Designing a Post-Quantum Ratchet for Signal

Benedikt Auerbach
Graeme Connell
Yevgeniy Dodis
Shuichi Katsumata
Daniel Jost
Thomas Prest
Rolfe Schmidt

CAW 2025 Madrid

The Signal Protocol

The Signal Protocol

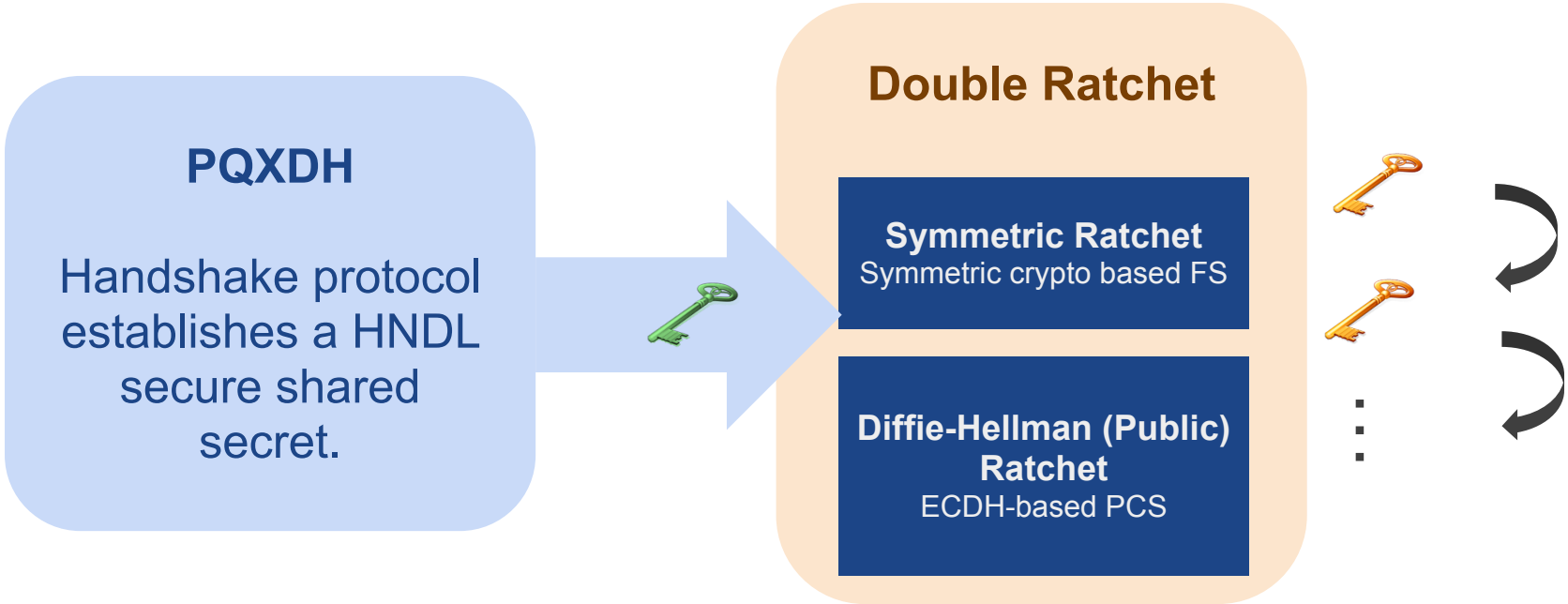
- Considered a standard for **two-user** Secure Messaging (SM)
- Used in Signal, WhatsApp, Google Messages, and more.
- **Not designed to be quantum safe.**

The Signal Protocol

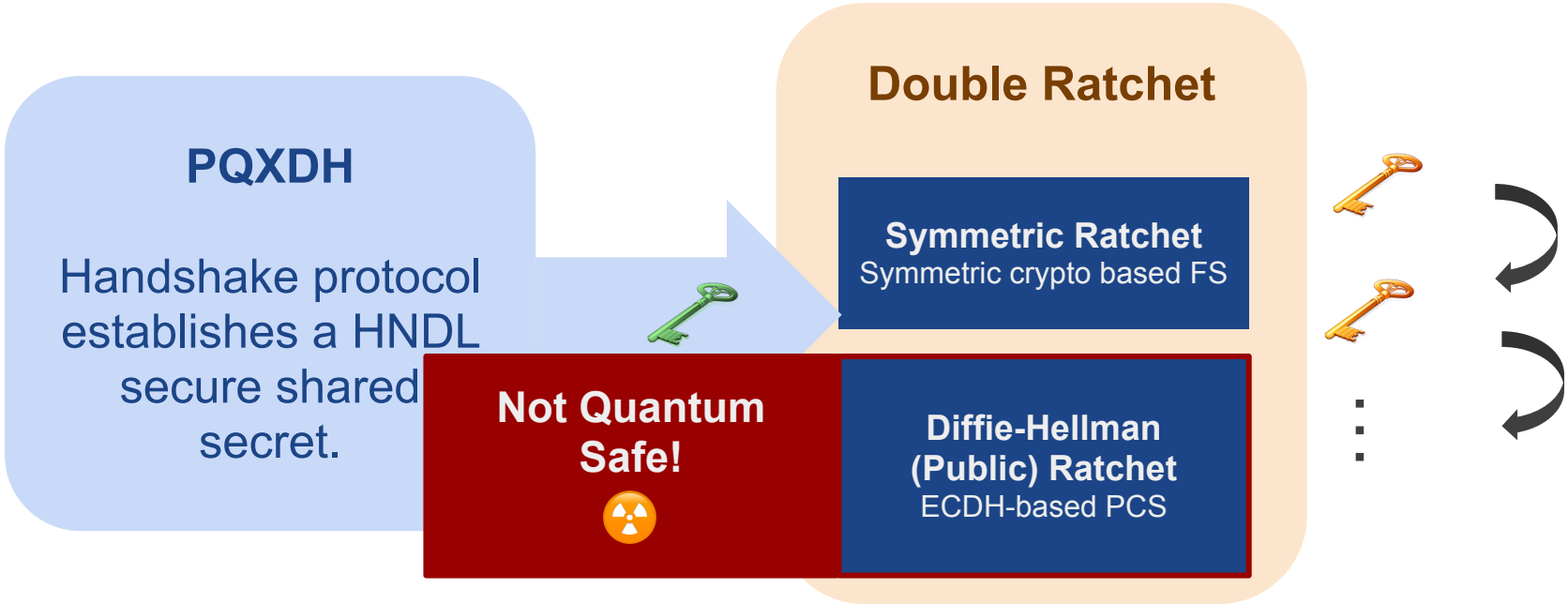
- Considered a standard for **two-user** Secure Messaging (SM)
- Used in Signal, WhatsApp, Google Messages, and more.
- **Not designed to be quantum safe.**

We want to be well ahead of the threat and ensure that our users' privacy is preserved in a post-quantum world.

Signal Protocol = PQXDH + Double Ratchet



Signal Protocol = PQXDH + Double Ratchet



We need quantum-safe PCS.

We need a new Public Ratchet.

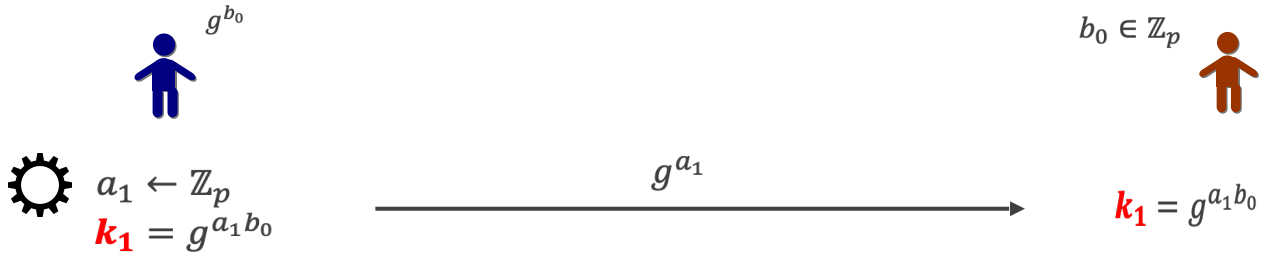
Post Quantum Secure Messaging

The Diffie-Hellman Ratchet

DH Ratchet is a **Continuous Key Agreement (CKA)** [EC:ACD19]

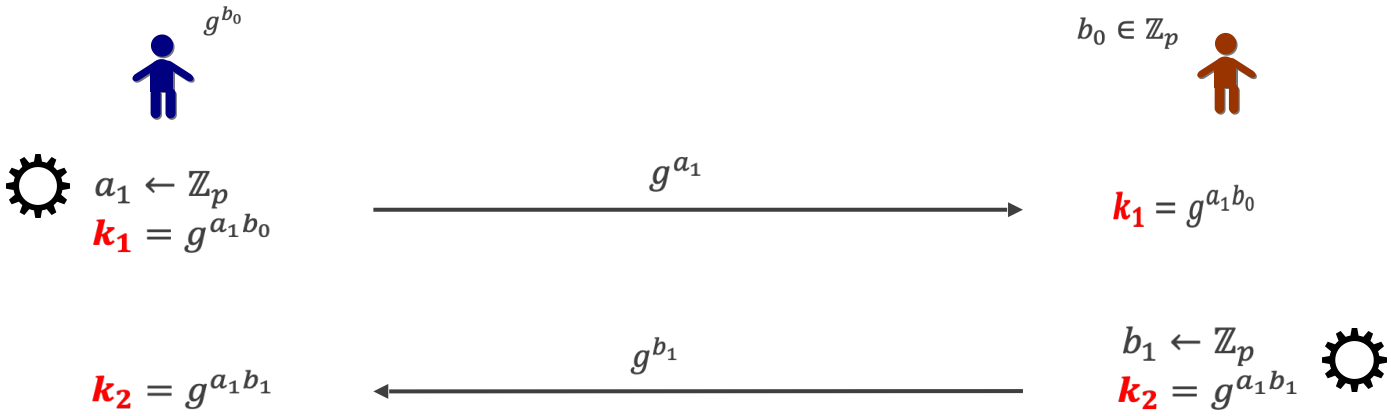
The Diffie-Hellman Ratchet

DH Ratchet is a **Continuous Key Agreement (CKA)** [EC:ACD19]



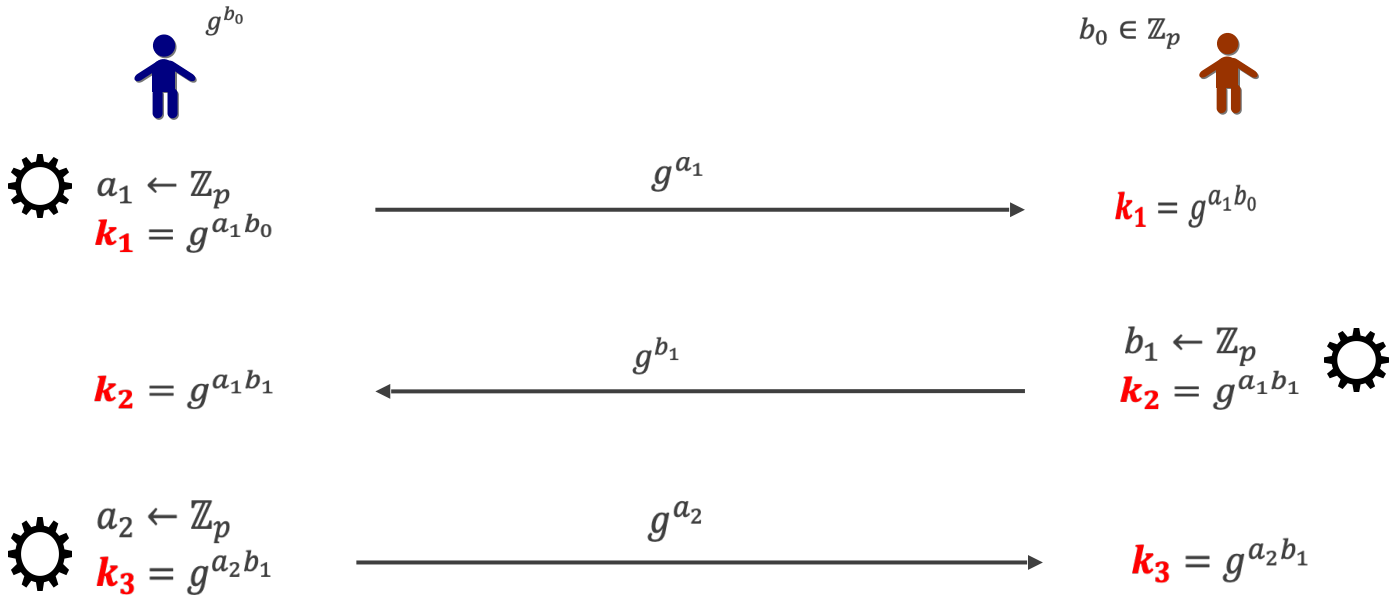
The Diffie-Hellman Ratchet

DH Ratchet is a **Continuous Key Agreement (CKA)** [EC:ACD19]



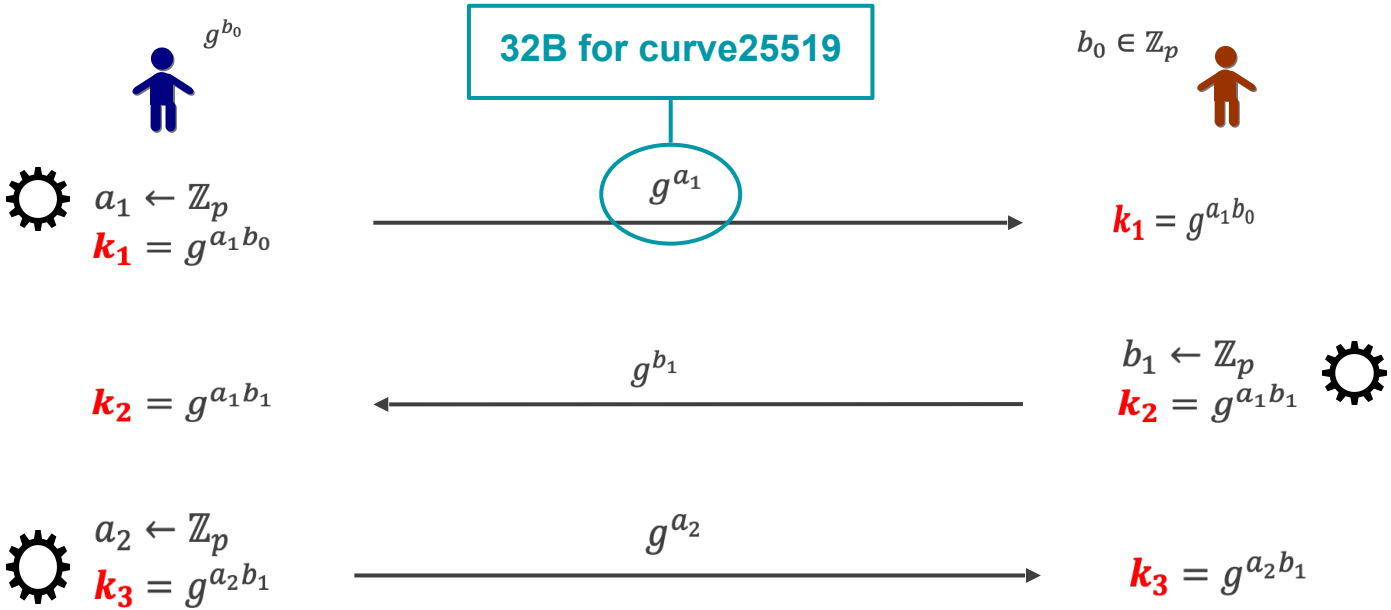
The Diffie-Hellman Ratchet

DH Ratchet is a **Continuous Key Agreement (CKA)** [EC:ACD19]



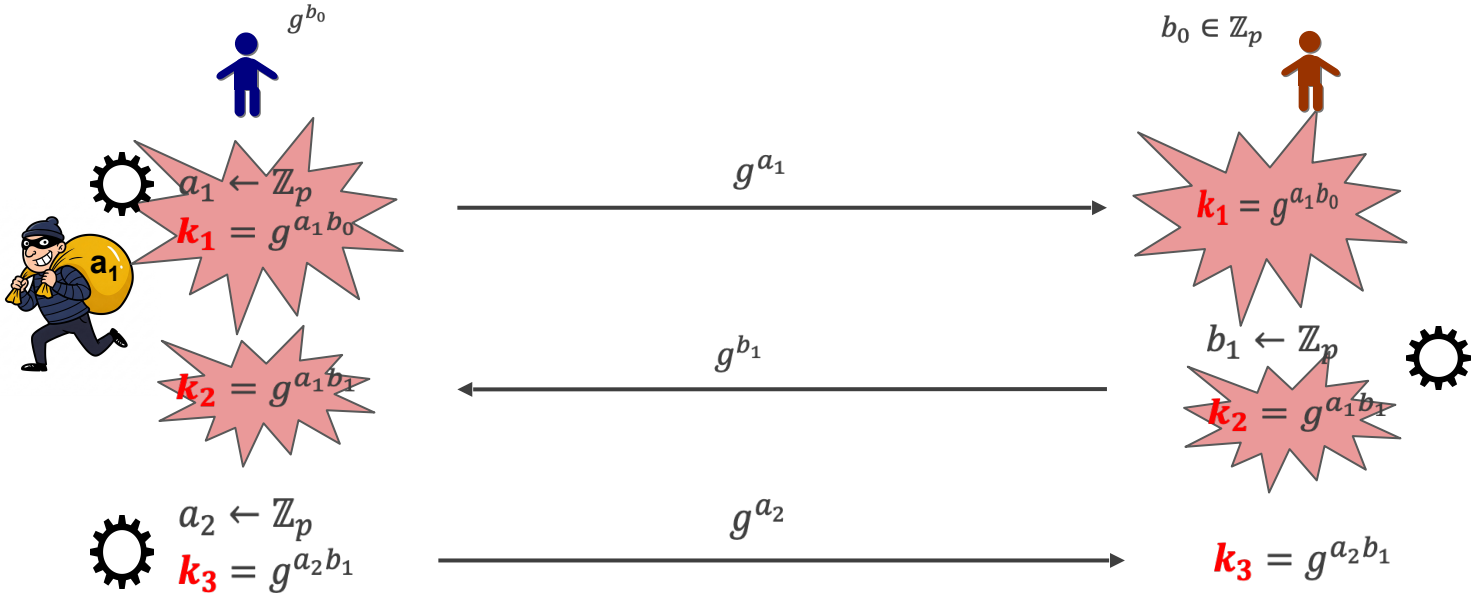
The Diffie-Hellman Ratchet

DH Ratchet is a **Continuous Key Agreement (CKA)** [EC:ACD19]



