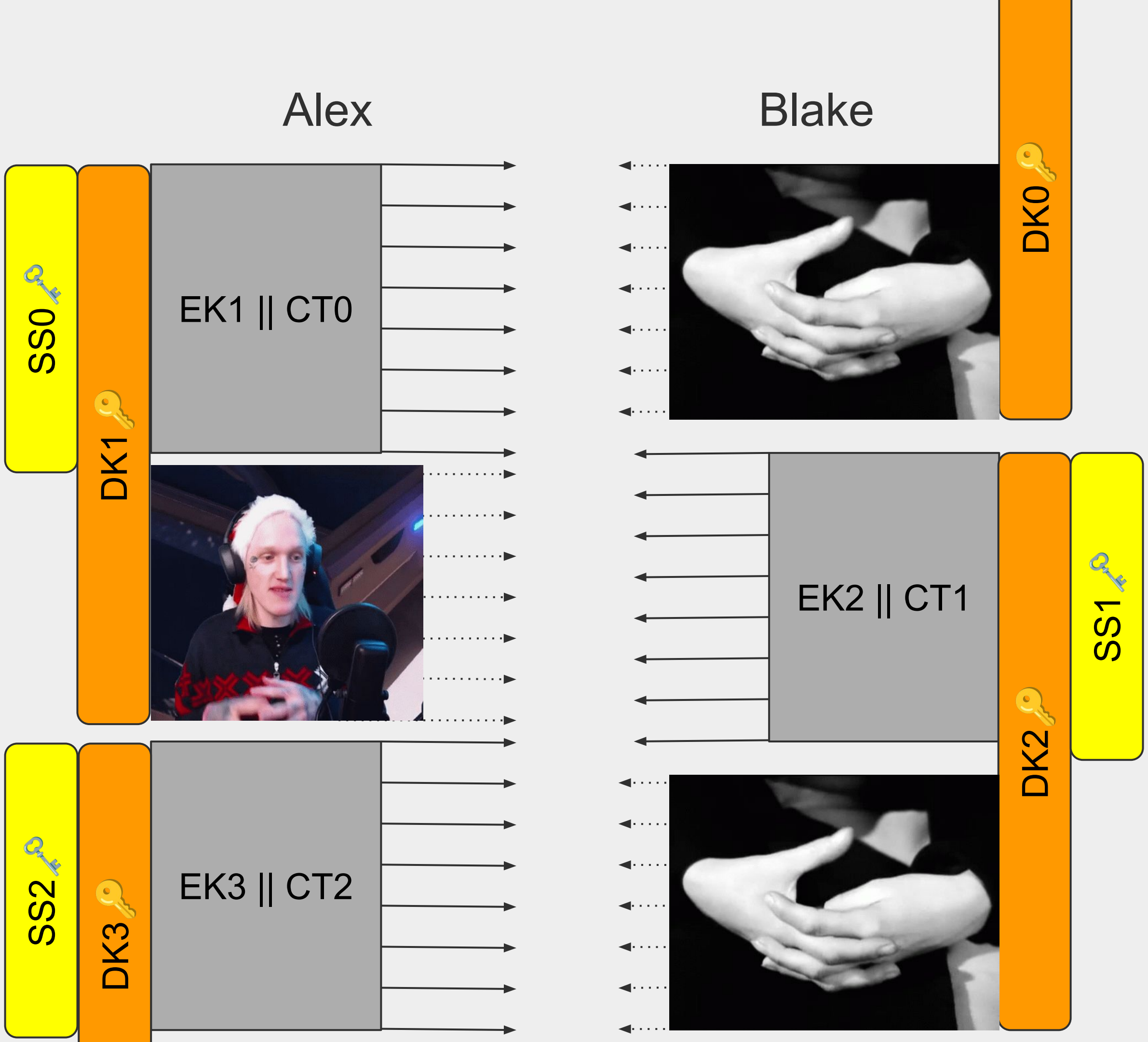
And look how long Alex and Blake have to hold onto their secrets. 😬.

Big attack surface, slow key emission.

Can’t they do *something*?

KEM Shared Secret 🗝️

Decapsulation Key 🔑



Ways to Do Better™:

1. Reduce the attack surface.
2. Blocked? Sample something and send!
3. Open the KEM black box.

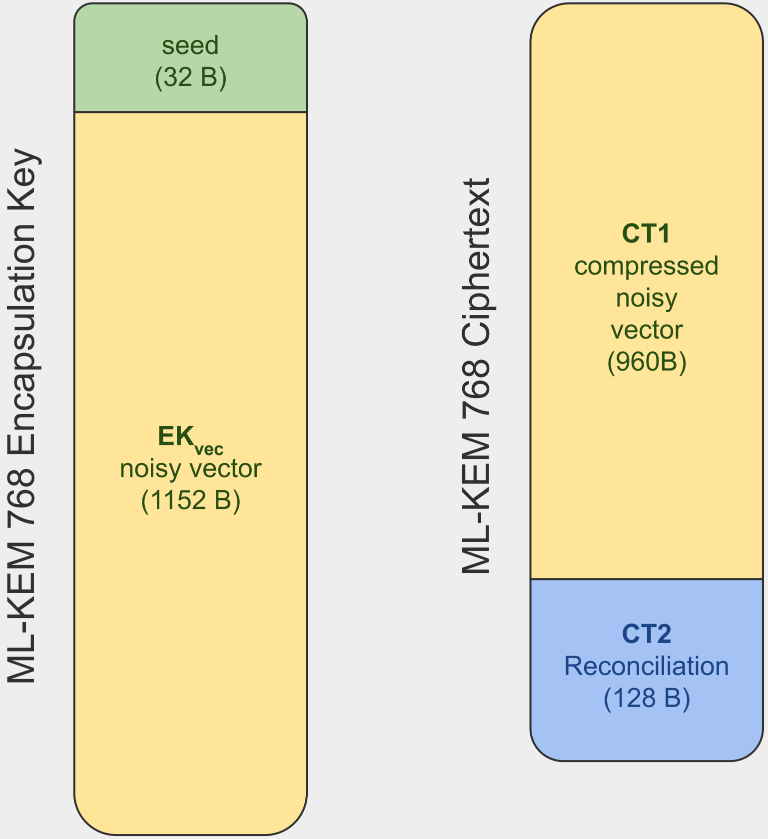
Open the KEM Black Box: Incremental KEM

An ML-KEM Encapsulation key has two parts:

1. A 32B seed that gets expanded into a matrix **A**.
2. A "noisy vector", **As** + **e**, where **s** is a decapsulation secret and **e** is small error.

