Finding Practically-exploitable Cryptographic Vulnerabilities in Matrix

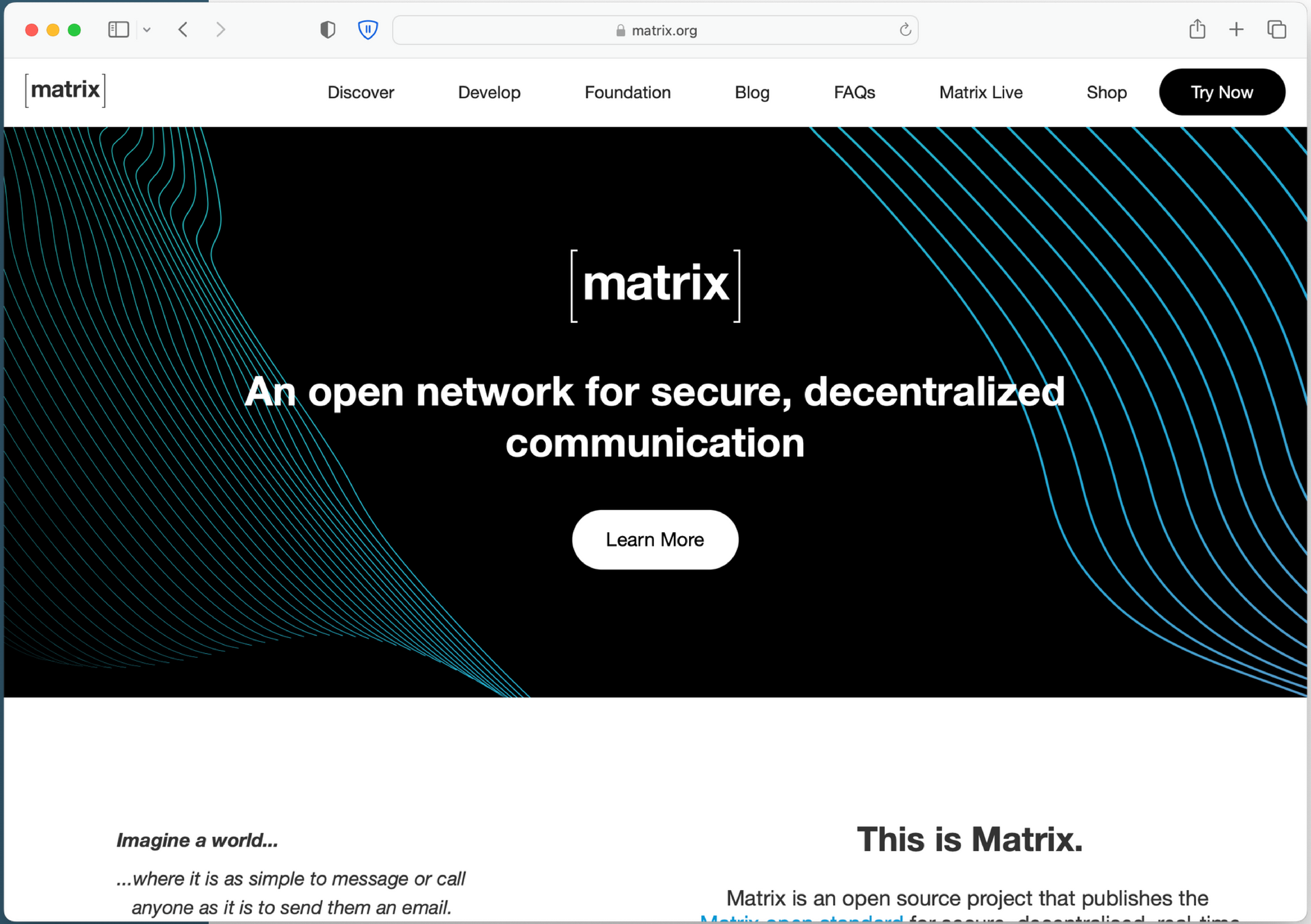
6th Workshop on Attacks in Cryptography, 20th August 2023

Martin Albrecht (King's College London*) *martin.albrecht@kcl.ac.uk*
Sofía Celi (Brave Software) *cherenkov@riseup.net*
Benjamin Dowling (University of Sheffield) *b.dowling@sheffield.ac.uk*
Dan Jones (Royal Holloway, University of London) *dan.jones@rhul.ac.uk*

me

* Work completed whilst working for Royal Holloway, University of London.

Matrix


matrix.org
matrix
Discover
Develop
Foundation
Blog
FAQs
Matrix Live
Shop
Try Now
matrix
An open network for secure, decentralized communication
Learn More
Imagine a world...
...where it is as simple to message or call anyone as it is to send them an email.
This is Matrix.
Matrix is an open source project that publishes the

Matrix

wired.co.uk

WIRED

LONG READS BUSINESS CULTURE GEAR SCIENCE SECURITY VIDEO

SUBSCRIBE

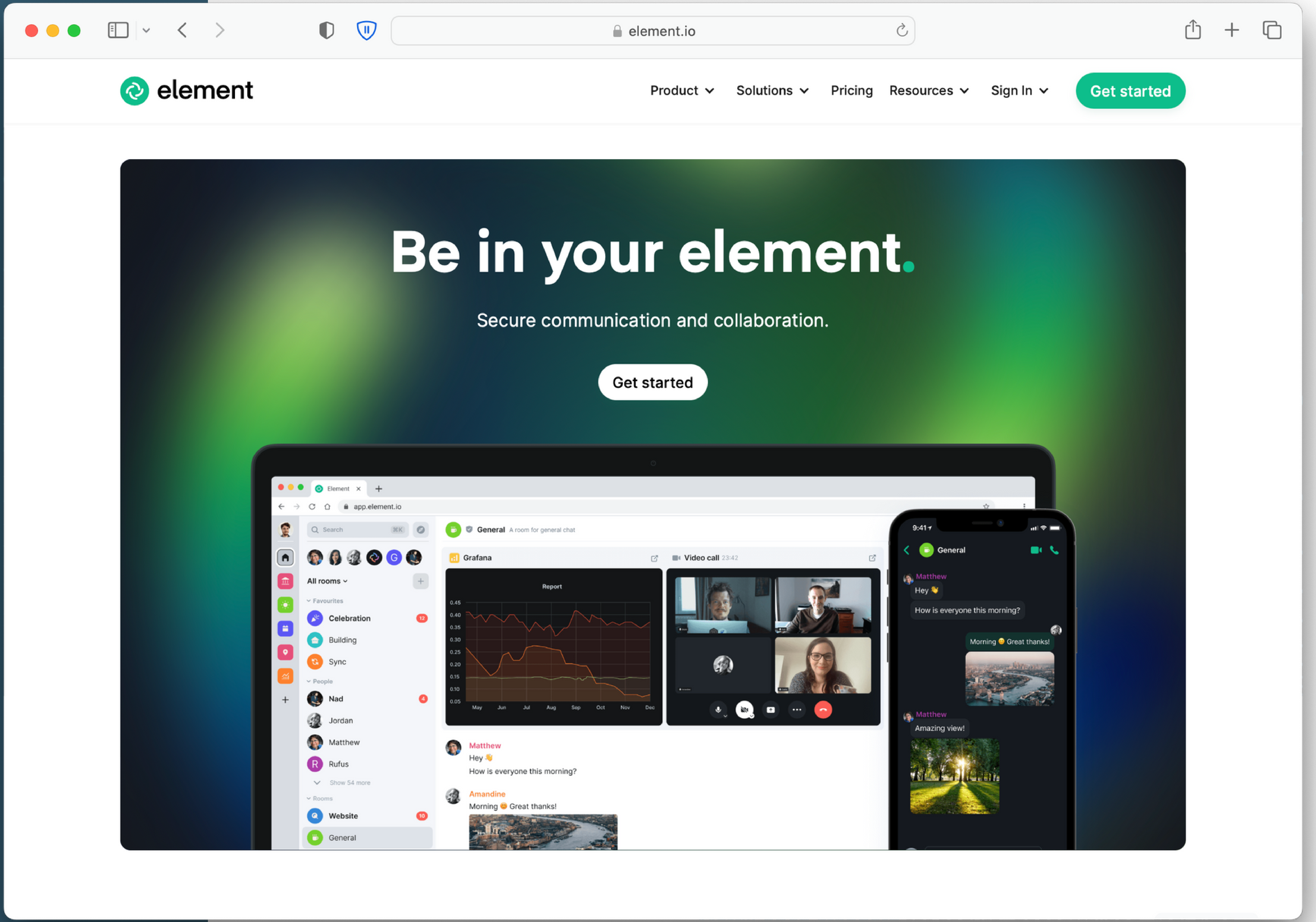
GIAN M. VOLPICELLI SECURITY 14.06.2021 06:00 AM

How governments and spies text each other

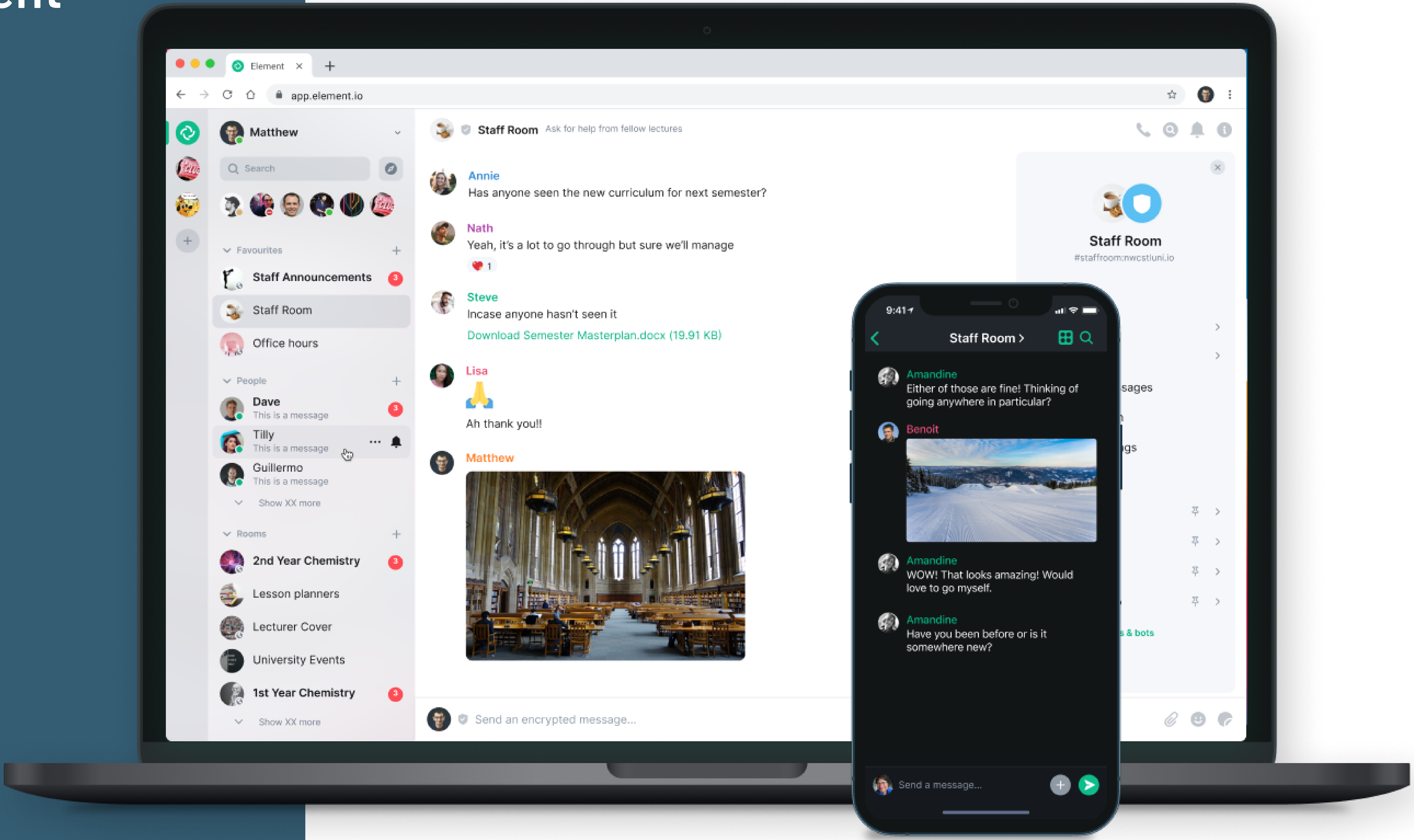
Matrix has become the messaging app of choice for top-secret communications

DAVID RYLE

Element


element.io
element
Product
Solutions
Pricing
Resources
Sign In
Get started
Be in your element.
Secure communication and collaboration.
Get started

Element



Element
app.element.io
Matthew
Search
Favourites
Staff Announcements
Staff Room
Office hours
People
Dave
This is a message
Tilly
This is a message
Guillermo
This is a message
Show XX more
Rooms
2nd Year Chemistry
Lesson planners
Lecturer Cover
University Events
1st Year Chemistry
Show XX more
Staff Room
Ask for help from fellow lectures
Annie
Has anyone seen the new curriculum for next semester?
Nath
Yeah, it's a lot to go through but sure we'll manage
Steve
Incase anyone hasn't seen it
Download Semester Masterplan.docx (19.91 KB)
Lisa
Ah thank you!!
Matthew
Send an encrypted message...
Staff Room
#staffroom:nwcstluni.io
9:41
Staff Room
Amandine
Either of those are fine! Thinking of going anywhere in particular?
Benoit
Amandine
WOW! That looks amazing! Would love to go myself.
Amandine
Have you been before or is it somewhere new?
Send a message...

Our Work

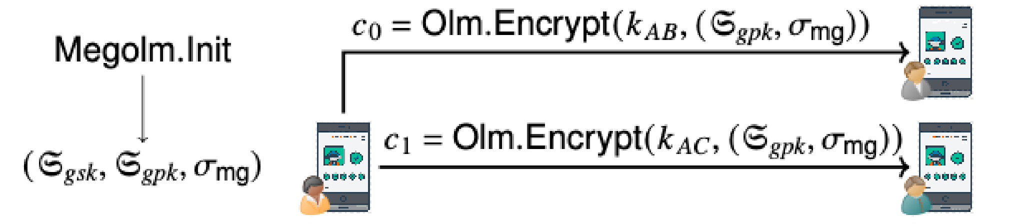
matrix-attack-paper-sp2023-camera-ready.pdf (page 1 of 18)

Practically-exploitable Cryptographic Vulnerabilities in Matrix

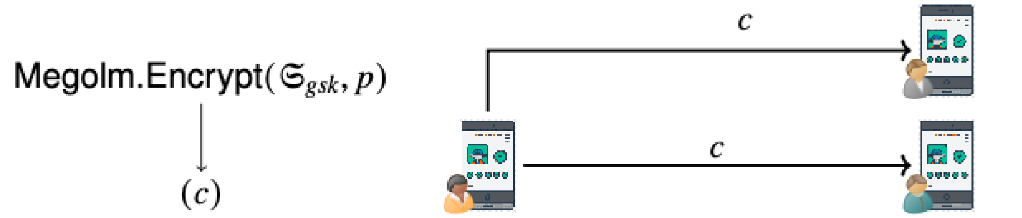
Martin R. Albrecht*, Sofía Celi[†], Benjamin Dowling[‡] and Daniel Jones[§]
[*] King's College London, martin.albrecht@kcl.ac.uk
[†] Brave Software, cherenkov@riseup.net
[‡] Security of Advanced Systems Group, University of Sheffield, b.dowling@sheffield.ac.uk
[§]Information Security Group, Royal Holloway, University of London, dan.jones@rhul.ac.uk

***Abstract*—We report several practically-exploitable cryptographic vulnerabilities in the Matrix standard for federated real-time communication and its flagship client and prototype implementation, Element. These, together, invalidate the confidentiality and authentication guarantees claimed by Matrix against a malicious server. This is despite Matrix' cryptographic routines being constructed from well-known and -studied cryptographic building blocks. The vulnerabilities we exploit differ in their nature (insecure by design, protocol confusion, lack of domain separation, implementation bugs) and are distributed broadly across the different subprotocols and libraries that make up the cryptographic core of Matrix and Element. Together, these vulnerabilities highlight the need for a systematic and formal analysis of the cryptography in the Matrix standard.**

I. INTRODUCTION



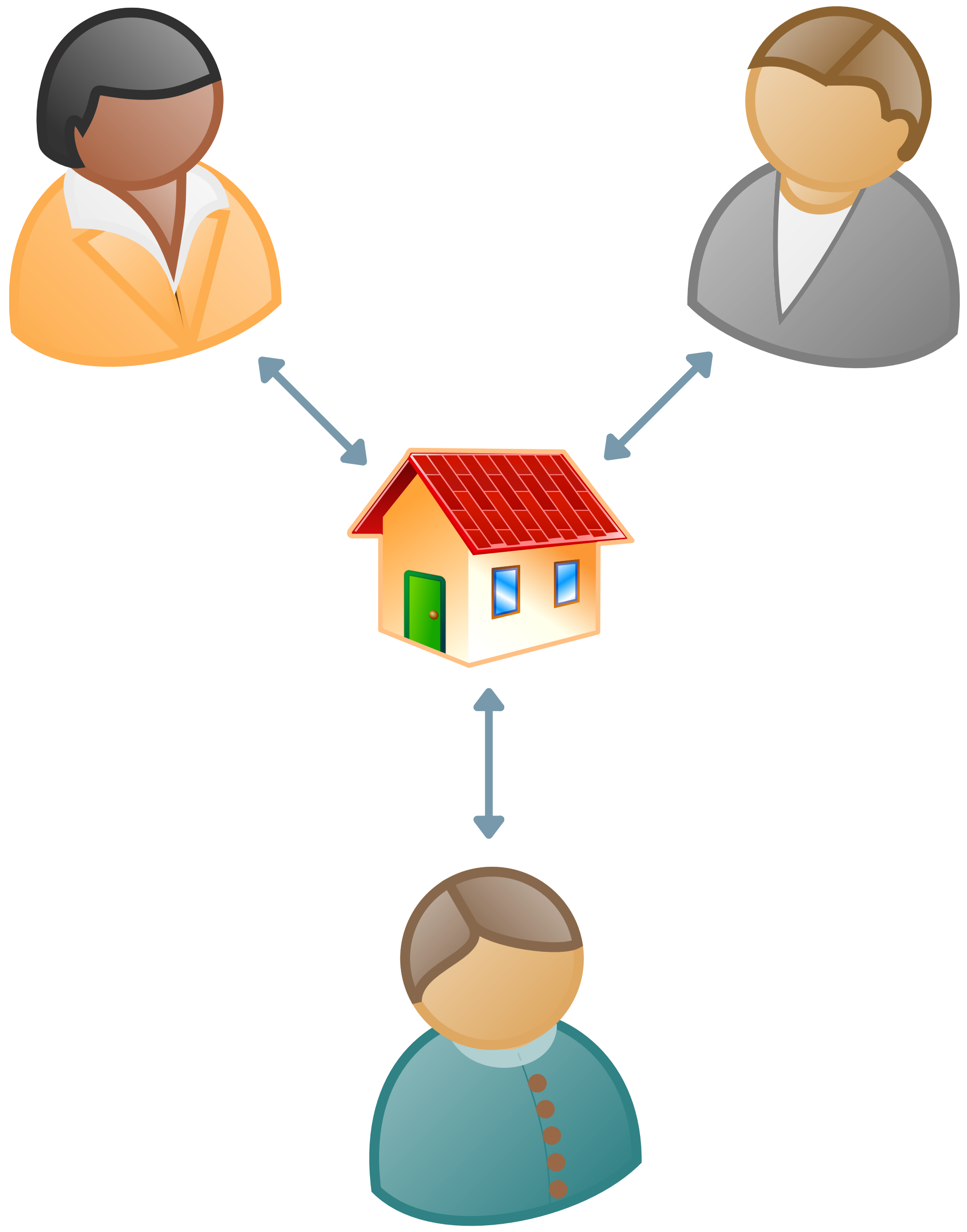
(a) Alice's device (left) sending Megolm channel information over pairwise Olm channels.



(b) Alice's device (left) sending payload data over the unidirectional Megolm channel.

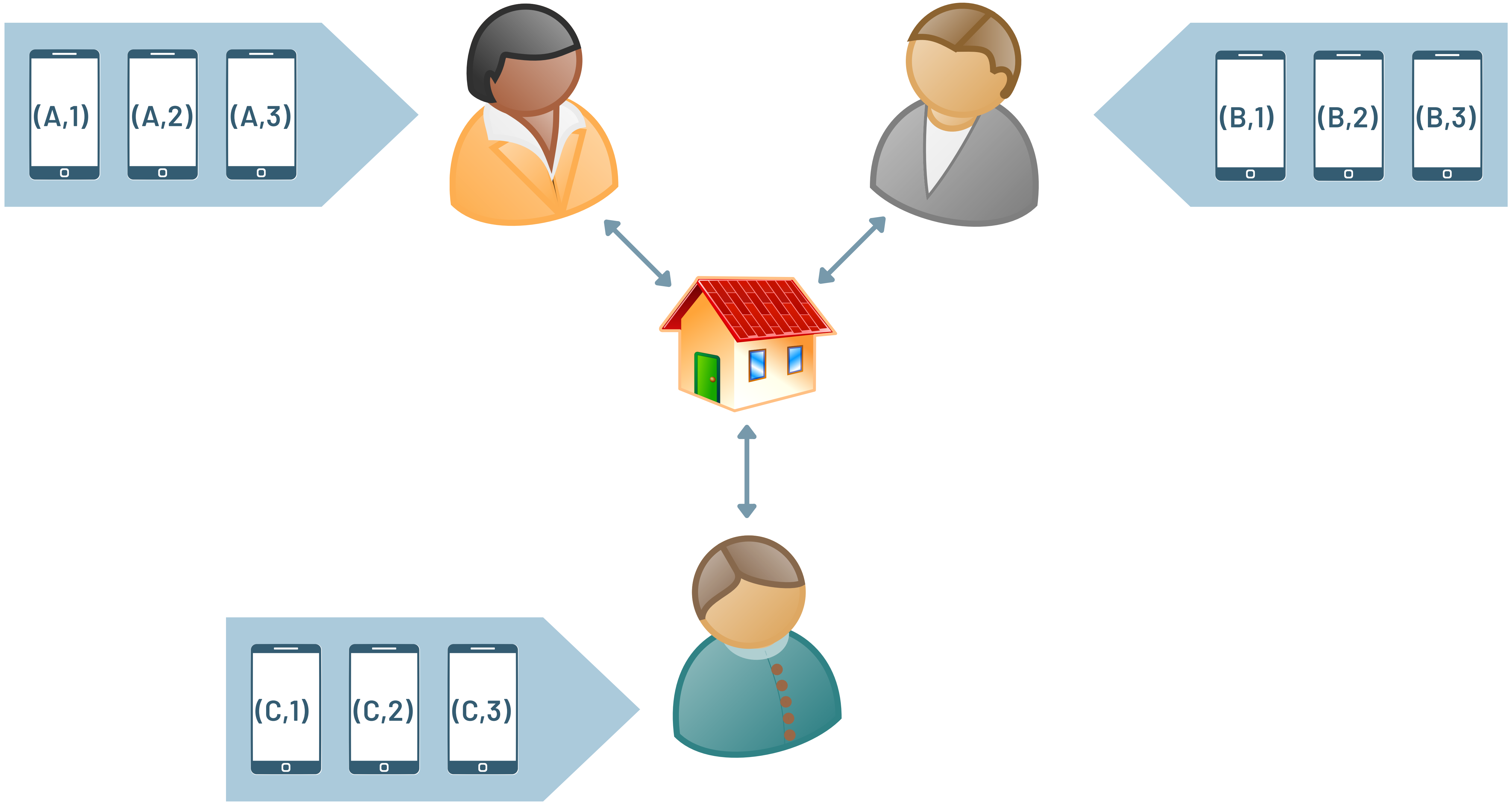
Matrix

A comprehensive secure messaging solution.



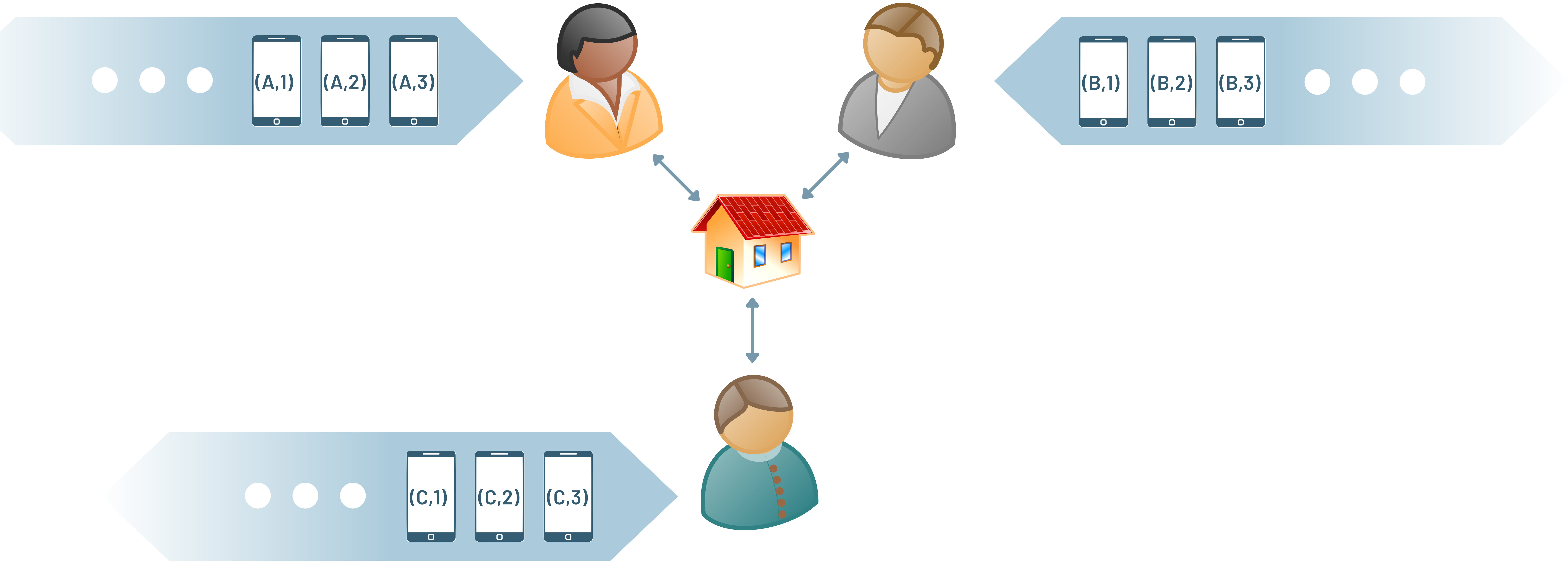
Matrix

A comprehensive secure messaging solution.



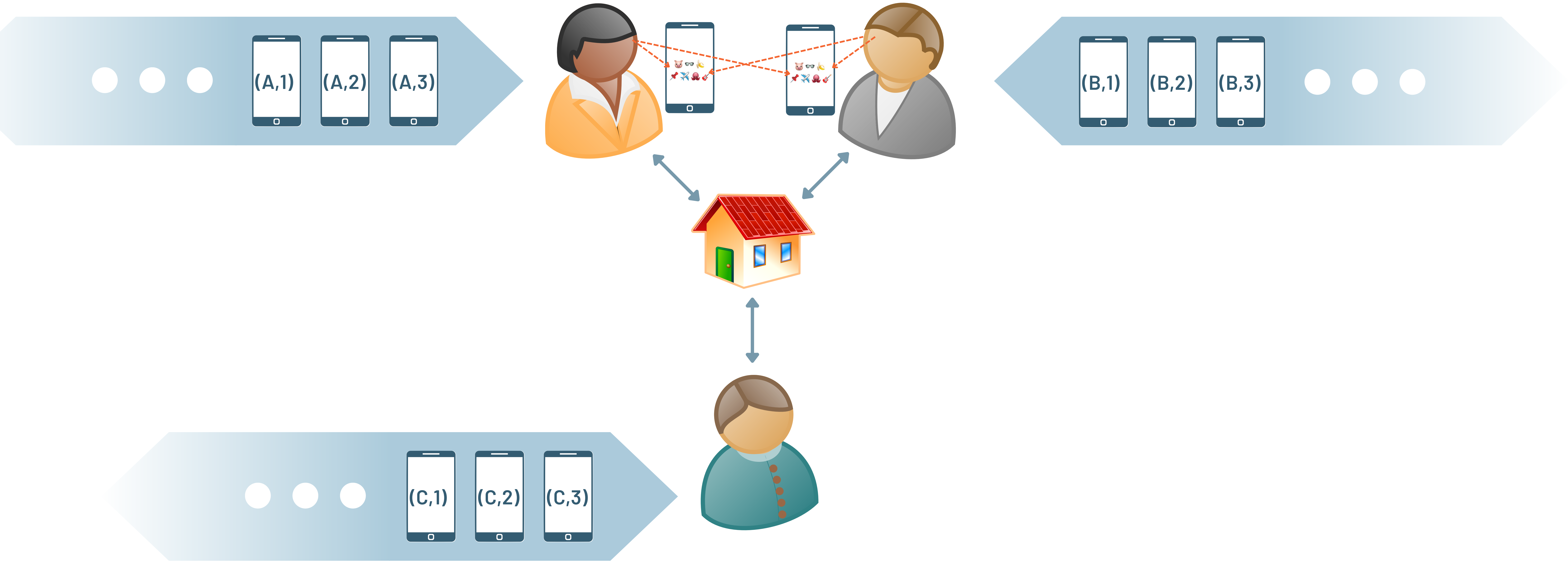
Matrix

A comprehensive secure messaging solution.



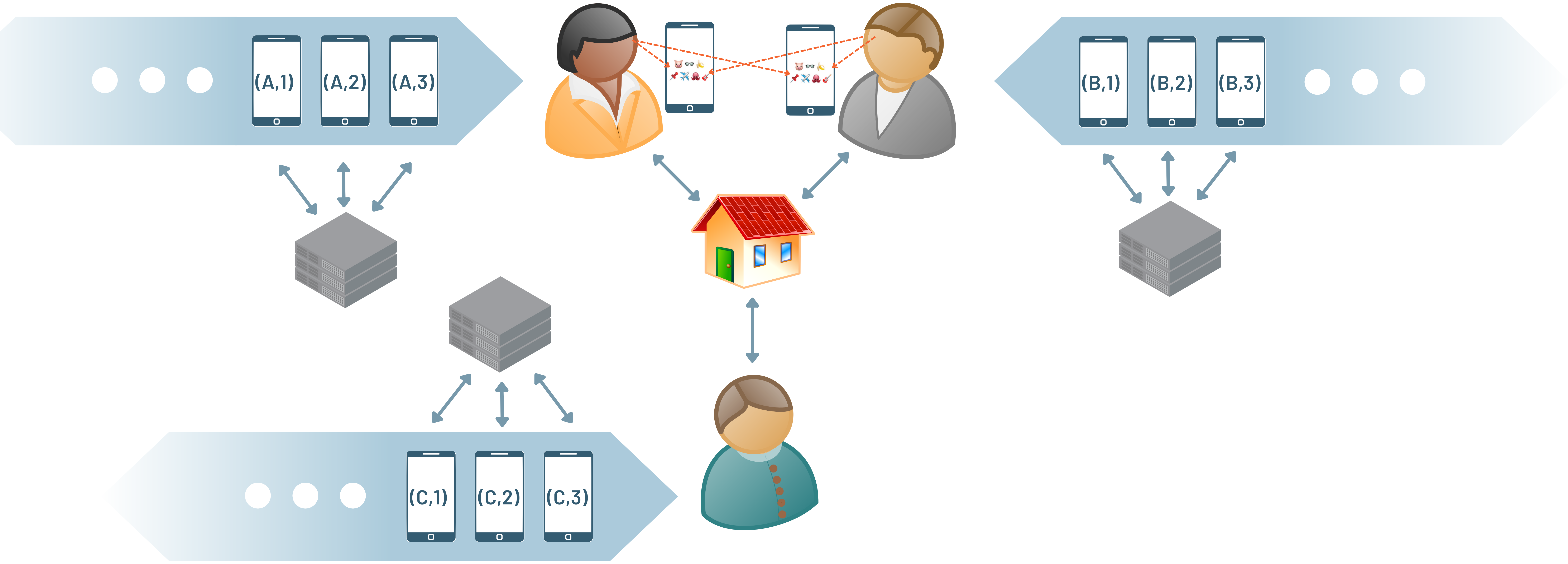
Matrix

A comprehensive secure messaging solution.