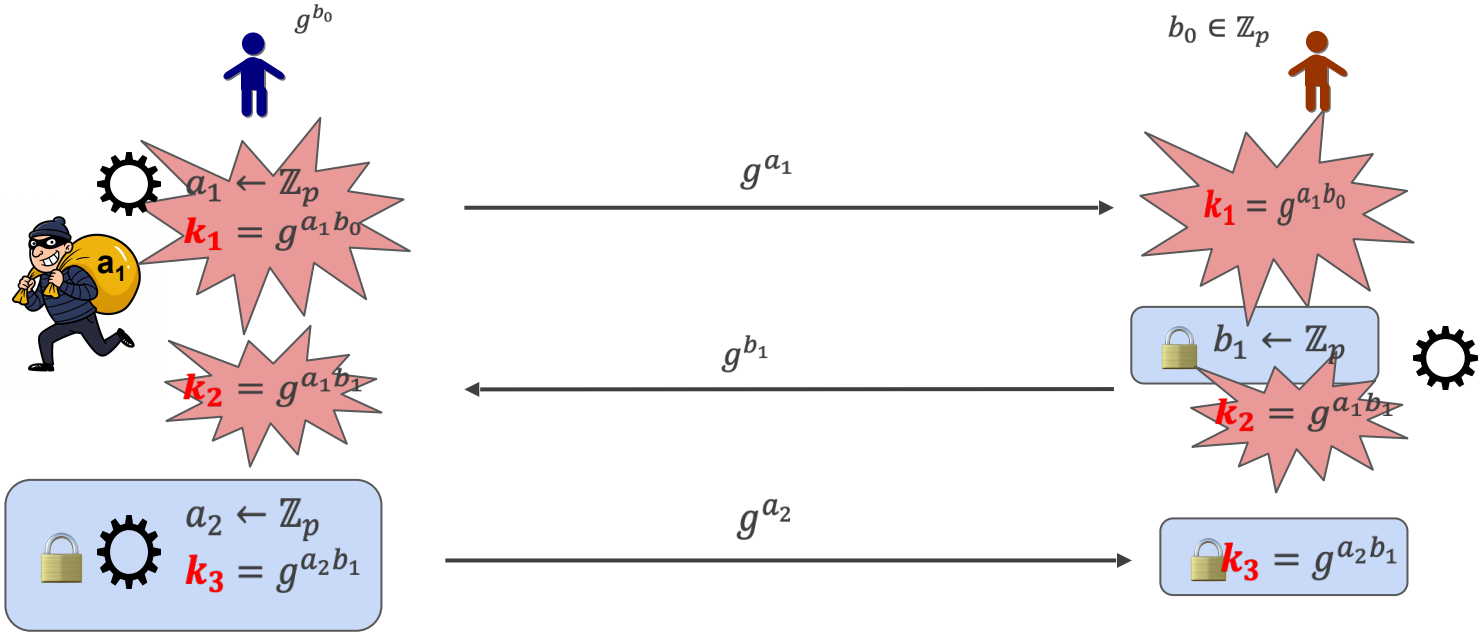
The Diffie-Hellman Ratchet

DH Ratchet is a **Continuous Key Agreement (CKA)** [EC:ACD19]



The Diffie-Hellman Ratchet

DH Ratchet is a **Continuous Key Agreement (CKA)** [EC:ACD19]



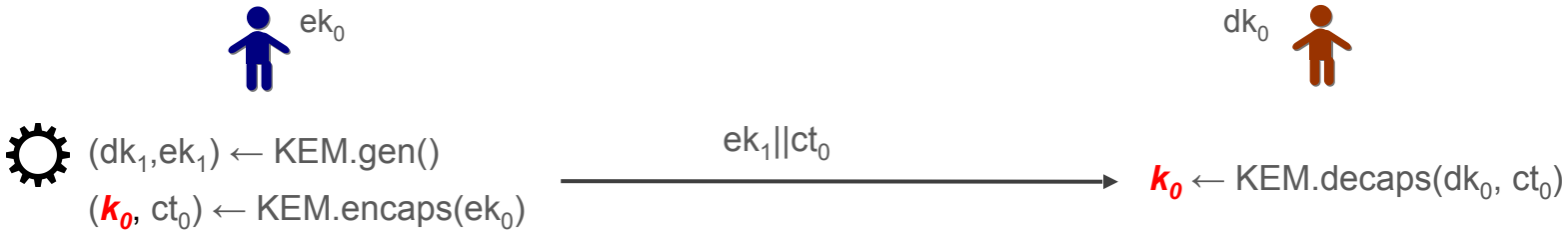
The Diffie-Hellman Ratchet

DH Ratchet is a **Continuous Key Agreement (CKA)** [EC:ACD19]

That’s Post Compromise Security (PCS).

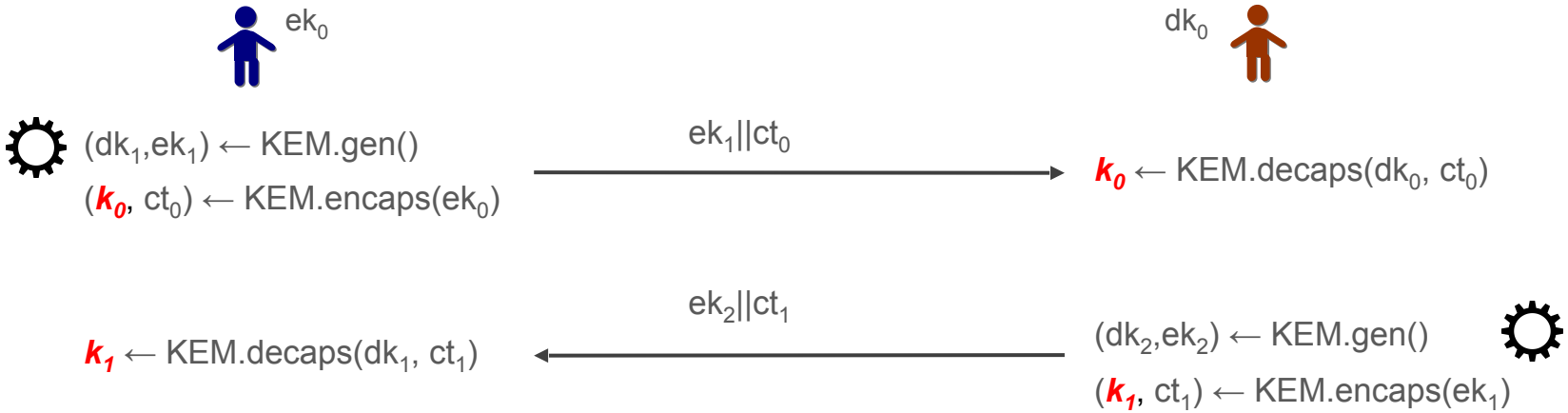
A Post Quantum Ratchet

We can also build **Continuous Key Agreement (CKA)** from a KEM (like ML-KEM) [EC:ACD19]



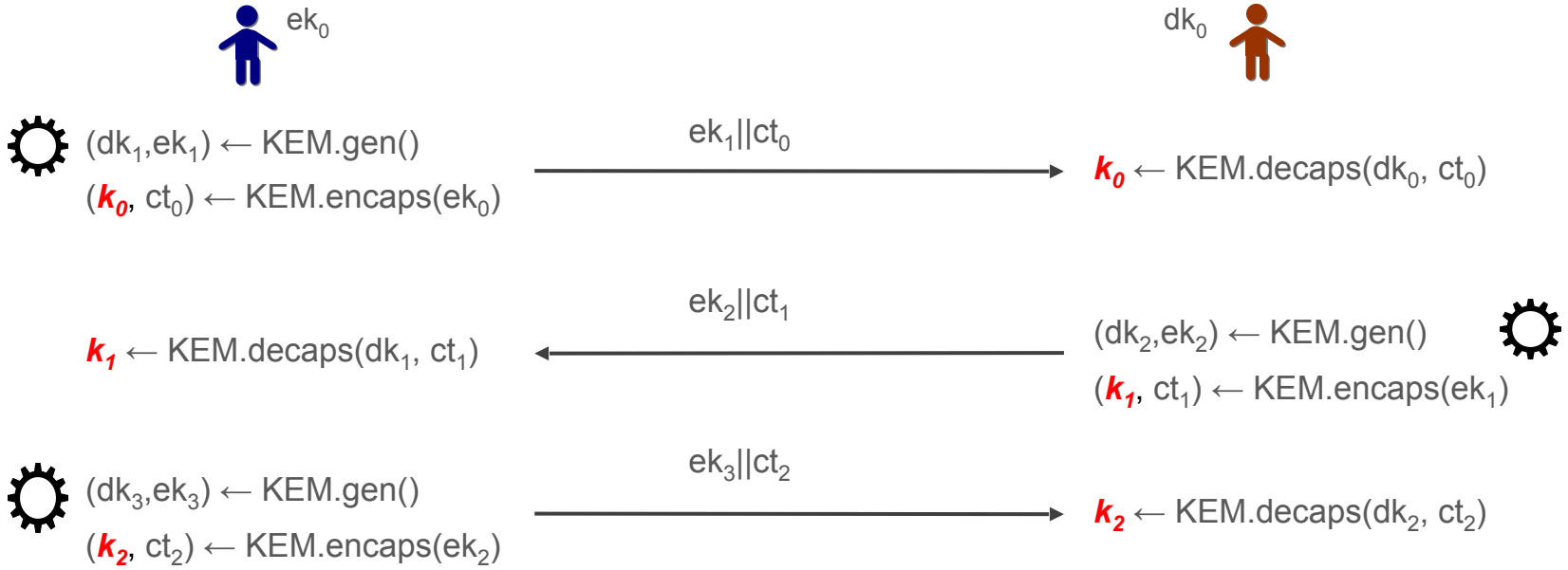
A Post Quantum Ratchet

We can also build **Continuous Key Agreement (CKA)** from a KEM (like ML-KEM) [EC:ACD19]



A Post Quantum Ratchet

We can also build **Continuous Key Agreement (CKA)** from a KEM (like ML-KEM) [EC:ACD19]



A Post Quantum Ratchet

We can also build **Continuous Key Agreem**
KEM (like ML-KEM) [EC:ACD19]

ek_0

$(dk_1, ek_1) \leftarrow KEM.gen()$

$(\boldsymbol{k_0}, ct_0) \leftarrow KEM.encaps(ek_0)$

$ek_1 || ct_0$

$ek_2 || ct_1$

$\boldsymbol{k_1} \leftarrow KEM.decaps(dk_1, ct_1)$

$(dk_3, ek_3) \leftarrow KEM.gen()$

$(\boldsymbol{k_2}, ct_2) \leftarrow KEM.encaps(ek_2)$

$ek_3 || ct_2$

$\boldsymbol{k_2} \leftarrow KEM.decaps(dk_2, ct_2)$

For ML-KEM 768 this is 2272 bytes.

Median DR message size today is 66 bytes - and that includes the 32B DH message.

Ouch.

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.

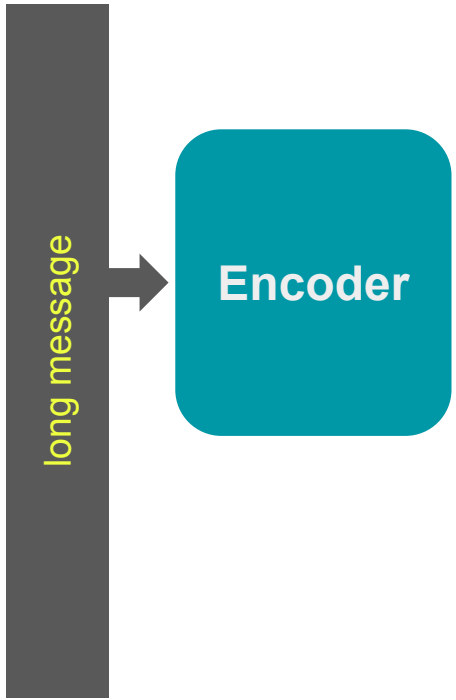
Working With Bandwidth Limits

Break a big message into small chunks.

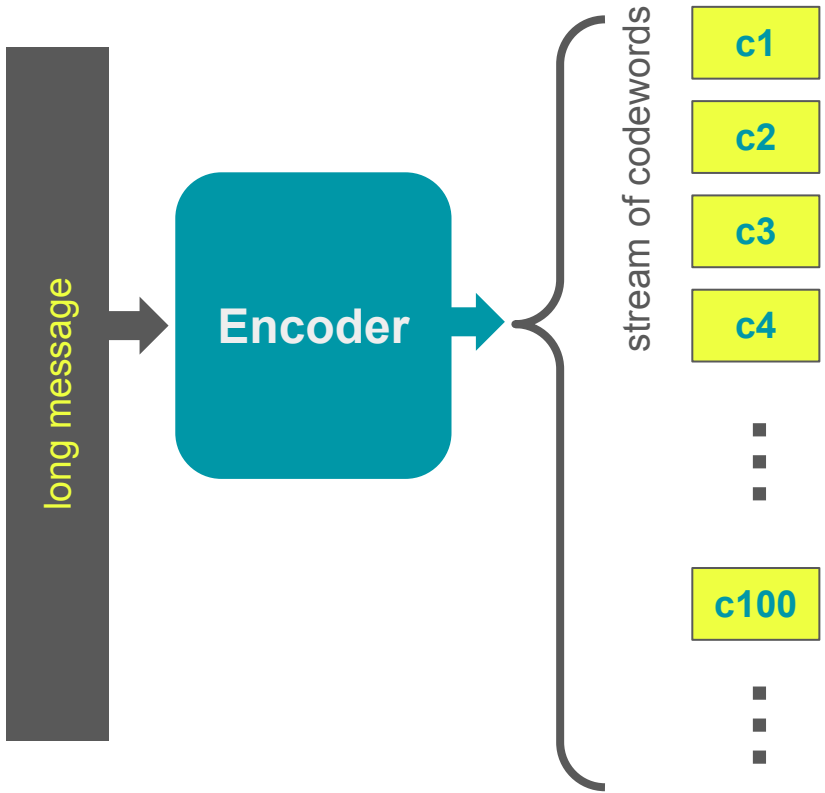
Send one chunk per message.

But it has to work even if messages are (maliciously) dropped!

Chunking with (Systematic) Erasure Codes



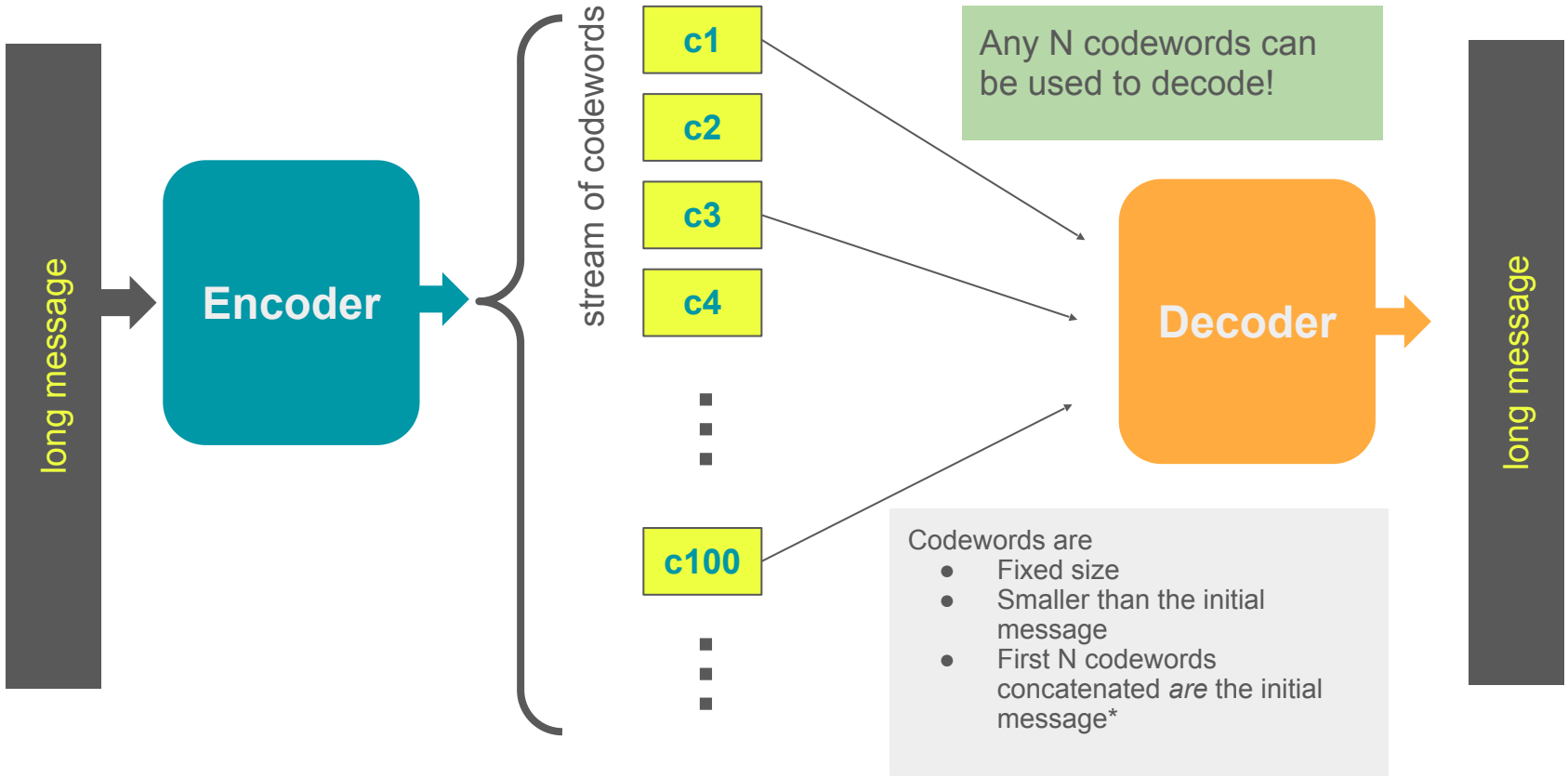
Chunking with (Systematic) Erasure Codes



Codewords are

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

Chunking with (Systematic) Erasure Codes

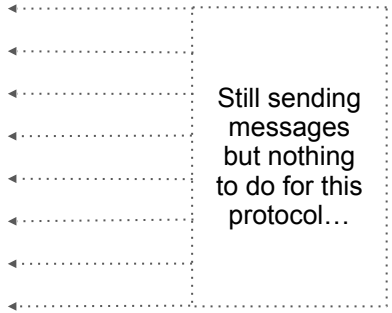


Secure Messaging with Sparse CKA

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.

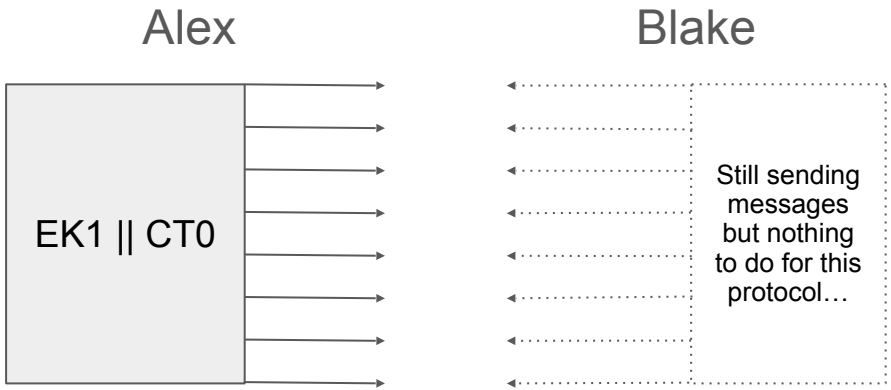


Secure Messaging with Sparse CKA

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.