An ML-KEM Ciphertext has two parts:

1. A "compressed noisy vector", **A s′** + **e′**, where **s′** is a decapsulation secret and **e′** is small error.
2. A "reconciliation message"



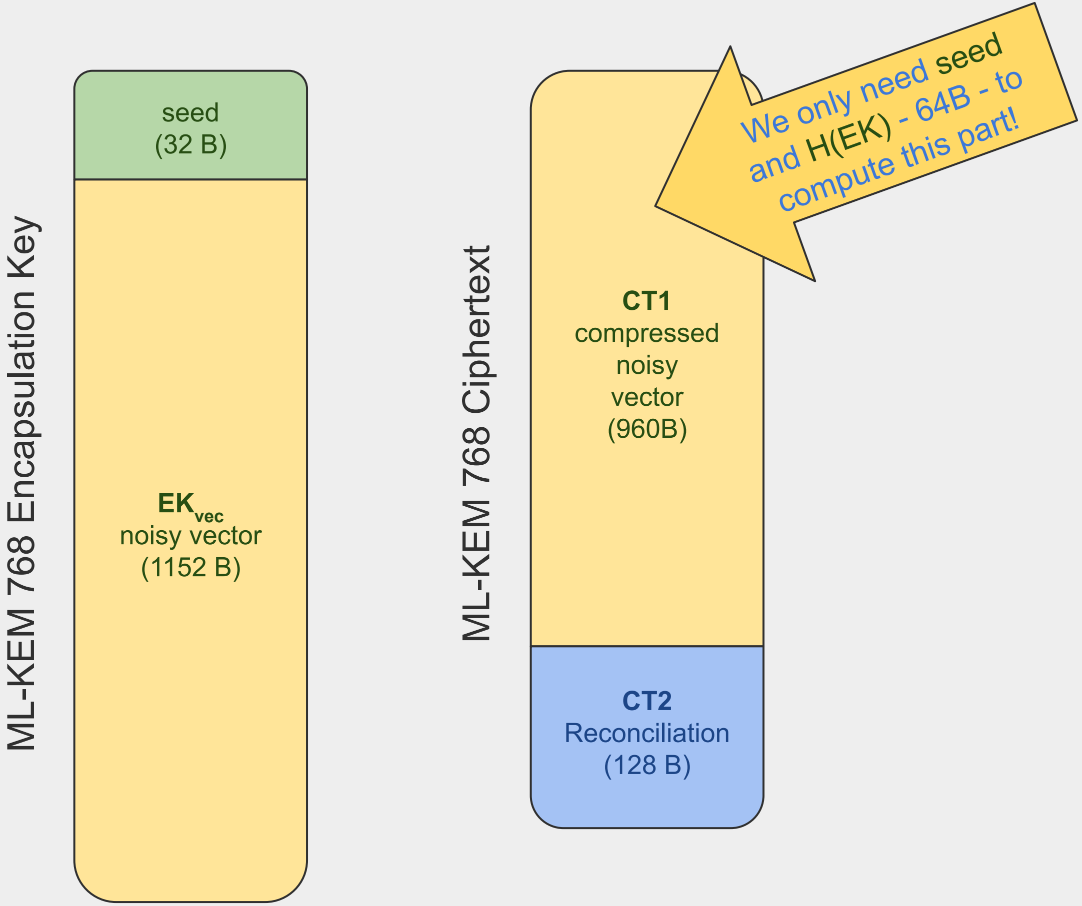
Open the KEM Black Box: Incremental KEM

An ML-KEM Encapsulation key has two parts:

1. A 32B seed that gets expanded into a matrix **A**.
2. A "noisy vector", **As** + **e**, where **s** is a decapsulation secret and **e** is small error.

An ML-KEM Ciphertext has two parts:

1. A "compressed noisy vector", **A s′** + **e′**, where **s′** is a decapsulation secret and **e′** is small error.
2. A "reconciliation message"