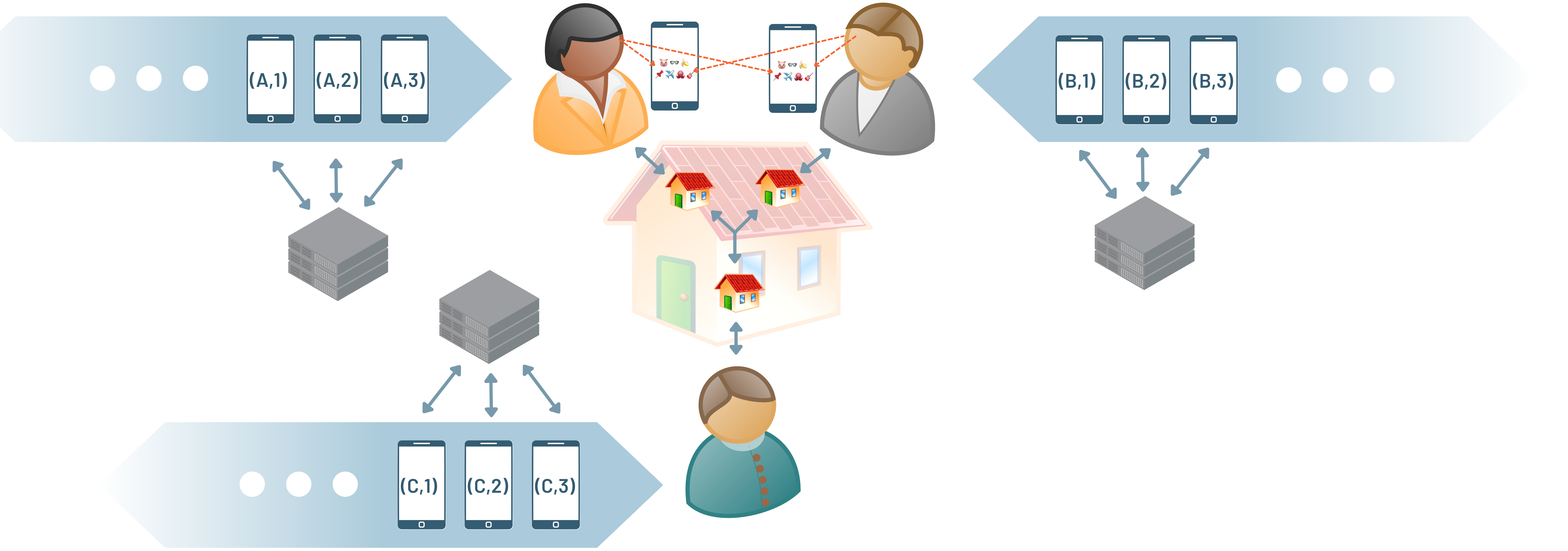
Matrix

A comprehensive secure messaging solution.



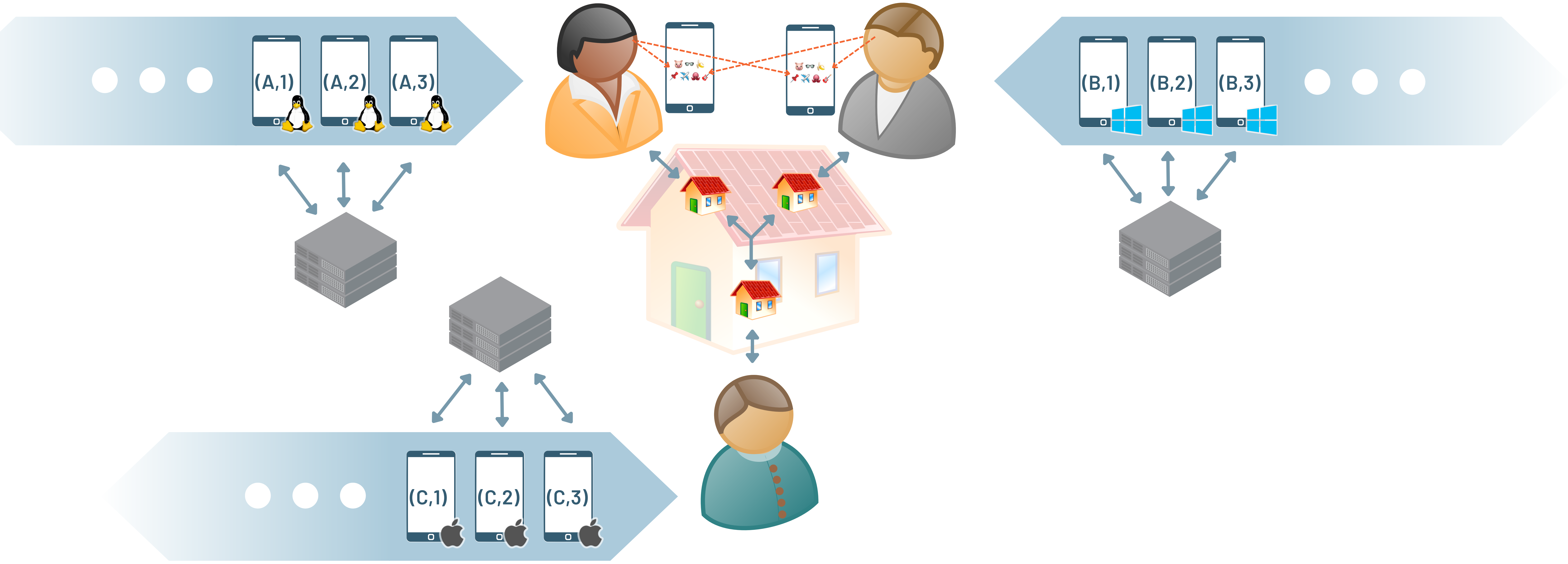
Matrix

A comprehensive secure messaging solution.

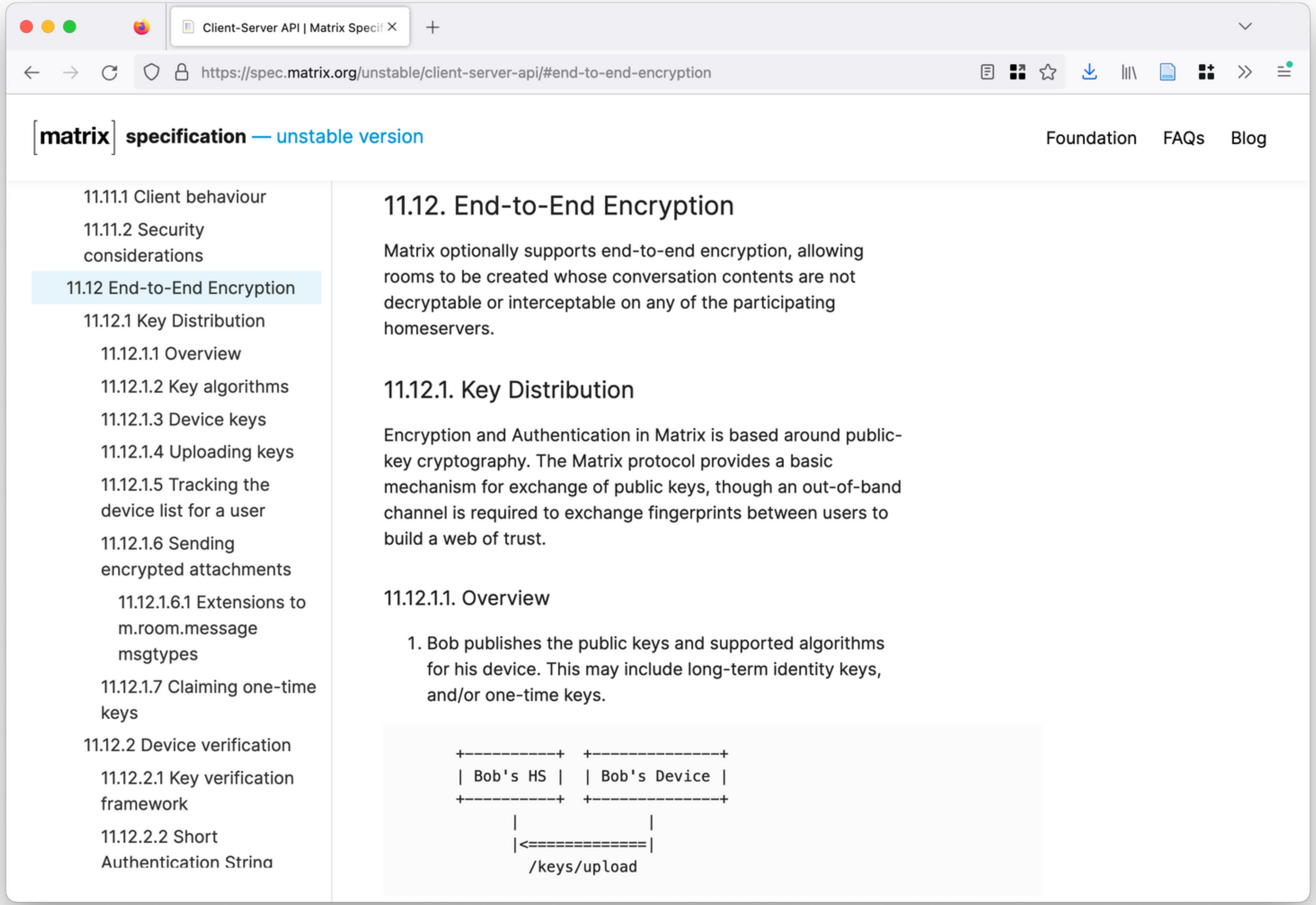


Matrix

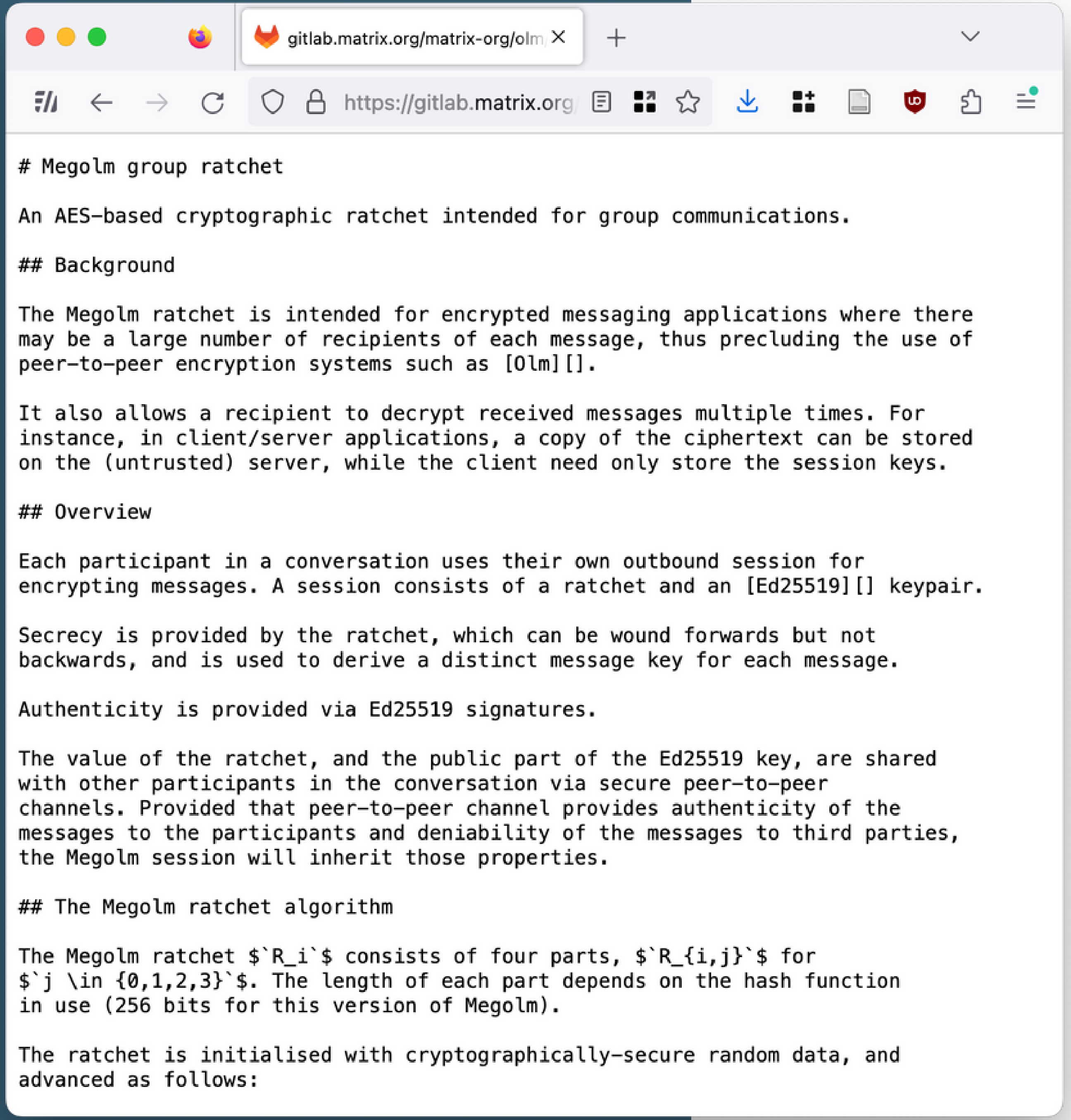
A comprehensive secure messaging solution.



Research


Client-Server API | Matrix Speci
https://spec.matrix.org/unstable/client-server-api/#end-to-end-encryption
[matrix] specification — unstable version
Foundation
FAQs
Blog
11.11.1 Client behaviour
11.11.2 Security considerations
11.12 End-to-End Encryption
11.12.1 Key Distribution
11.12.1.1 Overview
11.12.1.2 Key algorithms
11.12.1.3 Device keys
11.12.1.4 Uploading keys
11.12.1.5 Tracking the device list for a user
11.12.1.6 Sending encrypted attachments
11.12.1.6.1 Extensions to m.room.message msgtypes
11.12.1.7 Claiming one-time keys
11.12.2 Device verification
11.12.2.1 Key verification framework
11.12.2.2 Short Authentication String
11.12. End-to-End Encryption
Matrix optionally supports end-to-end encryption, allowing rooms to be created whose conversation contents are not decryptable or interceptable on any of the participating homeservers.
11.12.1. Key Distribution
Encryption and Authentication in Matrix is based around public-key cryptography. The Matrix protocol provides a basic mechanism for exchange of public keys, though an out-of-band channel is required to exchange fingerprints between users to build a web of trust.
11.12.1.1. Overview
1. Bob publishes the public keys and supported algorithms for his device. This may include long-term identity keys, and/or one-time keys.
Bob's HS
Bob's Device
/keys/upload

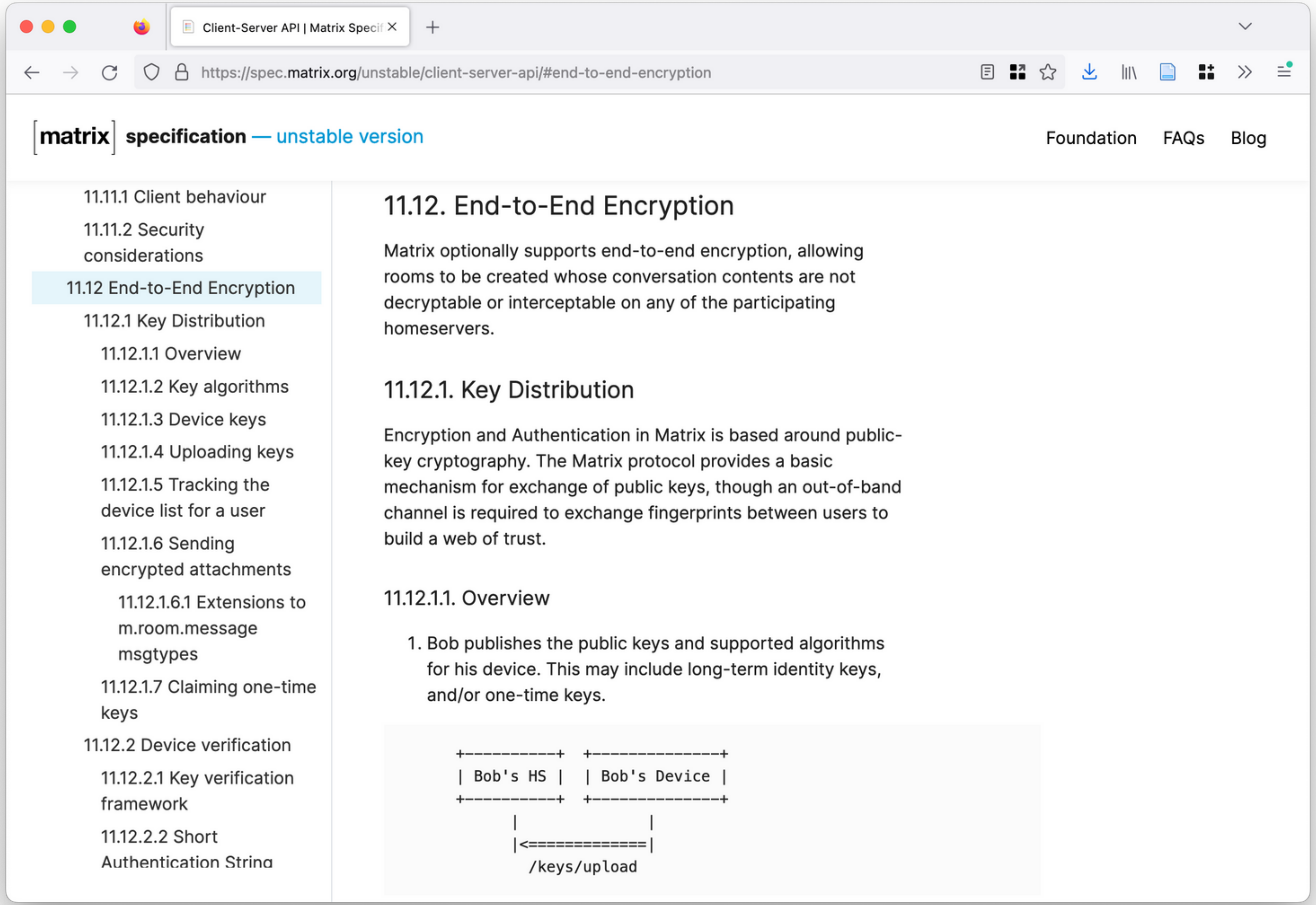
Research



gitlab.matrix.org/matrix-org/olm

https://gitlab.matrix.org

```
# Megolm group ratchet

An AES-based cryptographic ratchet intended for group communications.

## Background

The Megolm ratchet is intended for encrypted messaging applications where there
may be a large number of recipients of each message, thus precluding the use of
peer-to-peer encryption systems such as [Olm][].

It also allows a recipient to decrypt received messages multiple times. For
instance, in client/server applications, a copy of the ciphertext can be stored
on the (untrusted) server, while the client need only store the session keys.

## Overview

Each participant in a conversation uses their own outbound session for
encrypting messages. A session consists of a ratchet and an [Ed25519][] keypair.

Secrecy is provided by the ratchet, which can be wound forwards but not
backwards, and is used to derive a distinct message key for each message.

Authenticity is provided via Ed25519 signatures.

The value of the ratchet, and the public part of the Ed25519 key, are shared
with other participants in the conversation via secure peer-to-peer
channels. Provided that peer-to-peer channel provides authenticity of the
messages to the participants and deniability of the messages to third parties,
the Megolm session will inherit those properties.

## The Megolm ratchet algorithm

The Megolm ratchet $`R_i`$ consists of four parts, $`R_{i,j}`$ for
$`j \in {0,1,2,3}`$. The length of each part depends on the hash function
in use (256 bits for this version of Megolm).

The ratchet is initialised with cryptographically-secure random data, and
advanced as follows:
```



Client-Server API | Matrix Speci

https://spec.matrix.org/unstable/client-server-api/#end-to-end-encryption

[matrix] specification — unstable version

Foundation FAQs Blog

11.12. End-to-End Encryption

Matrix optionally supports end-to-end encryption, allowing rooms to be created whose conversation contents are not decryptable or interceptable on any of the participating homeservers.

11.12.1. Key Distribution

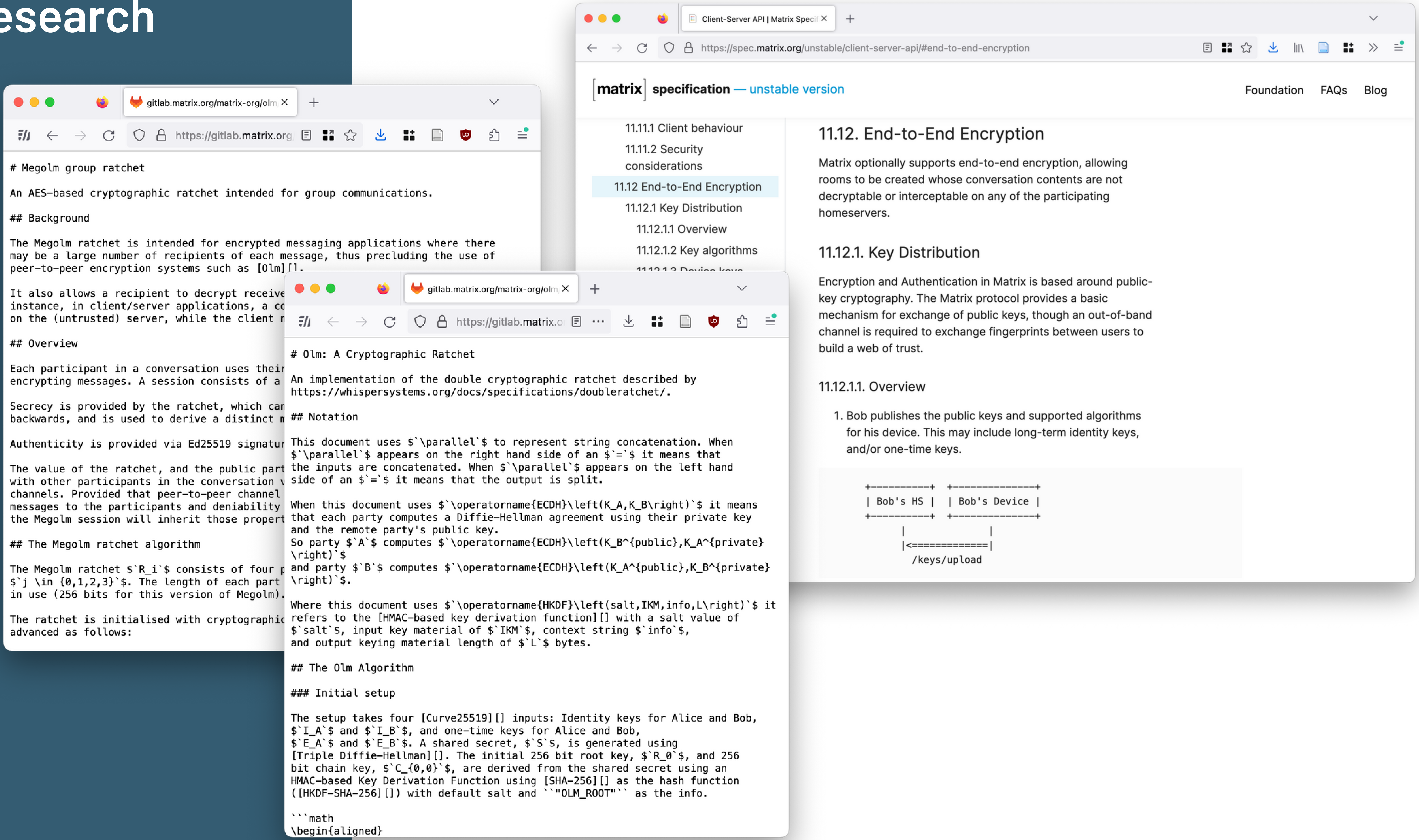
Encryption and Authentication in Matrix is based around public-key cryptography. The Matrix protocol provides a basic mechanism for exchange of public keys, though an out-of-band channel is required to exchange fingerprints between users to build a web of trust.

11.12.1.1. Overview

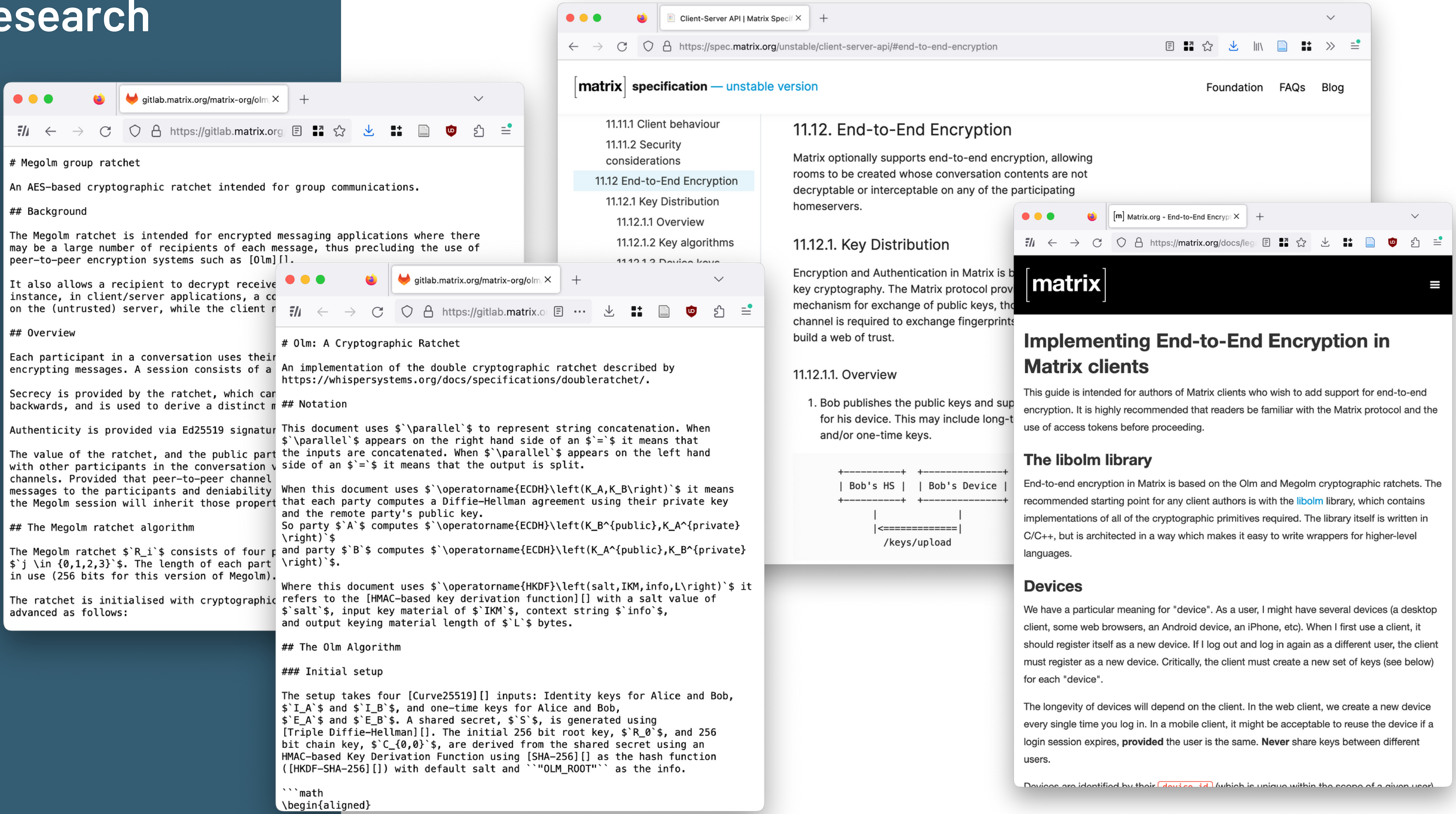
1. Bob publishes the public keys and supported algorithms for his device. This may include long-term identity keys, and/or one-time keys.

```
+----------+  +--------------+
| Bob's HS |  | Bob's Device |
+----------+  +--------------+
      |              |
      |<=============|
        /keys/upload
```