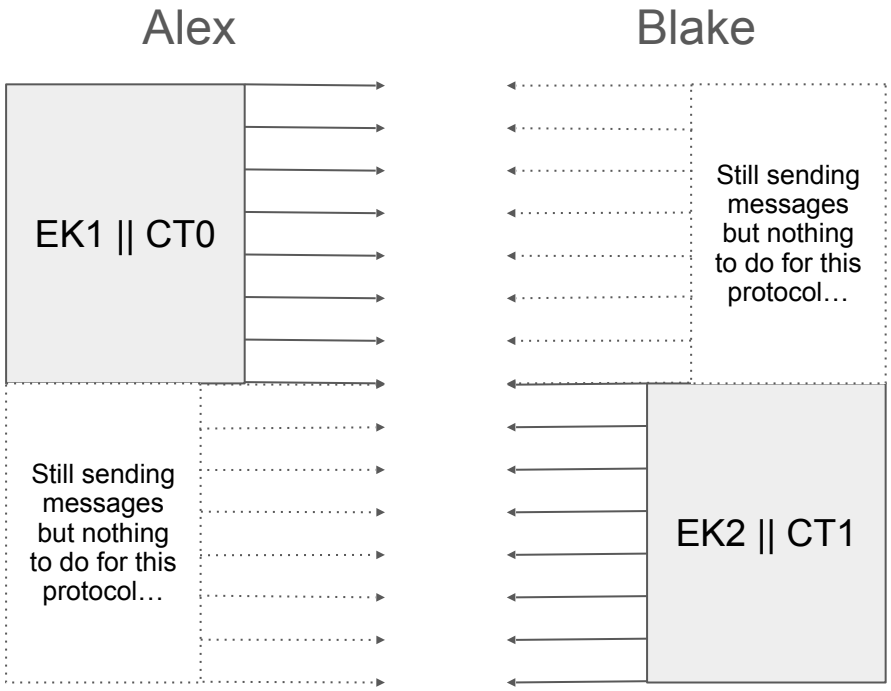


Secure Messaging with Sparse CKA

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.

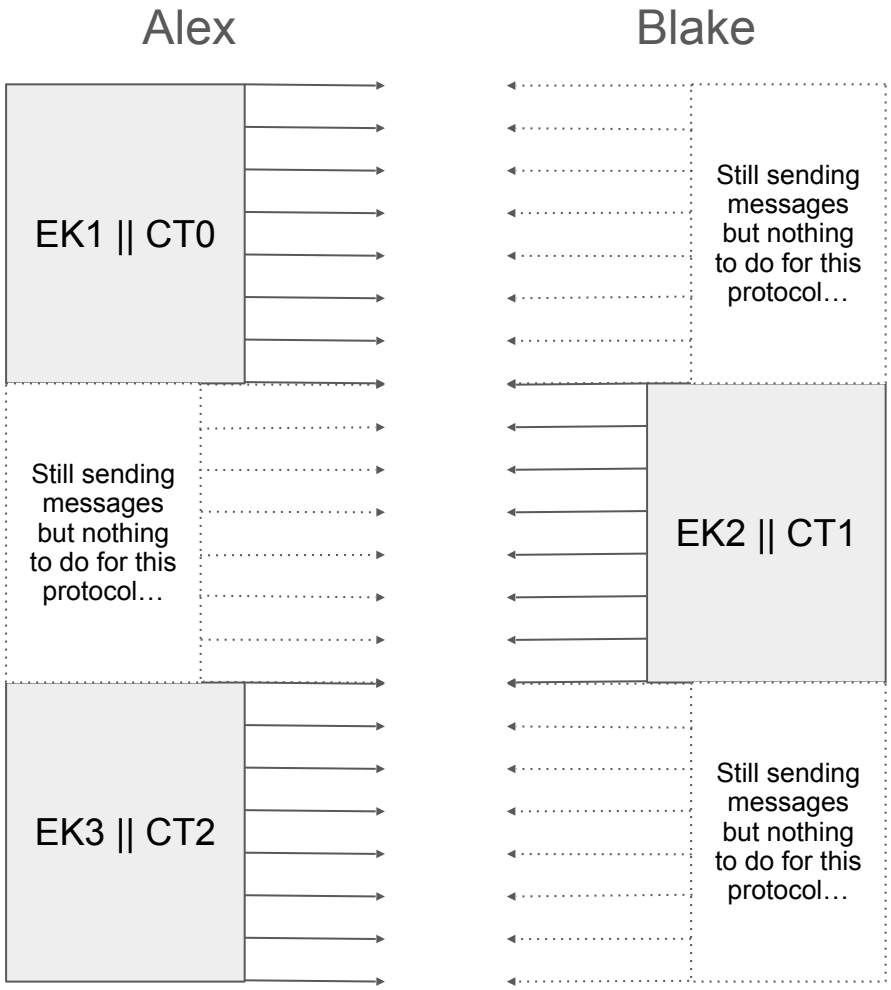


Secure Messaging with Sparse CKA

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.

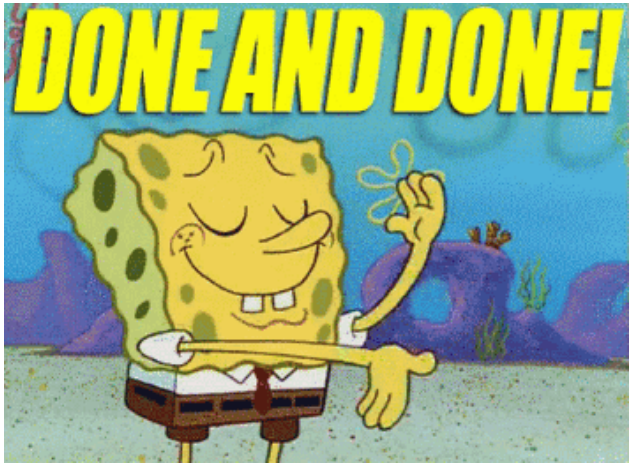


So we’re done?

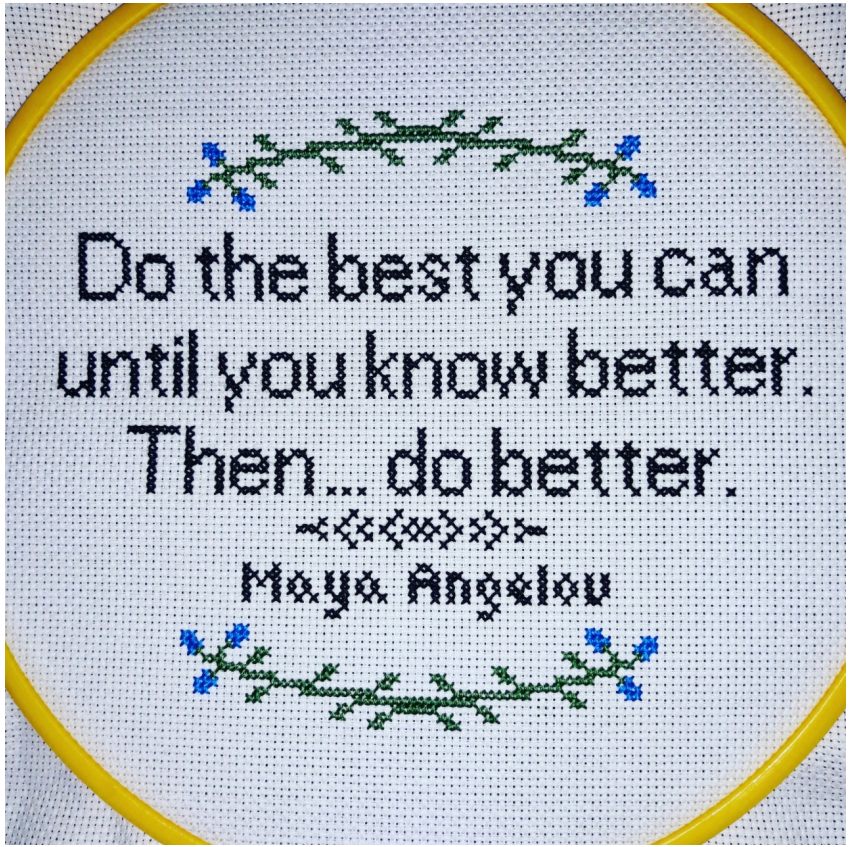
- Use ML-KEM to instantiate the KEM-based CKA from [EC:ACD19].
- Use our “chunking compiler” to turn it into an SCKA.
- Drop this into our SCKA-based Secure Messaging protocol to get messaging with MLWE-based security.
- Hybridize it with the classic double ratchet.

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.


Do the best you can
until you know better.
Then... do better.
Maya Angelou

Improving SCKA Protocols

The Problems

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

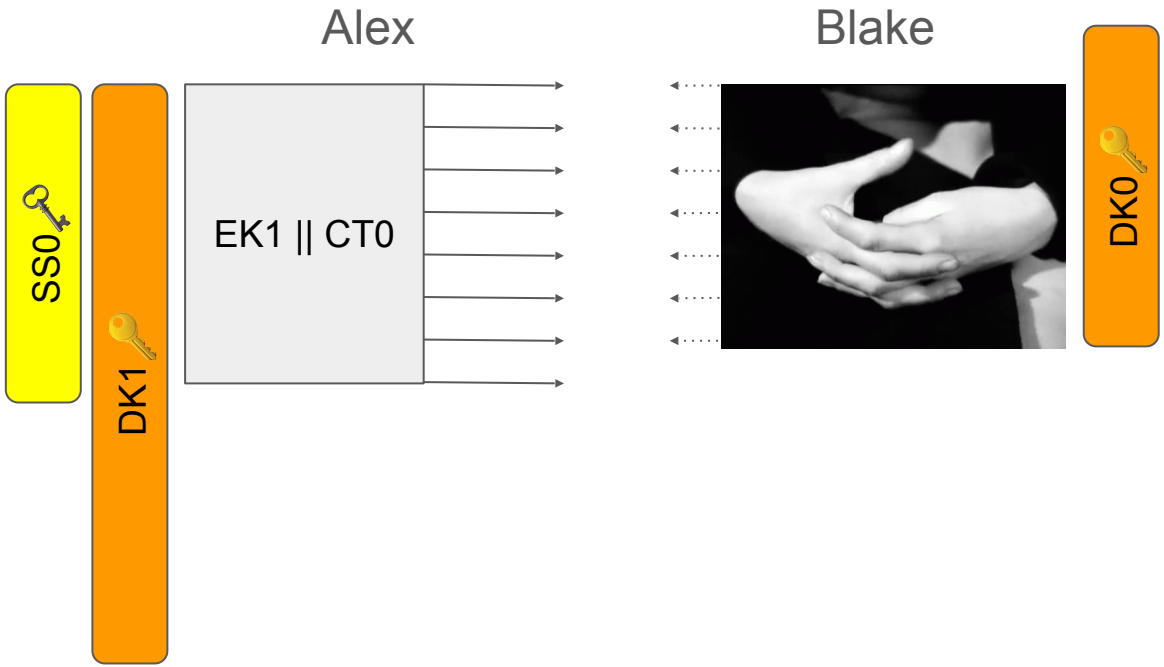
And look how long Alex and Blake have to hold onto their secrets. 😬.

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.

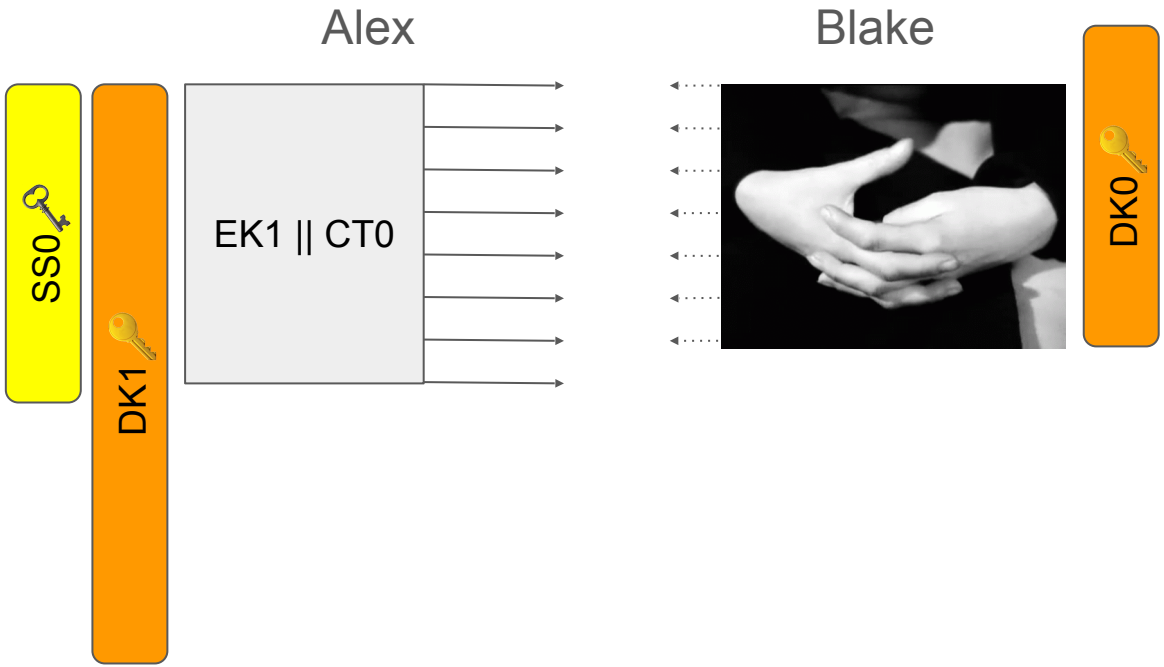
And look how long Alex and Blake have to hold onto their secrets. 😬.

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.

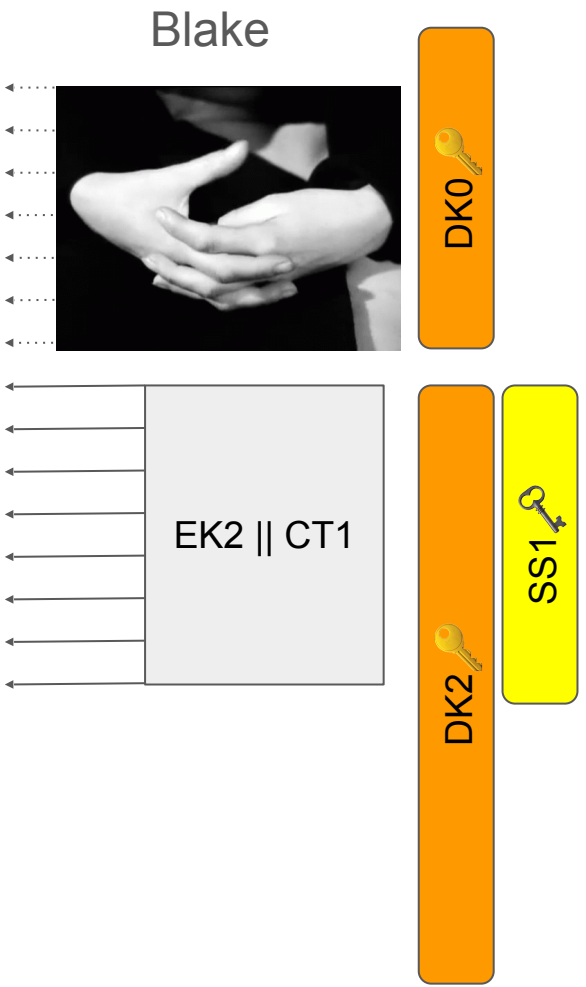
And look how long Alex and Blake have to hold onto their secrets. 😬.

Big attack surface, slow key emission.

Can’t they do *something*?

KEM Shared Secret 🔑

Decapsulation Key 🔑



The Problems

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

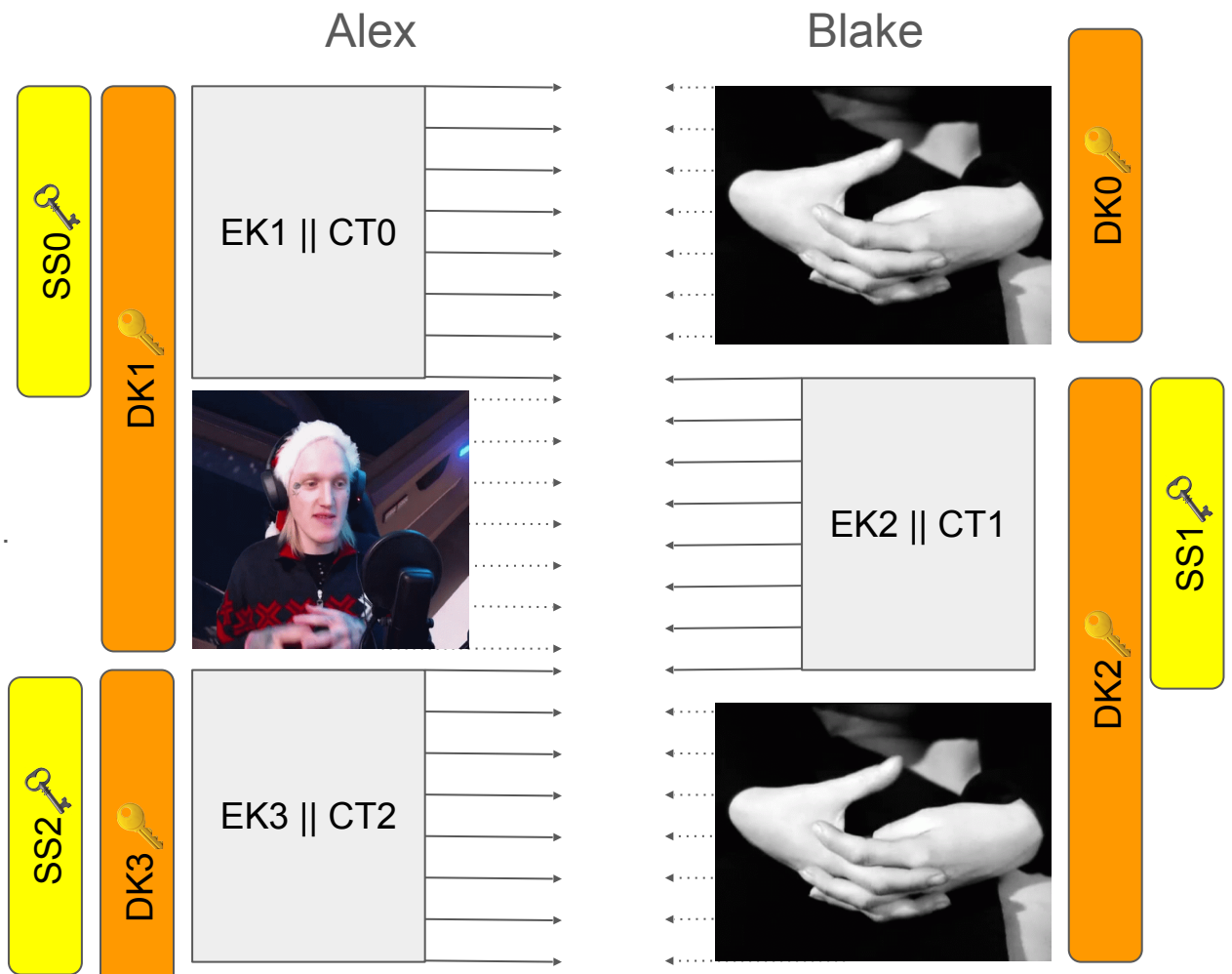
And look how long Alex and Blake have to hold onto their secrets. 😬.