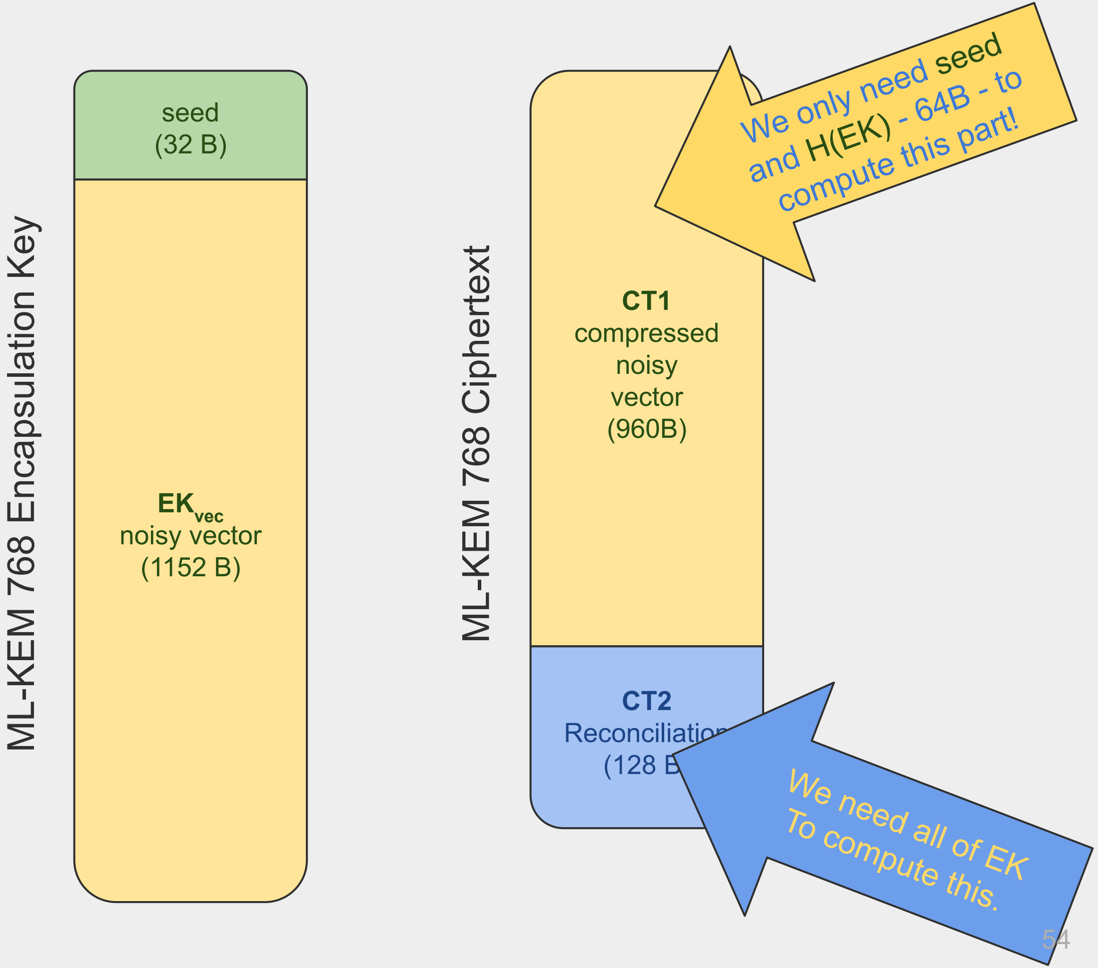
Open the KEM Black Box: Incremental KEM

An ML-KEM Encapsulation key has two parts:

1. A 32B seed that gets expanded into a matrix **A**.
2. A "noisy vector", **As** + **e**, where **s** is a decapsulation secret and **e** is small error.

An ML-KEM Ciphertext has two parts:

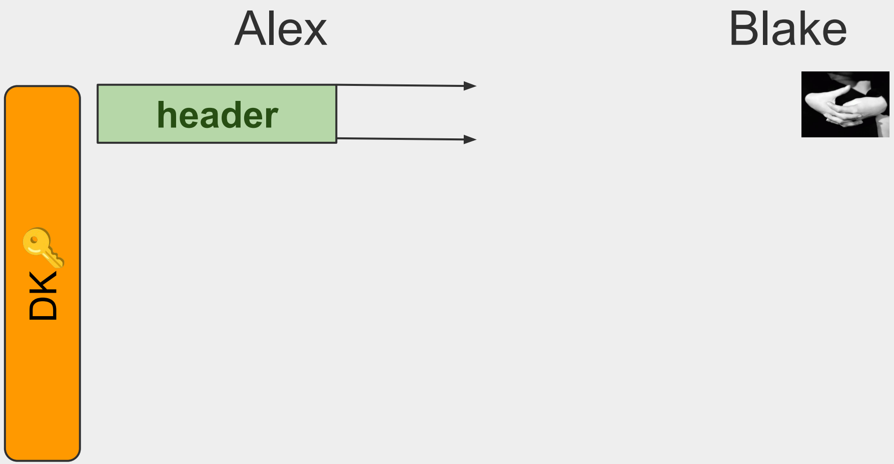
1. A "compressed noisy vector", **A s′** + **e′**, where **s′** is a decapsulation secret and **e′** is small error.
2. A "reconciliation message"



Idea: Now we can sample CT1 early and send it in parallel with EK.

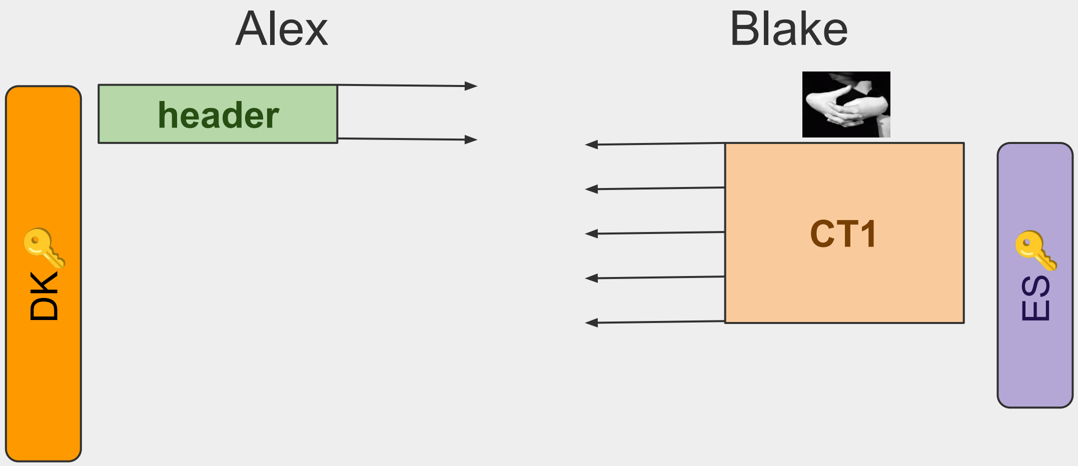
The ML-KEM Braid Protocol

- Alex sends the **seed** first
- When Blake gets **seed**, they sample **CT1** and start sending it.
- When Alex gets a chunk of **CT1** they can stop sending seed and start sending $\mathbf{EK}_{\mathbf{vec}}$.
- Once Blake has all of $\mathbf{EK}_{\mathbf{vec}}$ (and knows Alex has CT1!) they can start sending **CT2**.
- When Alex gets **CT2** they can start using the shared secret and ACK Blake.
- When Blake gets the ACK, they start using the shared secret and swap roles.