Research

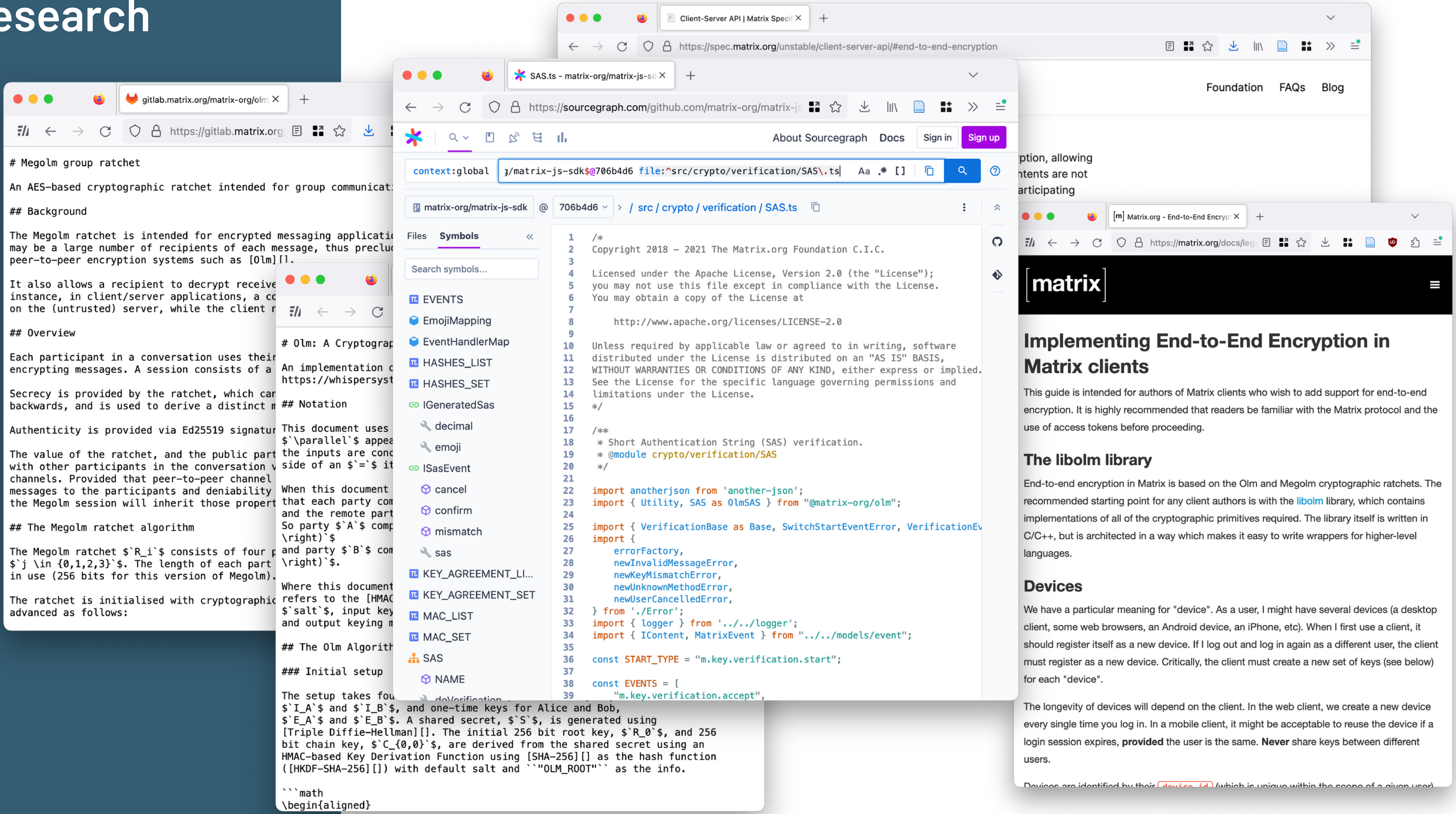

gitlab.matrix.org/matrix-org/olm
https://gitlab.matrix.org
Megolm group ratchet
An AES-based cryptographic ratchet intended for group communications.
Background
The Megolm ratchet is intended for encrypted messaging applications where there
may be a large number of recipients of each message, thus precluding the use of
peer-to-peer encryption systems such as [Olm][].
It also allows a recipient to decrypt receive
instance, in client/server applications, a co
on the (untrusted) server, while the client r
Overview
Each participant in a conversation uses their
encrypting messages. A session consists of a
Secrecy is provided by the ratchet, which can
backwards, and is used to derive a distinct m
Authenticity is provided via Ed25519 signatur
The value of the ratchet, and the public part
with other participants in the conversation v
channels. Provided that peer-to-peer channel
messages to the participants and deniability
the Megolm session will inherit those propert
The Megolm ratchet algorithm
The Megolm ratchet $`R_i`$ consists of four p
$`j \in {0,1,2,3}`$. The length of each part
in use (256 bits for this version of Megolm).
The ratchet is initialised with cryptographic
advanced as follows:
gitlab.matrix.org/matrix-org/olm
https://gitlab.matrix.o
Olm: A Cryptographic Ratchet
An implementation of the double cryptographic ratchet described by
https://whispersystems.org/docs/specifications/doubleratchet/.
Notation
This document uses $`\parallel`$ to represent string concatenation. When
$`\parallel`$ appears on the right hand side of an $`=`$ it means that
the inputs are concatenated. When $`\parallel`$ appears on the left hand
side of an $`=`$ it means that the output is split.
When this document uses $`\operatorname{ECDH}\left(K_A,K_B\right)`$ it means
that each party computes a Diffie-Hellman agreement using their private key
and the remote party's public key.
So party $`A`$ computes $`\operatorname{ECDH}\left(K_B^{public},K_A^{private}
\right)`$
and party $`B`$ computes $`\operatorname{ECDH}\left(K_A^{public},K_B^{private}
\right)`$.
Where this document uses $`\operatorname{HKDF}\left(salt,IKM,info,L\right)`$ it
refers to the [HMAC-based key derivation function][] with a salt value of
$`salt`$, input key material of $`IKM`$, context string $`info`$,
and output keying material length of $`L`$ bytes.
The Olm Algorithm
Initial setup
The setup takes four [Curve25519][] inputs: Identity keys for Alice and Bob,
$`I_A`$ and $`I_B`$, and one-time keys for Alice and Bob,
$`E_A`$ and $`E_B`$. A shared secret, $`S`$, is generated using
[Triple Diffie-Hellman][]. The initial 256 bit root key, $`R_0`$, and 256
bit chain key, $`C_{0,0}`$, are derived from the shared secret using an
HMAC-based Key Derivation Function using [SHA-256][] as the hash function
([HKDF-SHA-256][]) with default salt and ``"OLM_ROOT"`` as the info.
```math
\begin{aligned}
Client-Server API | Matrix Speci
https://spec.matrix.org/unstable/client-server-api/#end-to-end-encryption
matrix specification — unstable version
Foundation
FAQs
Blog
11.11.1 Client behaviour
11.11.2 Security considerations
11.12 End-to-End Encryption
11.12.1 Key Distribution
11.12.1.1 Overview
11.12.1.2 Key algorithms
11.12. End-to-End Encryption
Matrix optionally supports end-to-end encryption, allowing rooms to be created whose conversation contents are not decryptable or interceptable on any of the participating homeservers.
11.12.1. Key Distribution
Encryption and Authentication in Matrix is based around public-key cryptography. The Matrix protocol provides a basic mechanism for exchange of public keys, though an out-of-band channel is required to exchange fingerprints between users to build a web of trust.
11.12.1.1. Overview
1. Bob publishes the public keys and supported algorithms for his device. This may include long-term identity keys, and/or one-time keys.
Bob's HS
Bob's Device
/keys/upload
```

Research



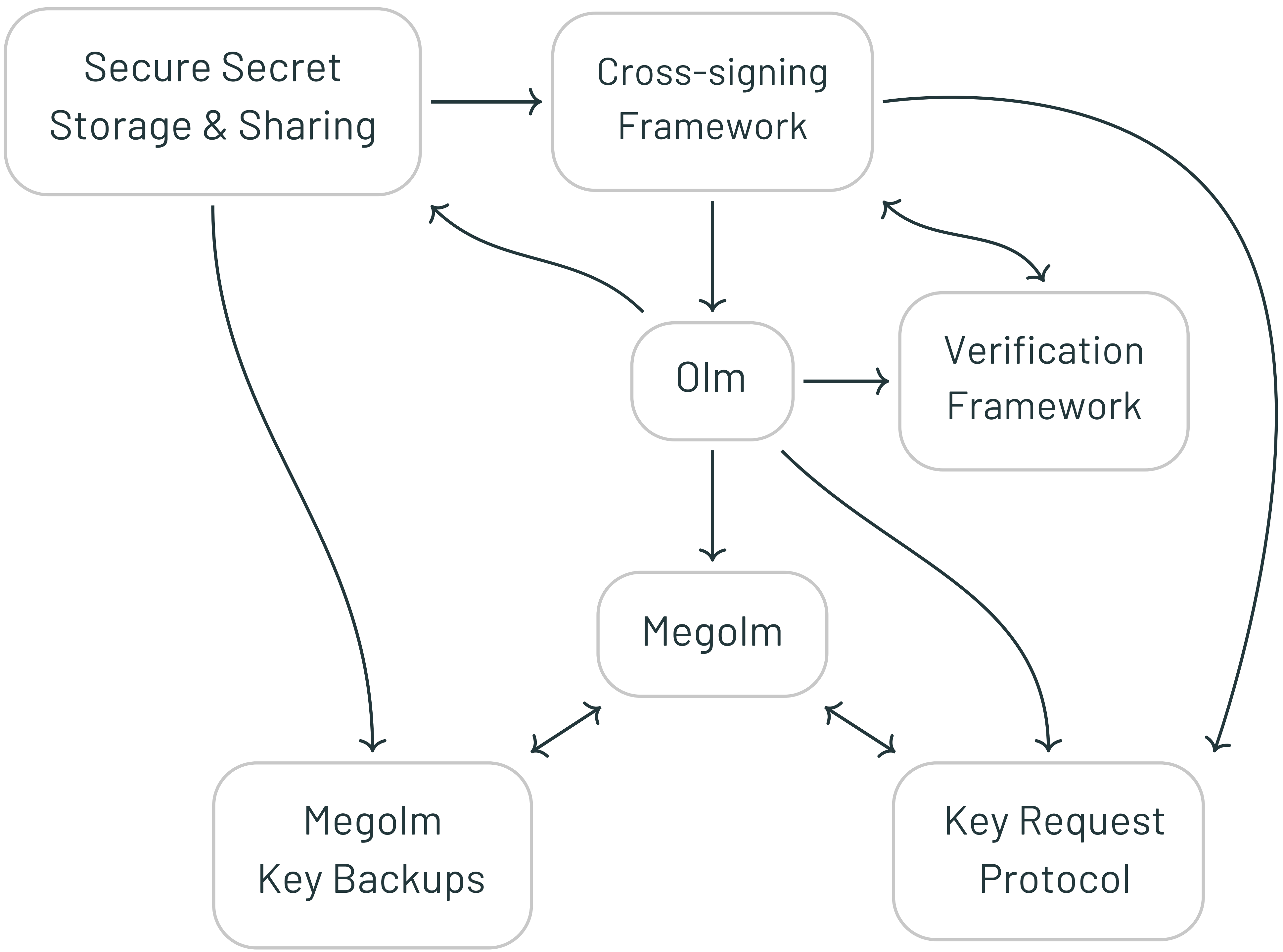
Client-Server API | Matrix Speci
https://spec.matrix.org/unstable/client-server-api/#end-to-end-encryption
matrix specification — unstable version
Foundation
FAQs
Blog
11.11.1 Client behaviour
11.11.2 Security considerations
11.12 End-to-End Encryption
11.12.1 Key Distribution
11.12.1.1 Overview
11.12.1.2 Key algorithms
11.12. End-to-End Encryption
Matrix optionally supports end-to-end encryption, allowing rooms to be created whose conversation contents are not decryptable or interceptable on any of the participating homeservers.
11.12.1. Key Distribution
11.12.1.1. Overview
Bob's HS
Bob's Device
/keys/upload
gitlab.matrix.org/matrix-org/olm
https://gitlab.matrix.org
Megolm group ratchet
An AES-based cryptographic ratchet intended for group communications.
Background
The Megolm ratchet is intended for encrypted messaging applications where there may be a large number of recipients of each message, thus precluding the use of peer-to-peer encryption systems such as [Olm][].
Overview
The Megolm ratchet algorithm
Olm: A Cryptographic Ratchet
An implementation of the double cryptographic ratchet described by https://whispersystems.org/docs/specifications/doubleratchet/.
Notation
This document uses $`\parallel`$ to represent string concatenation. When $`\parallel`$ appears on the right hand side of an $`=`$ it means that the inputs are concatenated. When $`\parallel`$ appears on the left hand side of an $`=`$ it means that the output is split.
When this document uses $`\operatorname{ECDH}\left(K_A,K_B\right)`$ it means that each party computes a Diffie-Hellman agreement using their private key and the remote party's public key. So party $`A`$ computes $`\operatorname{ECDH}\left(K_B^{public},K_A^{private}\right)`$ and party $`B`$ computes $`\operatorname{ECDH}\left(K_A^{public},K_B^{private}\right)`$.
Where this document uses $`\operatorname{HKDF}\left(salt,IKM,info,L\right)`$ it refers to the [HMAC-based key derivation function][] with a salt value of $`salt`$, input key material of $`IKM`$, context string $`info`$, and output keying material length of $`L`$ bytes.
The Olm Algorithm
Initial setup
The setup takes four [Curve25519][] inputs: Identity keys for Alice and Bob, $`I_A`$ and $`I_B`$, and one-time keys for Alice and Bob, $`E_A`$ and $`E_B`$. A shared secret, $`S`$, is generated using [Triple Diffie-Hellman][]. The initial 256 bit root key, $`R_0`$, and 256 bit chain key, $`C_{0,0}`$, are derived from the shared secret using an HMAC-based Key Derivation Function using [SHA-256][] as the hash function ([HKDF-SHA-256][]) with default salt and ``"OLM_ROOT"`` as the info.
```math
\begin{aligned}
Matrix.org - End-to-End Encryp
https://matrix.org/docs/leg
matrix
Implementing End-to-End Encryption in Matrix clients
This guide is intended for authors of Matrix clients who wish to add support for end-to-end encryption. It is highly recommended that readers be familiar with the Matrix protocol and the use of access tokens before proceeding.
The libolm library
End-to-end encryption in Matrix is based on the Olm and Megolm cryptographic ratchets. The recommended starting point for any client authors is with the libolm library, which contains implementations of all of the cryptographic primitives required. The library itself is written in C/C++, but is architected in a way which makes it easy to write wrappers for higher-level languages.
Devices
We have a particular meaning for "device". As a user, I might have several devices (a desktop client, some web browsers, an Android device, an iPhone, etc). When I first use a client, it should register itself as a new device. If I log out and log in again as a different user, the client must register as a new device. Critically, the client must create a new set of keys (see below) for each "device".
The longevity of devices will depend on the client. In the web client, we create a new device every single time you log in. In a mobile client, it might be acceptable to reuse the device if a login session expires, provided the user is the same. Never share keys between different users.
```

Research



gitlab.matrix.org/matrix-org/olm
https://gitlab.matrix.org
Megolm group ratchet
An AES-based cryptographic ratchet intended for group communicat
Background
The Megolm ratchet is intended for encrypted messaging applicati
may be a large number of recipients of each message, thus preclu
peer-to-peer encryption systems such as [Olm][].
It also allows a recipient to decrypt receive
instance, in client/server applications, a co
on the (untrusted) server, while the client r
Overview
Each participant in a conversation uses their
encrypting messages. A session consists of a
Secrecy is provided by the ratchet, which can
backwards, and is used to derive a distinct m
Authenticity is provided via Ed25519 signatur
The value of the ratchet, and the public part
with other participants in the conversation v
channels. Provided that peer-to-peer channel
messages to the participants and deniability
the Megolm session will inherit those propert
The Megolm ratchet algorithm
The Megolm ratchet $`R_i`$ consists of four p
$`j \in {0,1,2,3}`$. The length of each part
in use (256 bits for this version of Megolm).
The ratchet is initialised with cryptographic
advanced as follows:
Olm: A Cryptograp
An implementation c
https://whispersyst
Notation
This document uses
$`\parallel`$ appea
the inputs are conc
side of an $`=`$ it
When this document
that each party com
and the remote part
So party $`A`$ comp
\right)`$
and party $`B`$ com
\right)`$.
Where this document
refers to the [HMAC
$`salt`$, input key
and output keying m
The Olm Algorith
Initial setup
The setup takes fou
$`I_A`$ and $`I_B`$, and one-time keys for Alice and Bob,
$`E_A`$ and $`E_B`$. A shared secret, $`S`$, is generated using
[Triple Diffie-Hellman][]. The initial 256 bit root key, $`R_0`$, and 256
bit chain key, $`C_{0,0}`$, are derived from the shared secret using an
HMAC-based Key Derivation Function using [SHA-256][] as the hash function
([HKDF-SHA-256][]) with default salt and ``"OLM_ROOT"`` as the info.
```math
\begin{aligned}
SAS.ts - matrix-org/matrix-js-sd
https://sourcegraph.com/github.com/matrix-org/matrix-js
About Sourcegraph
Docs
Sign in
Sign up
context:global
/matrix-js-sdk$@706b4d6 file:^src/crypto/verification/SAS\.ts
matrix-org/matrix-js-sdk
706b4d6
/ src / crypto / verification / SAS.ts
Files
Symbols
Search symbols...
EVENTS
EmojiMapping
EventHandlerMap
HASHES_LIST
HASHES_SET
IGeneratedSas
decimal
emoji
ISasEvent
cancel
confirm
mismatch
sas
KEY_AGREEMENT_LI...
KEY_AGREEMENT_SET
MAC_LIST
MAC_SET
SAS
NAME
/*
Copyright 2018 - 2021 The Matrix.org Foundation C.I.C.
Licensed under the Apache License, Version 2.0 (the "License");
you may not use this file except in compliance with the License.
You may obtain a copy of the License at
http://www.apache.org/licenses/LICENSE-2.0
Unless required by applicable law or agreed to in writing, software
distributed under the License is distributed on an "AS IS" BASIS,
WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
See the License for the specific language governing permissions and
limitations under the License.
*/
/**
* Short Authentication String (SAS) verification.
* @module crypto/verification/SAS
*/
import anotherjson from 'another-json';
import { Utility, SAS as OlmSAS } from "@matrix-org/olm";
import { VerificationBase as Base, SwitchStartEventError, VerificationEv
import {
errorFactory,
newInvalidMessageError,
newKeyMismatchError,
newUnknownMethodError,
newUserCancelledError,
} from './Error';
import { logger } from '../../logger';
import { IContent, MatrixEvent } from "../../models/event";
const START_TYPE = "m.key.verification.start";
const EVENTS = [
"m.key.verification.accept",
Client-Server API | Matrix Speci
https://spec.matrix.org/unstable/client-server-api/#end-to-end-encryption
Foundation
FAQs
Blog
ption, allowing
ntents are not
articipating
Matrix.org - End-to-End Encryp
https://matrix.org/docs/leg
[matrix]
Implementing End-to-End Encryption in Matrix clients
This guide is intended for authors of Matrix clients who wish to add support for end-to-end encryption. It is highly recommended that readers be familiar with the Matrix protocol and the use of access tokens before proceeding.
The libolm library
End-to-end encryption in Matrix is based on the Olm and Megolm cryptographic ratchets. The recommended starting point for any client authors is with the libolm library, which contains implementations of all of the cryptographic primitives required. The library itself is written in C/C++, but is architected in a way which makes it easy to write wrappers for higher-level languages.
Devices
We have a particular meaning for "device". As a user, I might have several devices (a desktop client, some web browsers, an Android device, an iPhone, etc). When I first use a client, it should register itself as a new device. If I log out and log in again as a different user, the client must register as a new device. Critically, the client must create a new set of keys (see below) for each "device".
The longevity of devices will depend on the client. In the web client, we create a new device every single time you log in. In a mobile client, it might be acceptable to reuse the device if a login session expires, provided the user is the same. Never share keys between different users.
```

Matrix Overview



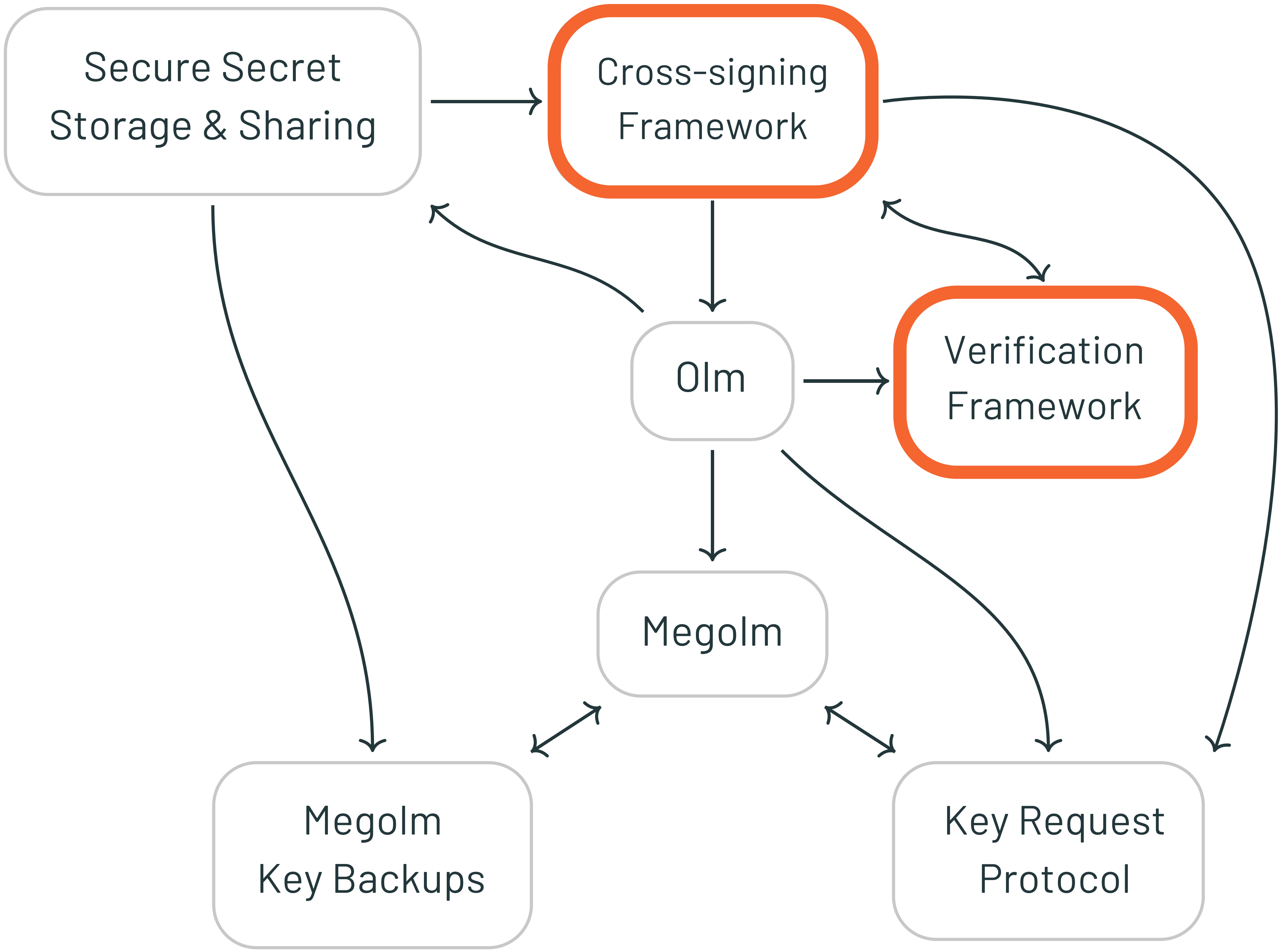
Cross-signing

- Cryptographic identities for users and their devices.

& Verification

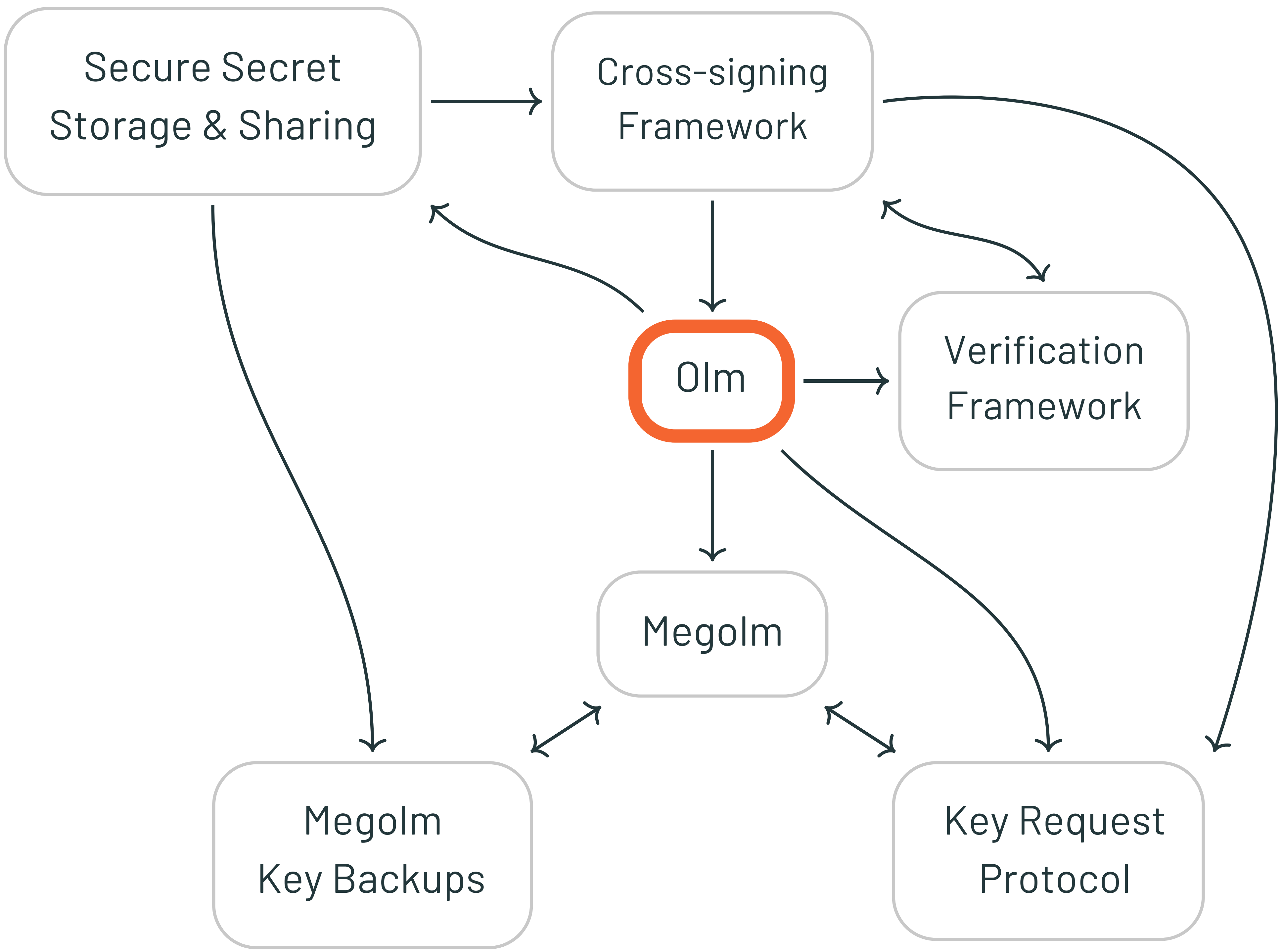
- *Self-verification*
  Users sign their own devices to indicate trust.

- *Cross-signing*
  Users sign each other's identities.



Olm

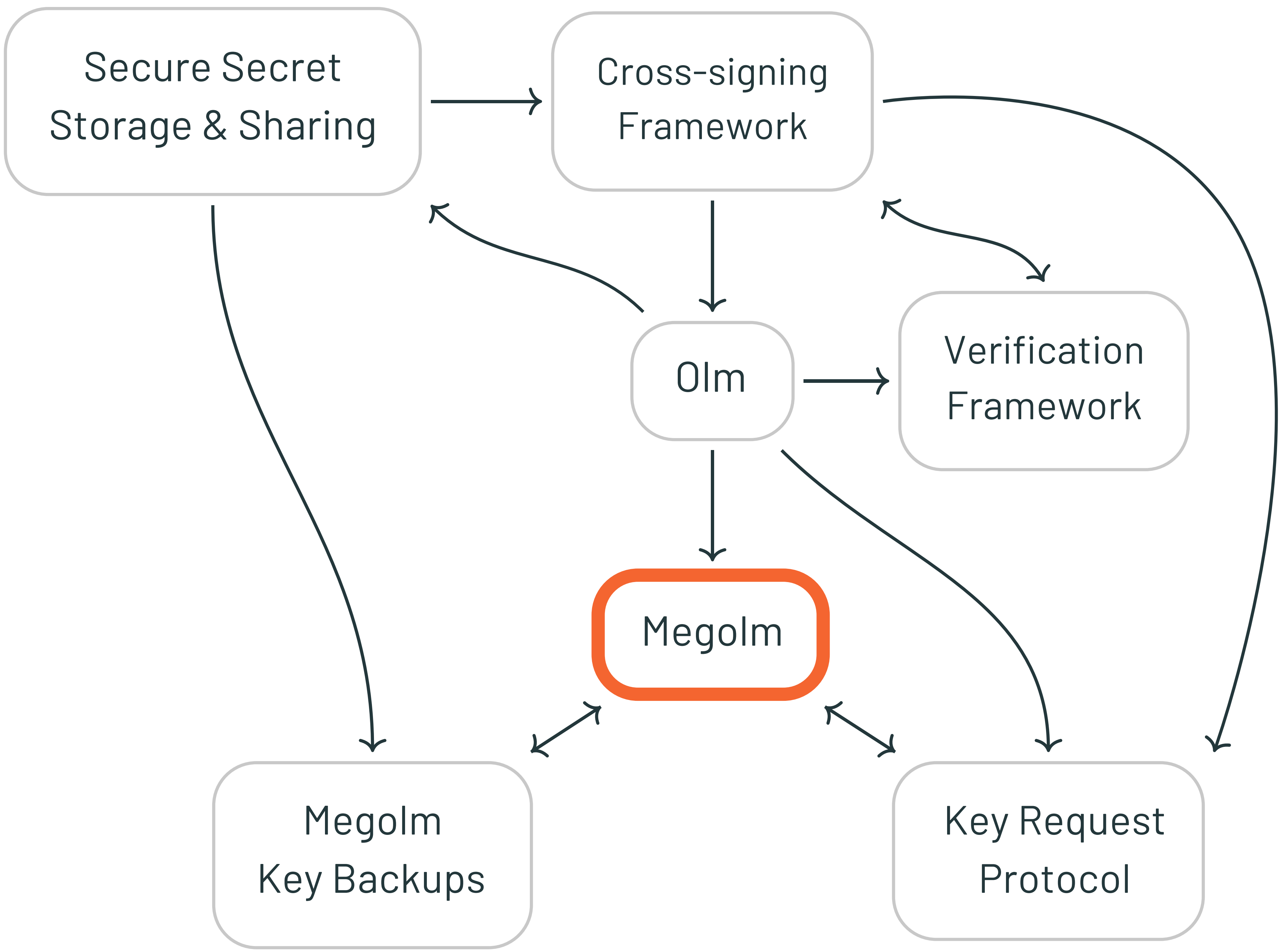
- Secure pairwise channels between devices.
- Initial key exchange via *Triple Diffie-Hellman.*
- Continuous key exchange via *Double Ratchet.*
- Signalling layer for other sub-protocols in Matrix.



Megolm

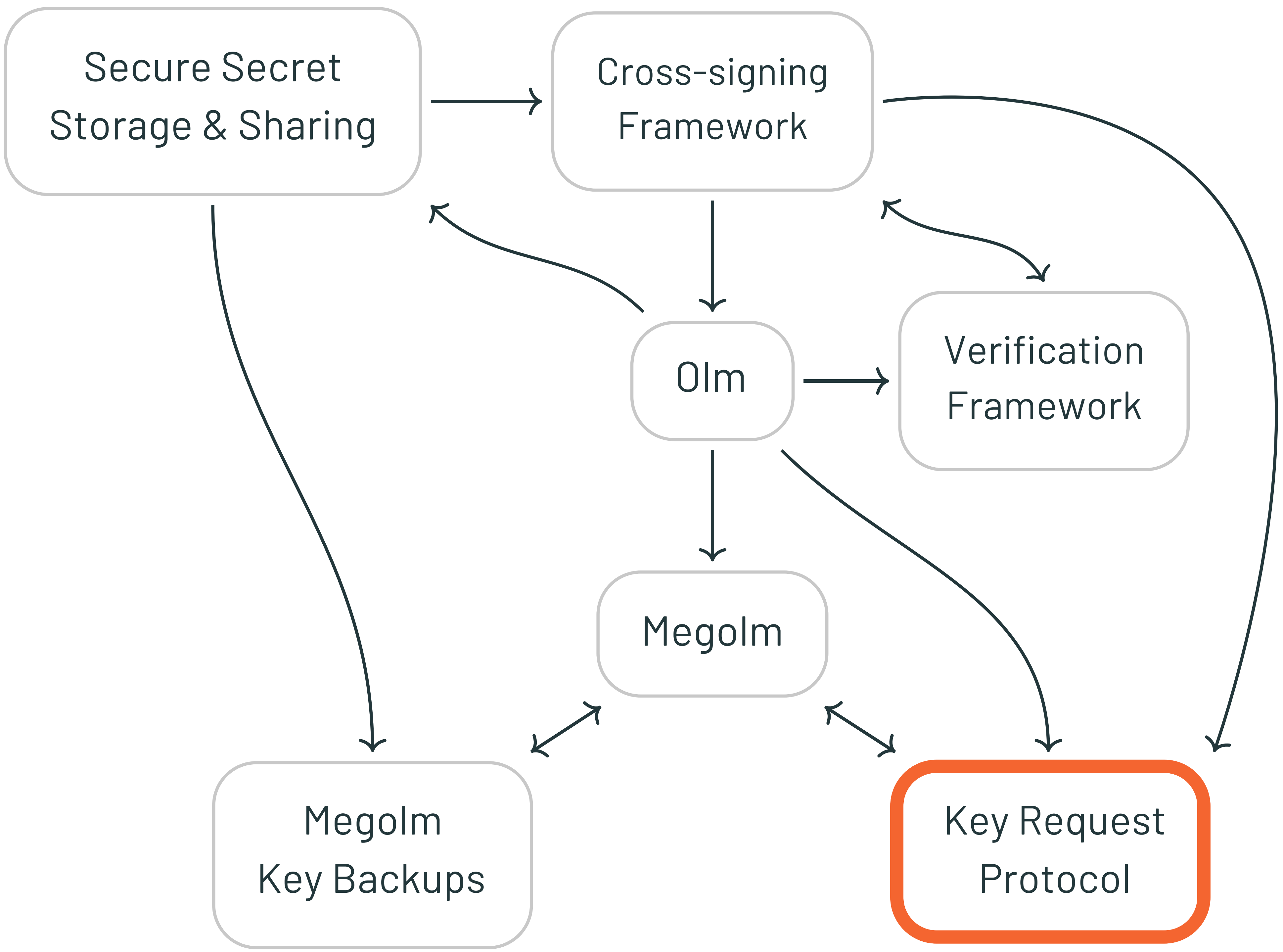
- Secure one-to-many channel.
- Symmetric ratchet for forward security.
- Session keys are distributed over Olm.
- Each sender maintains their own Megolm session.