Big attack surface, slow key emission.

Can’t they do *something*?

KEM Shared Secret 🔑

Decapsulation Key 🔑

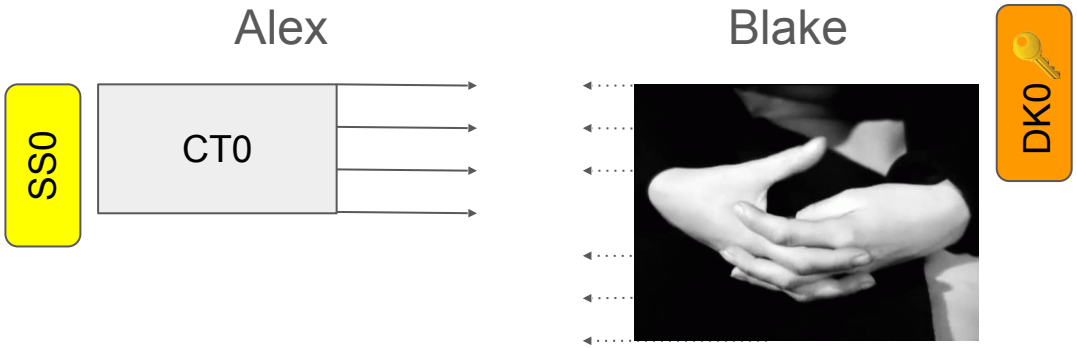


Competing ways to Do Better™:

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

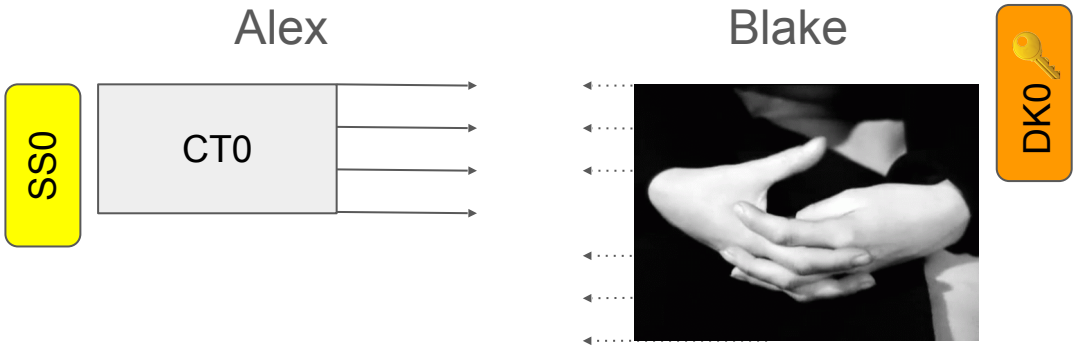
1. Reduce the Attack Surface

We can do a little better by sending EK and CT separately and minimize the time we need to store decapsulation keys



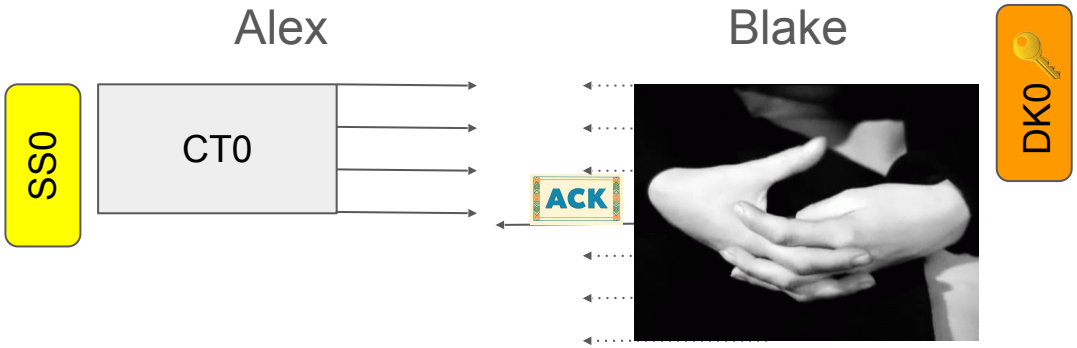
1. Reduce the Attack Surface

We can do a little better by sending EK and CT separately and minimize the time we need to store decapsulation keys



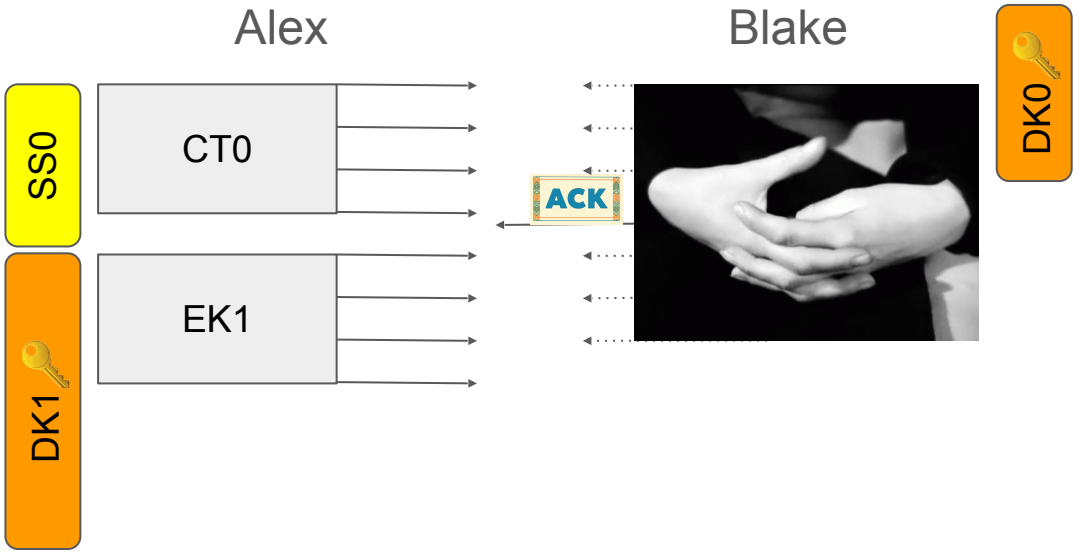
1. Reduce the Attack Surface

We can do a little better by sending EK and CT separately and minimize the time we need to store decapsulation keys



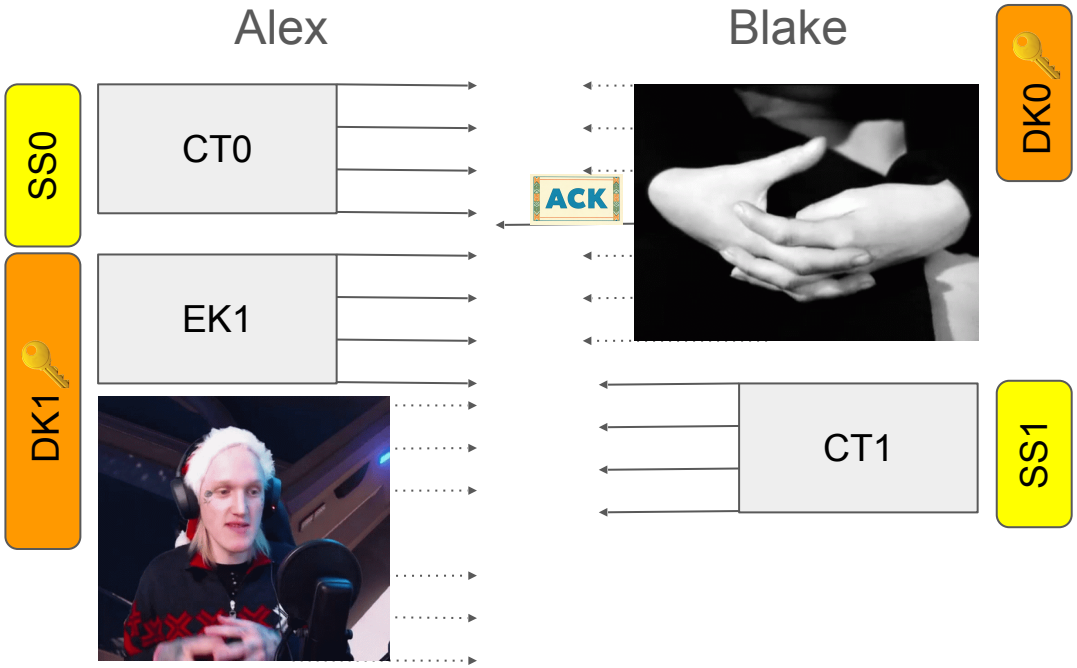
1. Reduce the Attack Surface

We can do a little better by sending EK and CT separately and minimize the time we need to store decapsulation keys



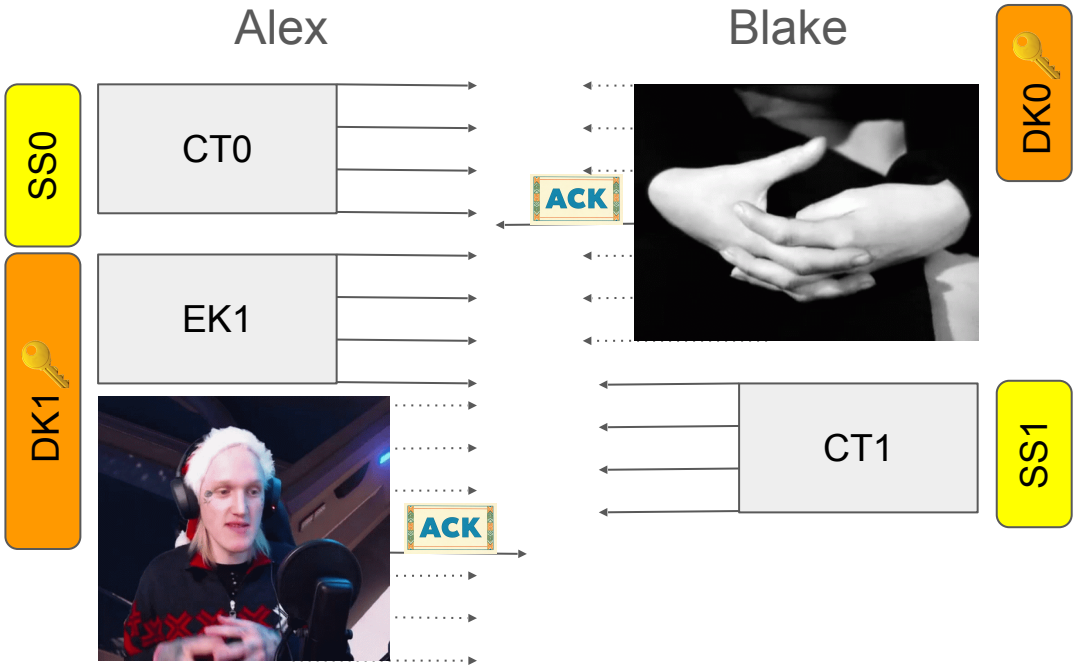
1. Reduce the Attack Surface

We can do a little better by sending EK and CT separately and minimize the time we need to store decapsulation keys



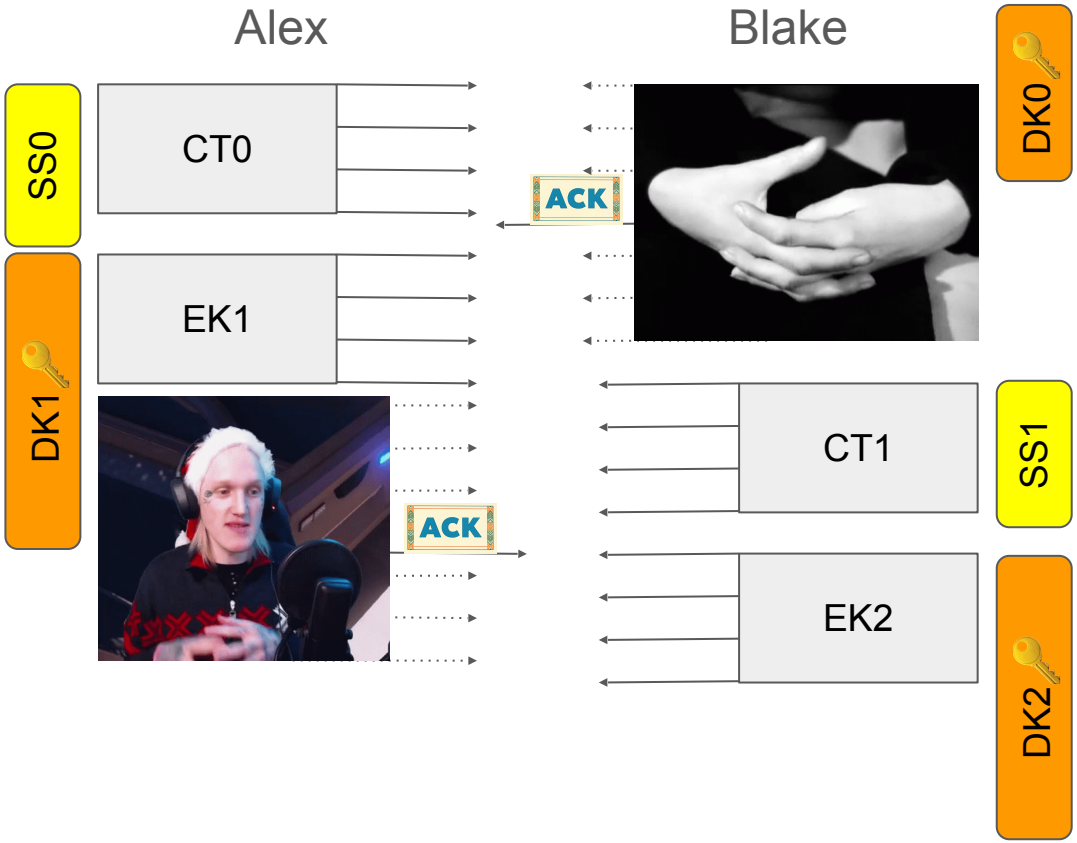
1. Reduce the Attack Surface

We can do a little better by sending EK and CT separately and minimize the time we need to store decapsulation keys



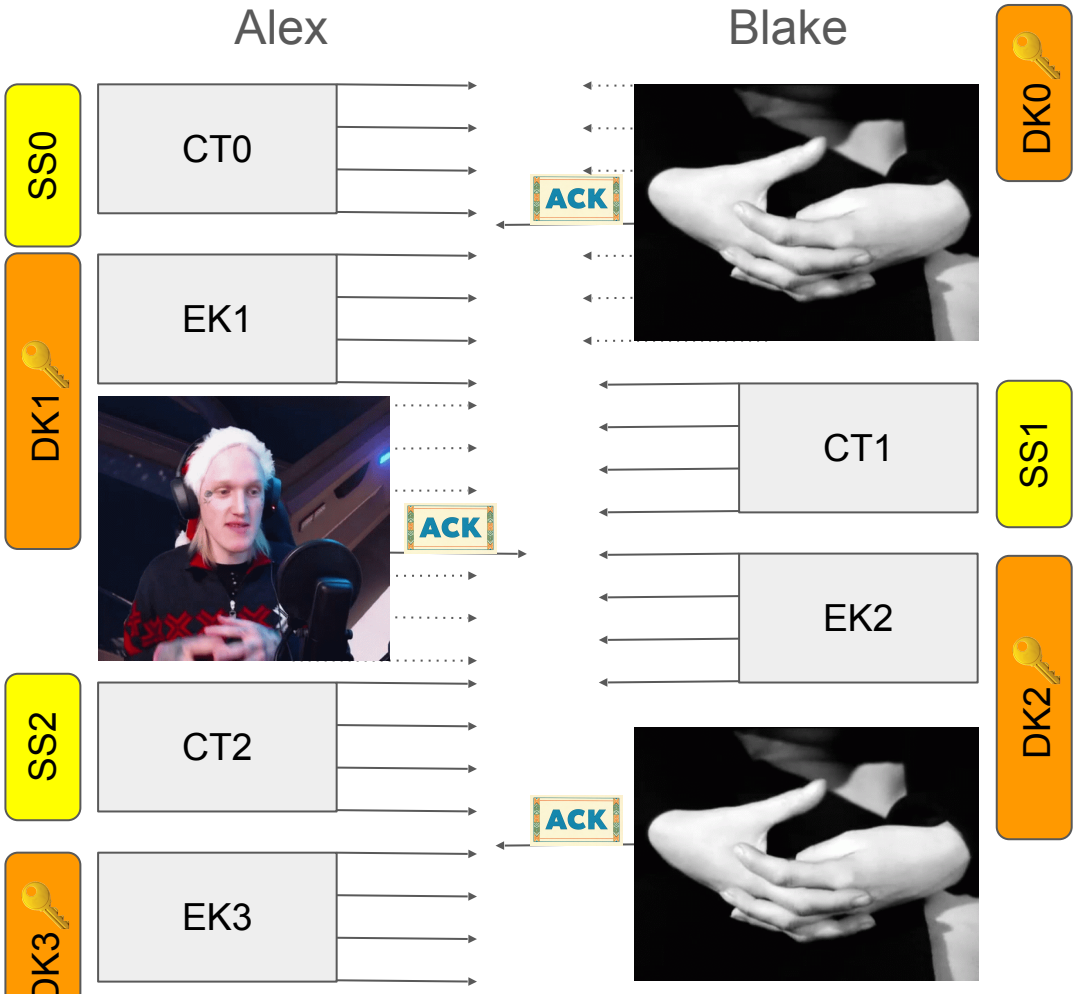
1. Reduce the Attack Surface

We can do a little better by sending EK and CT separately and minimize the time we need to store decapsulation keys



1. Reduce the Attack Surface

We can do a little better by sending EK and CT separately and minimize the time we need to store decapsulation keys

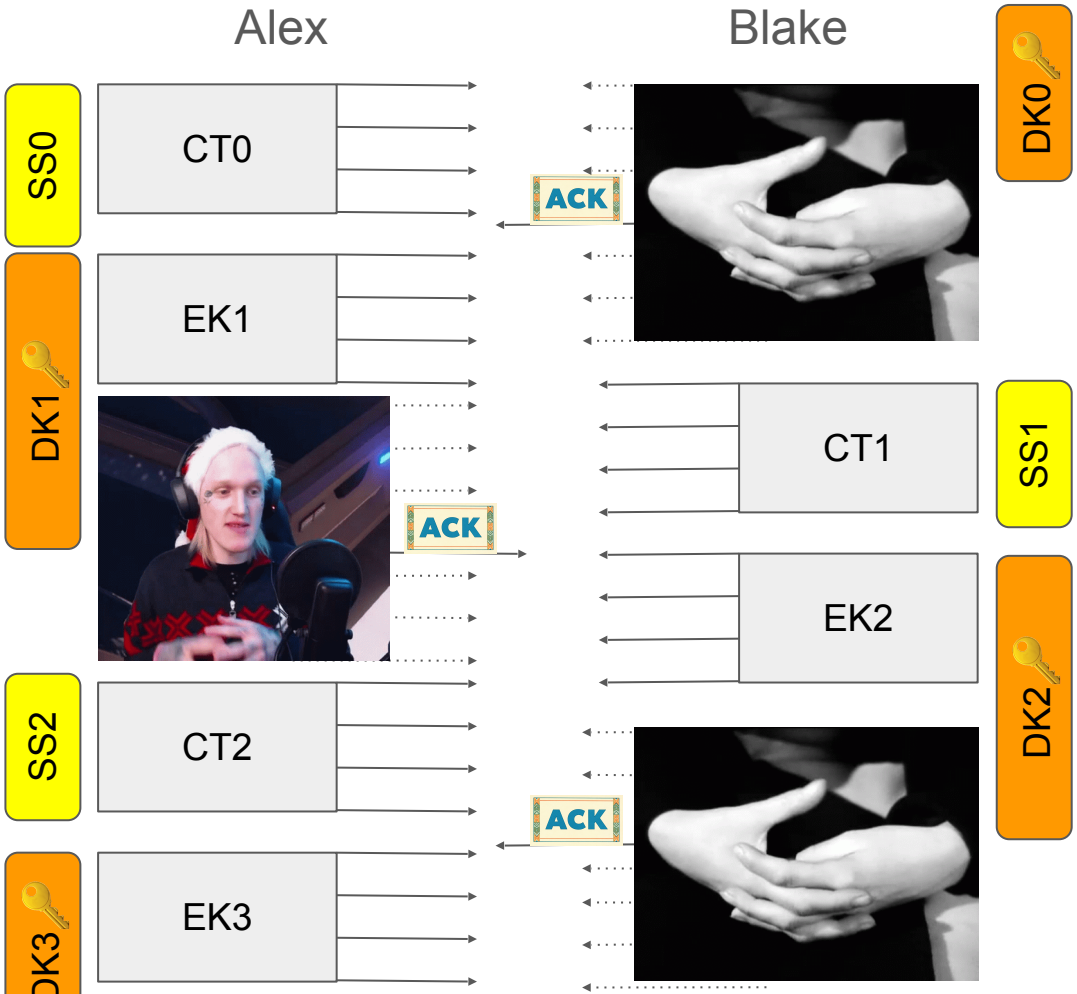


1. Reduce the Attack Surface

We can do a little better by sending EK and CT separately and minimize the time we need to store decapsulation keys

But notice! this protocol *seems* better, but it doesn't emit new shared secrets any faster on average.