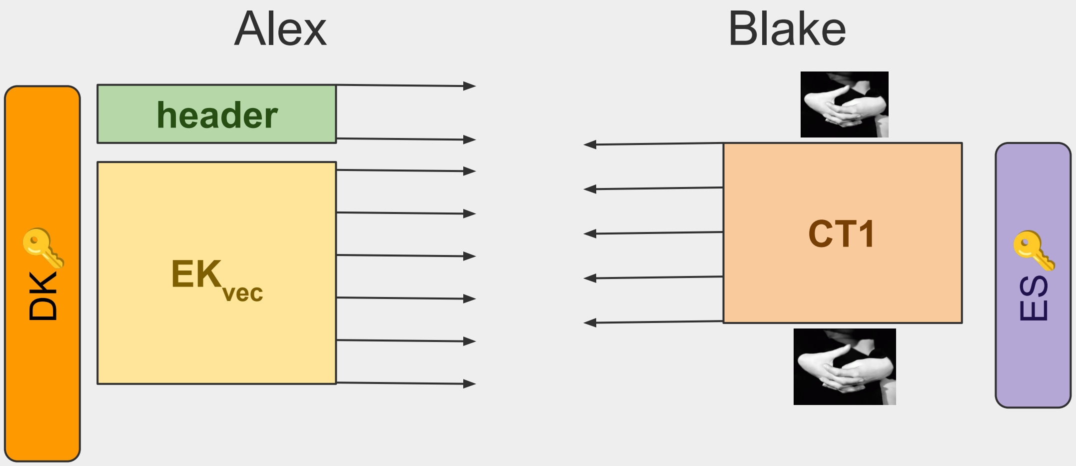
The ML-KEM Braid Protocol

- Alex sends the **seed** first
- When Blake gets **seed**, they sample **CT1** and start sending it.
- When Alex gets a chunk of **CT1** they can stop sending seed and start sending $\mathbf{EK}_{\mathbf{vec}}$.
- Once Blake has all of $\mathbf{EK}_{\mathbf{vec}}$ (and knows Alex has CT1!) they can start sending **CT2**.
- When Alex gets **CT2** they can start using the shared secret and ACK Blake.
- When Blake gets the ACK, they start using the shared secret and swap roles.



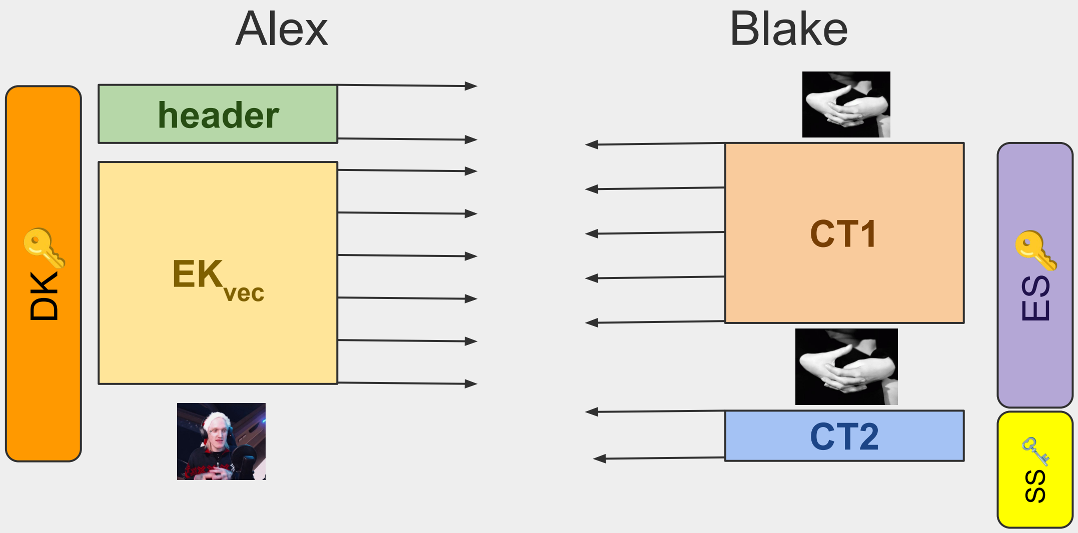
The ML-KEM Braid Protocol

- Alex sends the **seed** first
- When Blake gets **seed**, they sample **CT1** and start sending it.
- When Alex gets a chunk of **CT1** they can stop sending seed and start sending $\mathbf{EK_{vec}}$.
- Once Blake has all of $\mathbf{EK_{vec}}$ (and knows Alex has CT1!) they can start sending **CT2**.
- When Alex gets **CT2** they can start using the shared secret and ACK Blake.
- When Blake gets the ACK, they start using the shared secret and swap roles.