⤷ Compose together to form group chat.



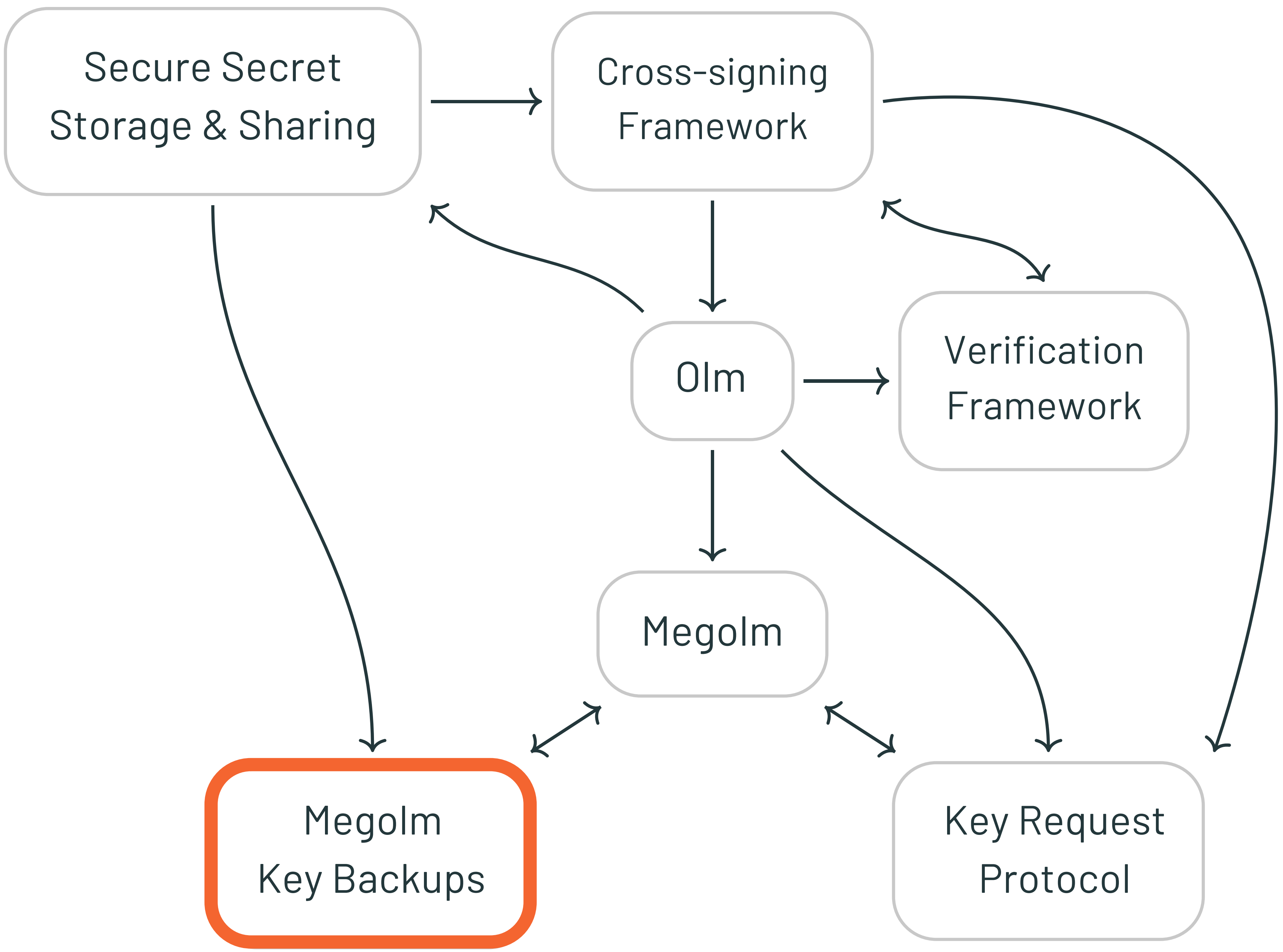
Key Request Protocol

- Request-response protocol.
- Allows devices to request and share keys between each other.
- Responding device must ensure requesting device is entitled to the keys.
- Keys are shared over Olm.



Megolm Key Backups

- Asynchronous alternative to Key Request protocol.
- Inbound Megolm sessions are encrypted and saved to the homeserver.
- Encrypt using a secret key shared between a user's devices.



Secure Secret Storage & Sharing

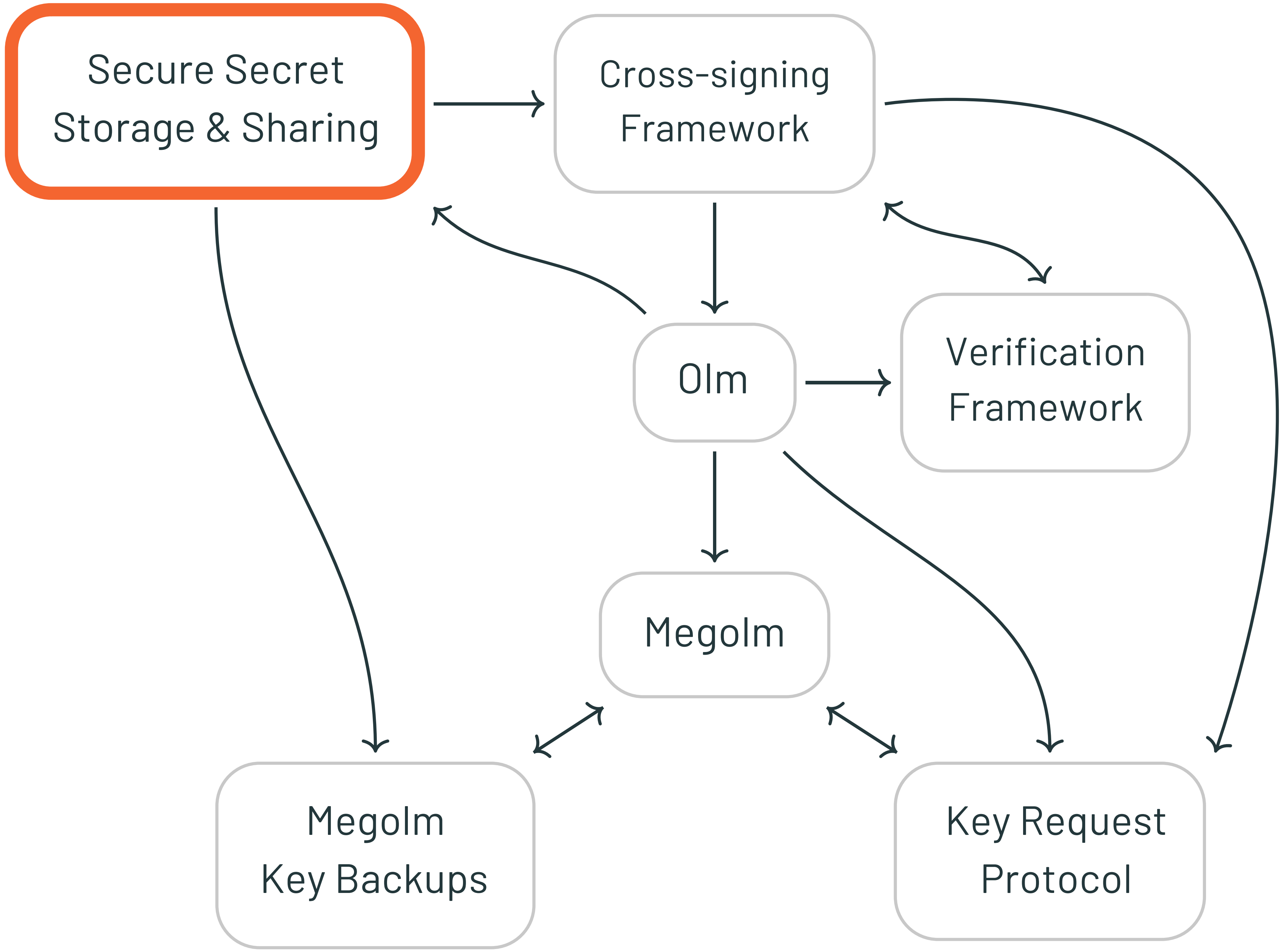
Backup, recover and share *user-level secrets.*

Secret Storage:

- Encrypt secrets and store on homeserver.
- Shared symmetric key (may be password-derived).

Secret Sharing:

- Use Olm to share secrets to newly verified devices.



Secure Secret Storage & Sharing

Backup, recover and share *user-level secrets.*

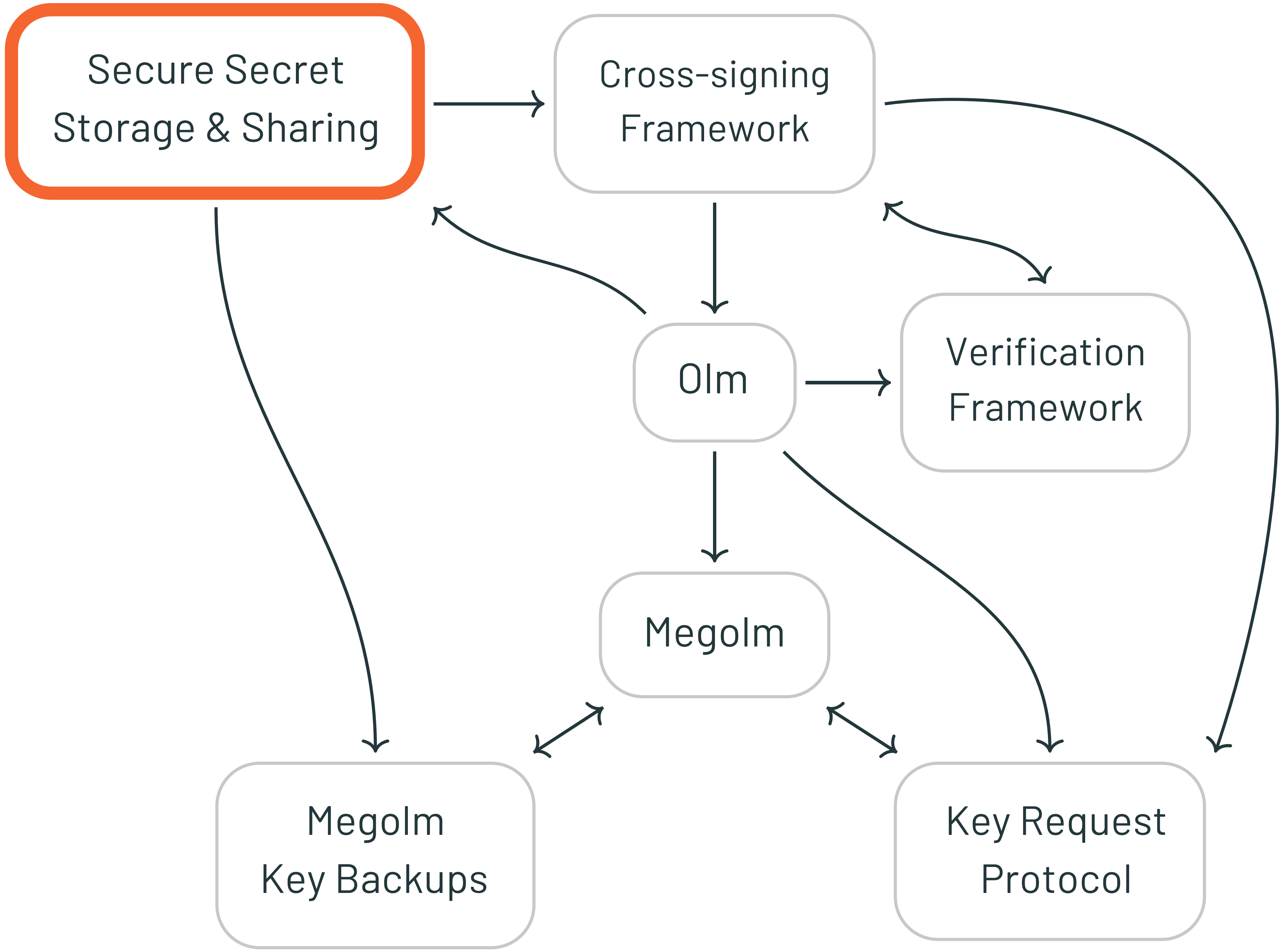
Secret Storage:

- Encrypt secrets and store on homeserver.
- Shared symmetric key (may be password-derived).

Root secret for a user's account

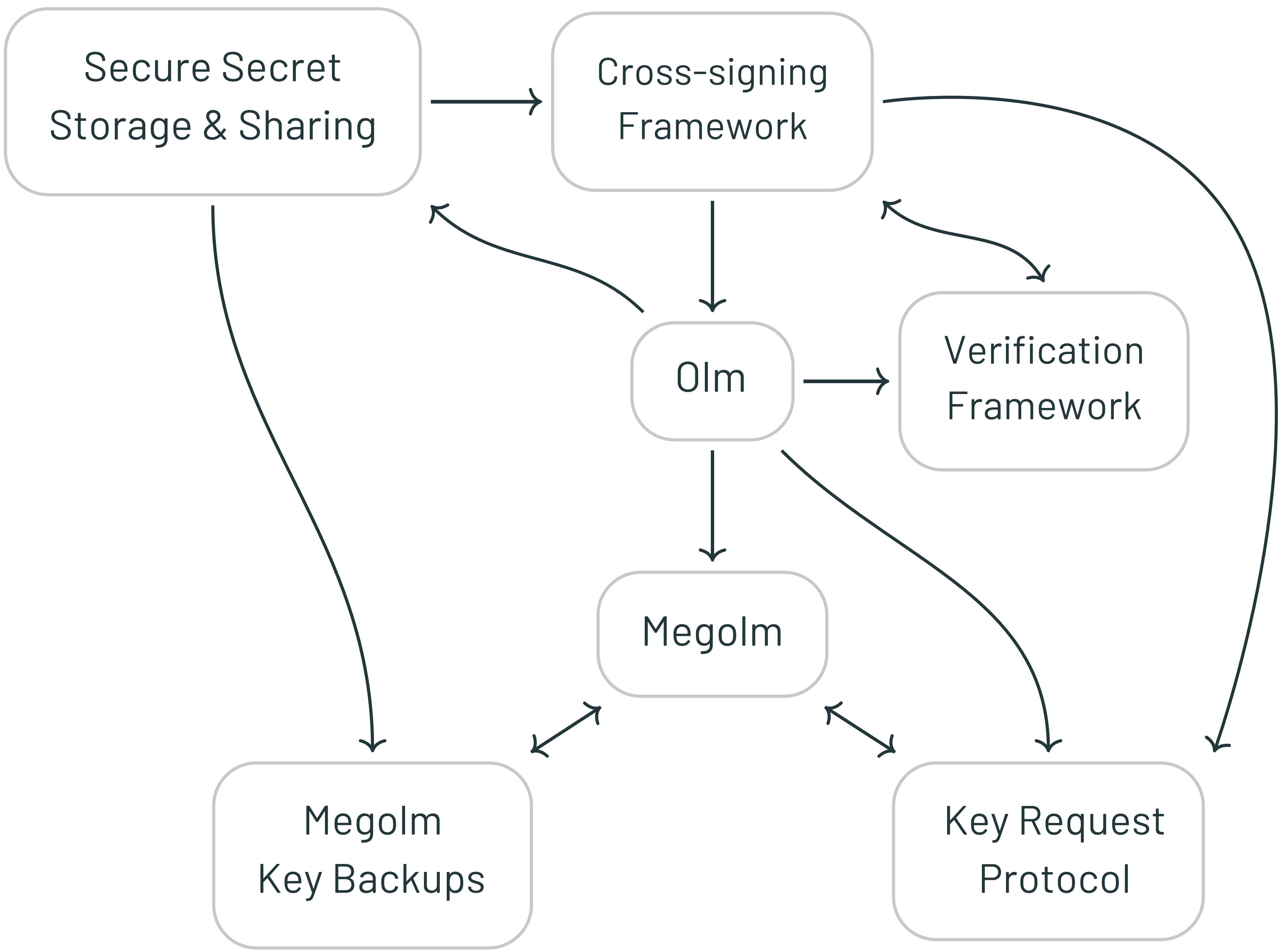
Secret Sharing:

- Use Olm to share secrets to newly verified devices.



Modelling Matrix & Finding Attacks

We found many of these attacks while formalising each sub-protocol as part of the modelling work.



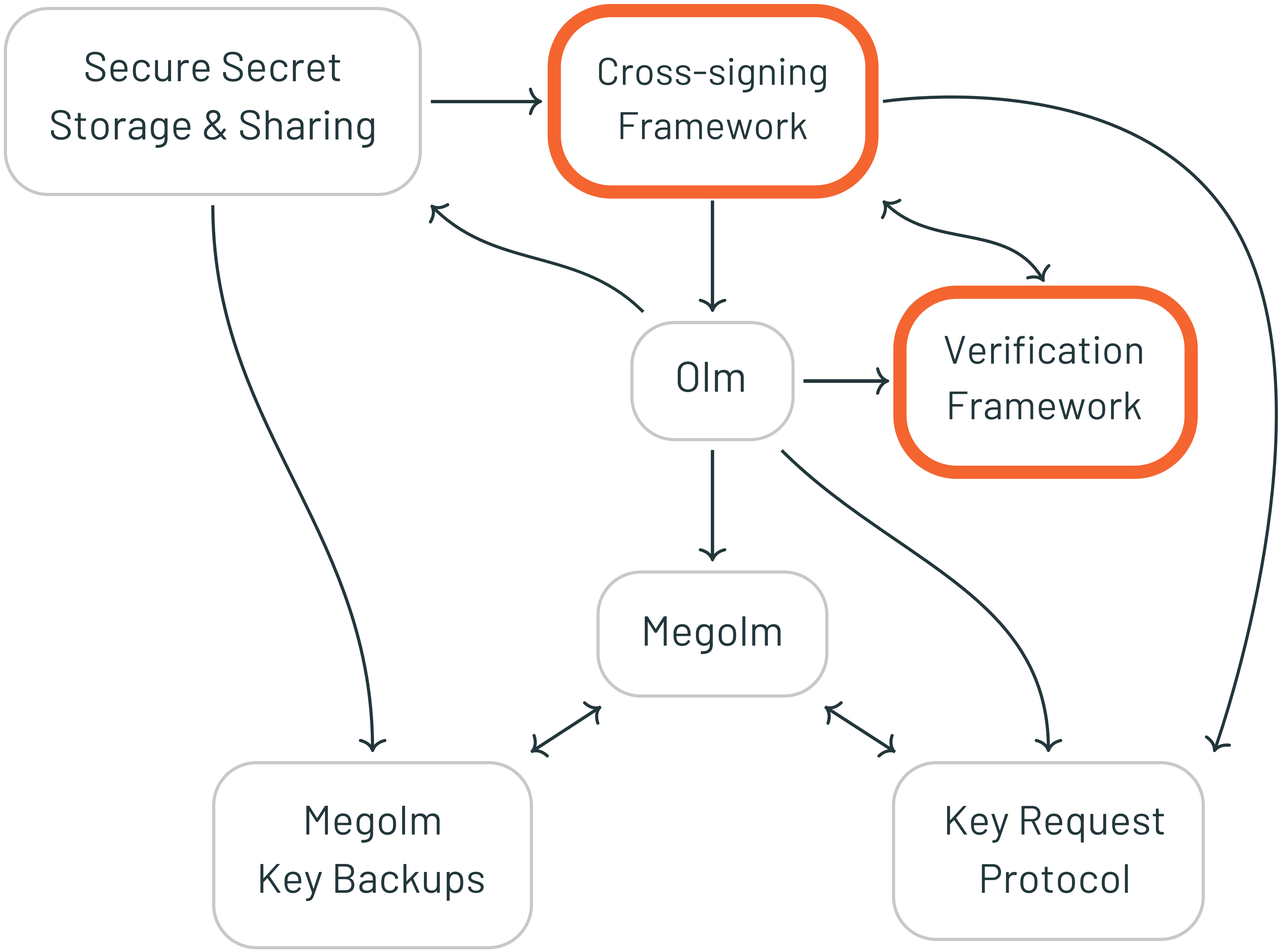
Cross-signing

- Cryptographic identities for users and their devices.

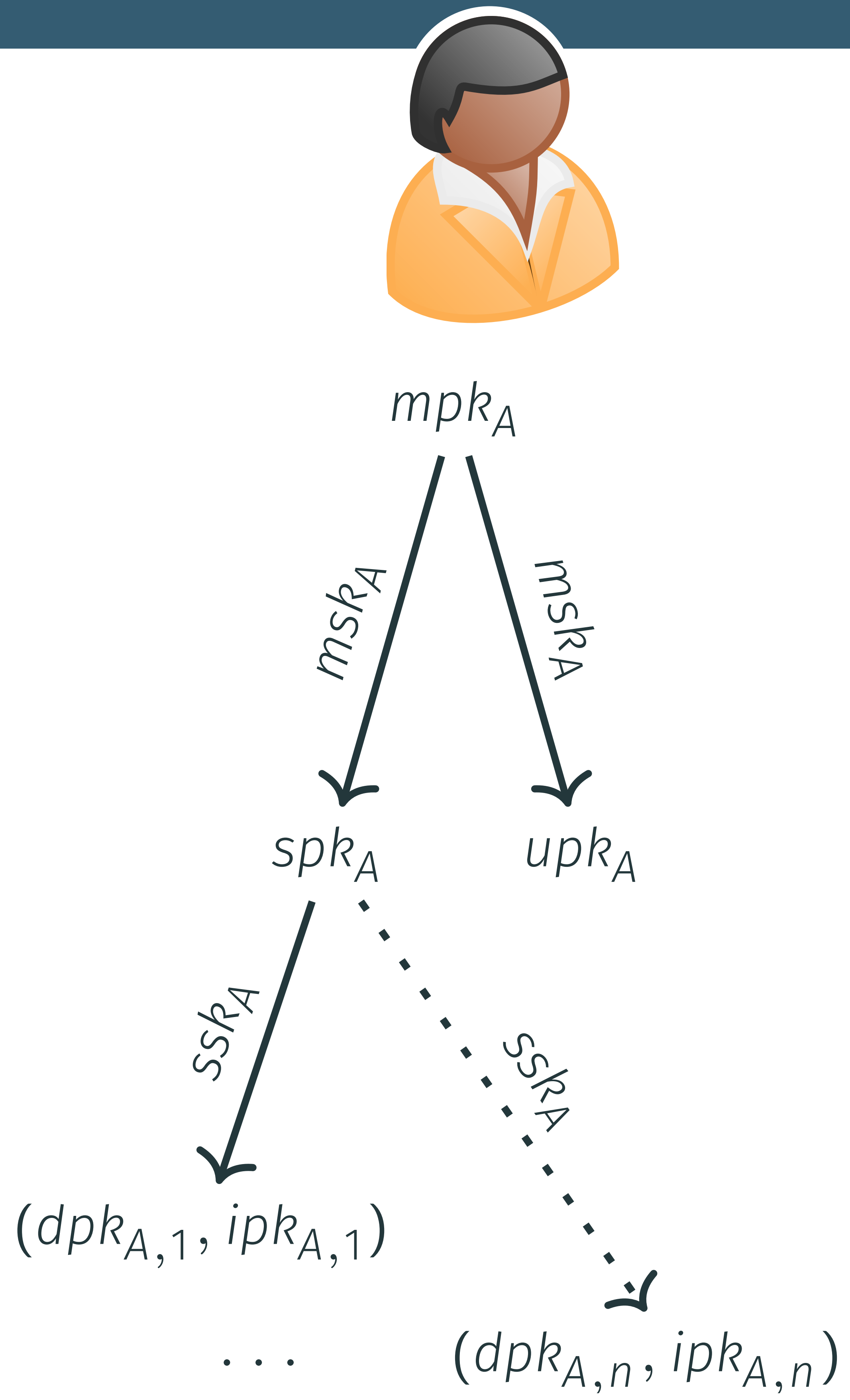
& Verification

- *Self-verification*
  Users sign their own devices to indicate trust.

- *Cross-signing*
  Users sign each other's identities.