🤔



Alex

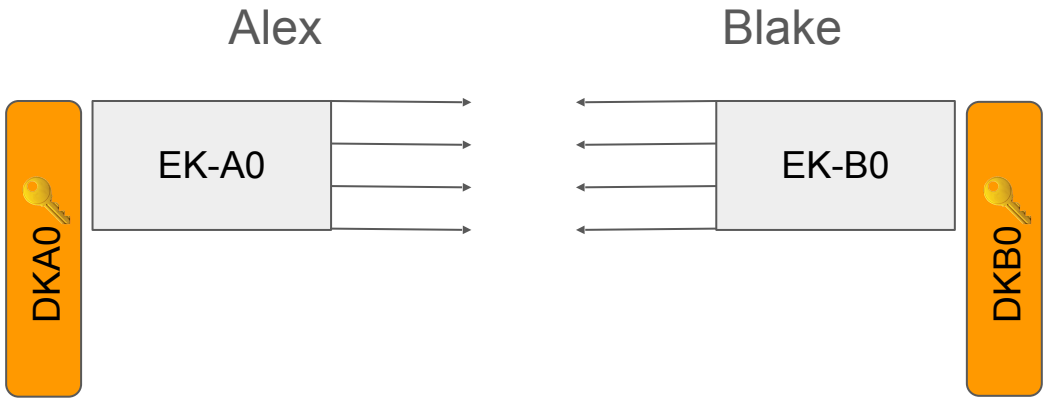
Blake

2. Sample Early!

We can emit keys almost twice as fast as before by having Blake just sample a new keypair when they are stuck and start sending.

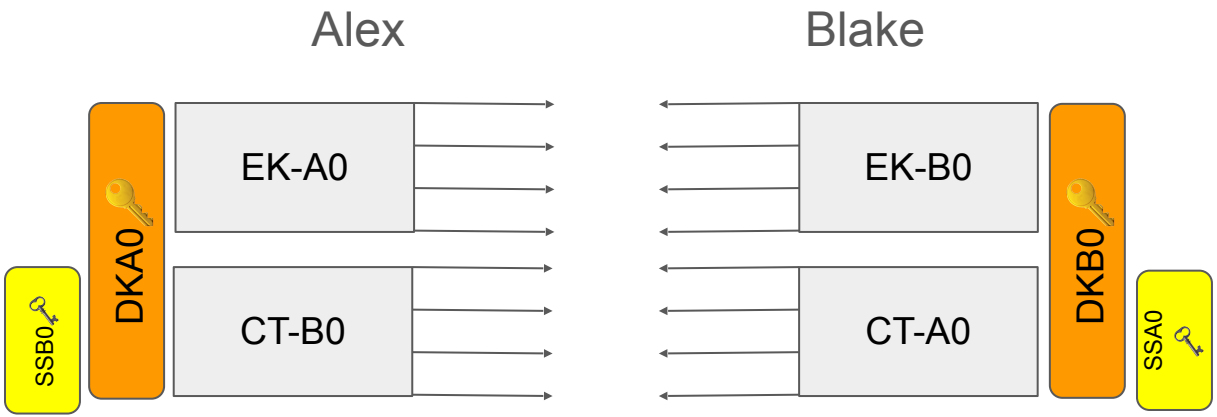
2. Sample Early!

We can emit keys almost twice as fast as before by having Blake just sample a new keypair when they are stuck and start sending.



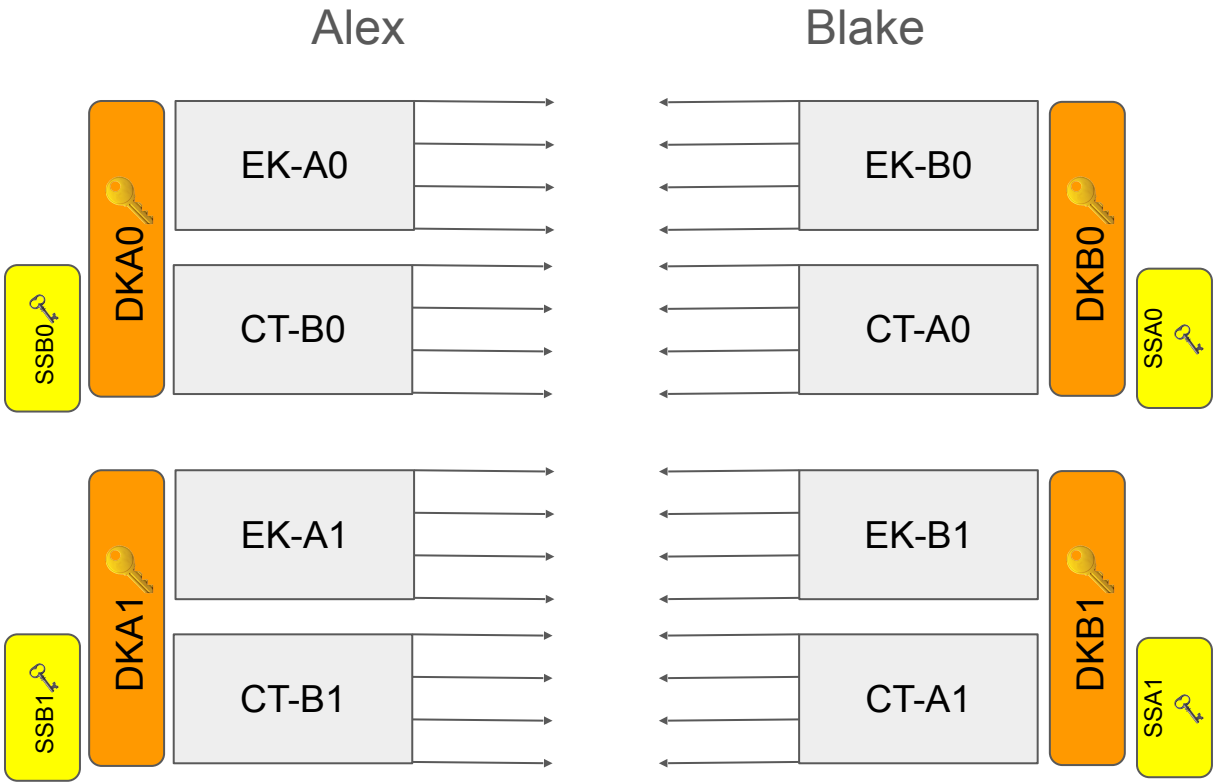
2. Sample Early!

We can emit keys almost twice as fast as before by having Blake just sample a new keypair when they are stuck and start sending.



2. Sample Early!

We can emit keys almost twice as fast as before by having Blake just sample a new keypair when they are stuck and start sending.

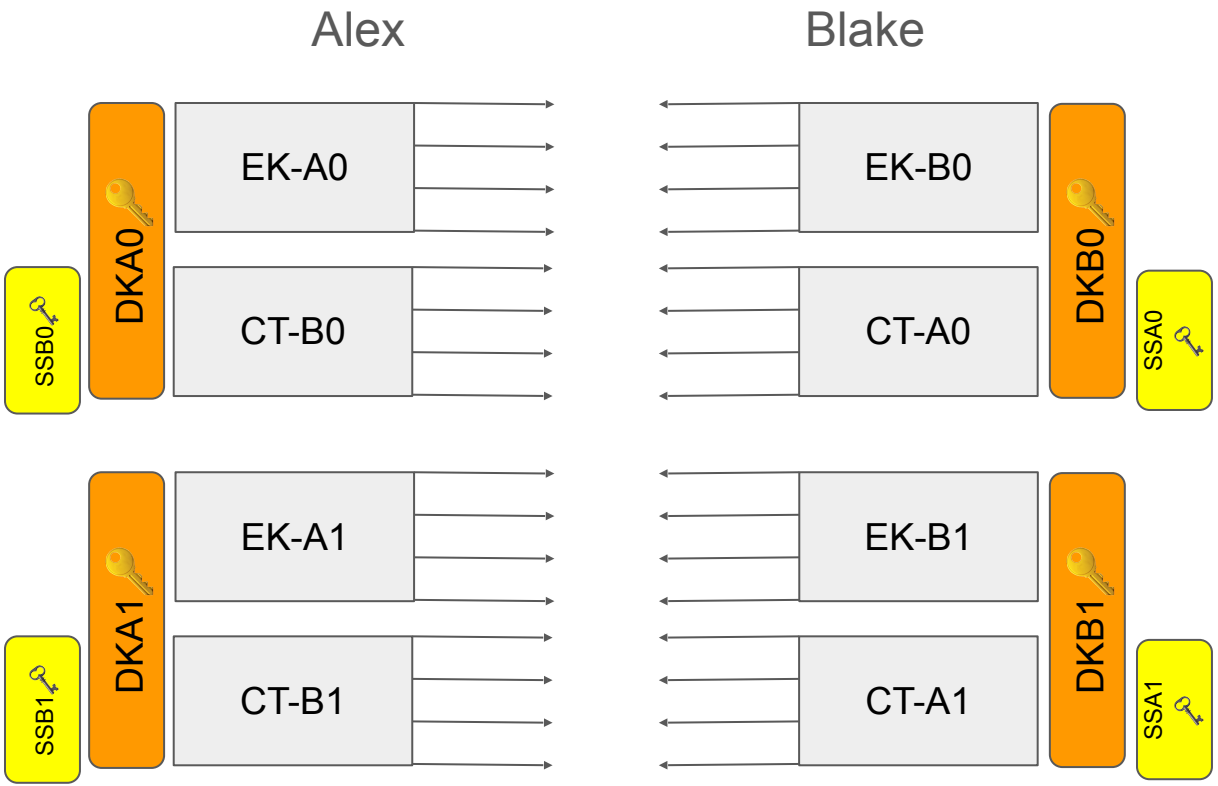


2. Sample Early!

We can emit keys almost twice as fast as before by having Blake just sample a new keypair when they are stuck and start sending.

But notice! this protocol emits keys almost 2x faster but the attacker also gets more secrets per compromise. Is it 2x as good?

🤔

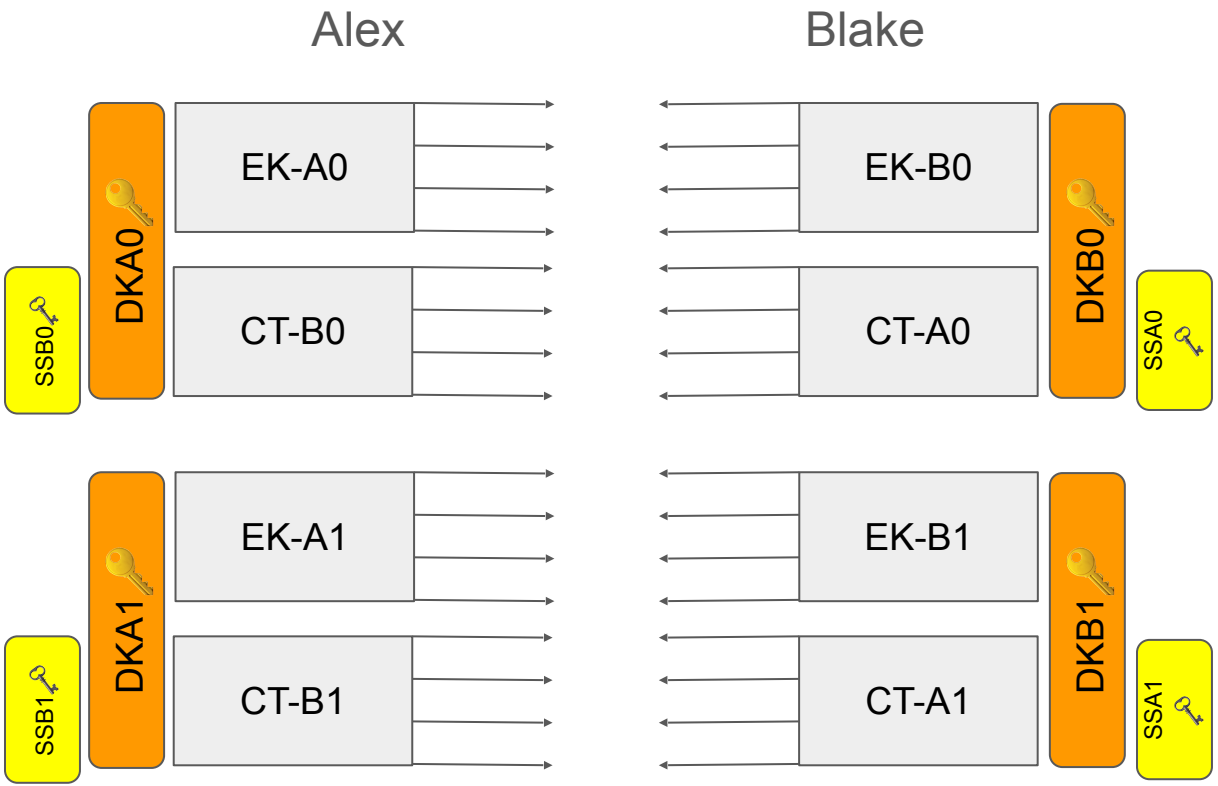


2. Sample Early!

We can emit keys almost twice as fast as before by having Blake just sample a new keypair when they are stuck and start sending.

But notice! this protocol emits keys almost 2x faster but the attacker also gets more secrets per compromise. Is it 2x as good?

🤔



And how do they agree on the order of the keys? What happens when messaging is unbalanced? Et cetera…

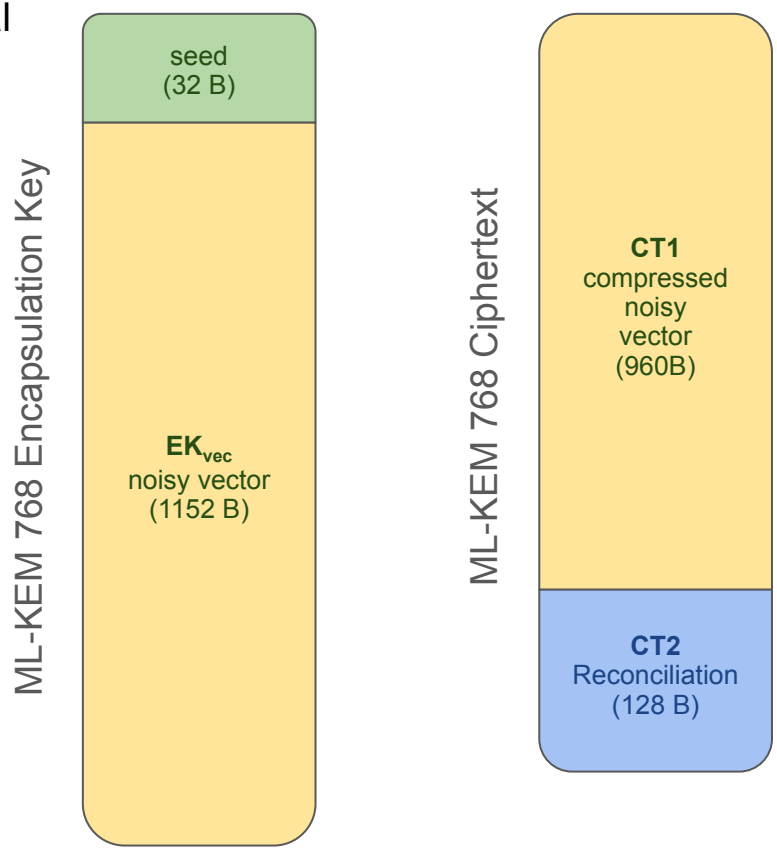
3. Open the KEM Black Box: Incremental KEM

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”, $\mathbf{A}^T\mathbf{s'} + \mathbf{e'}$, where **s’** is a decapsulation secret and **e’** is small error.
2. A “reconciliation message”



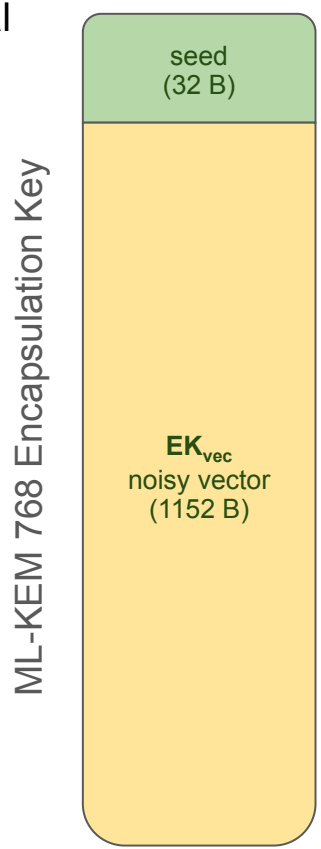
3. Open the KEM Black Box: Incremental KEM