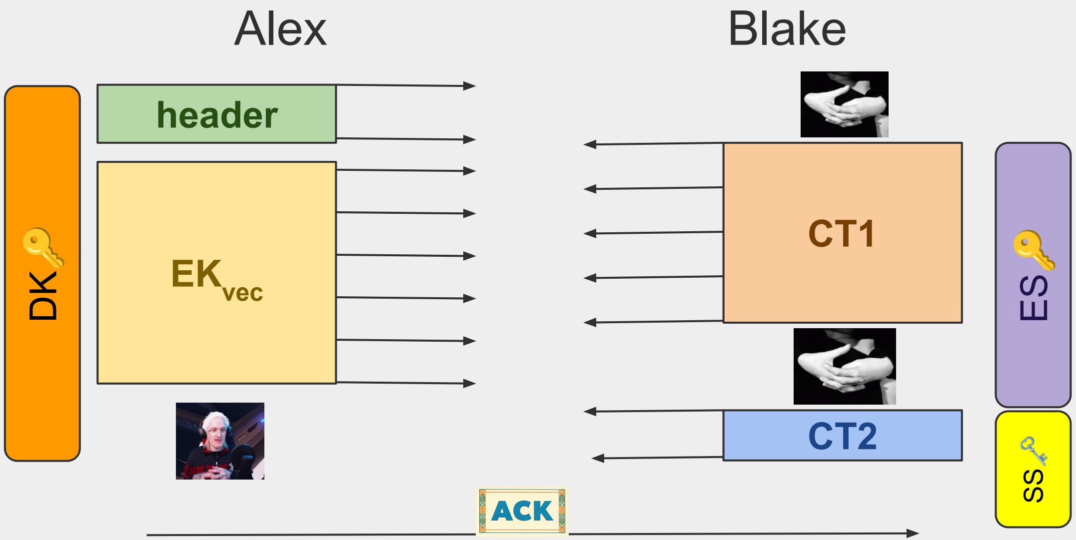
The ML-KEM Braid Protocol

- Alex sends the **seed** first
- When Blake gets **seed**, they sample **CT1** and start sending it.
- When Alex gets a chunk of **CT1** they can stop sending seed and start sending **EK$_{vec}$**.
- Once Blake has all of **EK$_{vec}$** (and knows Alex has CT1!) they can start sending **CT2**.
- When Alex gets **CT2** they can start using the shared secret and ACK Blake.
- When Blake gets the ACK, they start using the shared secret and swap roles.



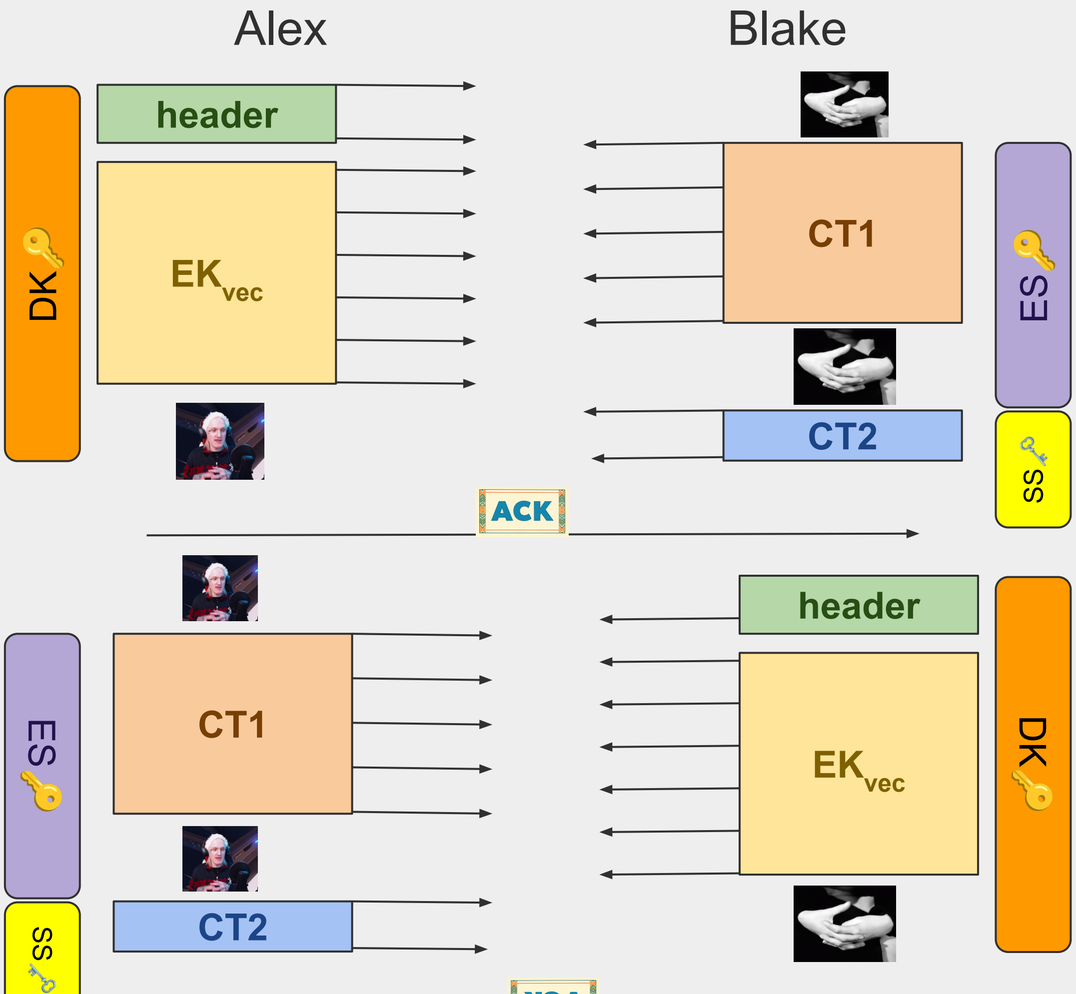
The ML-KEM Braid Protocol

- Alex sends the **seed** first
- When Blake gets **seed**, they sample **CT1** and start sending it.
- When Alex gets a chunk of **CT1** they can stop sending seed and start sending **EK**$_{\mathbf{vec}}$.
- Once Blake has all of **EK**$_{\mathbf{vec}}$ (and knows Alex has CT1!) they can start sending **CT2**.
- When Alex gets **CT2** they can start using the shared secret and ACK Blake.
- When Blake gets the ACK, they start using the shared secret and swap roles.



The ML-KEM Braid Protocol

- Alex sends the **seed** first
- When Blake gets **seed**, they sample **CT1** and start sending it.
- When Alex gets a chunk of **CT1** they can stop sending seed and start sending **EK$_{vec}$**.
- Once Blake has all of **EK$_{vec}$** (and knows Alex has CT1!) they can start sending **CT2**.
- When Alex gets **CT2** they can start using the shared secret and ACK Blake.
- When Blake gets the ACK, they start using the shared secret and swap roles.