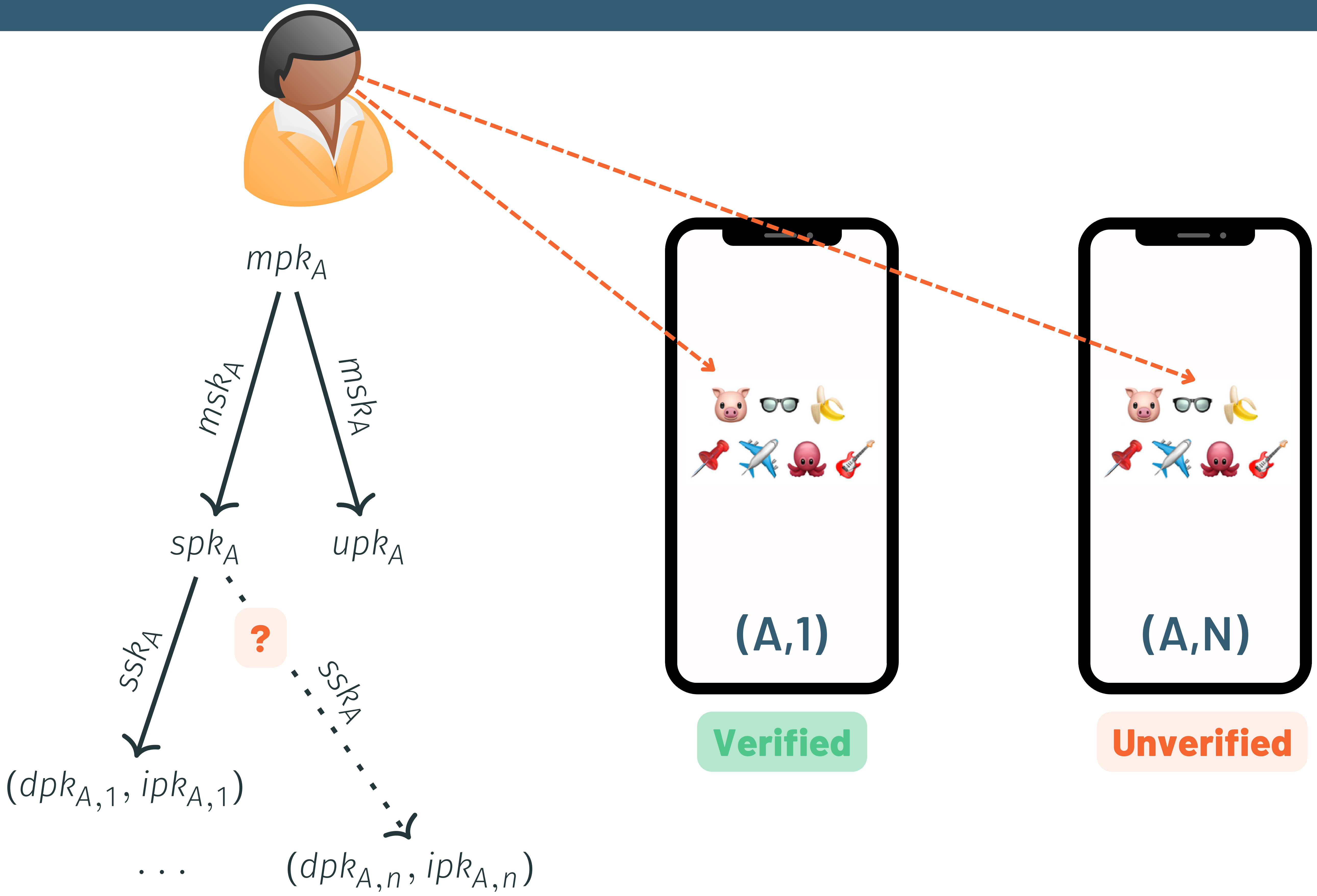


Cross-signing and Verification



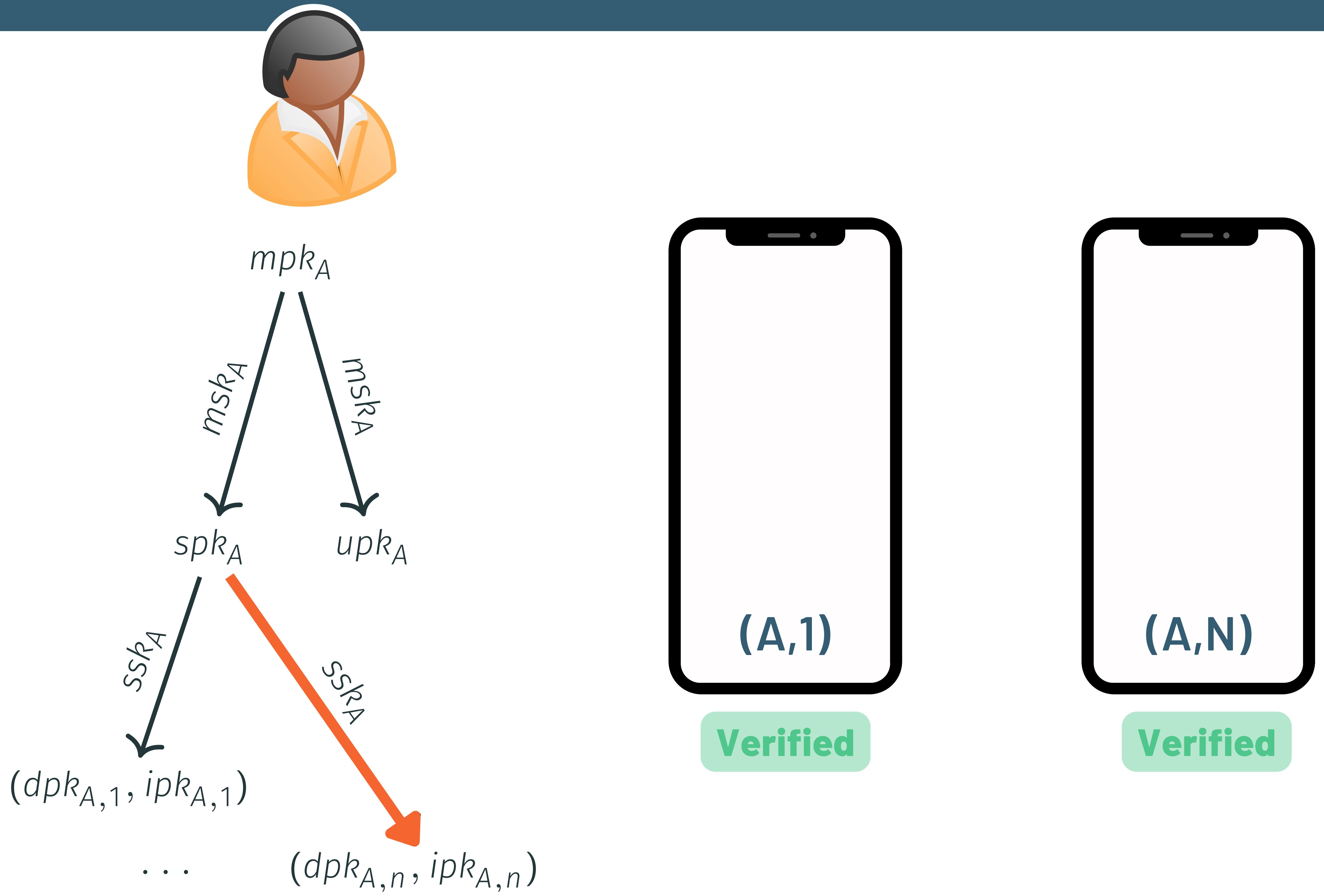
Cross-signing and Verification

Device-to-device verification

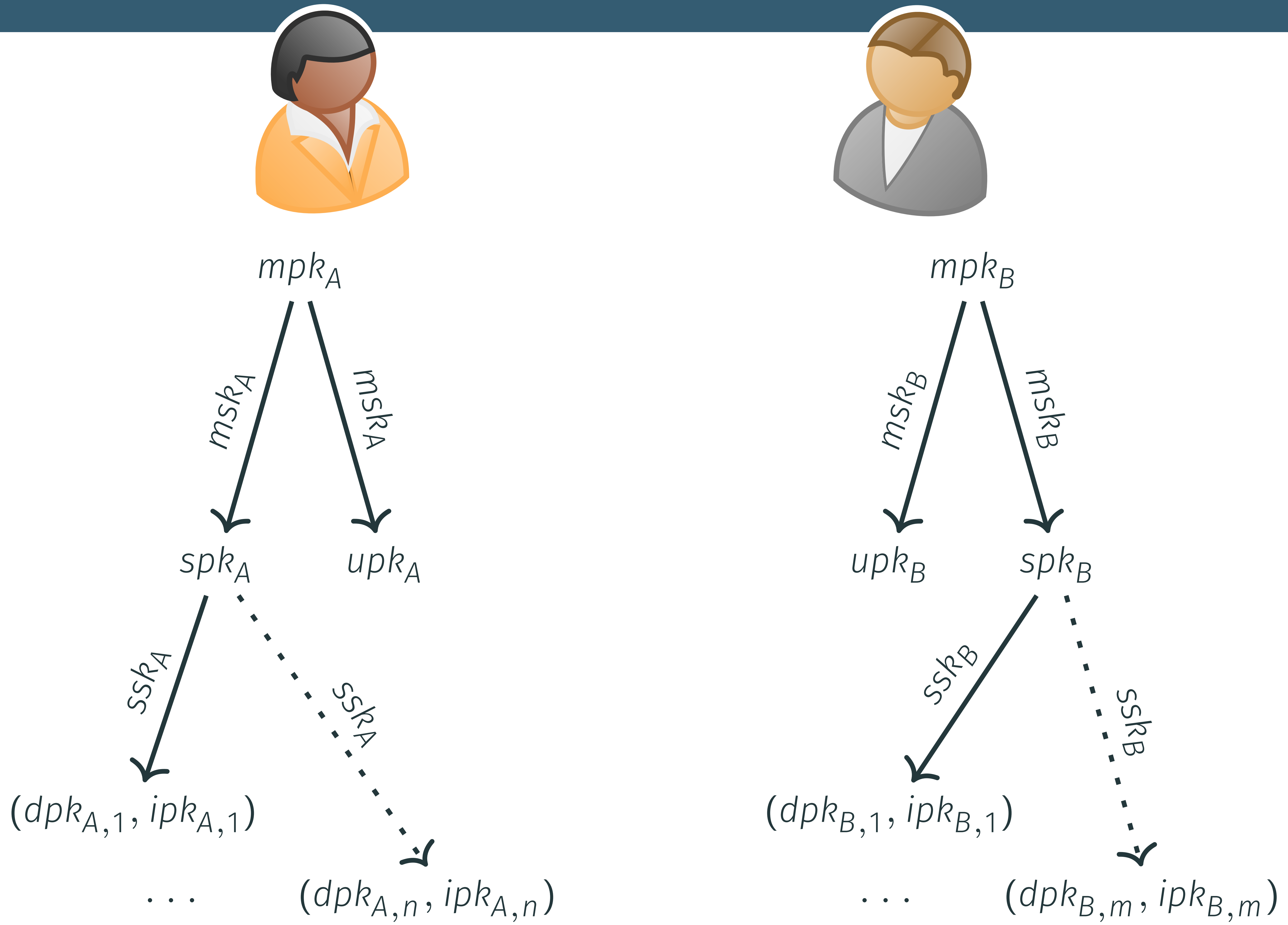


Cross-signing and Verification

Device-to-device verification

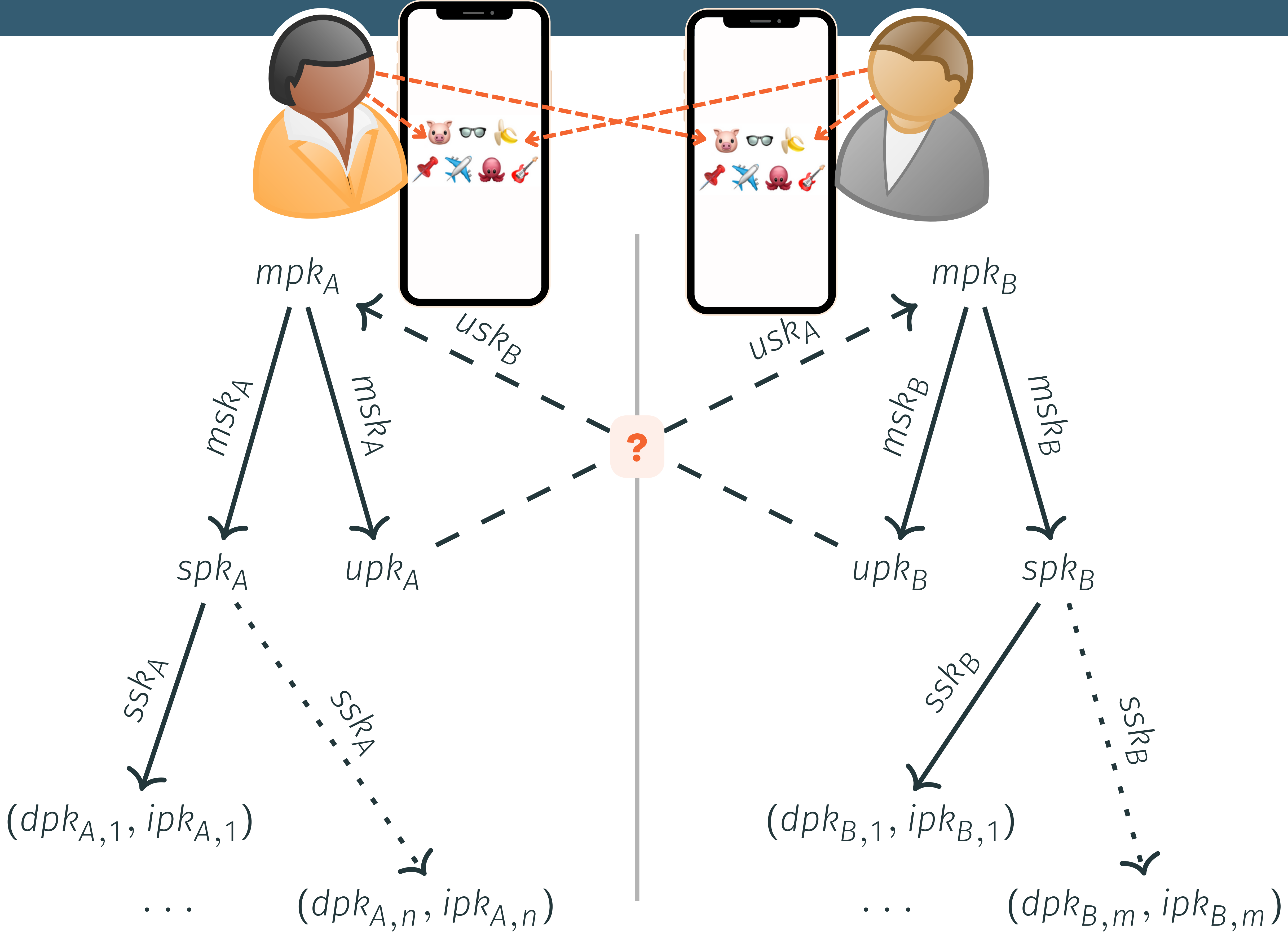


Cross-signing and Verification



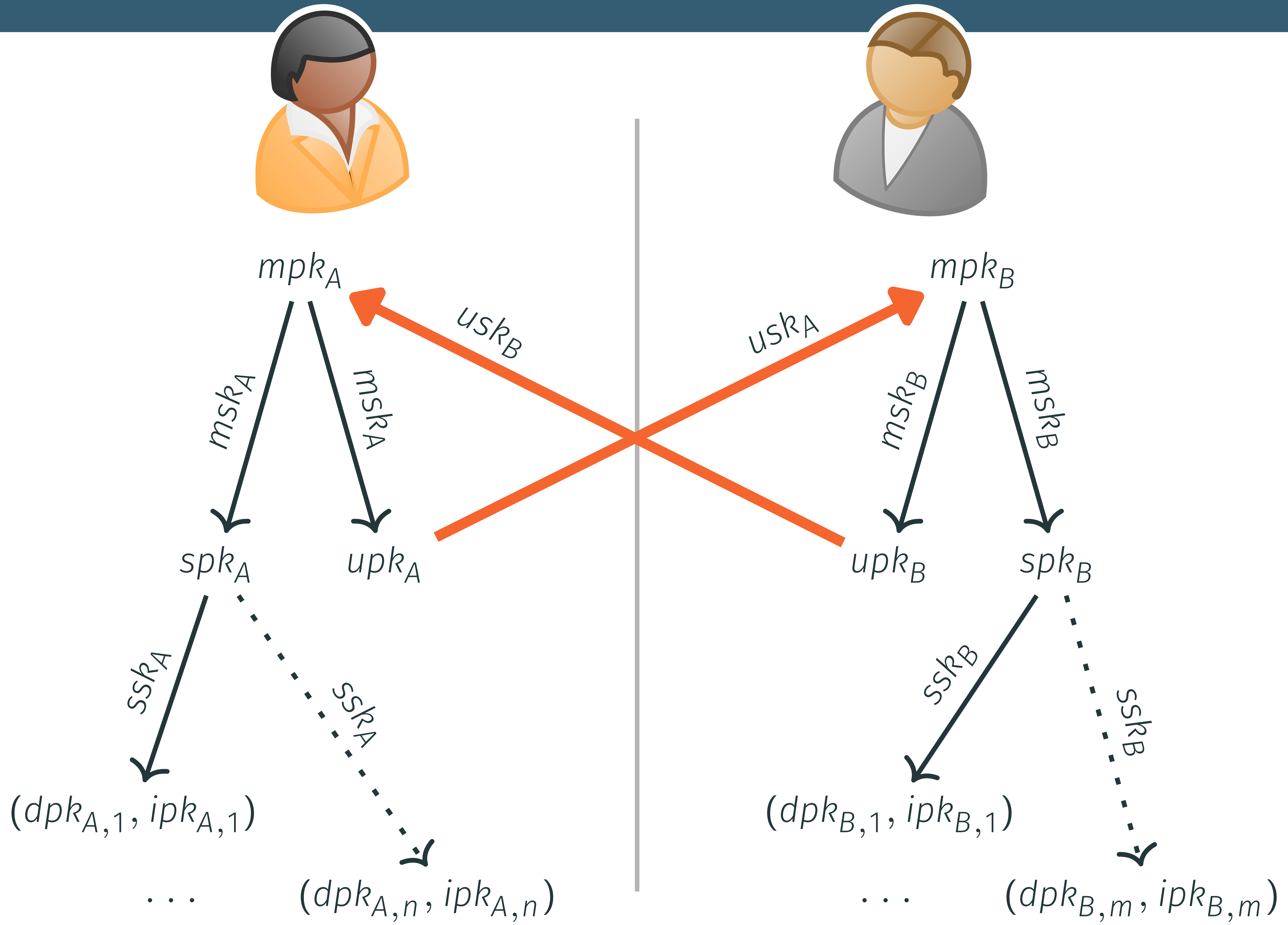
Cross-signing and Verification

User-to-user verification



Cross-signing and Verification

User-to-user verification

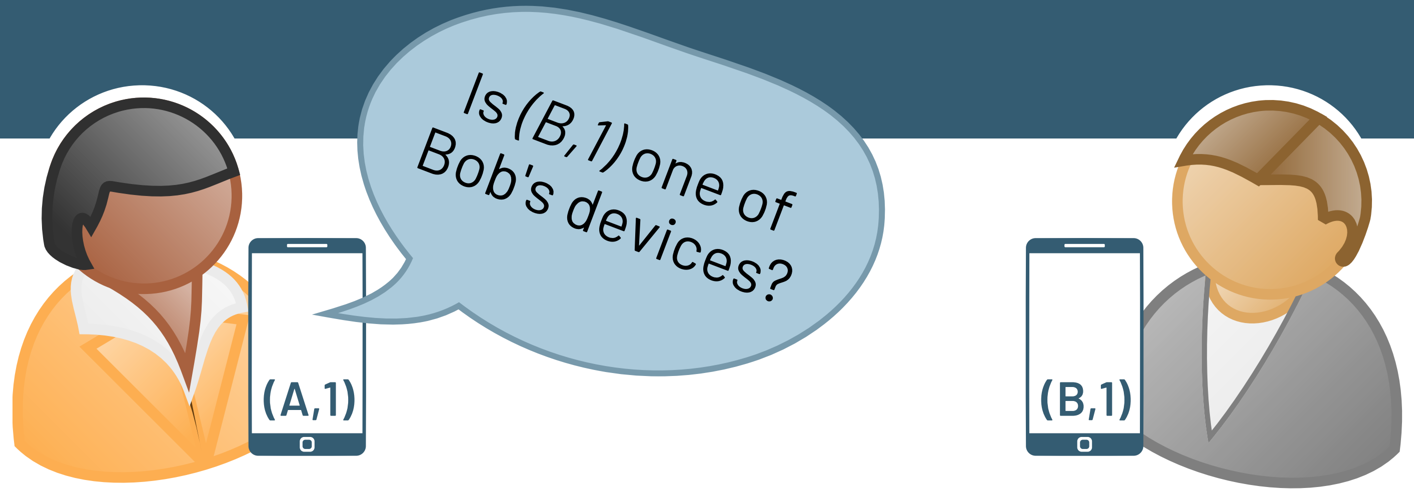


Cross-signing and Verification

Verifying device identities

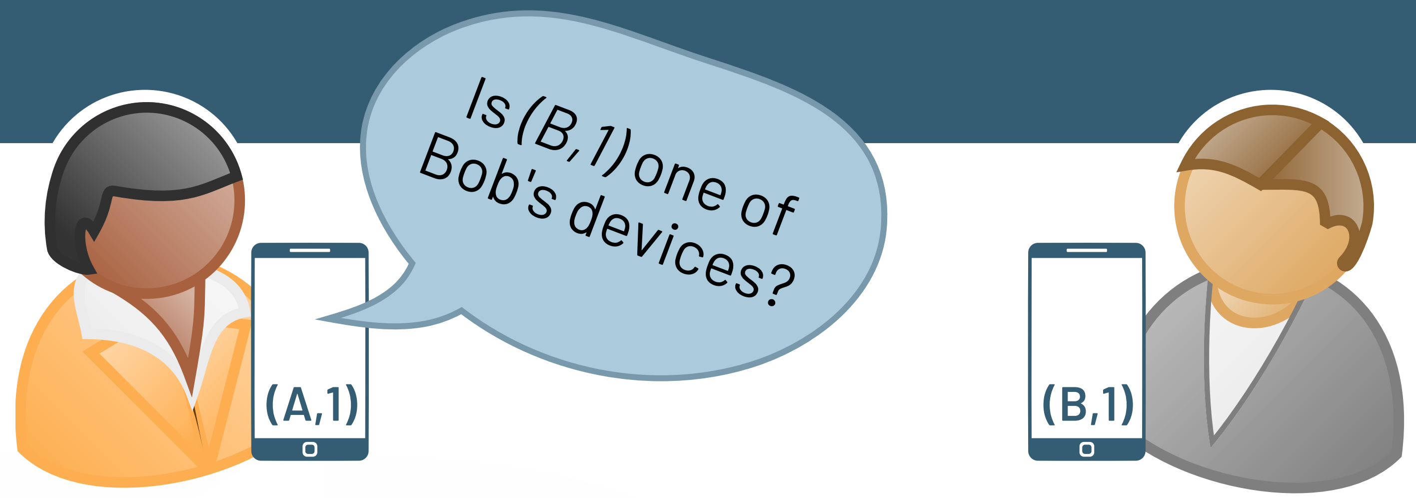
Cross-signing and Verification

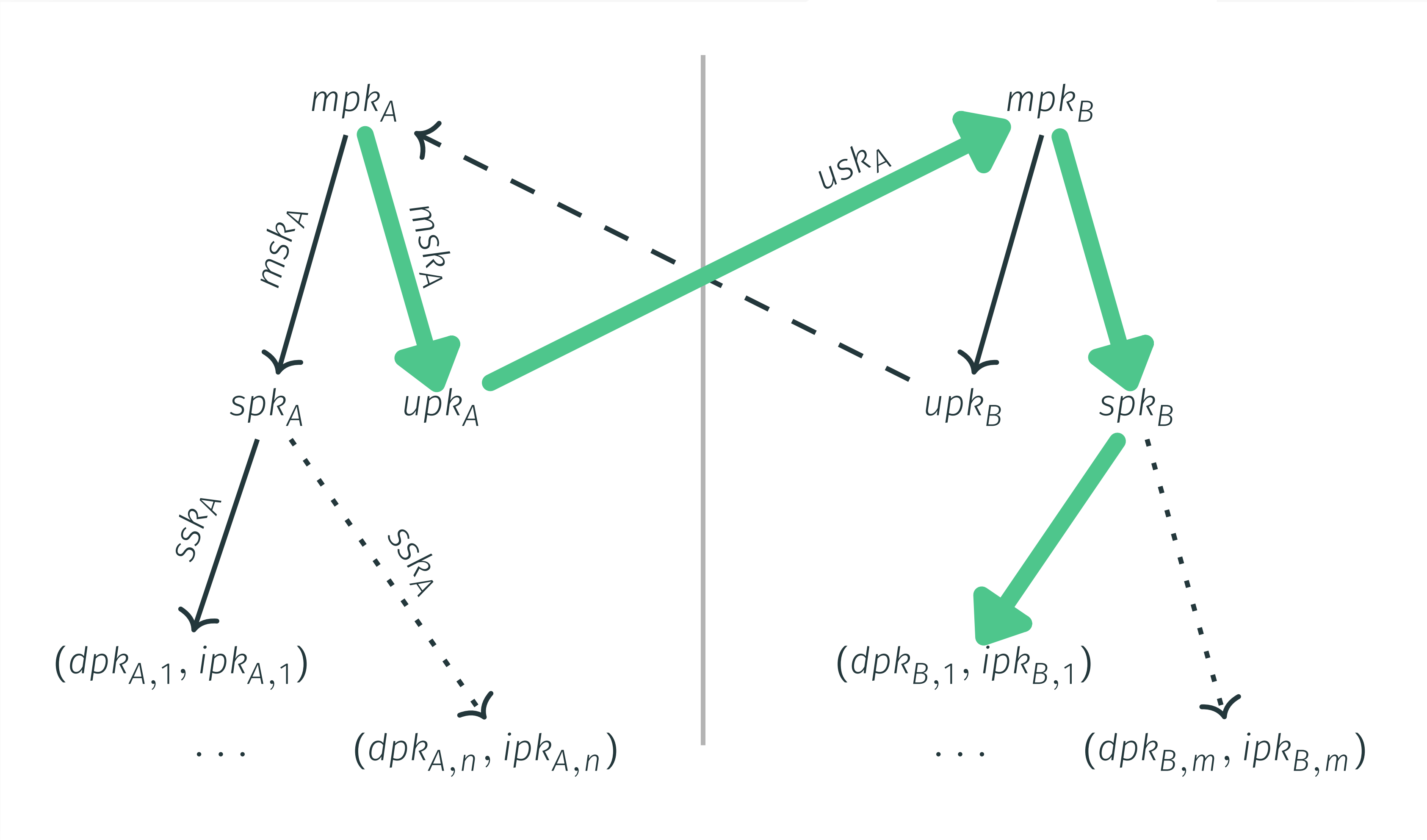
Verifying device identities



Cross-signing and Verification

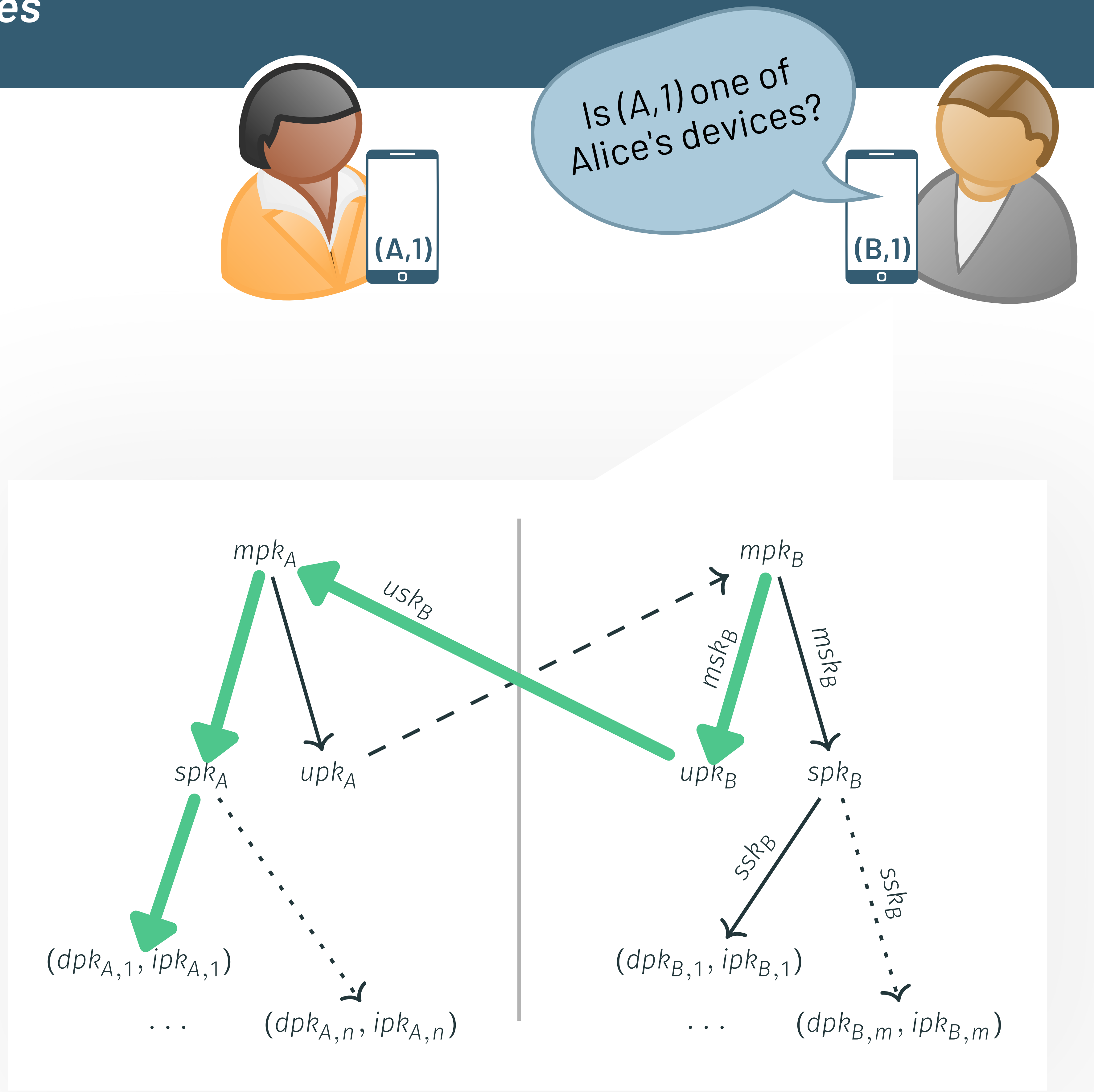
Verifying device identities





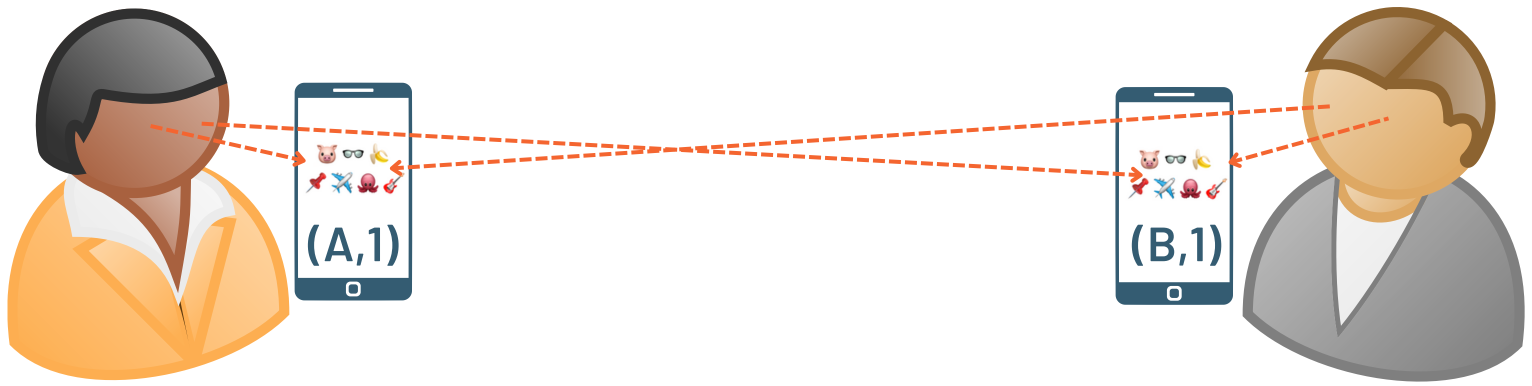
Cross-signing and Verification

Verifying device identities



Cross-signing and Verification

Short Authentication String Protocol



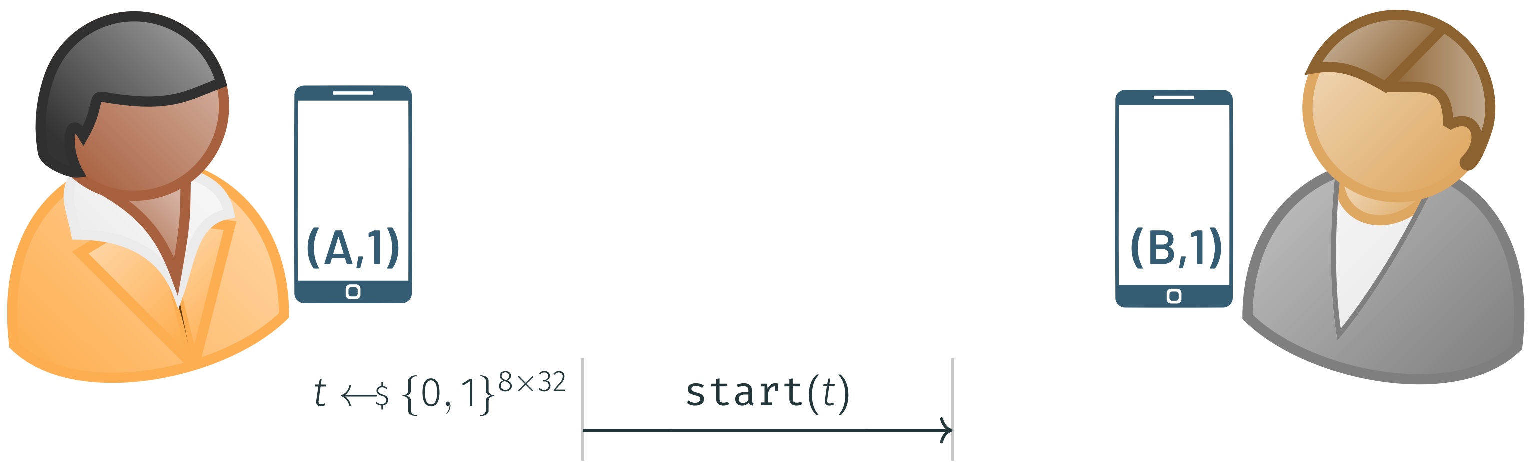
Cross-signing and Verification

Short Authentication String Protocol

1. Setup tamperproof channel with known identities

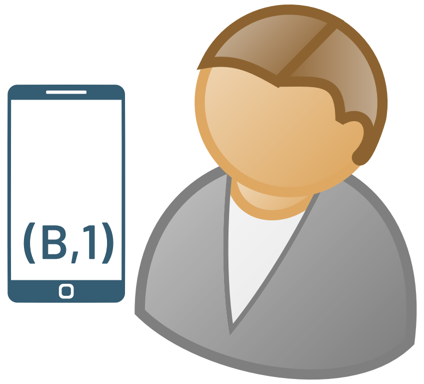
Cross-signing and Verification

Short Authentication String Protocol



Cross-signing and Verification

Short Authentication String Protocol



$t \leftarrow\!\!\$ \{0,1\}^{8\times 32}$ — $\mathtt{start}(t)$ →

$sk_B, pk_B \leftarrow \mathsf{X25519.KGen}(1^n)$

$com \leftarrow \mathsf{SHA\text{-}256}(pk_B)$

← $\mathtt{accept}(t, com)$

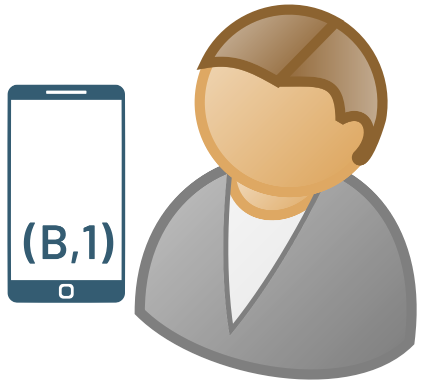
$sk_A, pk_A \leftarrow \mathsf{X25519.KGen}(1^n)$ — $\mathtt{key}(t, pk_A)$ →

← $\mathtt{key}(t, pk_B)$

(abort if $\mathsf{SHA\text{-}256}(pk_B) \neq com$)

Cross-signing and Verification

Short Authentication String Protocol



A	Messages	B
$t \leftarrow\!\!\$ \{0,1\}^{8\times 32}$	$\texttt{start}(t)$ →	
	← $\texttt{accept}(t, com)$	$sk_B, pk_B \leftarrow \text{X25519.KGen}(1^n)$ $com \leftarrow \text{SHA-256}(pk_B)$
$sk_A, pk_A \leftarrow \text{X25519.KGen}(1^n)$	$\texttt{key}(t, pk_A)$ →	
	← $\texttt{key}(t, pk_B)$	
(abort if $\text{SHA-256}(pk_B) \neq com$)		
$k \leftarrow sk_A \times pk_B$ $c \leftarrow$ "MATRIX_KEY_VERIFICATION_SAS" $\Vert A \Vert D_{A,1} \Vert pk_A \Vert B \Vert D_{B,1} \Vert pk_B \Vert t$ $sas_A \leftarrow \text{HKDF-SHA-256}(0, k, c, 32)$		$k \leftarrow sk_B \times pk_A$ $c \leftarrow$ "MATRIX_KEY_VERIFICATION_SAS" $\Vert A \Vert D_{A,1} \Vert pk_A \Vert B \Vert D_{B,1} \Vert pk_B \Vert t$ $sas_B \leftarrow \text{HKDF-SHA-256}(0, k, c, 32)$

Cross-signing and Verification

Short Authentication String Protocol

Alice: $t \leftarrow_{\$} \{0,1\}^{8\times 32}$

$\texttt{start}(t)$ →

Bob: $sk_B, pk_B \leftarrow \text{X25519.KGen}(1^n)$

Bob: $com \leftarrow \text{SHA-256}(pk_B)$

← $\texttt{accept}(t, com)$

Alice: $sk_A, pk_A \leftarrow \text{X25519.KGen}(1^n)$

$\texttt{key}(t, pk_A)$ →

← $\texttt{key}(t, pk_B)$

Alice: (abort if $\text{SHA-256}(pk_B) \neq com$)

Alice:

$k \leftarrow sk_A \times pk_B$

$c \leftarrow$ "MATRIX_KEY_VERIFICATION_SAS" $\| A \| D_{A,1} \| pk_A \| B \| D_{B,1} \| pk_B \| t$

$sas_A \leftarrow \text{HKDF-SHA-256}(0, k, c, 32)$

Bob:

$k \leftarrow sk_B \times pk_A$

$c \leftarrow$ "MATRIX_KEY_VERIFICATION_SAS" $\| A \| D_{A,1} \| pk_A \| B \| D_{B,1} \| pk_B \| t$

$sas_B \leftarrow \text{HKDF-SHA-256}(0, k, c, 32)$

Alice and Bob compare sas_A and sas_B out-of-band.