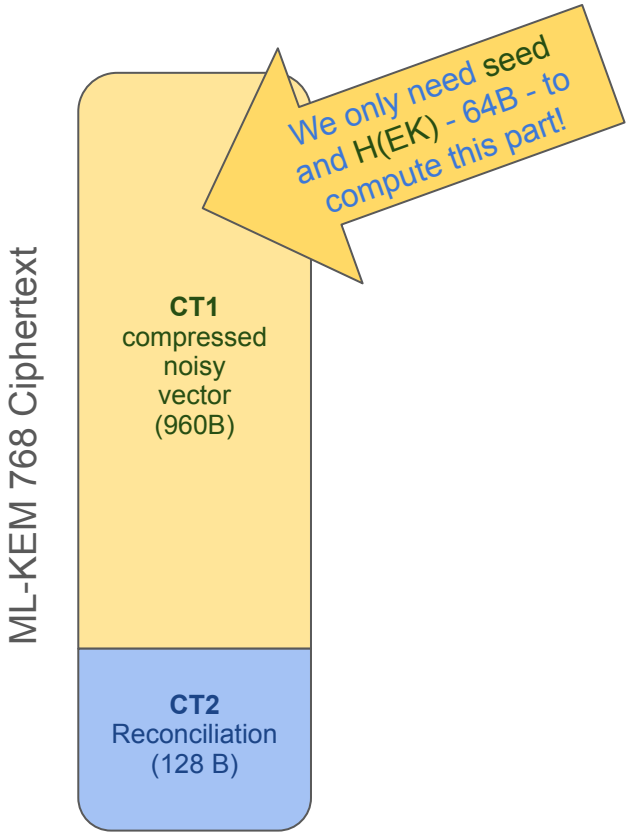
An ML-KEM Encapsulation key has two parts:

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”, $\mathbf{A}^T\mathbf{s'} + \mathbf{e'}$, where **s’** is a decapsulation secret and **e’** is small error.
2. A “reconciliation message”





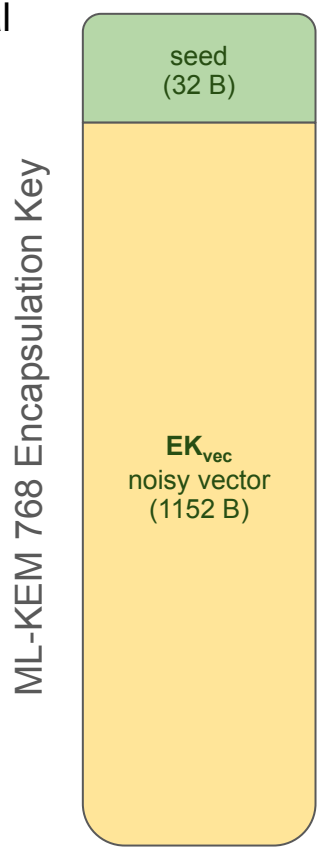
3. Open the KEM Black Box: Incremental KEM

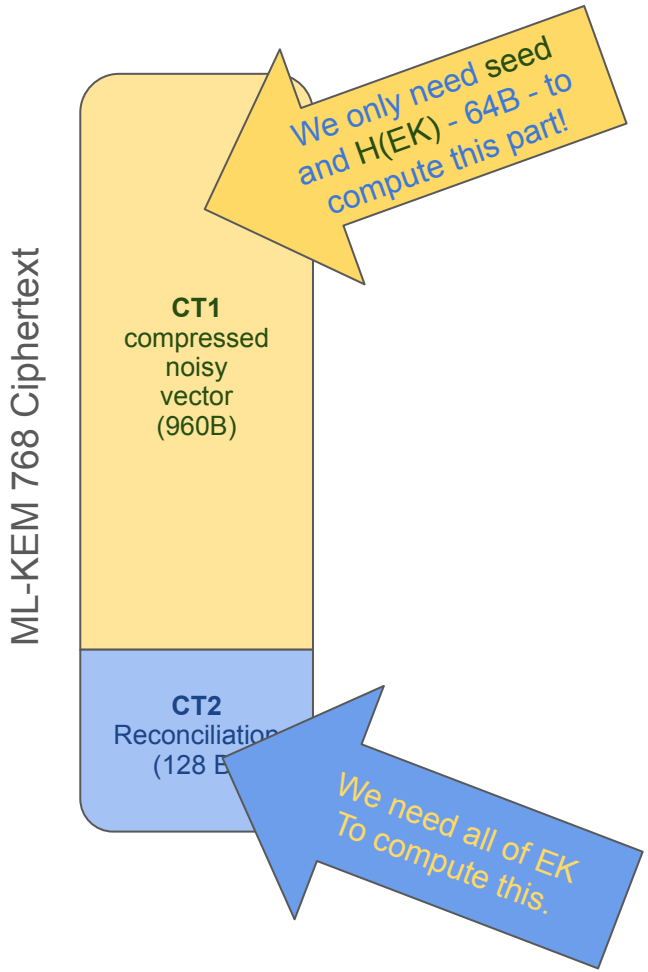
An ML-KEM Encapsulation key has two parts:

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”, $\mathbf{A}^T\mathbf{s'} + \mathbf{e'}$, where **s’** is a decapsulation secret and **e’** is small error.
2. A “reconciliation message”





Idea: Now we can sample CT1 instead of a new keypair.

Also: Ratcheting KEM [EC:DJKP25]. See our talk later this week!

The Protocol Design Space

The design space is large!

- What sort of KEM (Standard, Ratcheting, Incremental)?
- When do you sample a new keypair?
- If you can send an Encapsulation Key or a Ciphertext, which one do you prioritize?
- When do you send extra data (ACKs) to let the other know what state we're in?

How to choose?

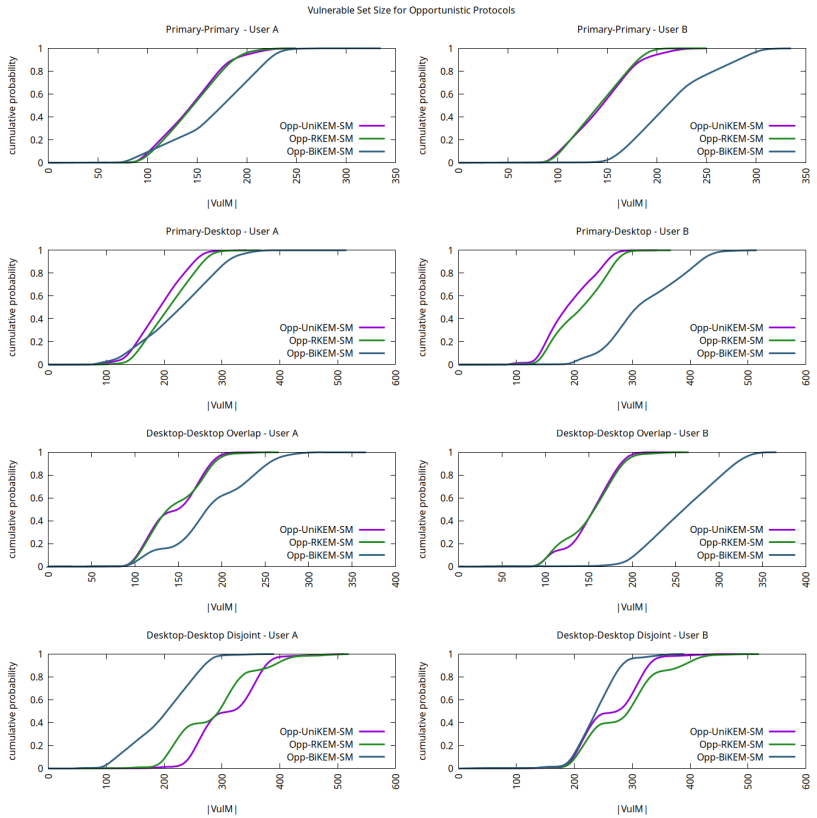
- We evaluated over a dozen candidate protocols.
- Our metric is the size of the **Vulnerable Message Set**: number of messages were leaked to an attacker by a compromise.
  - This is *why* we want a “small attack surface” or “fast key emission”.
- This depends on messaging behavior!
  - Balanced or unbalanced?
  - How often are parties online?
- Developed statistical models of typical behaviors and simulated attacks to compare distributions.

How to choose?

It turns out protocols aren't comparable by this metric!

But evaluating many protocols in a variety of realistic conditions, two basic protocol types consistently revealed fewer messages to an attacker:

- Opportunistic Katana ⚔️(see our EC talk!)
- Opportunistic Incremental KEM



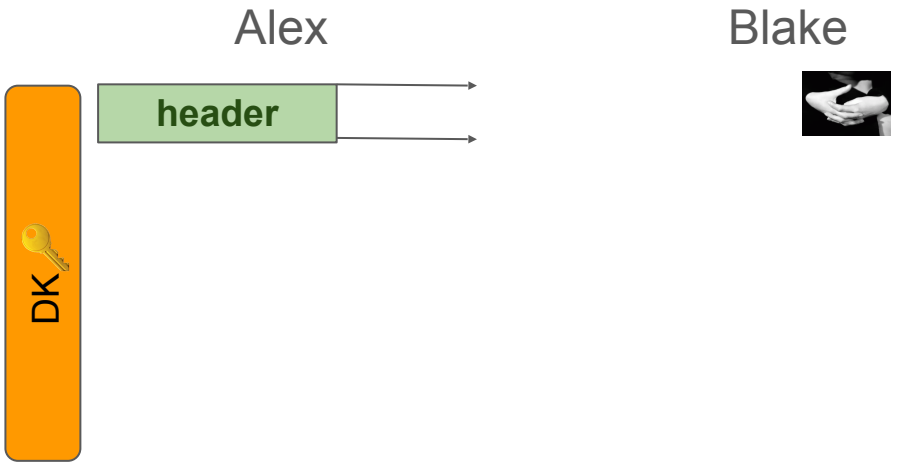
Here’s what we chose.

The ML-KEM Braid

Main Idea: Send “header” with **seed** and **H(EK)** first so Blake can sample **CT1** early.

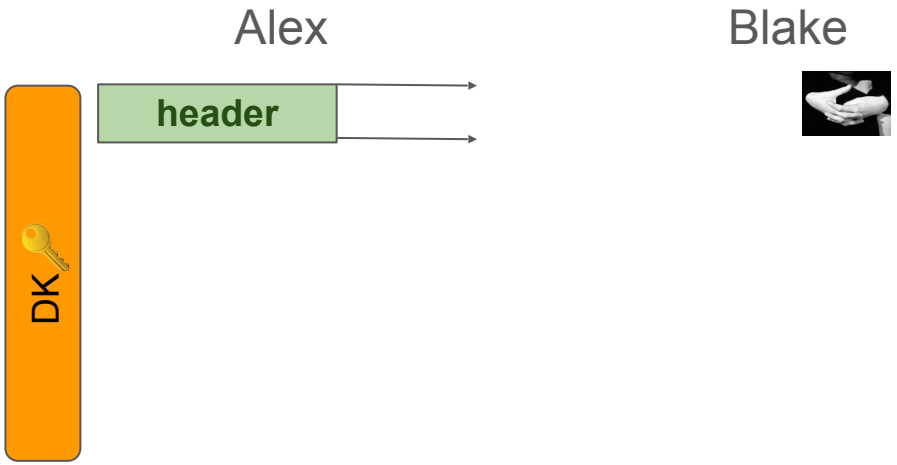
We can do most of the work in parallel!

- 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.



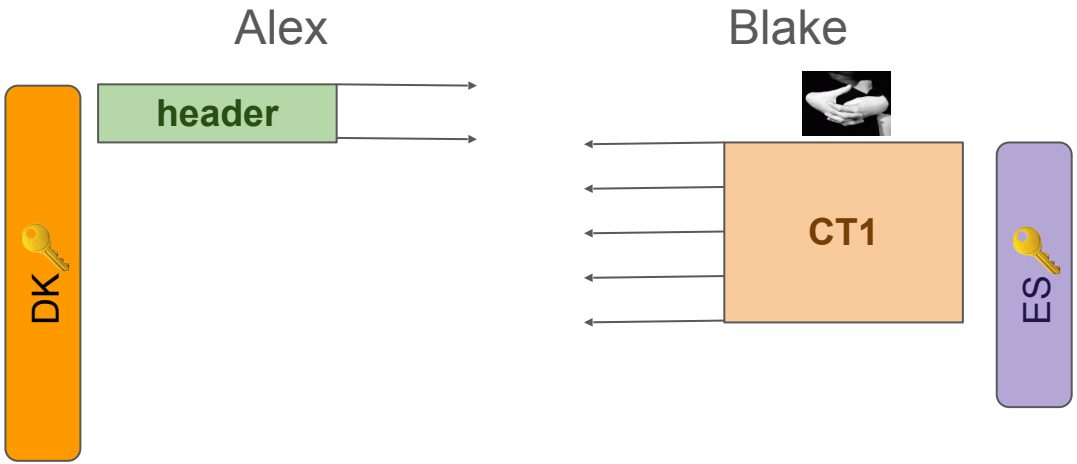
We can do most of the work in parallel!

- 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.



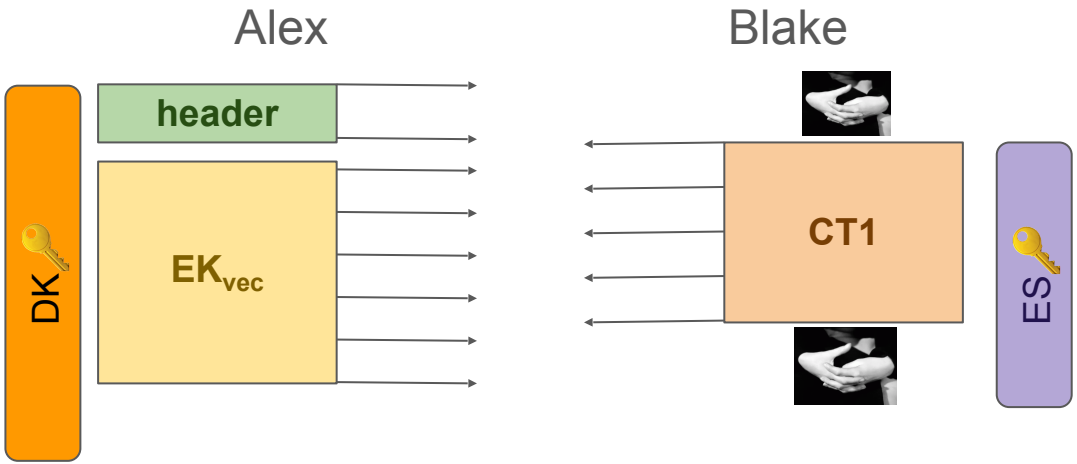
We can do most of the work in parallel!

- 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.



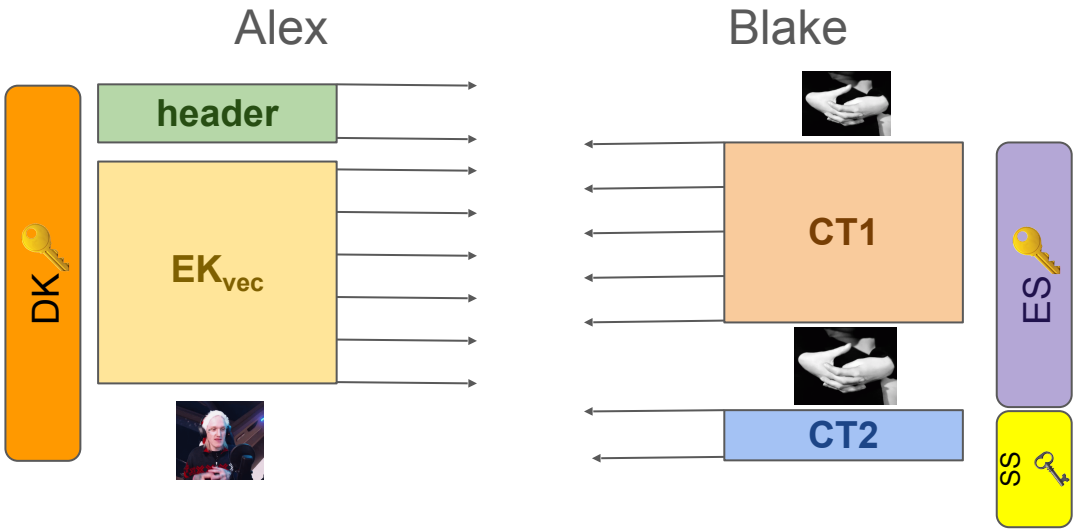
We can do most of the work in parallel!

- 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.



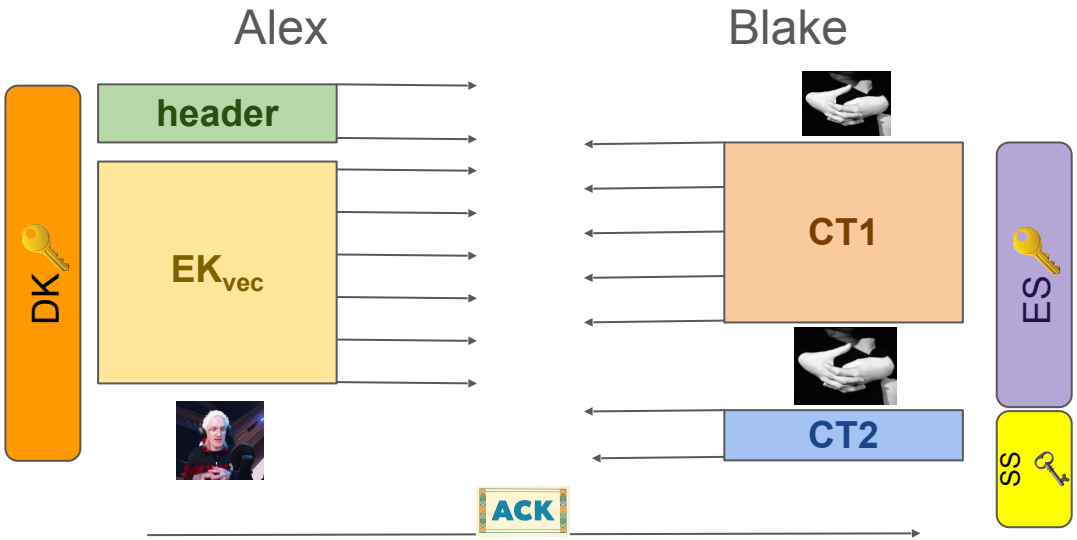
We can do most of the work in parallel!

- 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.