~~35x~~ 1.6x

Using a 42B per-message bandwidth limit increases the size of a typical small message by a factor of 1.6.

Still costly, but **consistent** and **reasonable**.

But we ratchet much more slowly.

We call Secure Messaging with the **ML-KEM Braid** the

Sparse Post Quantum Ratchet

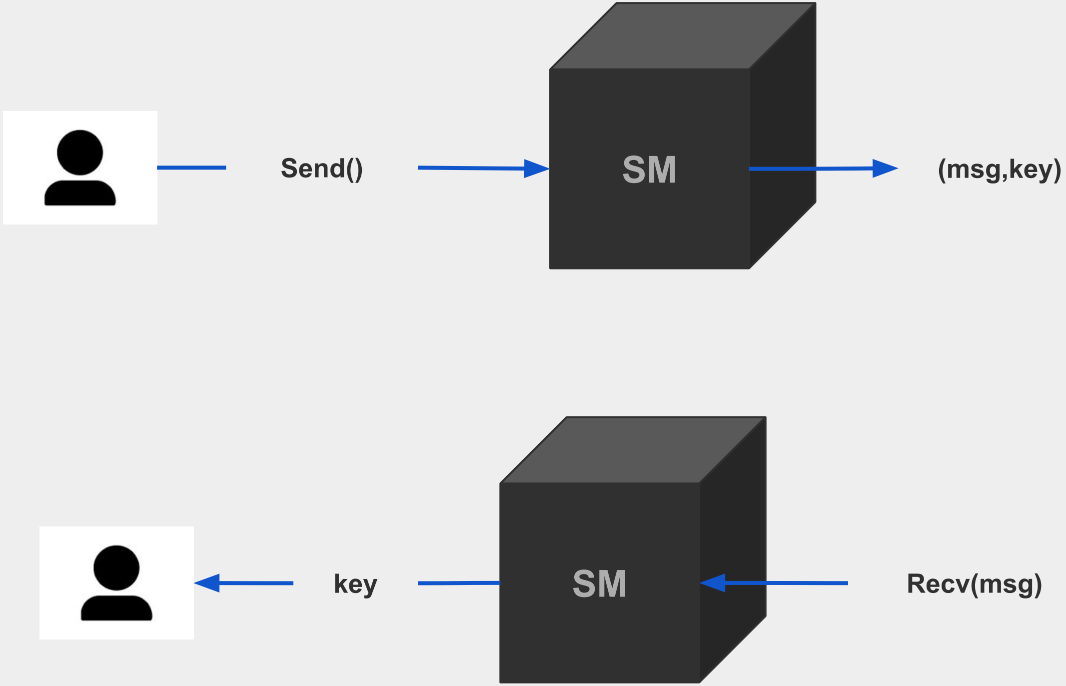
SPQR

Last step: Integrate the PQ ratchet with the classic ratchet for hybrid security.

Secure Messaging as a Black Box

To hybridize, think of the Double Ratchet as a Secure Messaging black box

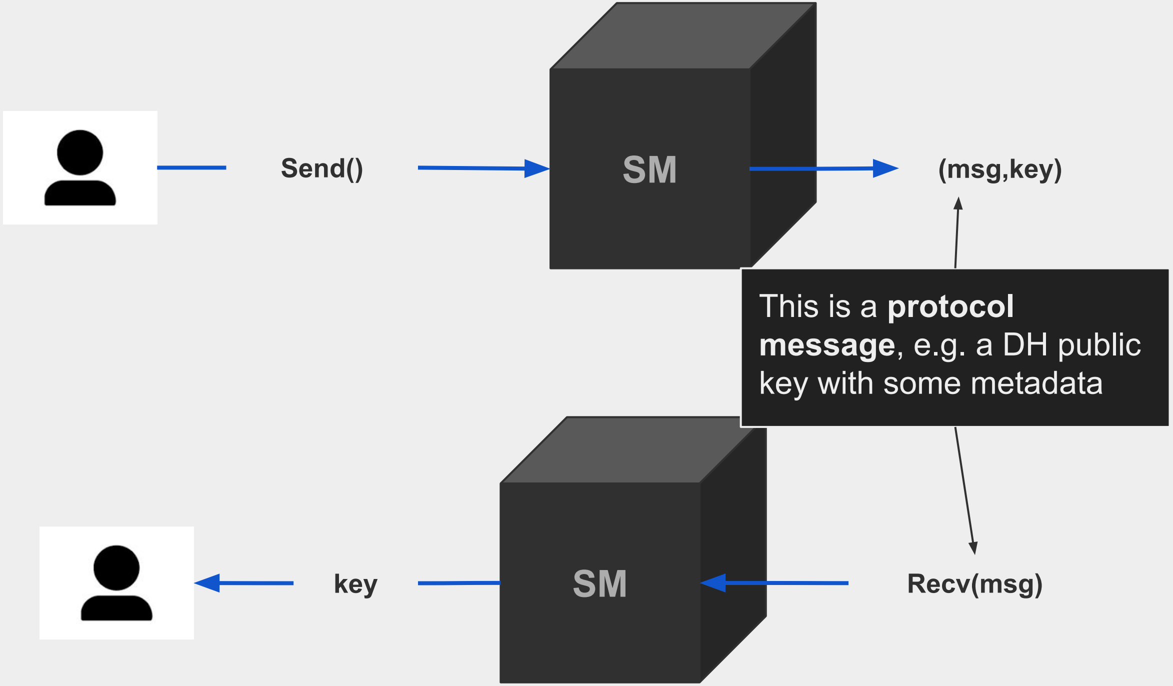
- **Init(secret)**
- **Send() → (msg, mk_{enc})**: get a protocol message and an encryption key, no input needed.
- **Recv(msg) → mk_{enc}**: Take a protocol message and get a decryption key.