(abort if $sas_A \neq sas_B$)

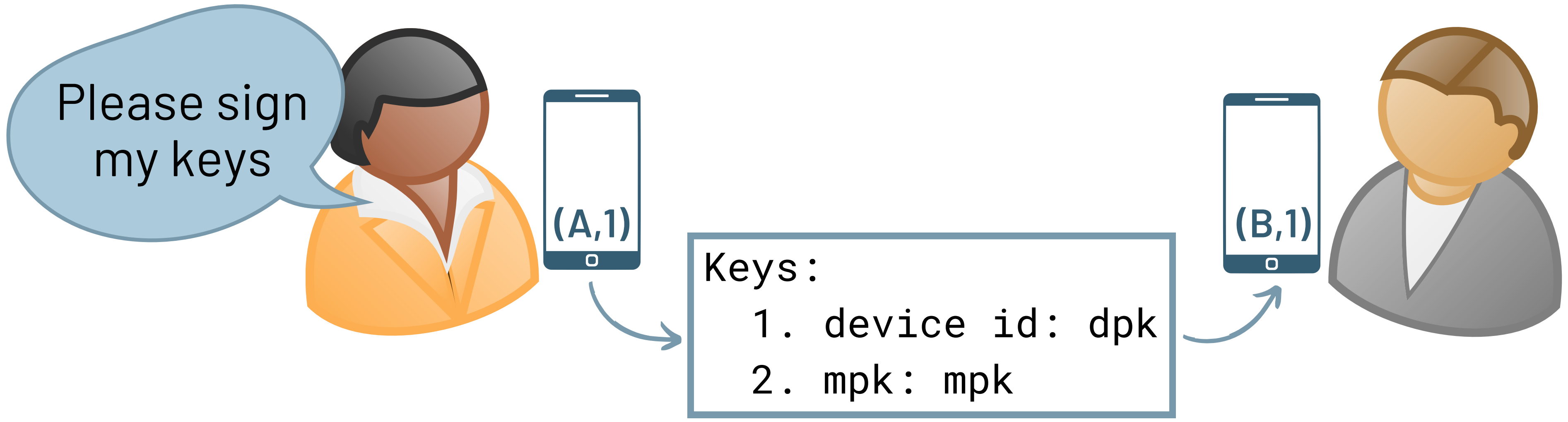
Cross-signing and Verification

Short Authentication String Protocol

Alice and Bob may use k to communicate securely through the homeserver

Cross-signing and Verification

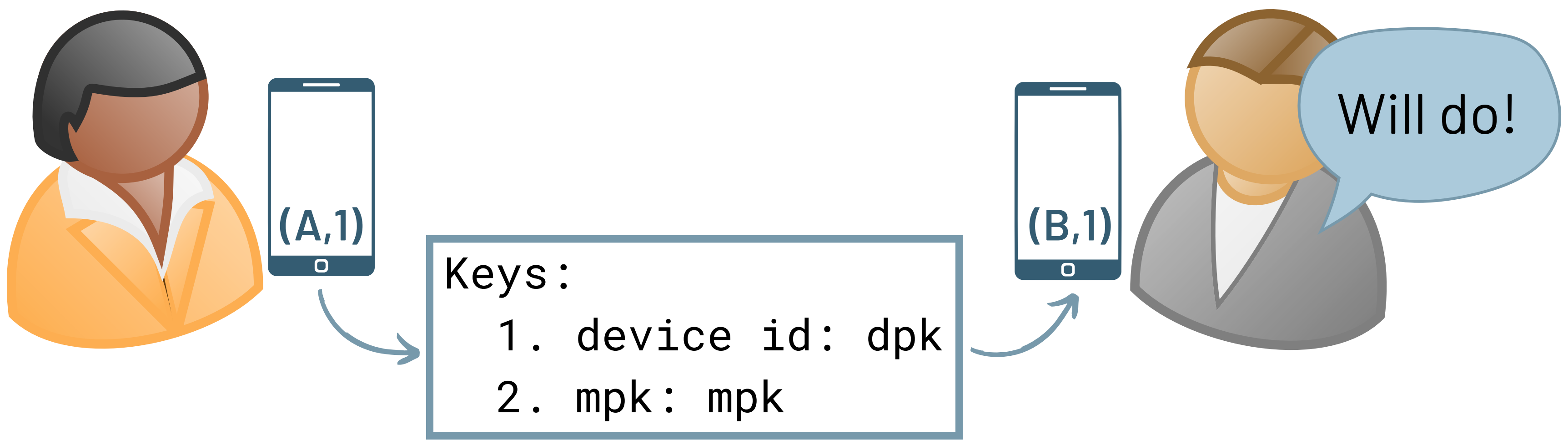
Short Authentication String Protocol



2. Send keys for signing

Cross-signing and Verification

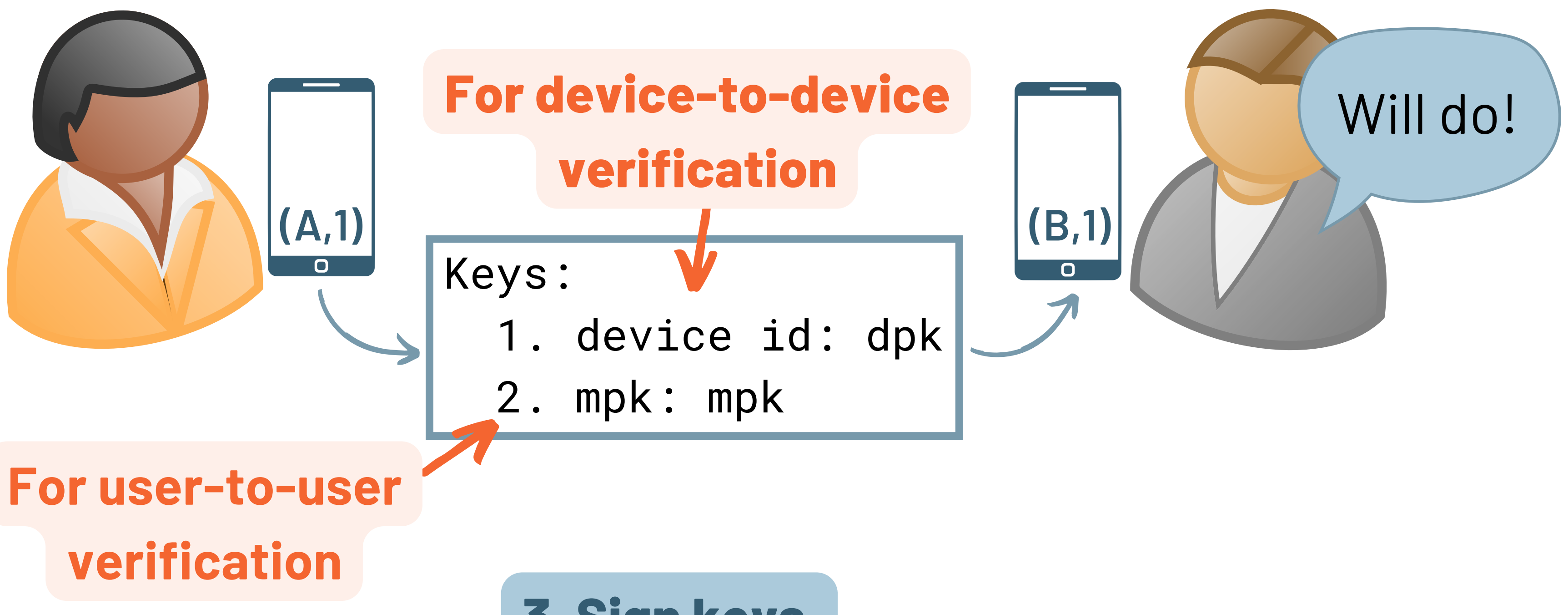
Short Authentication String Protocol



3. Sign keys

Cross-signing and Verification

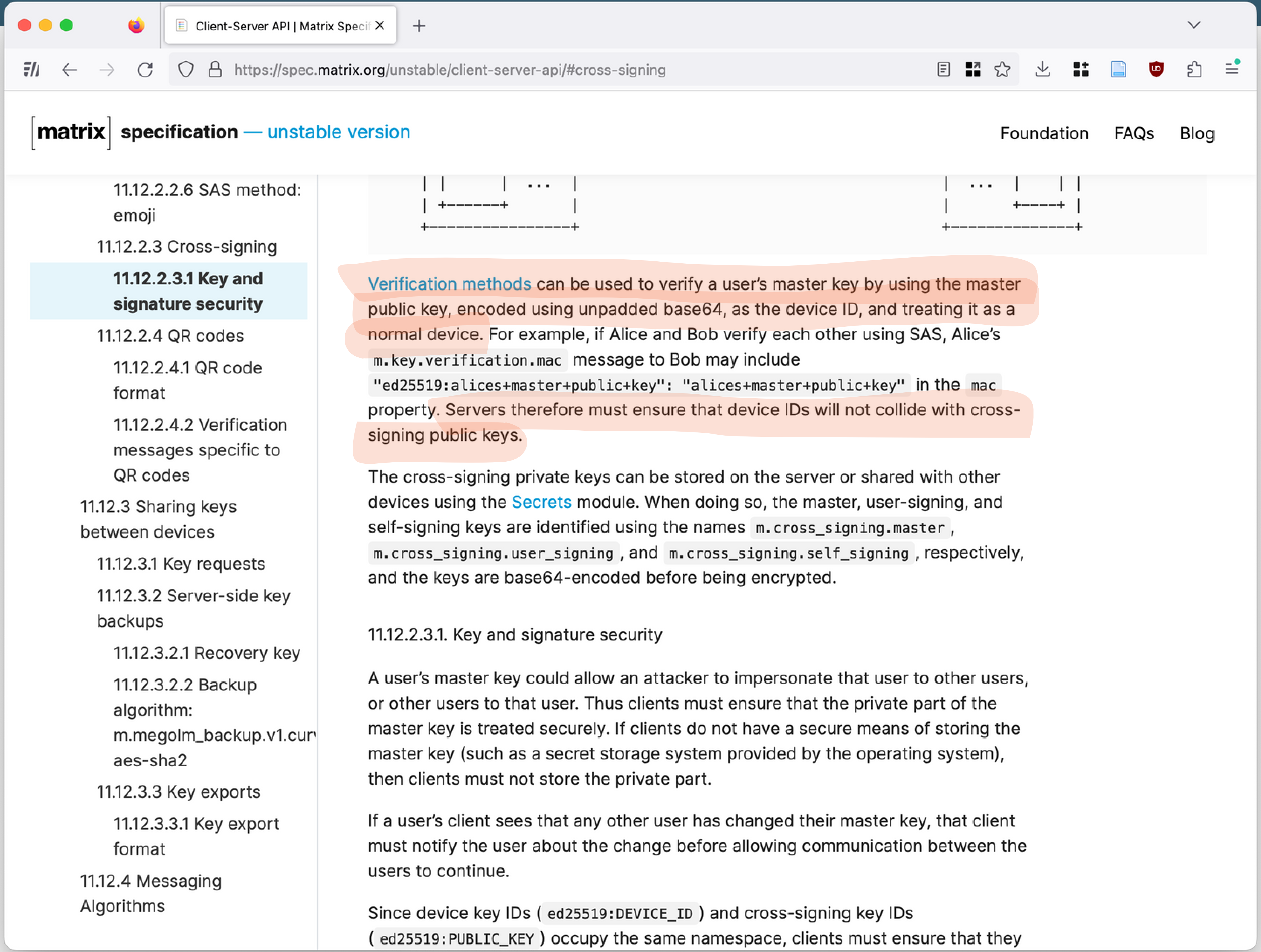
Short Authentication String Protocol



3. Sign keys

Cross-signing and Verification

Short Authentication String Protocol



Client-Server API | Matrix Speci

https://spec.matrix.org/unstable/client-server-api/#cross-signing

[matrix] specification — unstable version

Foundation FAQs Blog

Verification methods can be used to verify a user's master key by using the master public key, encoded using unpadded base64, as the device ID, and treating it as a normal device. For example, if Alice and Bob verify each other using SAS, Alice's `m.key.verification.mac` message to Bob may include `"ed25519:alices+master+public+key": "alices+master+public+key"` in the `mac` property. Servers therefore must ensure that device IDs will not collide with cross-signing public keys.

The cross-signing private keys can be stored on the server or shared with other devices using the Secrets module. When doing so, the master, user-signing, and self-signing keys are identified using the names `m.cross_signing.master`, `m.cross_signing.user_signing`, and `m.cross_signing.self_signing`, respectively, and the keys are base64-encoded before being encrypted.

11.12.2.3.1. Key and signature security

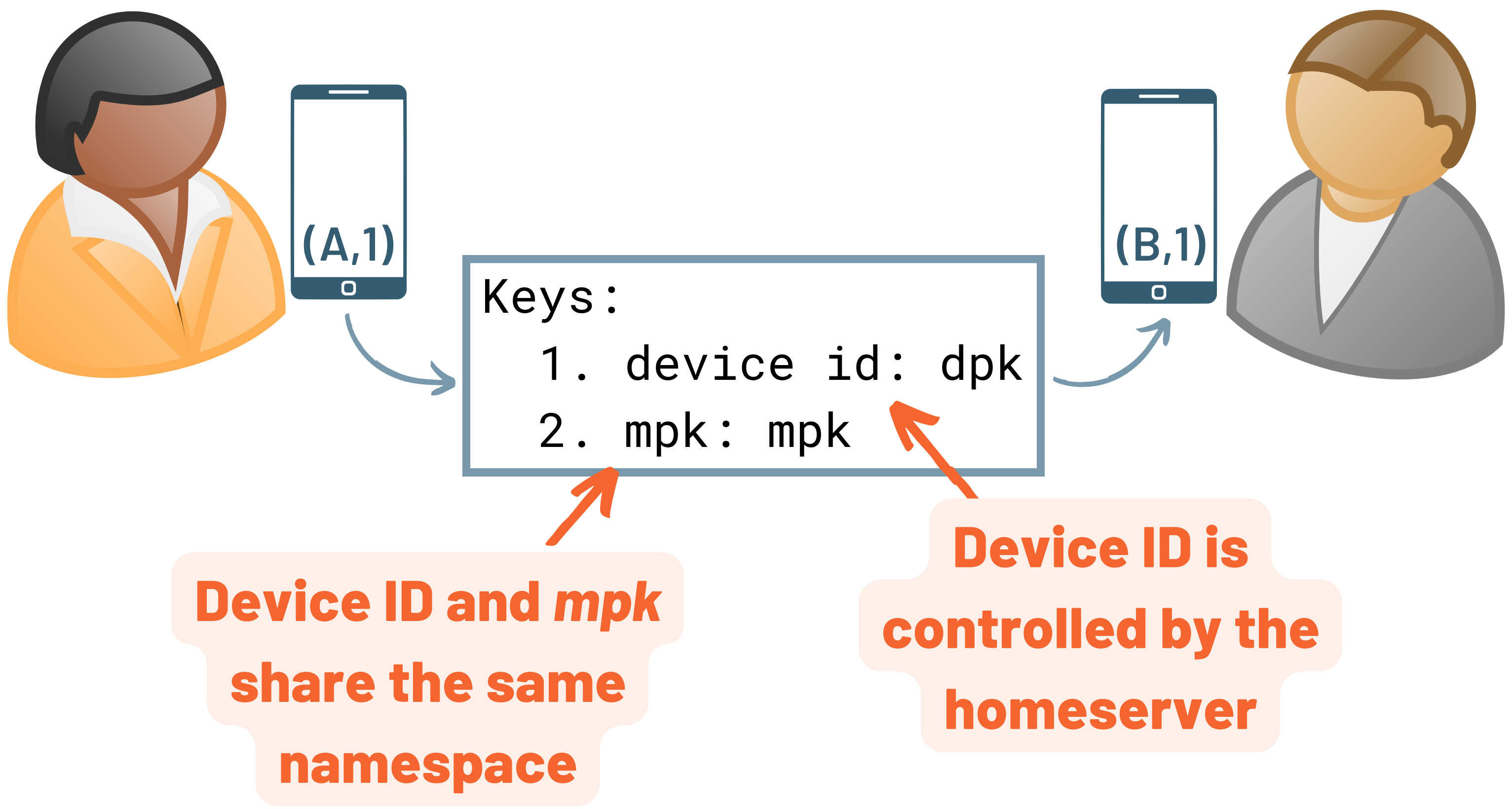
A user's master key could allow an attacker to impersonate that user to other users, or other users to that user. Thus clients must ensure that the private part of the master key is treated securely. If clients do not have a secure means of storing the master key (such as a secret storage system provided by the operating system), then clients must not store the private part.

If a user's client sees that any other user has changed their master key, that client must notify the user about the change before allowing communication between the users to continue.

Since device key IDs (`ed25519:DEVICE_ID`) and cross-signing key IDs (`ed25519:PUBLIC_KEY`) occupy the same namespace, clients must ensure that they

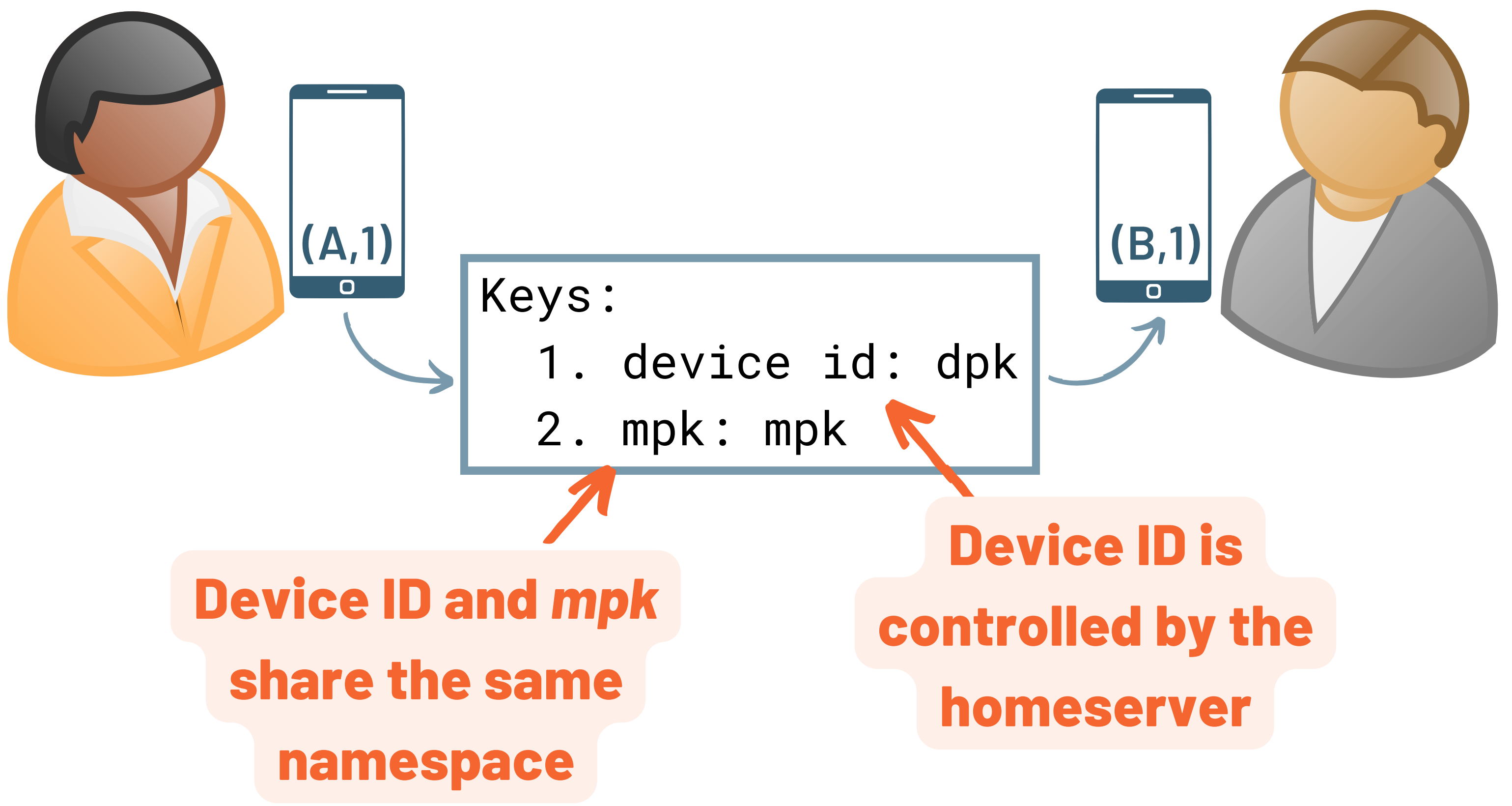
Cross-signing and Verification

Short Authentication String Protocol



User/Device Confusion in Out-of-Band Verification

Aim: Trick clients into signing an attacker controlled identity.

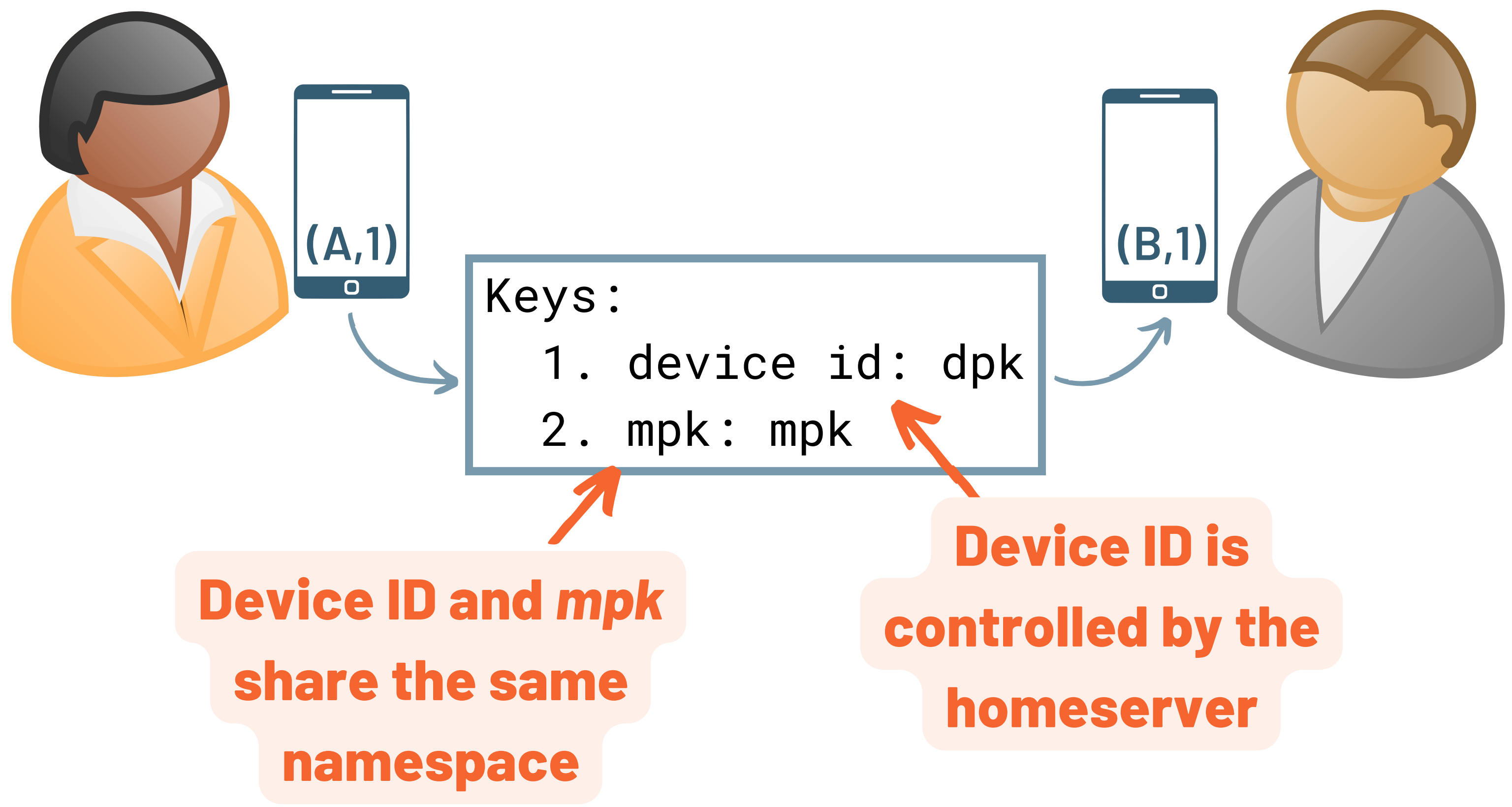


User/Device Confusion in Out-of-Band Verification

Aim: Trick clients into signing an attacker controlled identity.

If we set Alice's device identifier to a valid cross-signing key, ***can we trick Bob into signing it as if it were Alice's?***

ATTACK

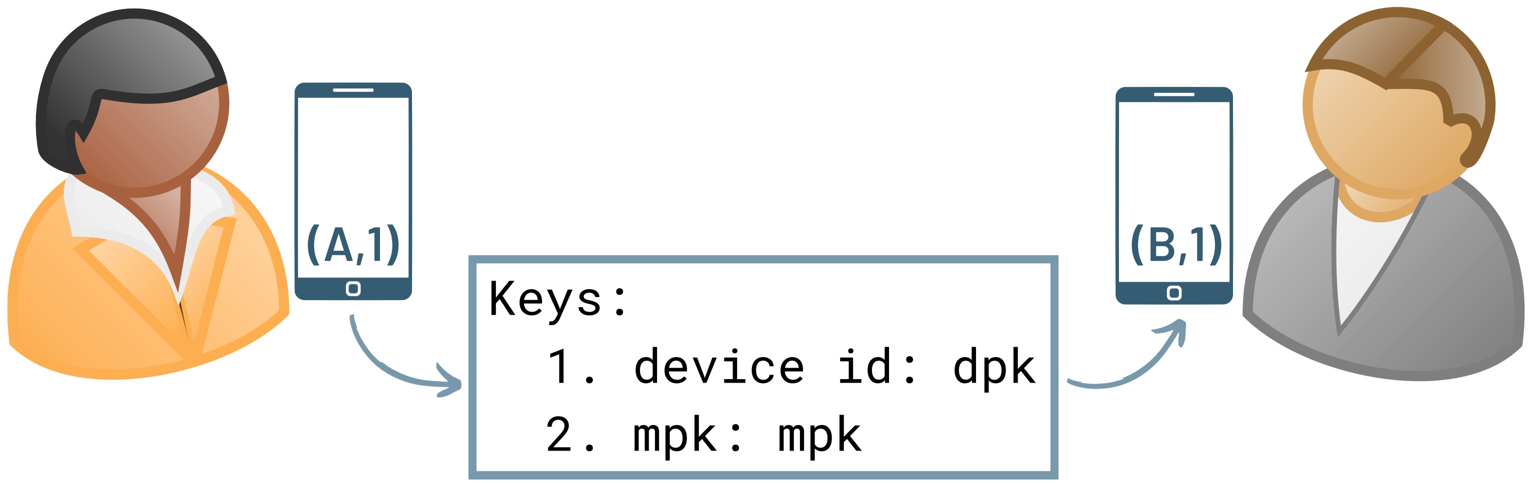


User/Device Confusion in Out-of-Band Verification

ATTACK

Aim: Trick clients into signing an attacker controlled identity.

If we set Alice's device identifier to a valid cross-signing key, can we trick Bob into signing it as if it were Alice's? Yes



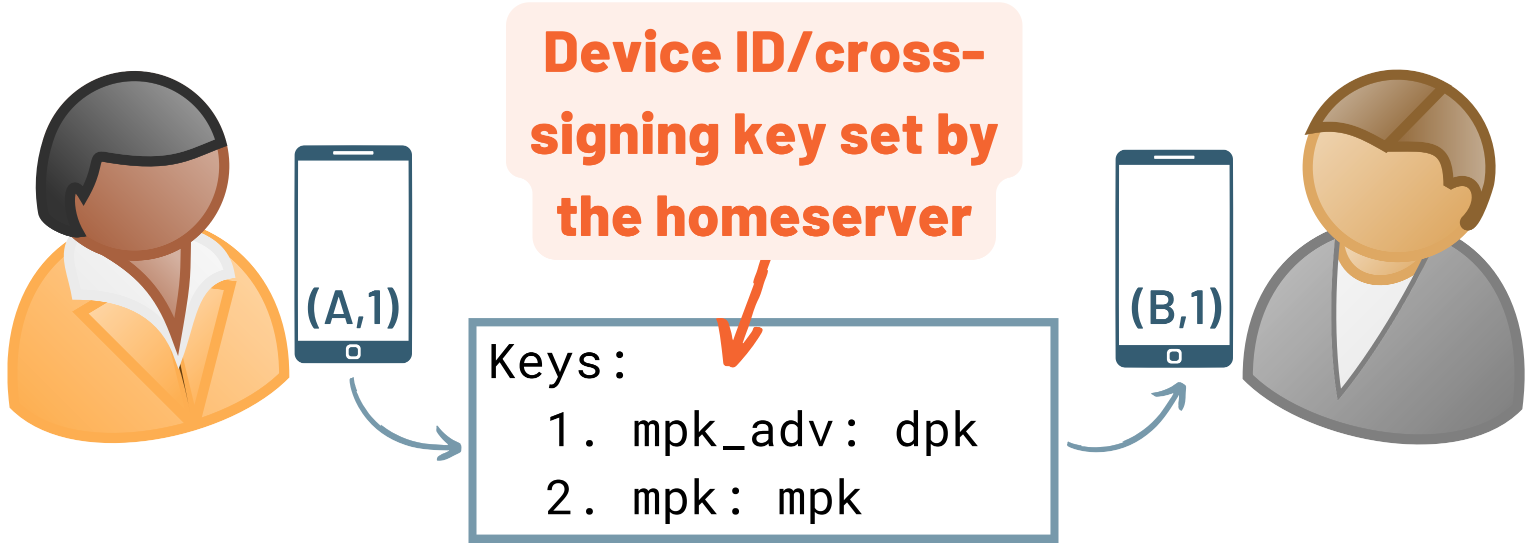
1. Construct a string mpk_adv as a valid Device ID and *user cross-signing key (mpk)*

User/Device Confusion in Out-of-Band Verification

Aim: Trick clients into signing an attacker controlled identity.