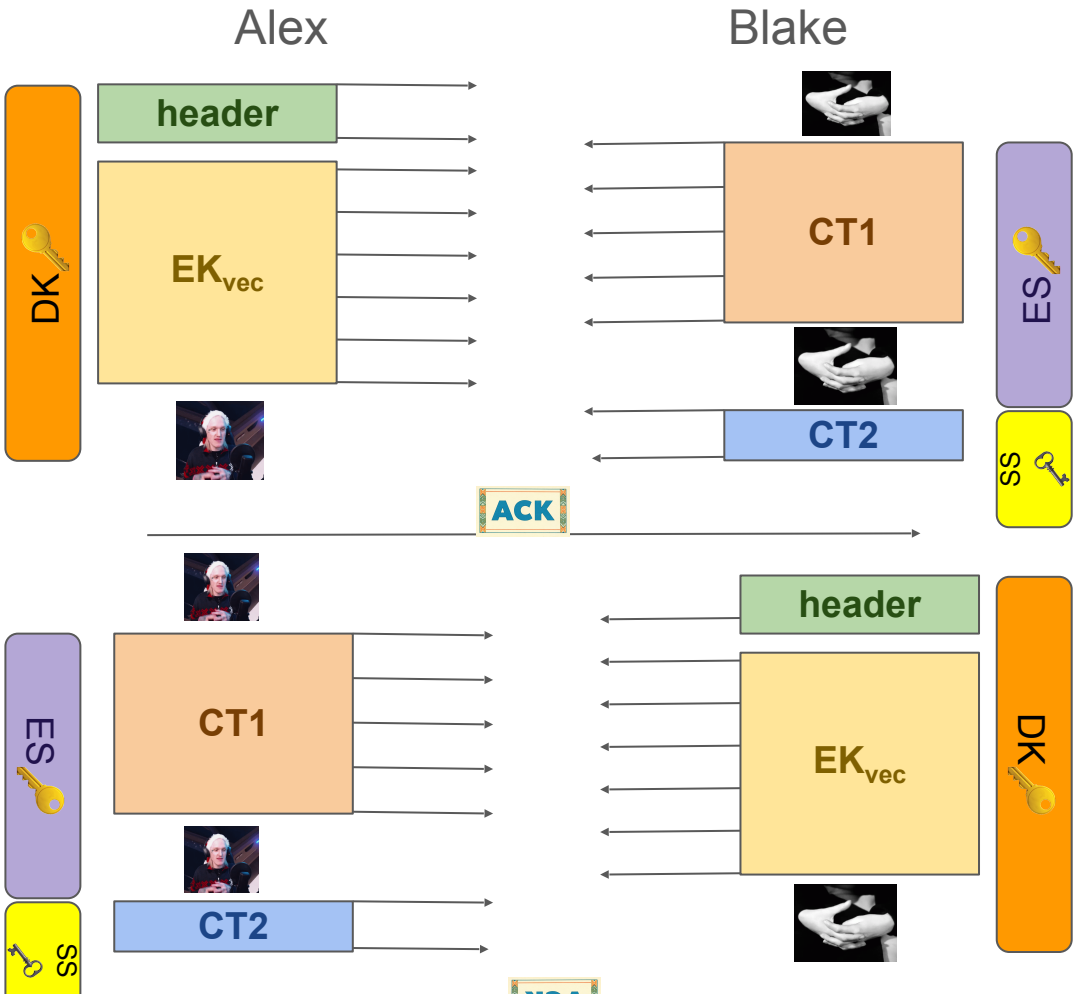
We can do most of the work in parallel!

- 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.



We can do most of the work in parallel!

- 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.



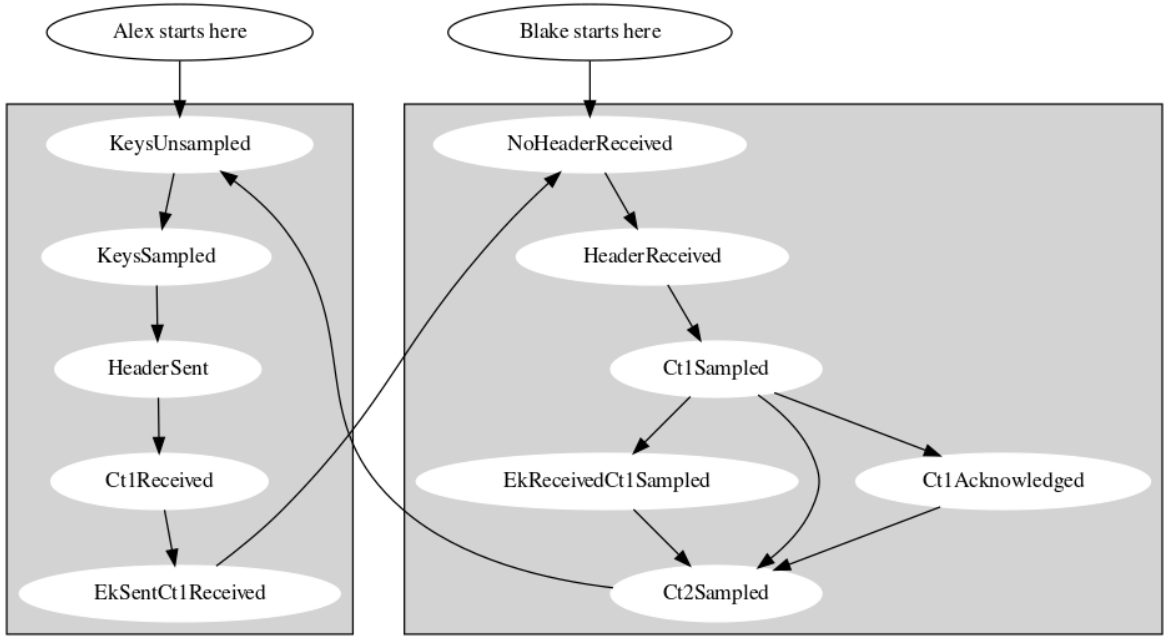
Real communication isn’t balanced.

We need to handle all of the corner cases.

State Machines

We handle all possible cases by describing a protocol participant as a state machine.

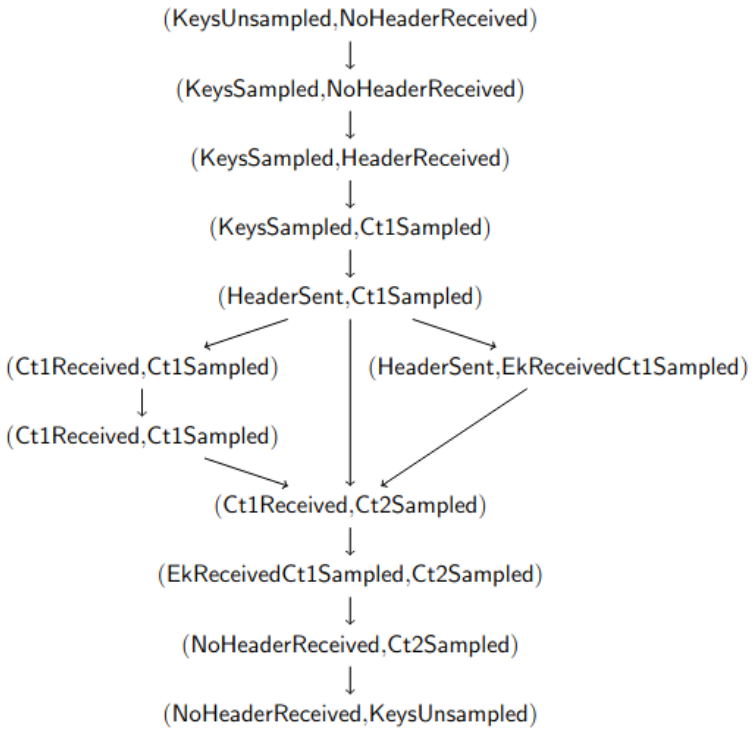
This maps easily to ProVerif allowing us to formally verify that the protocol has the desired security properties.



State Machines

Even without machine checked proofs, it is straightforward (if tedious) to see that with the right initial states for Alex and Blake, the set of accessible pairs of states is highly constrained.

And can always make forward progress!



~~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.

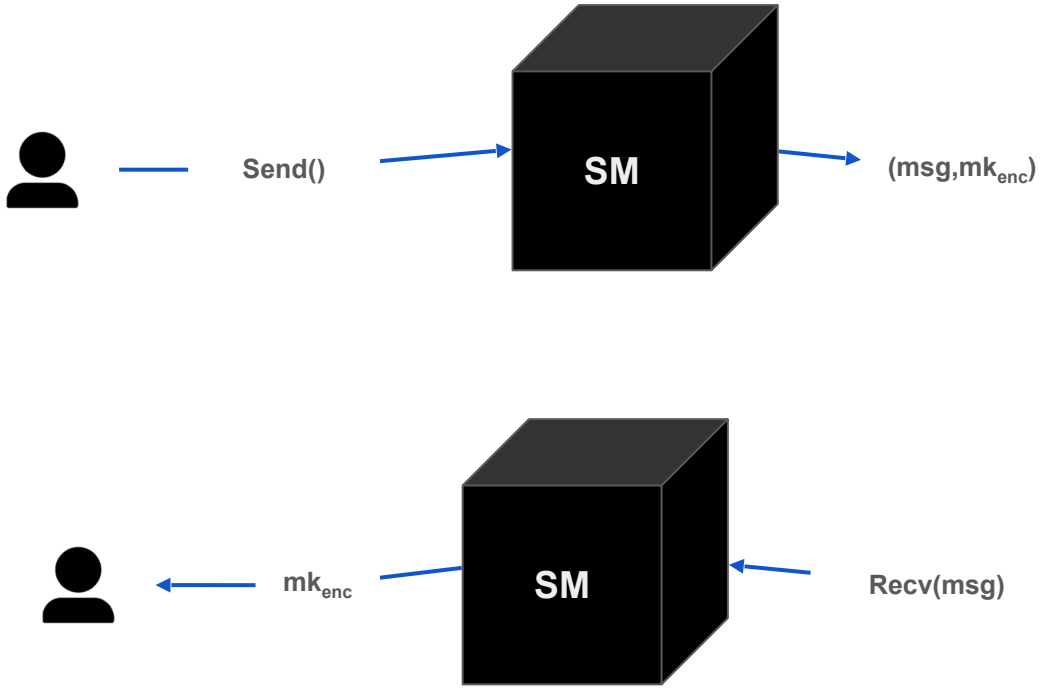
Putting it All Together

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

Double Ratchet as a Black Box

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

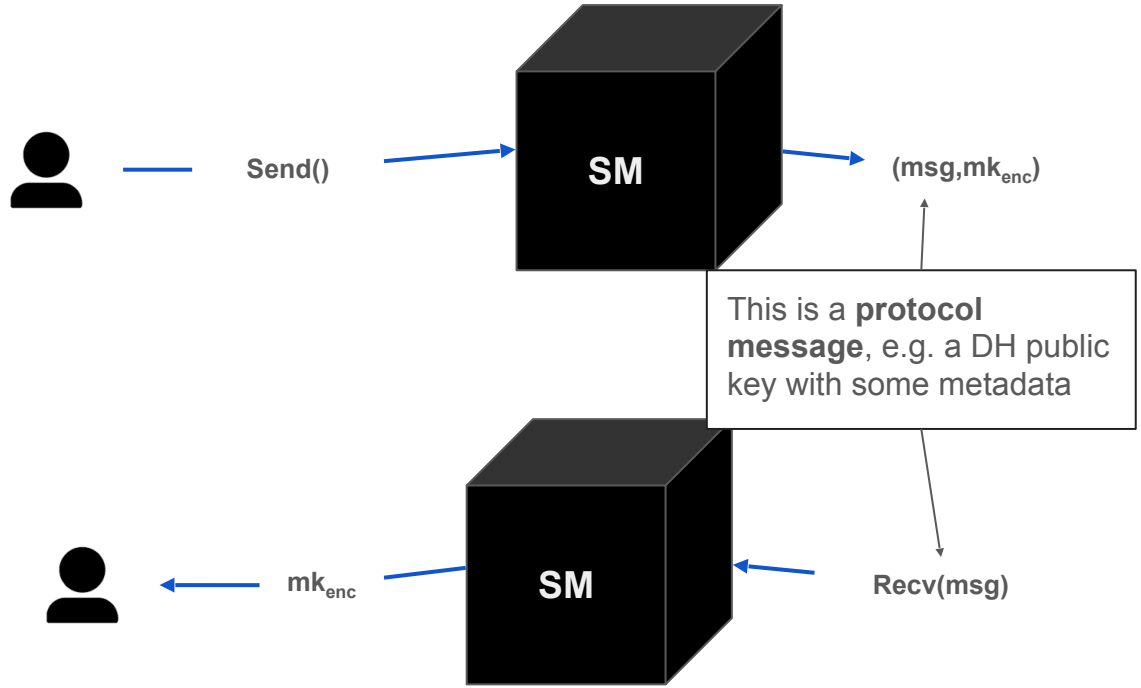
- **Init(secret)**
- **Send() → (msg, mk_{enc})**: get a protocol message and an encryption key, no input needed.
- **Recv(msg) → mk_{enc}**: Take a protocol message and get a decryption key.



Double Ratchet 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.

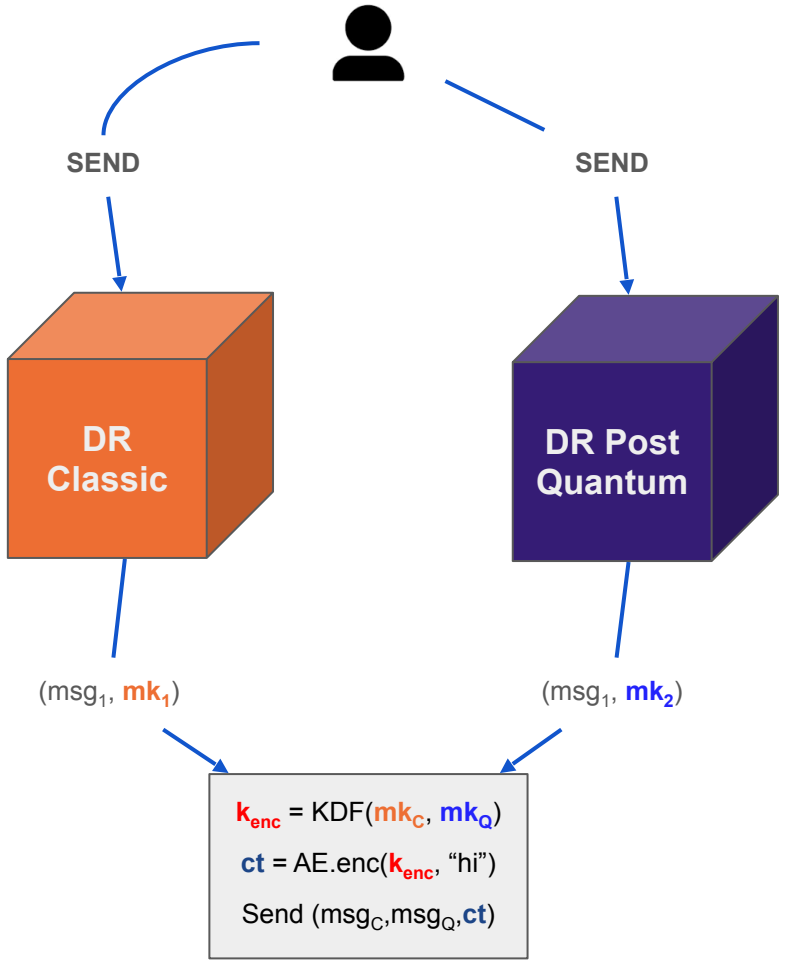


Now we can hybridize

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!



Formal Verification

- We worked with Cryspen from the beginning of implementation
  - Encouraged a proof-friendly modular design that can be analyzed independently
  - Modeled design in ProVerif from the start - and we used this to check design changes ourselves!
  - Prepared a CI pipeline integrating **hax**.
  - Gave guidance on proof-friendly Rust style (there aren't many restrictions!)
- Formally verified implementation
  - Proven panic-free with Hax/F*
    - We create security relevant asserts and prove they do not happen!
  - Proven correct finite field operations (for chunking) with Hax/F*
  - Proverif models of ML-KEM Braid and Symmetric Ratchet
- Formal verification is a dynamic part of the engineering process
  - On every push we extract models from the code, then prove safety and correctness.
  - If the proofs fail, the build fails - and we can fix it.

There’s Code!

https://github.com/signalapp/SparsePostQuantumRatchet

This is not the end of the story

- We can Do Better if we don’t send that Header.
  - We could derive the seed from session state
  - But we couldn’t use standardized ML-KEM.
- Formal verification will expand.
- Evaluation needs more work!
  - Better models of messaging behavior
  - Quantifiable measures of “robustness” of a protocol?
- **There’s another way to open the KEM black box**
  - **Opportunistic Katana ⚔️ [EC:DJKP25] may be the future.**
  - **See our talk later this week!**

Thank you!

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

rolfe@signal.org

Amortize the Cost?

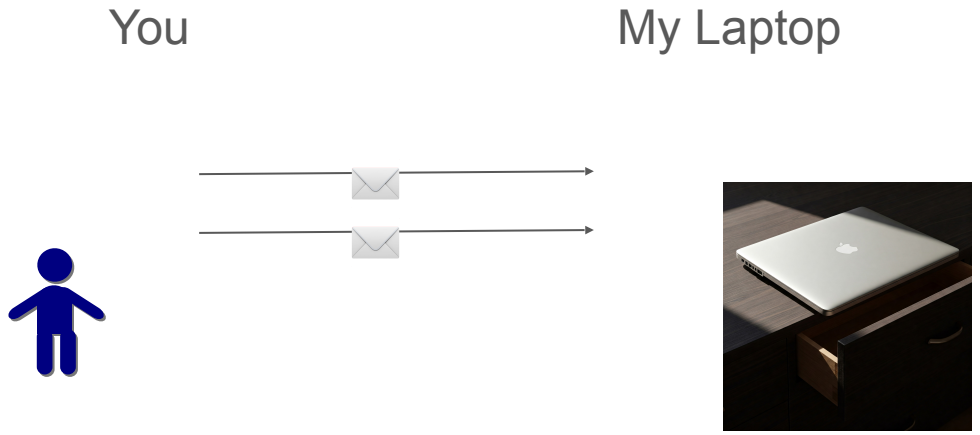
Apple's PQ3 addresses this by amortizing the cost of the large message:

- They send one large PQ CKA message every ~50 messages or once a week.
- To get immediate decryption they MUST repeat this large message until they get a response.
- Great when two parties are online!
- But…

Amortize the Cost?