Secure Messaging as a Black Box

To hybridize, think of the Double Ratchet as a Secure Messaging black box

- **Init(secret)**
- **Send() → (msg, mk_{enc})**: get a protocol message and an encryption key, no input needed.
- **Recv(msg) → mk_{enc}**: Take a protocol message and get a decryption key.



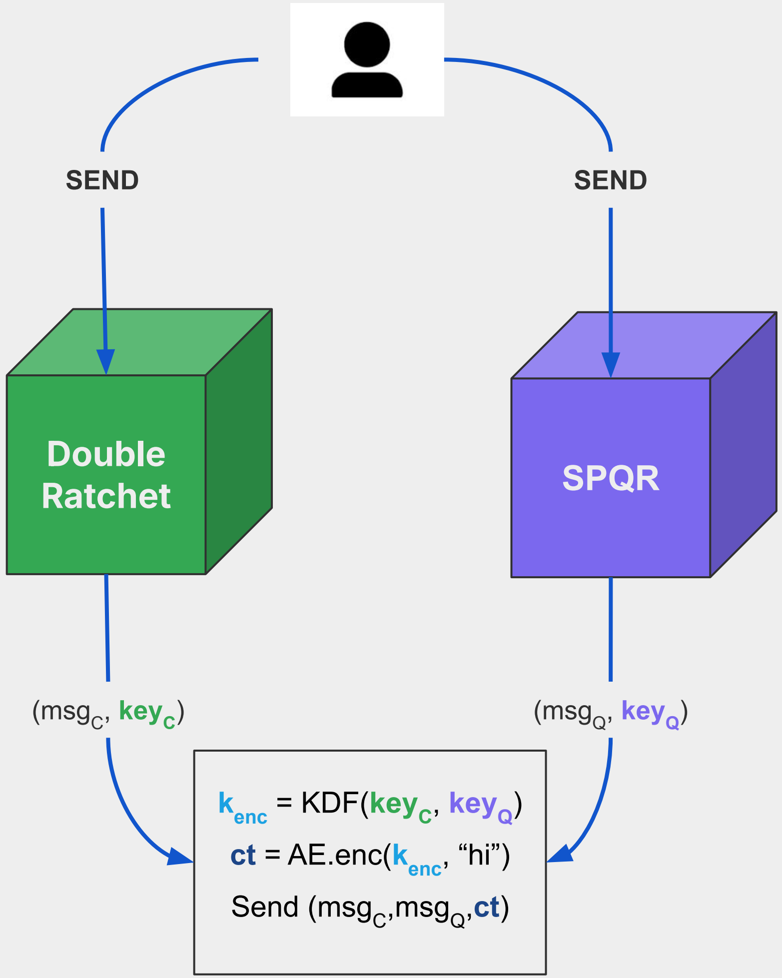
The Triple Ratchet

Compose to SM protocols by KDF-ing the output keys together.

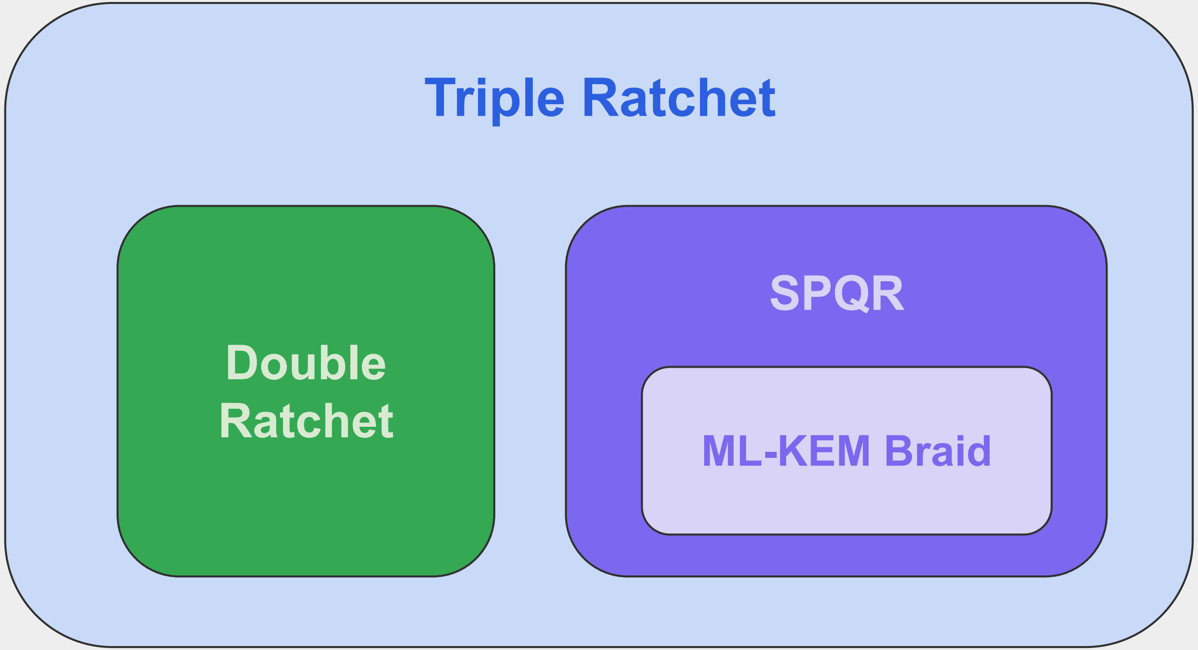
Combining the existing Double Ratchet and the ML-KEM Braid based Double Ratchet we get **hybrid DH+MLWE PCS**.

Bonus: changes to existing code are minimal!

https://eprint.iacr.org/2025/078



All the names



Formal Verification

- Formal Verification was part of the process from the beginning.
- ProVerif was used to evaluate protocol candidates.
- Hax/F* used to verify Rust implementation is panic free
    - Also prove correctness of Galois field arithmetic
- Our CI pipeline runs the proofs on every push.
- Have a look: https://github.com/signalapp/SparsePostQuantumRatchet

Formal verification doesn't freeze your implementation. It's an important part of the dynamic development process.

with CRYSPEN

Triple Ratchet Impact, Risk, and Effort

Impact	Development Effort and Risk
 Targeted.	 Manageable risk. Significant effort.