If we set Alice's device identifier to a valid cross-signing key, can we trick Bob into signing it as if it were Alice's? Yes

ATTACK



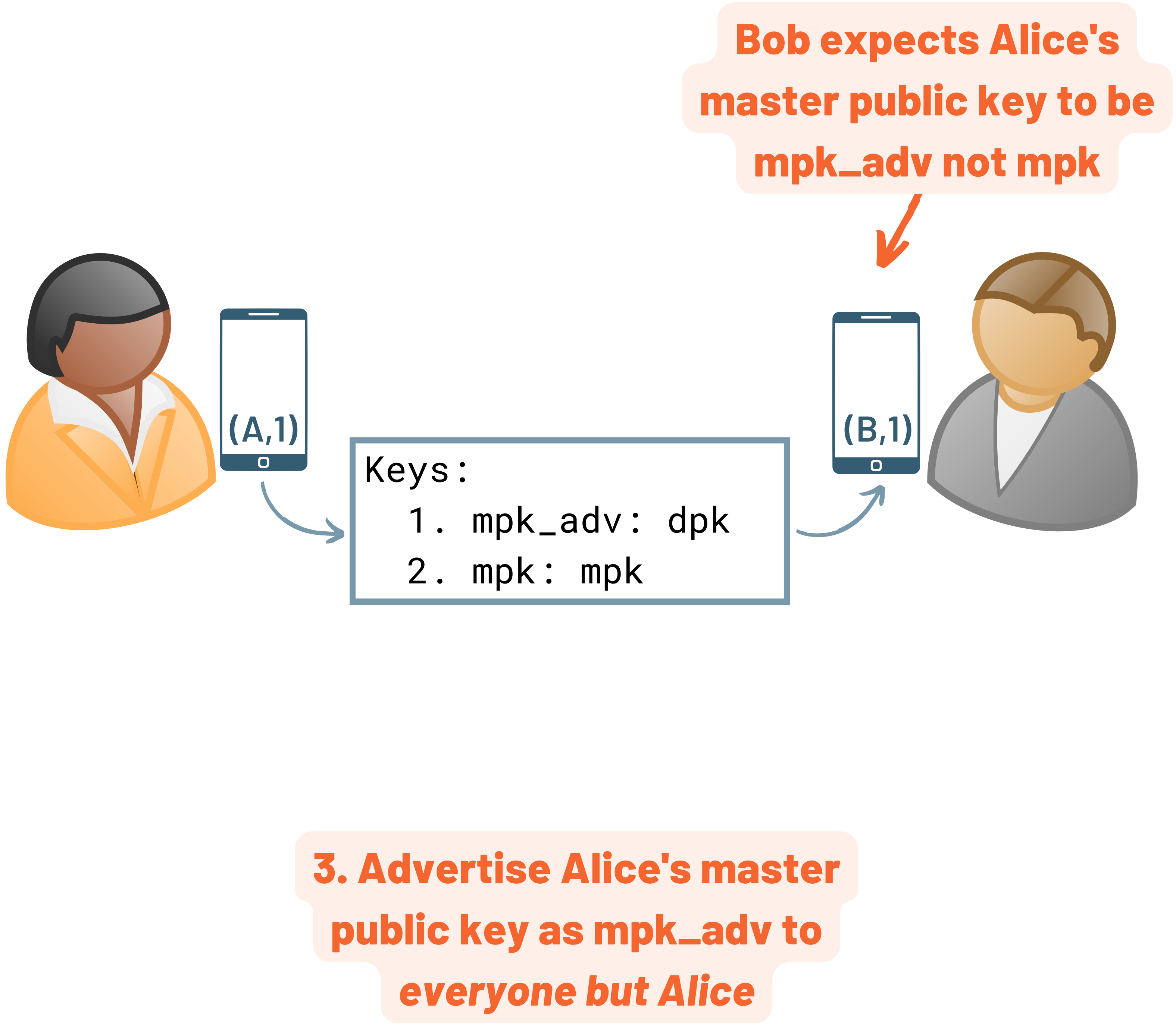
2. Assign mpk_adv to (A,1) as its device identifier

ATTACK

User/Device Confusion in Out-of-Band Verification

Aim: Trick clients into signing an attacker controlled identity.

If we set Alice's device identifier to a valid cross-signing key, can we trick Bob into signing it as if it were Alice's? Yes

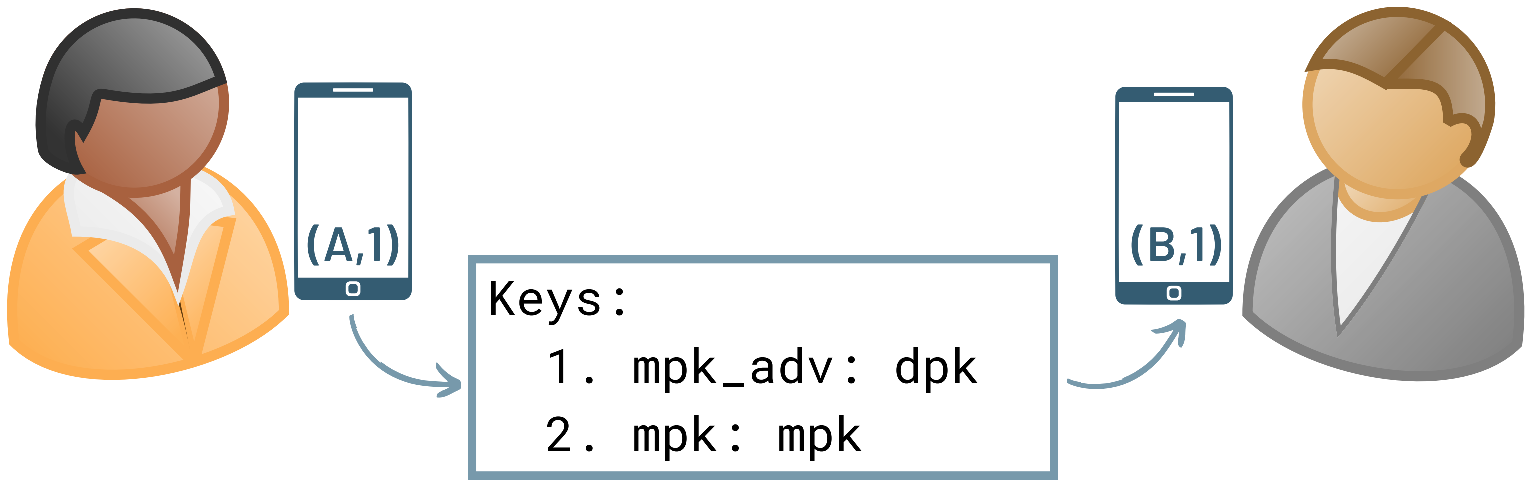


User/Device Confusion in Out-of-Band Verification

Aim: Trick clients into signing an attacker controlled identity.

If we set Alice's device identifier to a valid cross-signing key, can we trick Bob into signing it as if it were Alice's? Yes

ATTACK



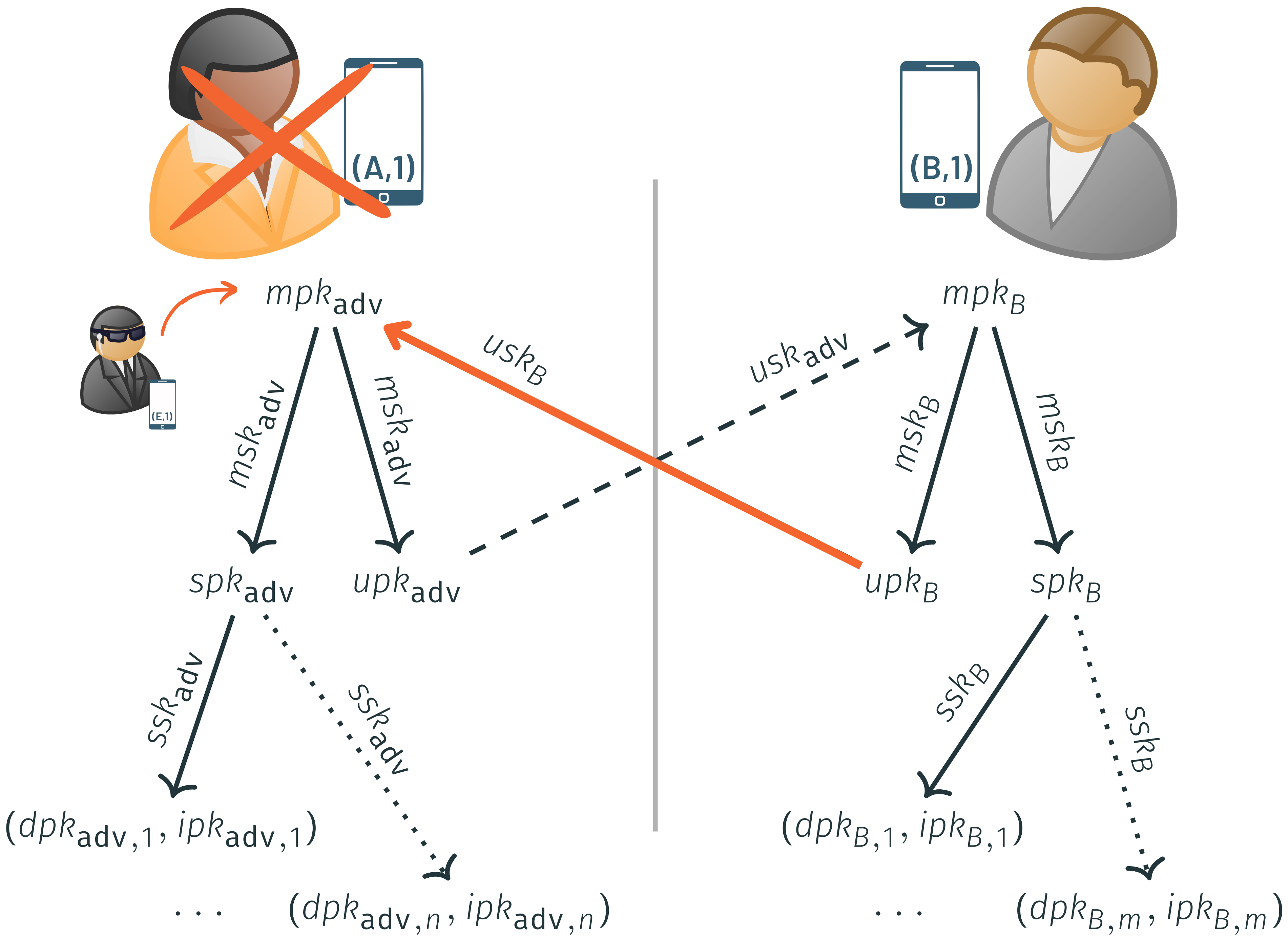
4. Alice's client asks Bob to sign mpk_adv!

User/Device Confusion in Out-of-Band Verification

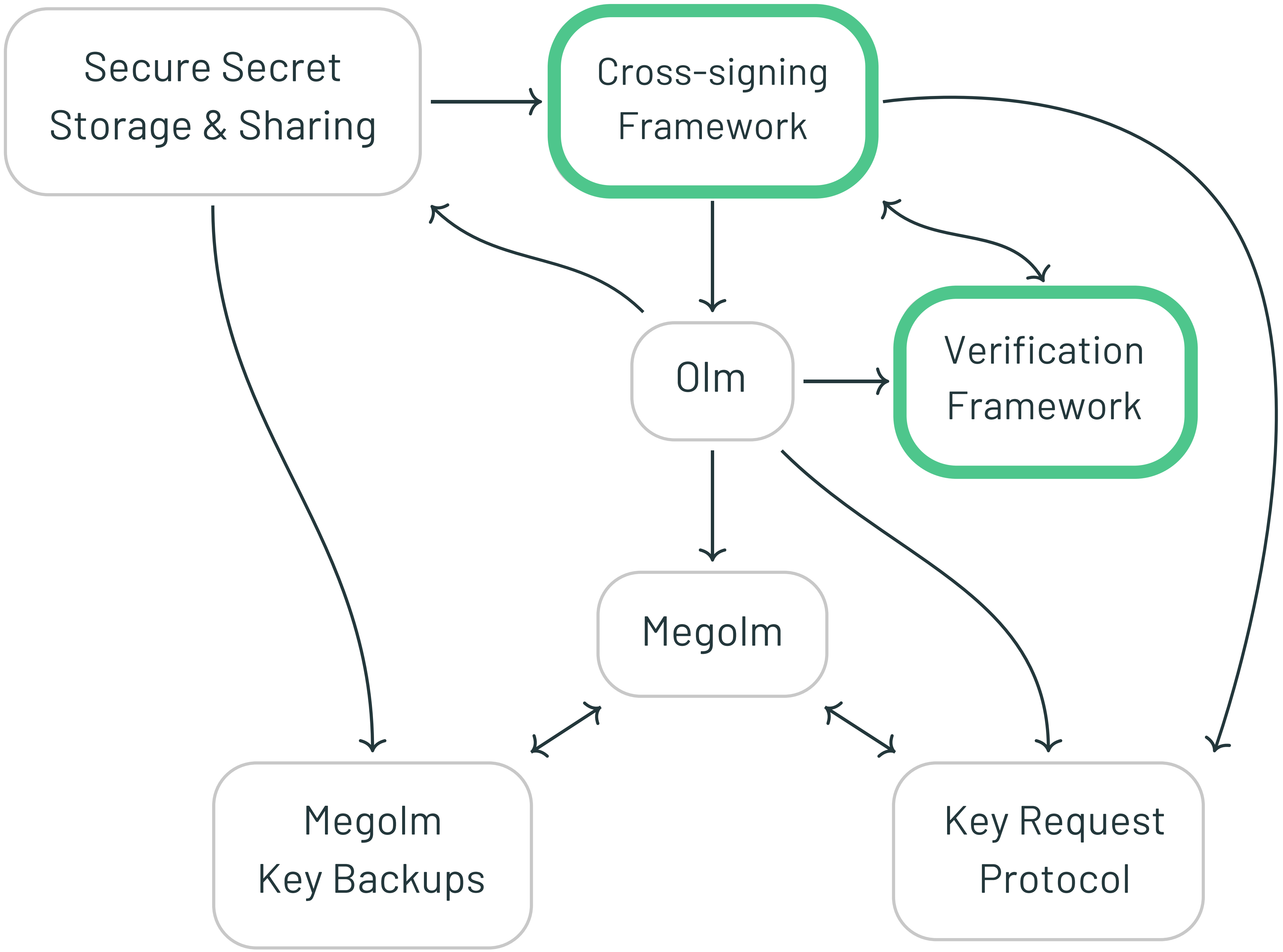
Aim: Trick clients into signing an attacker controlled identity.

If we set Alice's device identifier to a valid cross-signing key, can we trick Bob into signing it as if it were Alice's? Yes

↪ Active MITM



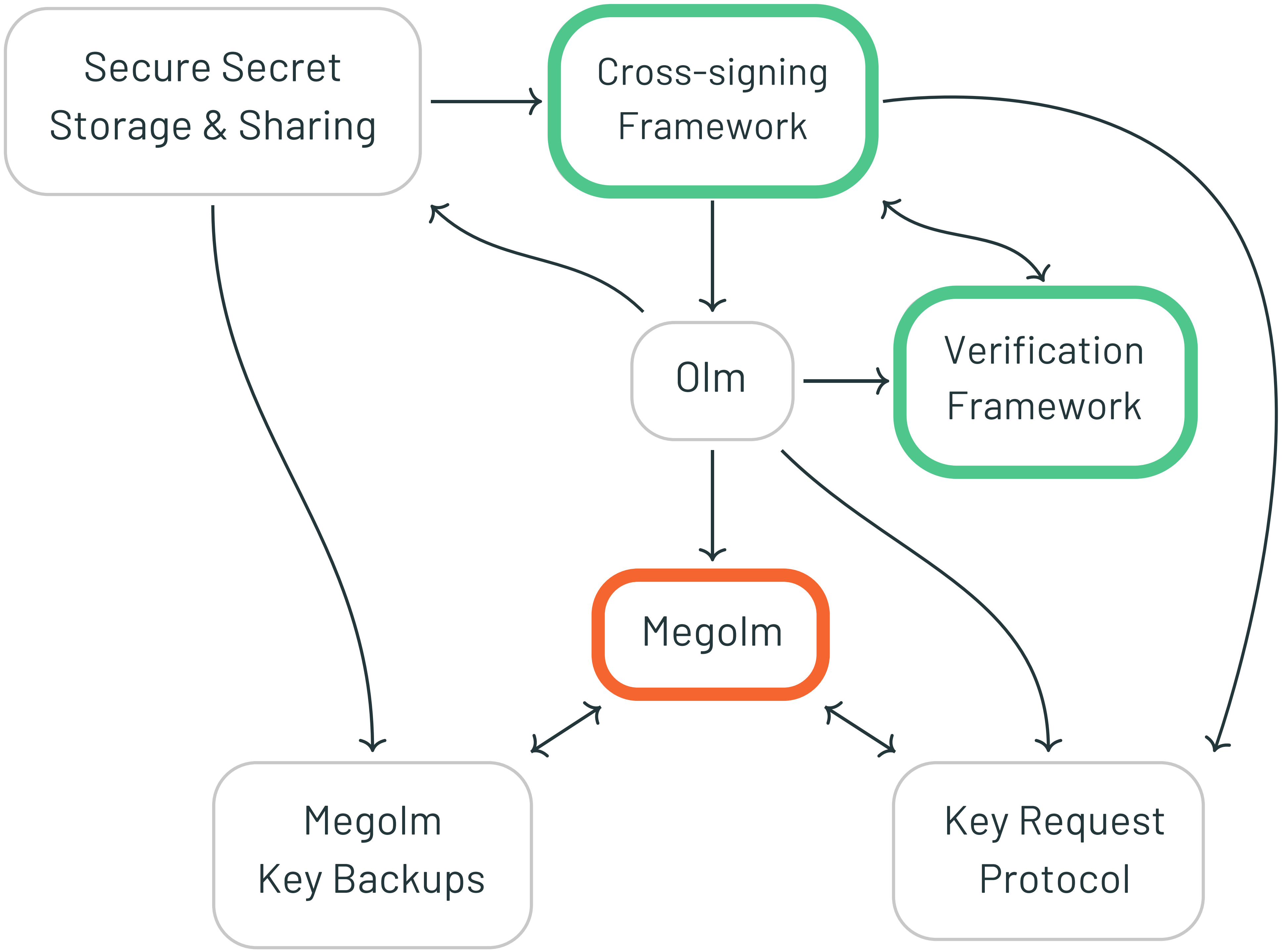
Modelling Matrix & Finding Attacks



Megolm

- Secure one-to-many channel.
- Symmetric ratchet for forward security.
- Session keys are distributed over Olm.
- Each sender maintains their own Megolm session.

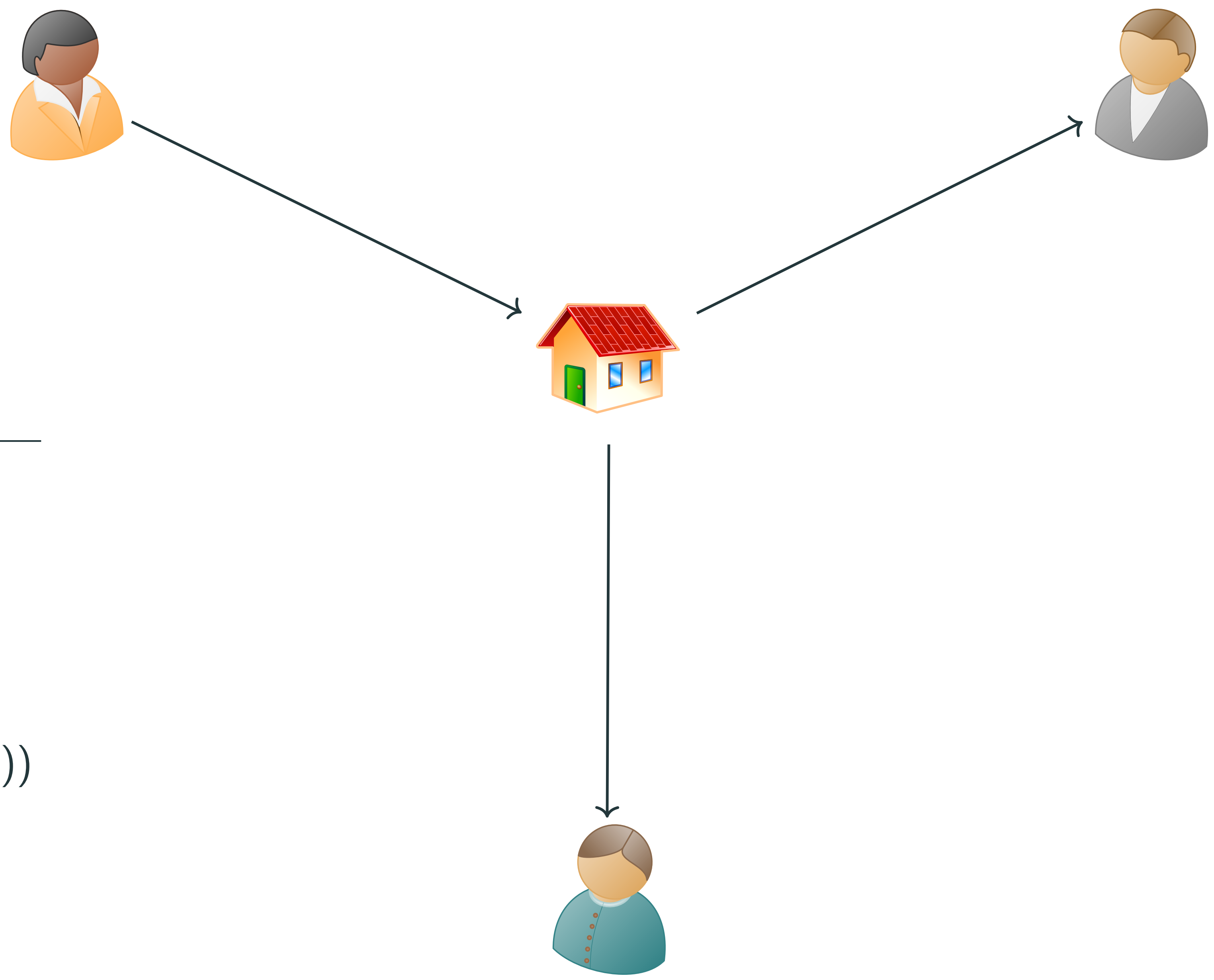
↪ Compose together to form group chat.



Megolm

1. Alice initialises and distributes session.

$(\mathfrak{S}^A_{gsk}, \mathfrak{S}^A_{gpk}, \sigma^A_{\mathrm{mg}}) \leftarrow \mathrm{Megolm.Init}()$



$\underline{\mathrm{Megolm.Init}()}$

$i \leftarrow 0$

$R \leftarrow \{0,1\}^{1024}$

$(gsk, gpk) \leftarrow \mathrm{Ed25519.KGen}(1^n)$

$\mathrm{ver} \leftarrow \mathtt{0x03}$

$\sigma_{\mathrm{mg}} \leftarrow \mathrm{Ed25519.Sign}(gsk, (\mathrm{ver}, i, R, gpk))$

$\mathfrak{S}_{gsk} \leftarrow (\mathrm{ver}, i, R, gsk, gpk)$

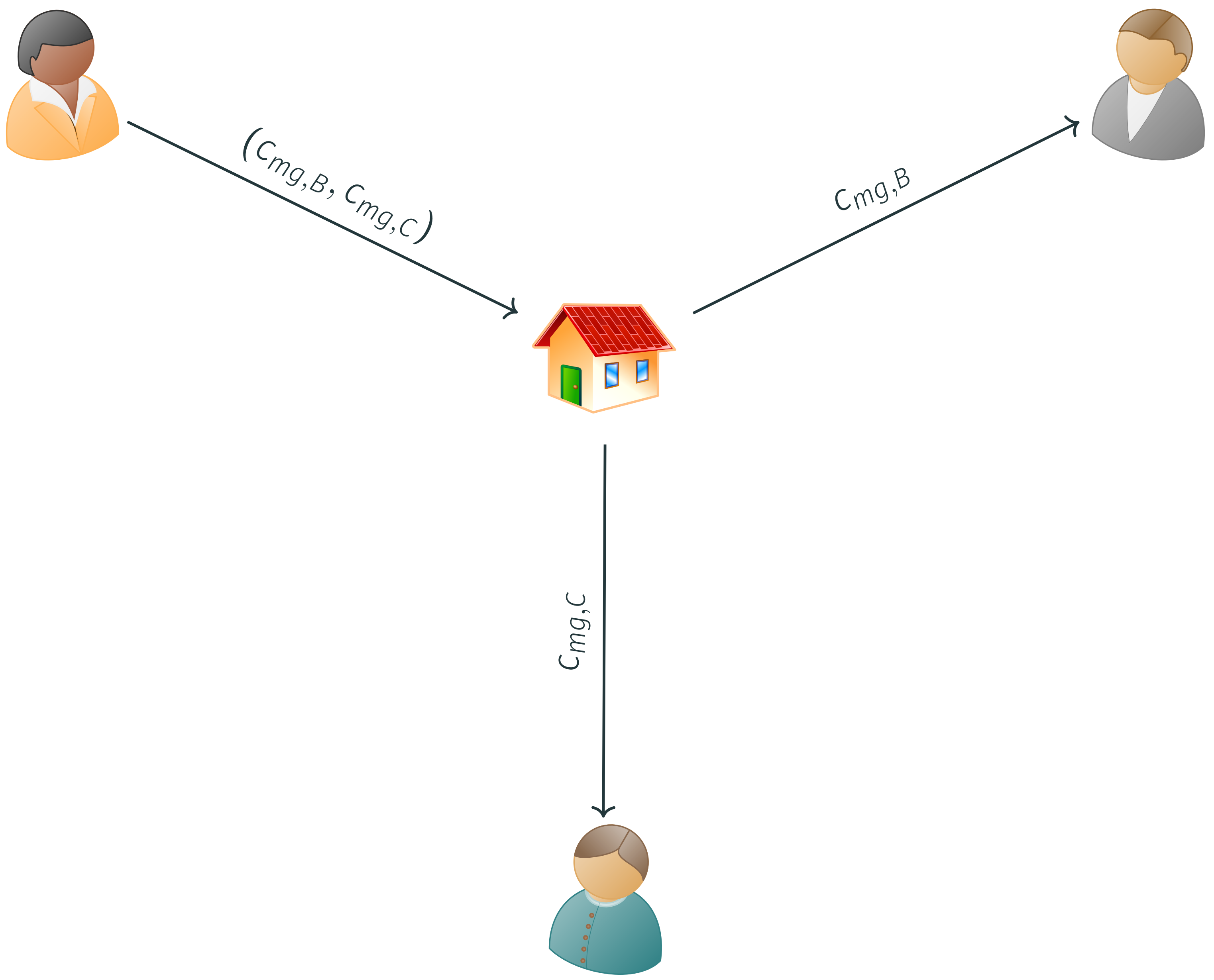
$\mathfrak{S}_{gpk} \leftarrow (\mathrm{ver}, i, R, gpk)$

$\textbf{return } \mathfrak{S}_{gsk}, \mathfrak{S}_{gpk}, \sigma_{\mathrm{mg}}$

Megolm

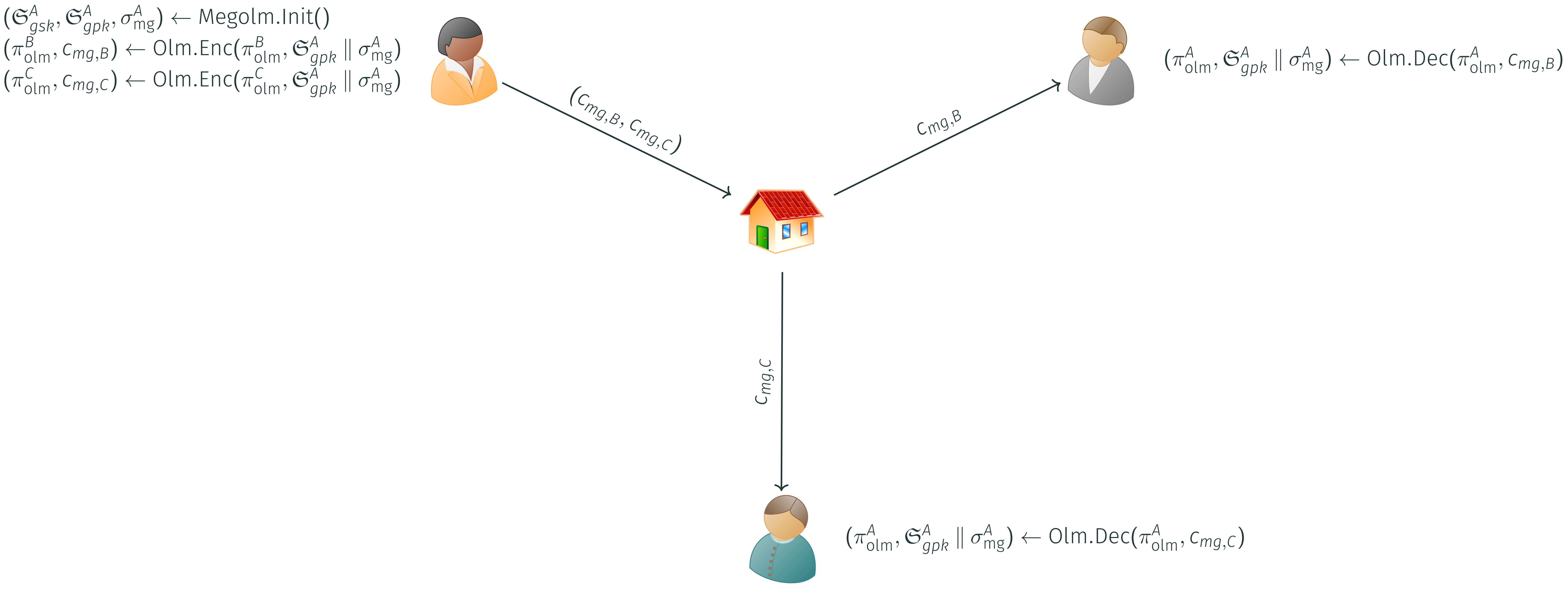
1. Alice initialises and distributes session.

$$(\mathfrak{S}^A_{gsk}, \mathfrak{S}^A_{gpk}, \sigma^A_{\text{mg}}) \leftarrow \text{Megolm.Init}()$$
$$(\pi^B_{\text{olm}}, c_{mg,B}) \leftarrow \text{Olm.Enc}(\pi^B_{\text{olm}}, \mathfrak{S}^A_{gpk} \parallel \sigma^A_{\text{mg}})$$
$$(\pi^C_{\text{olm}}, c_{mg,C}) \leftarrow \text{Olm.Enc}(\pi^C_{\text{olm}}, \mathfrak{S}^A_{gpk} \parallel \sigma^A_{\text{mg}})$$