Apple's PQ3 addresses this by amortizing the cost of the large message:

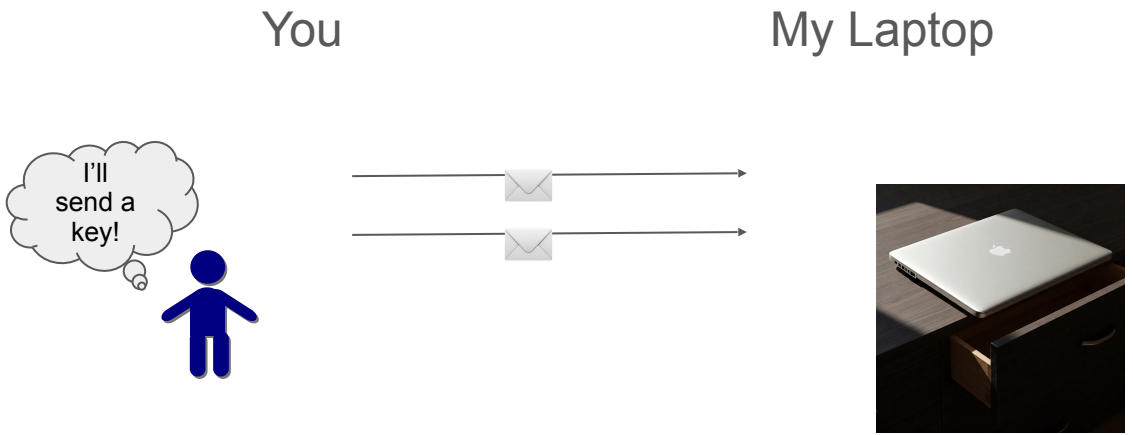
- They send one large PQ CKA message every ~50 messages or once a week.
- To get immediate decryption they MUST repeat this large message until they get a response.
- Great when two parties are online!
- But…



Amortize the Cost?

Apple's PQ3 addresses this by amortizing the cost of the large message:

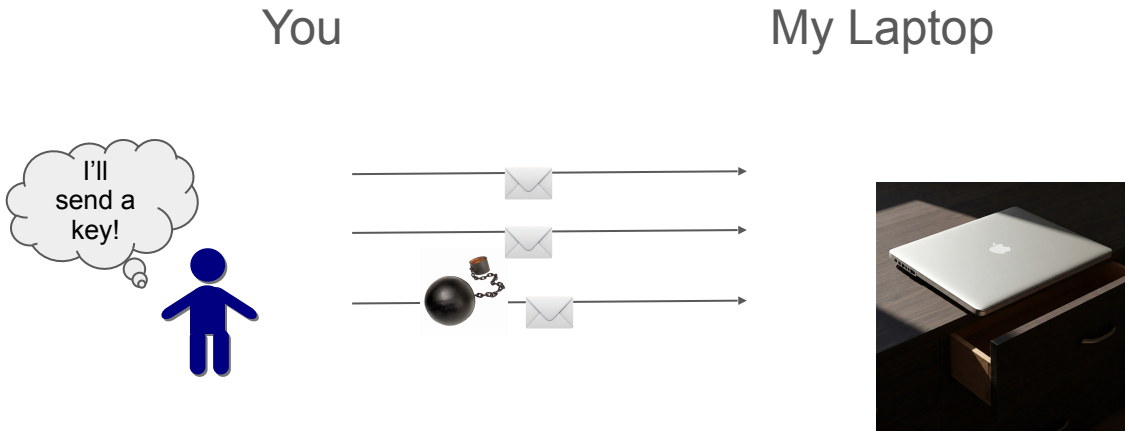
- They send one large PQ CKA message every ~50 messages or once a week.
- To get immediate decryption they MUST repeat this large message until they get a response.
- Great when two parties are online!
- But…



Amortize the Cost?

Apple's PQ3 addresses this by amortizing the cost of the large message:

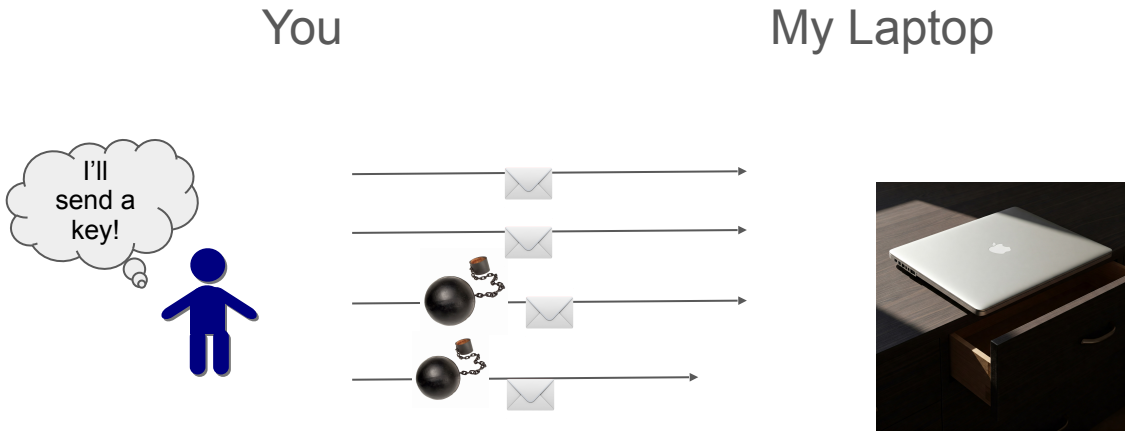
- They send one large PQ CKA message every ~50 messages or once a week.
- To get immediate decryption they MUST repeat this large message until they get a response.
- Great when two parties are online!
- But…



Amortize the Cost?

Apple's PQ3 addresses this by amortizing the cost of the large message:

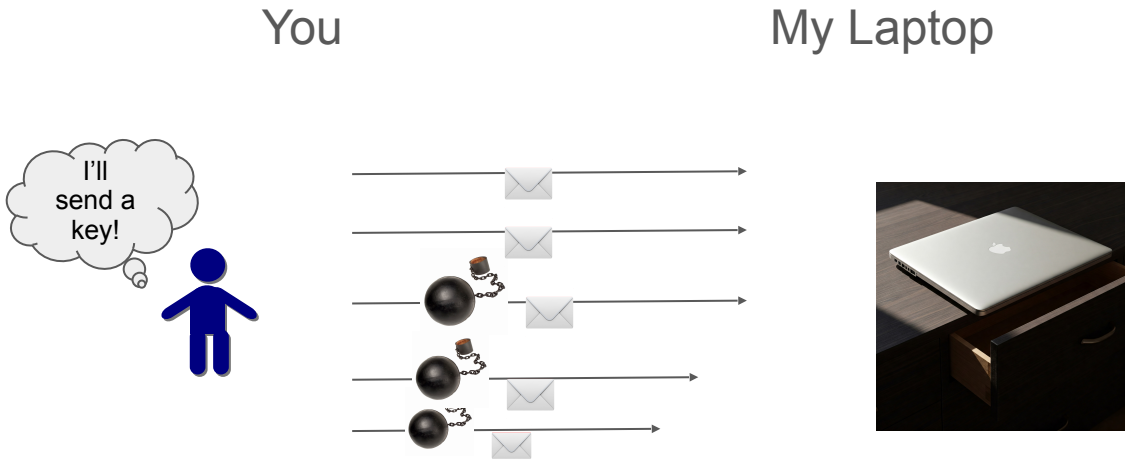
- They send one large PQ CKA message every ~50 messages or once a week.
- To get immediate decryption they MUST repeat this large message until they get a response.
- Great when two parties are online!
- But…



Amortize the Cost?

Apple's PQ3 addresses this by amortizing the cost of the large message:

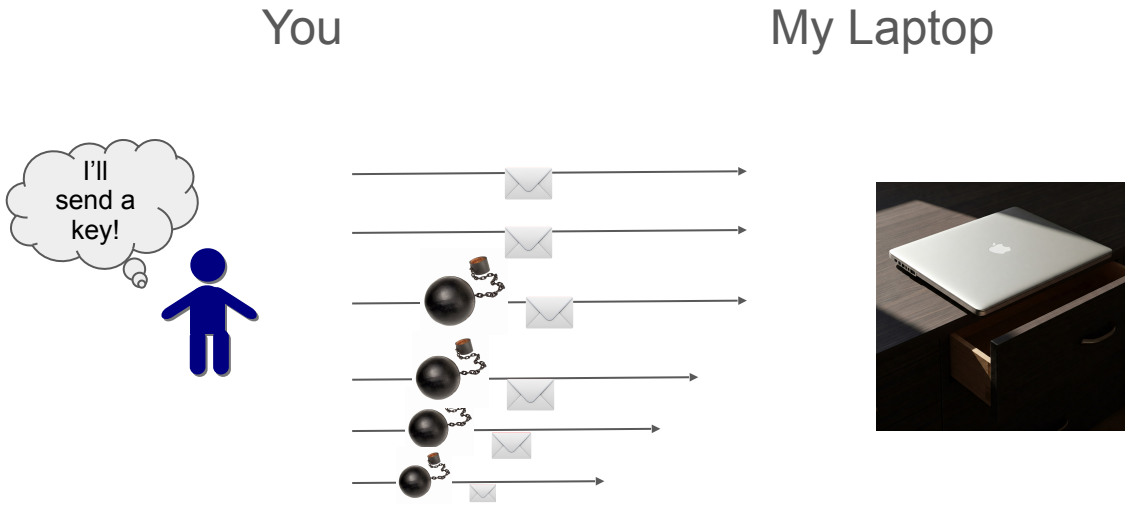
- They send one large PQ CKA message every ~50 messages or once a week.
- To get immediate decryption they MUST repeat this large message until they get a response.
- Great when two parties are online!
- But…



Amortize the Cost?

Apple's PQ3 addresses this by amortizing the cost of the large message:

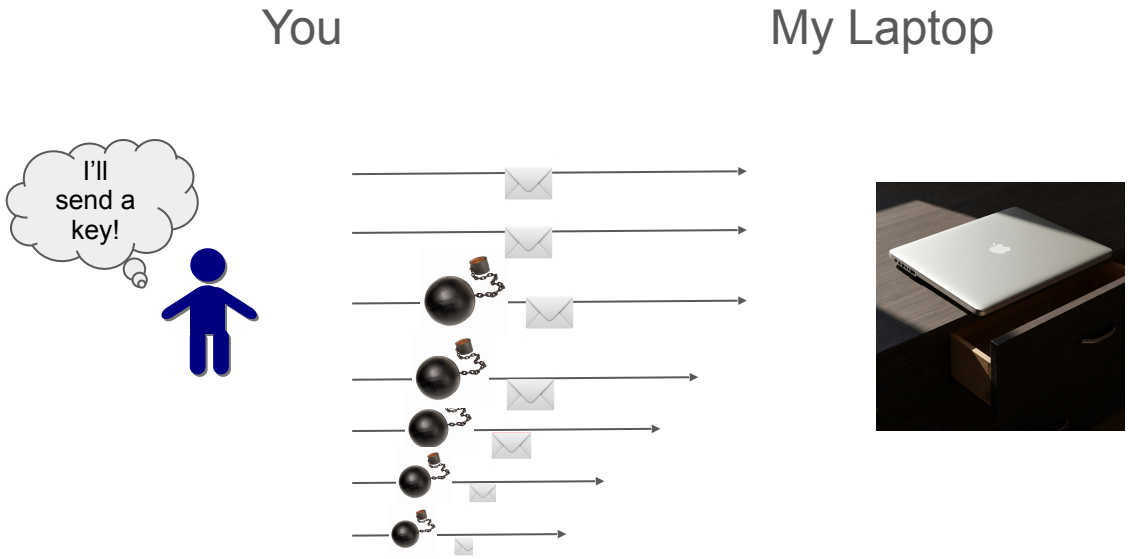
- They send one large PQ CKA message every ~50 messages or once a week.
- To get immediate decryption they MUST repeat this large message until they get a response.
- Great when two parties are online!
- But…



Amortize the Cost?

Apple's PQ3 addresses this by amortizing the cost of the large message:

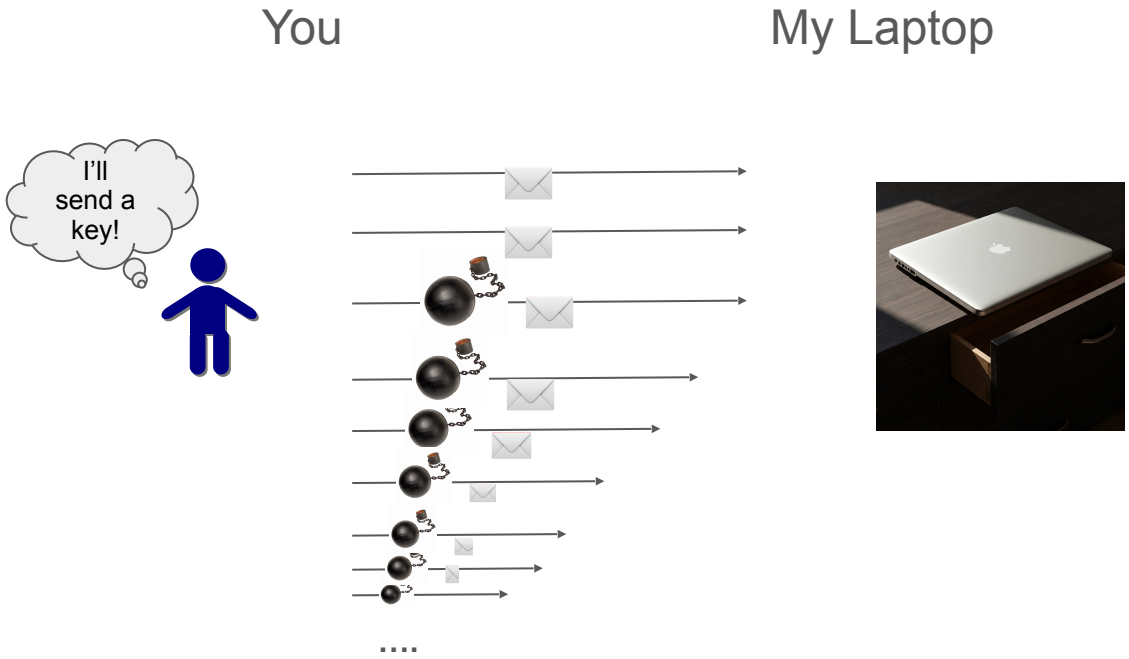
- They send one large PQ CKA message every ~50 messages or once a week.
- To get immediate decryption they MUST repeat this large message until they get a response.
- Great when two parties are online!
- But…



Amortize the Cost?

Apple's PQ3 addresses this by amortizing the cost of the large message:

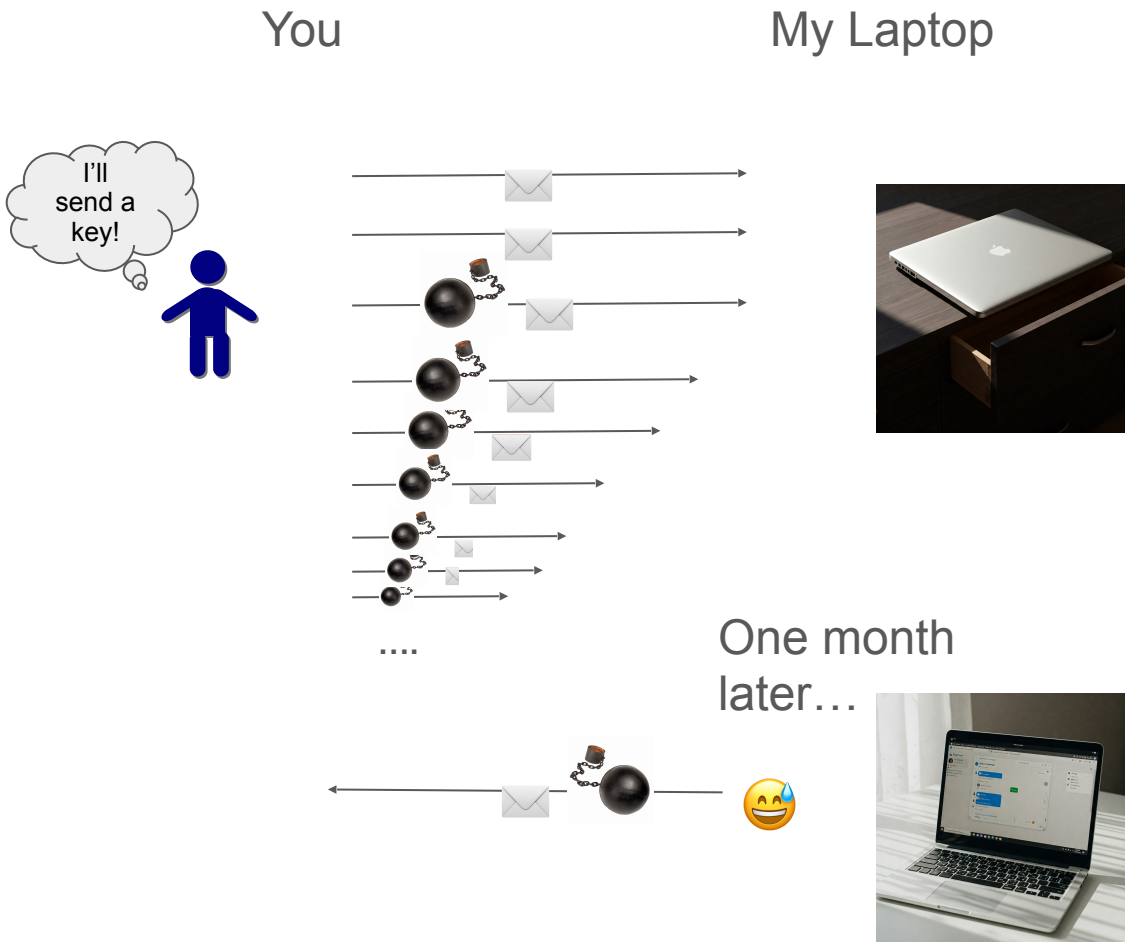
- They send one large PQ CKA message every ~50 messages or once a week.
- To get immediate decryption they MUST repeat this large message until they get a response.
- Great when two parties are online!
- But…



Amortize the Cost?

Apple's PQ3 addresses this by amortizing the cost of the large message:

- They send one large PQ CKA message every ~50 messages or once a week.
- To get immediate decryption they MUST repeat this large message until they get a response.
- Great when two parties are online!
- But…



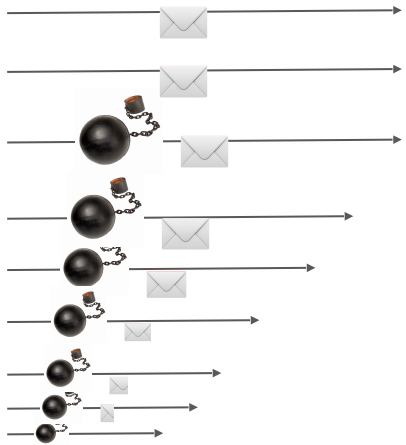
Amortize the Cost?

Apple's PQ3 addresses this by amortizing the cost of the large message:

- They send one large PQ CKA message every ~50 messages or once a week.
- To get immediate decryption they MUST repeat this large message until they get a response.
- Great when two parties are online!
- But…

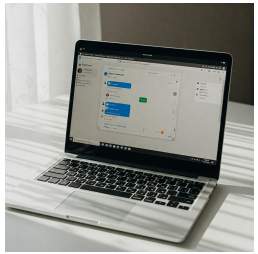
You

My Laptop



....

One month later…



Amortize the Cost?

Apple's PQ3 addresses thi
amortizing the cost of the
message:

- They send one larg
message every ~50
or once a week.
- To get immediate d
they MUST repeat t
message until they
response.
- Great when two pa
online!
- But…

You

My Laptop

You're not happy.

Signal's not happy.

I'm not happy because I'm getting this from *everyone*.

Maybe your mobile carrier is happy if you don't have an unlimited plan?

One month
later…