What this means for our users

Once the Triple Ratchet is fully deployed, Signal users will have Forward Secrecy and Post Compromise Security even against quantum adversaries.

Once the session is established it is quantum safe - and still has all of the ECDH-based security guarantees.

But about that session establishment...

A Fully Quantum Safe Protocol

We aren’t there yet

Once the session is established, the Triple Ratchet provides full hybrid security.

But PQXDH is trivially insecure against active quantum attacks: a quantum attacker can compute the secret key for your Identity Key and impersonate you.

We need post-quantum authentication in the handshake protocol...

...without breaking the other promises:

- DH Authentication
- DH+MLWE Forward Secrecy
- “Deniability”

We have good options:

- Well studied protocols
- Nuanced understanding of deniability
- Efficient 2-Ring signatures

If we set out to design a concrete protocol now we are likely to succeed.

Full Hybrid Protocol Impact, Risk, and Effort

Impact	Development Effort and Risk
 Anticipated/widespread.	 Moderate risk and effort. Research will minimize risk.

What this will mean for our users

Once we deploy a hybrid secure handshake, the updated Signal Protocol will have full post-quantum security.

This is the root of all data security throughout our app and gives us a solid foundation for the rest of our PQ transition.

We will be ready.

Stay tuned.

The Big Picture

- The transition so far
- Looking forward

Our PQ Transition So Far

✅ PQXDH (2023): HNDL protection deployed

🐴 Workhorse crypto (2024-∞): transitioning secure channels and more

🔄 Triple Ratchet (2025): Post-quantum PCS coming soon

We are prioritizing and progressing.

Looking Forward

📋 Full Hybrid Signal Protocol: Research phase

🐴 More workhorse crypto (much is blocked until we have PQ Identity)

🔬 Beyond 1:1:

Sealed Sender

Group Messaging

Anonymous Credentials

We are just getting started.

Thank you!

rolfe@signal.org

rolfe.1729

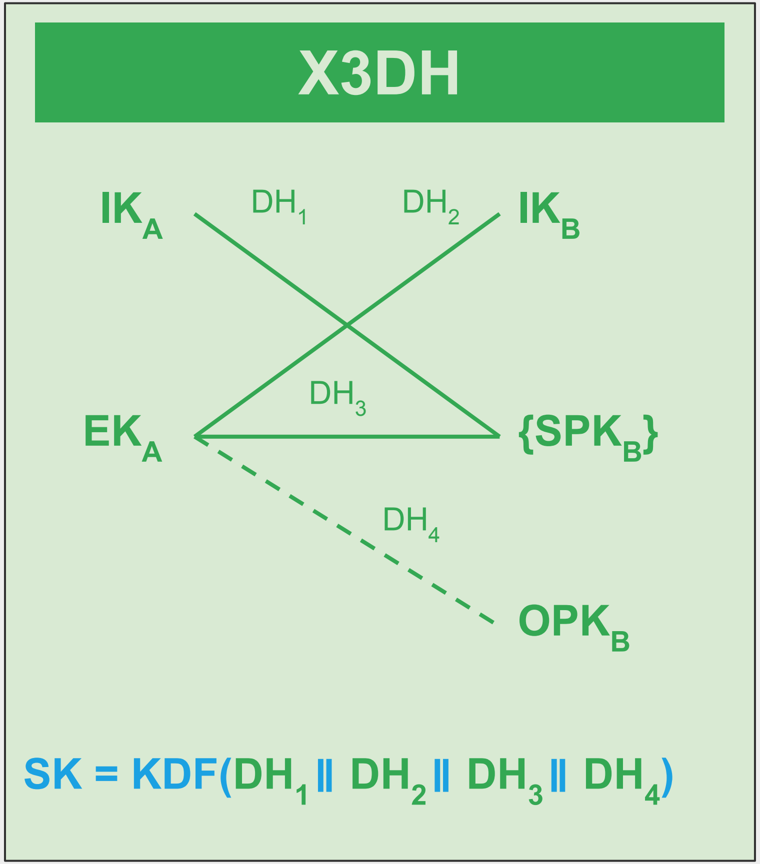
Scan this QR code with your phone to chat with me on Signal.

Signal Protocol 1: Before PQXDH

1. Perform 3 or 4 Elliptic Curve Diffie-Hellman agreements DH_1, DH_2, DH_3, DH_4 .
2. Feed DH_1, DH_2, DH_3, DH_4 a into a Key Derivation Function to attain a session secret SK.
3. Use AEAD to encrypt an initial message with Identity Keys as associated data:

   $AD = IK_A \,||\, IK_B$
   CT_{msg} = AEAD.Enc(SK, "hello", AD)
4. Send along with info about what keys were used:
   msg = (CT_{msg}, EK_A^{PK}, IK_A, SPK_B.id, OPK_B.id)

The receiver can compute SK and decrypt CT_{msg}.



Signal Protocol 1: PQXDH

1. Perform 3 or 4 Elliptic Curve Diffie-Hellman agreements DH_1, DH_2, DH_3, DH_4 .
2. **Use a KEM Encapsulation Key to encapsulate a new shared secret SS in CT_{KEM}.**
3. Feed DH_1, DH_2, DH_3, DH_4 and SS into a Key Derivation Function to attain a session secret SK.
4. Use AEAD to encrypt an initial message with Identity Keys as associated data:

   $AD = IK_A \,||\, IK_B$
   CT_{msg} = AEAD.Enc(SK, "hello", AD)
5. Send along with info about what keys were used:
   msg = (CT_{msg}, EK_A^{PK}, IK_A, CT_{KEM}, SPK_B.id, OPK_B.id)

The receiver can compute SK and decrypt CT_{msg}.