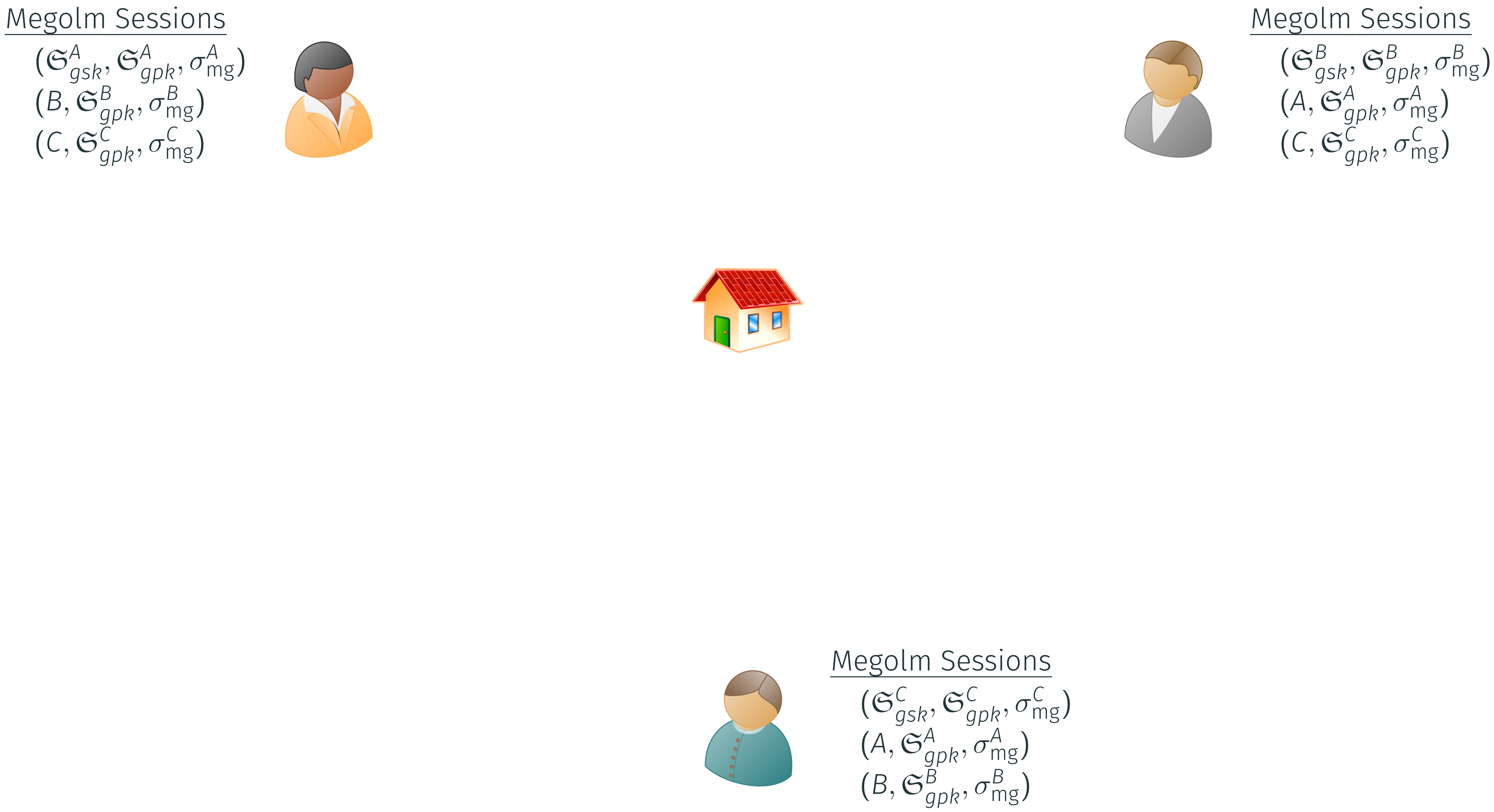
Megolm

2. Bob and Claire recieve and verify Alice's session.



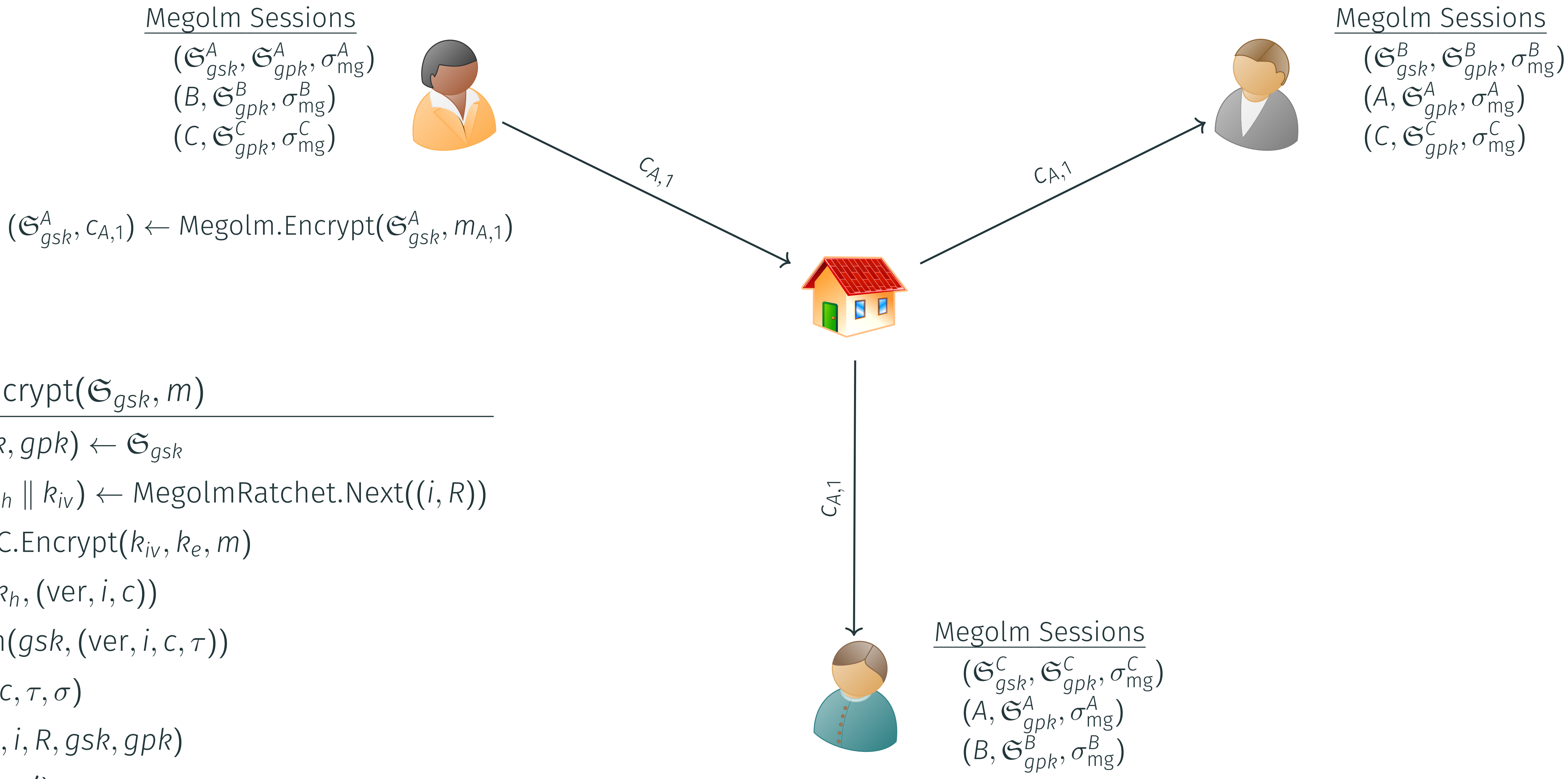
Megolm

3. Bob and Claire generate and distribute their own sessions.



Megolm

4. Alice sends a Megolm message to Bob and Claire.



$\mathrm{Megolm.Encrypt}(\mathfrak{S}_{gsk}, m)$

$(\mathrm{ver}, i, R, gsk, gpk) \leftarrow \mathfrak{S}_{gsk}$

$(i, R), (k_e \parallel k_h \parallel k_{iv}) \leftarrow \mathrm{MegolmRatchet.Next}((i, R))$

$c \leftarrow \mathrm{AES\text{-}CBC.Encrypt}(k_{iv}, k_e, m)$

$\tau \leftarrow \mathrm{HMAC}(k_h, (\mathrm{ver}, i, c))$

$\sigma \leftarrow \mathrm{DS.Sign}(gsk, (\mathrm{ver}, i, c, \tau))$

$c' \leftarrow (\mathrm{ver}, i, c, \tau, \sigma)$

$\mathfrak{S}_{gsk} \leftarrow (\mathrm{ver}, i, R, gsk, gpk)$

return $(\mathfrak{S}_{gsk}, c')$

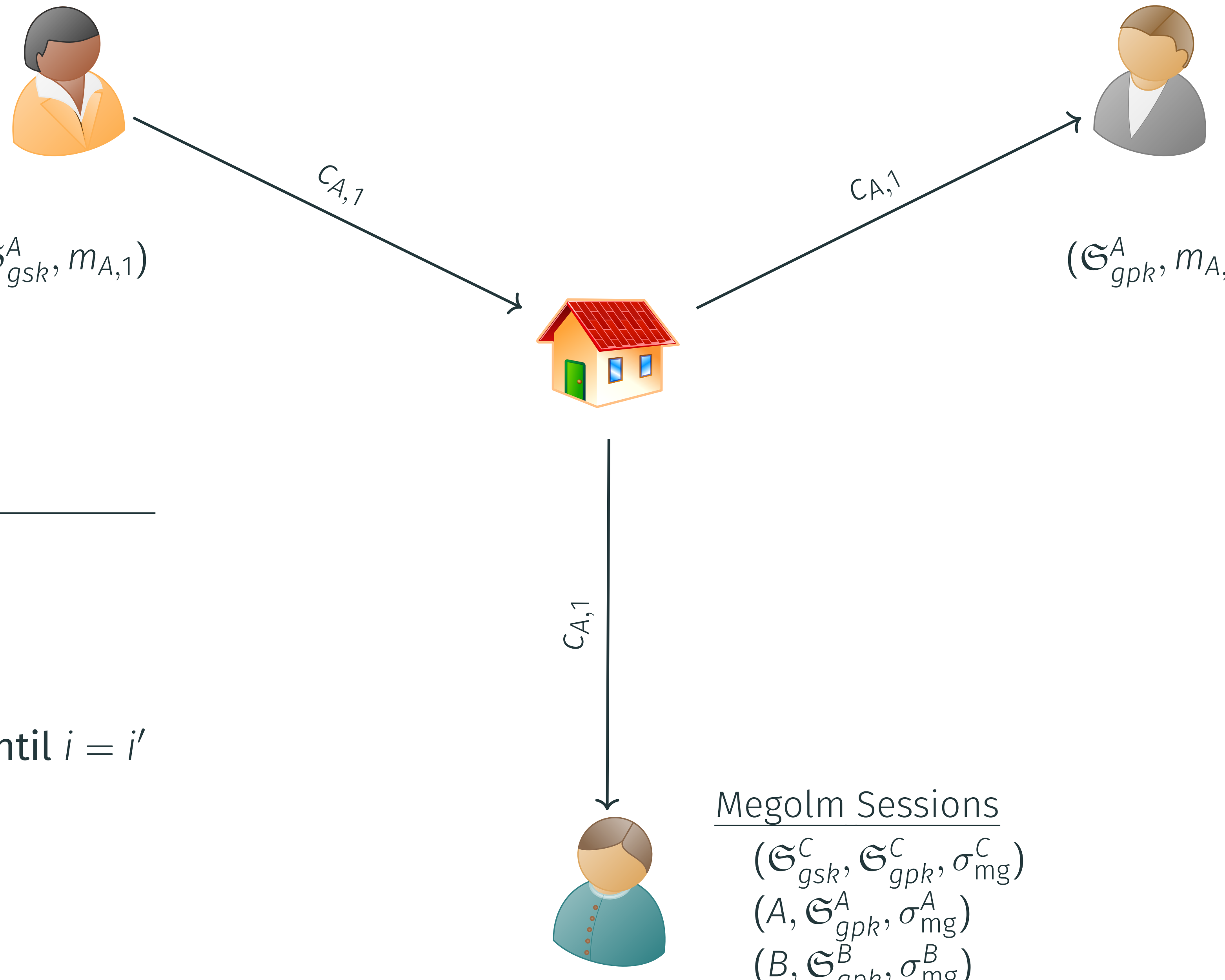
Megolm

5. *Bob and Claire decrypt and verify Alice's message.*

Megolm Sessions

$(\mathfrak{S}^A_{gsk}, \mathfrak{S}^A_{gpk}, \sigma^A_{\mathrm{mg}})$
$(B, \mathfrak{S}^B_{gpk}, \sigma^B_{\mathrm{mg}})$
$(C, \mathfrak{S}^C_{gpk}, \sigma^C_{\mathrm{mg}})$

$(\mathfrak{S}^A_{gsk}, c_{A,1}) \leftarrow \text{Megolm.Encrypt}(\mathfrak{S}^A_{gsk}, m_{A,1})$



Megolm Sessions

$(\mathfrak{S}^B_{gsk}, \mathfrak{S}^B_{gpk}, \sigma^B_{\mathrm{mg}})$
$(A, \mathfrak{S}^A_{gpk}, \sigma^A_{\mathrm{mg}})$
$(C, \mathfrak{S}^C_{gpk}, \sigma^C_{\mathrm{mg}})$

$(\mathfrak{S}^A_{gpk}, m_{A,1}) \leftarrow \text{Megolm.Decrypt}(\mathfrak{S}^B_{gpk}, c_{A,1})$

Megolm Sessions

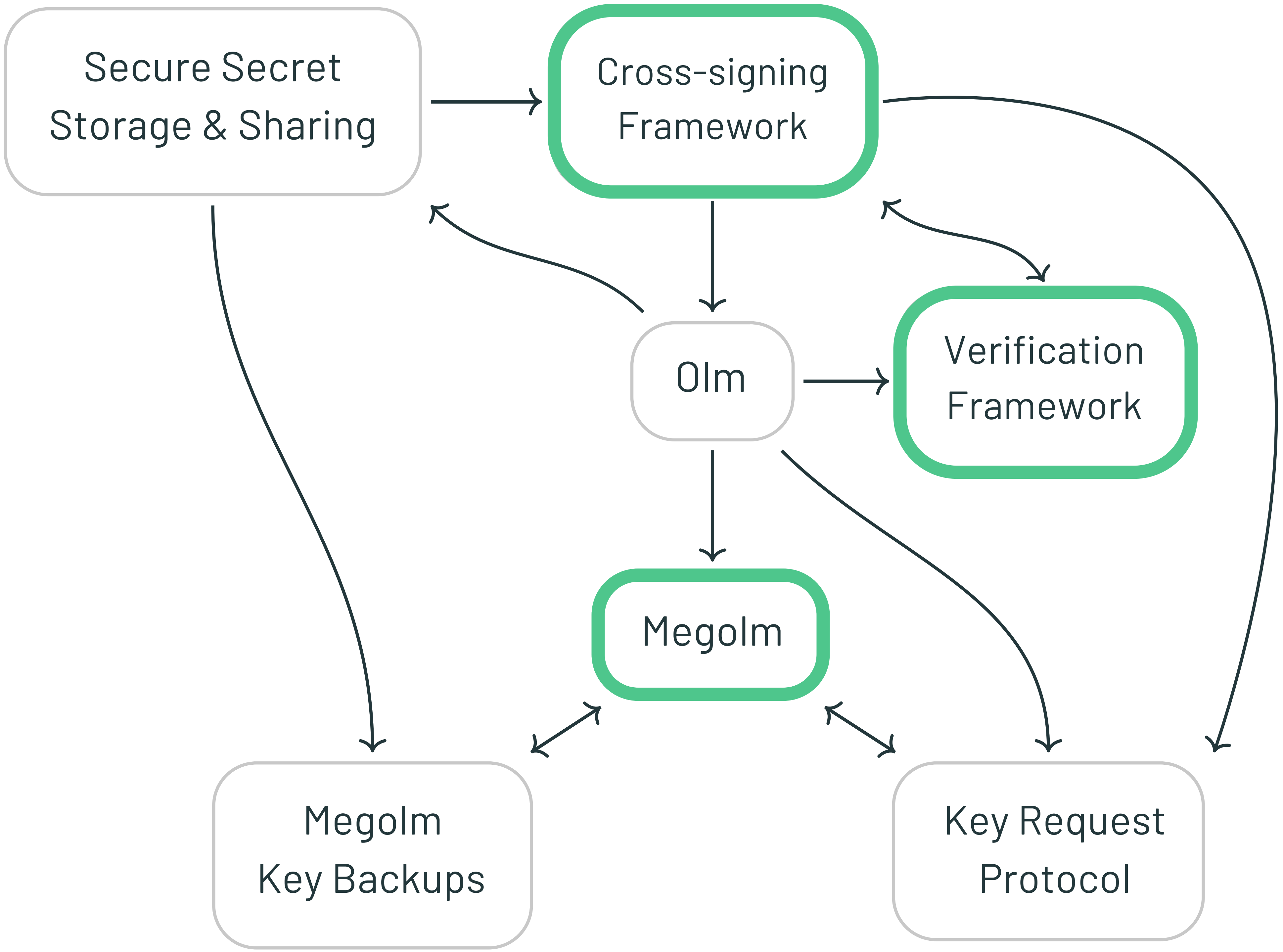
$(\mathfrak{S}^C_{gsk}, \mathfrak{S}^C_{gpk}, \sigma^C_{\mathrm{mg}})$
$(A, \mathfrak{S}^A_{gpk}, \sigma^A_{\mathrm{mg}})$
$(B, \mathfrak{S}^B_{gpk}, \sigma^B_{\mathrm{mg}})$

$(\mathfrak{S}^A_{gpk}, m_{A,1}) \leftarrow \text{Megolm.Decrypt}(\mathfrak{S}^B_{gpk}, c_{A,1})$

$\text{Megolm.Decrypt}(\mathfrak{S}_{gpk}, c)$

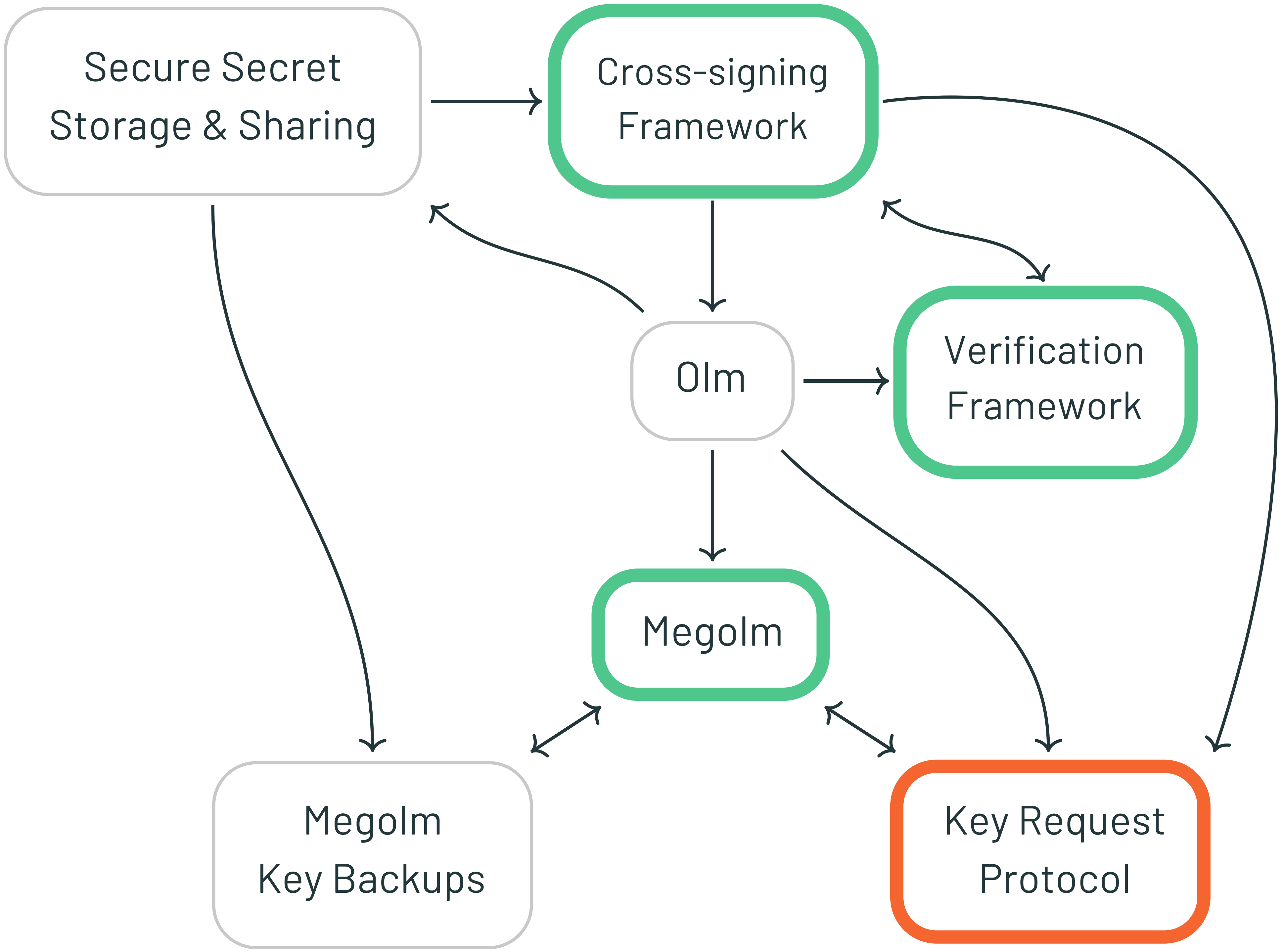
$(\mathrm{ver}, i, R, gpk) \leftarrow \mathfrak{S}_{gpk}$
$(\mathrm{ver}', i', c', \tau, \sigma) \leftarrow c$
assert $\text{DS.Verify}(gpk, \sigma, (\mathrm{ver}, i', c', \tau))$
do $(i, R), k \leftarrow \text{MegolmRatchet.Next}((i, R))$ **until** $i = i'$
$(k_e \parallel k_h \parallel k_{iv}) \leftarrow k$
assert $\tau = \text{HMAC}(k_h, (\mathrm{ver}, i, c'))$
$m \leftarrow \text{AES-CBC.Decrypt}(k_{iv}, k_e, c')$
$\mathfrak{S}_{gpk} \leftarrow (\mathrm{ver}, i, R, gpk)$
return $(\mathfrak{S}_{gpk}, m)$

Modelling Matrix & Finding Attacks



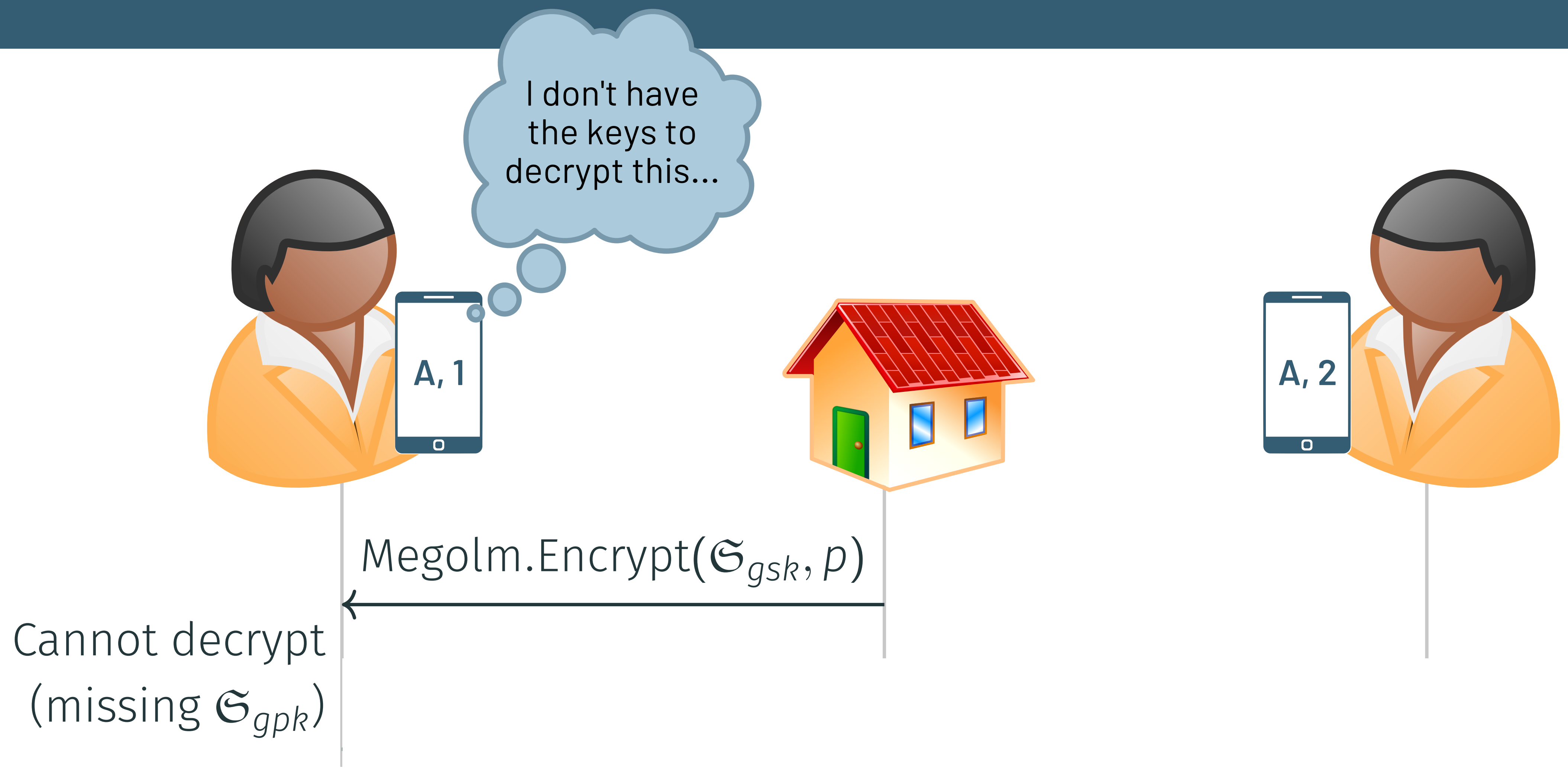
Key Request Protocol

- Request-response protocol.
- Allows devices to request and share keys between each other.
- Responding device must ensure requesting device is entitled to the keys.
- Keys are shared over Olm.



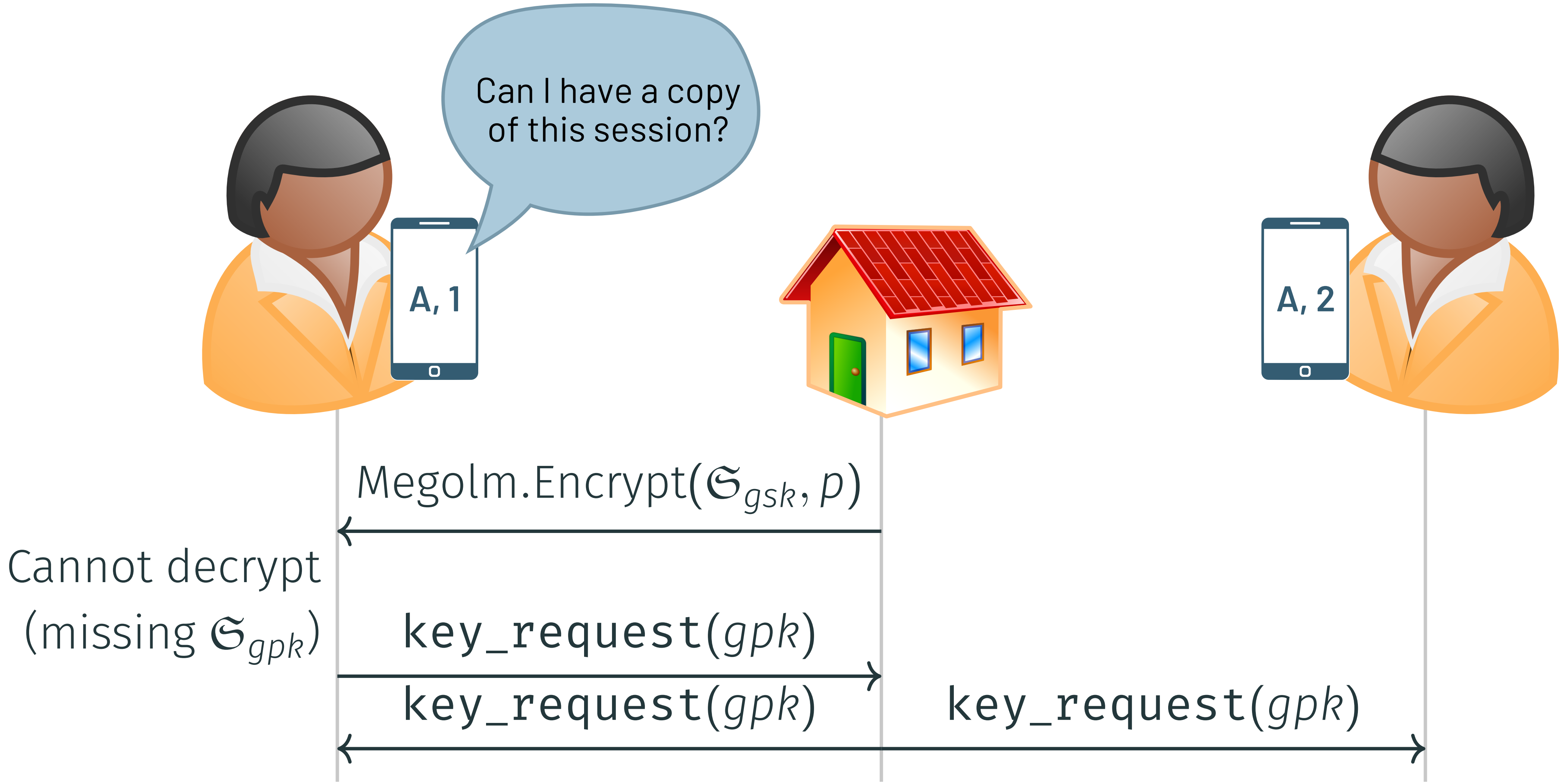
Key Request Protocol

1. Alice's device receives a ciphertext it can't decrypt.



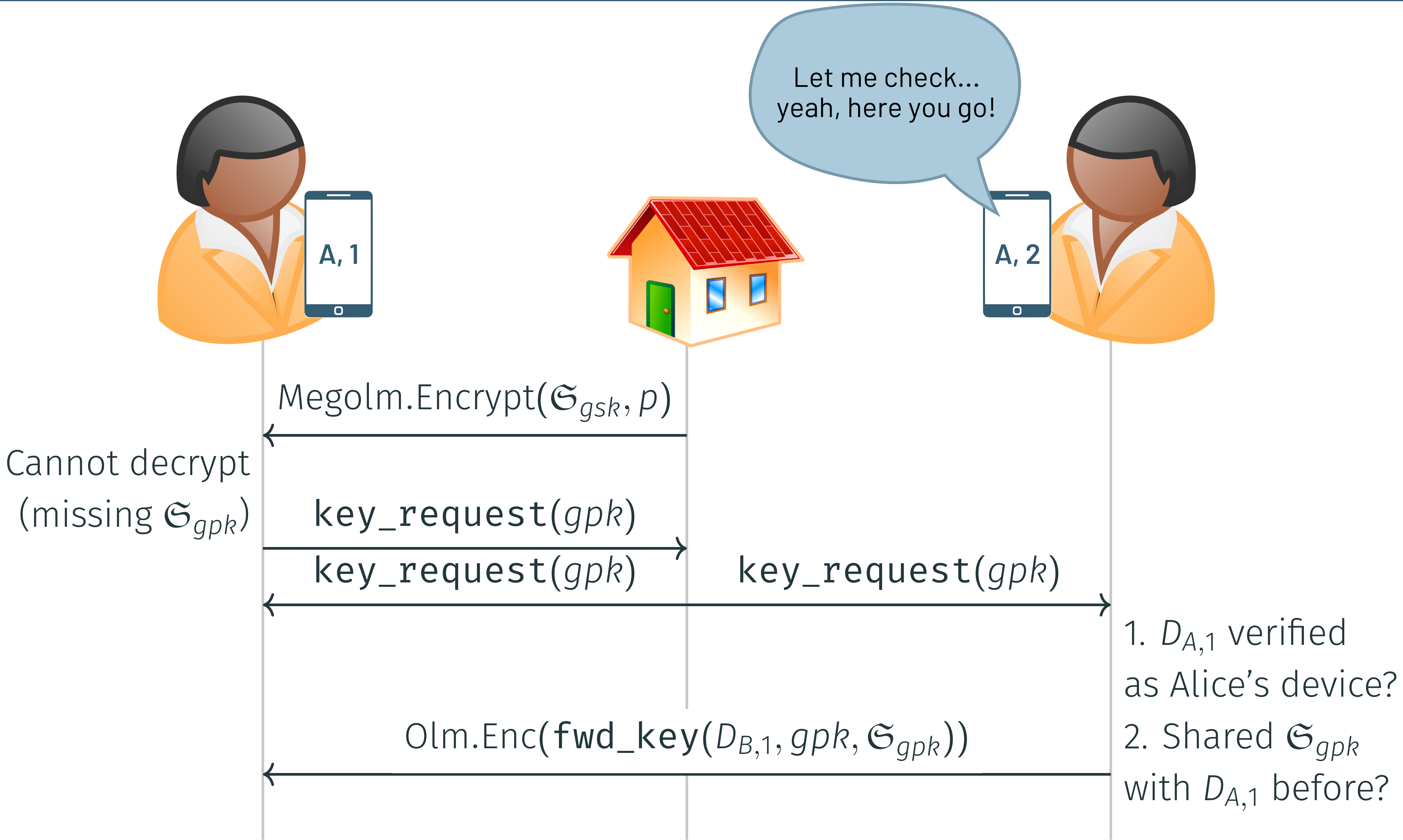
Key Request Protocol

2. Alice's client requests a copy of the decryption keys.



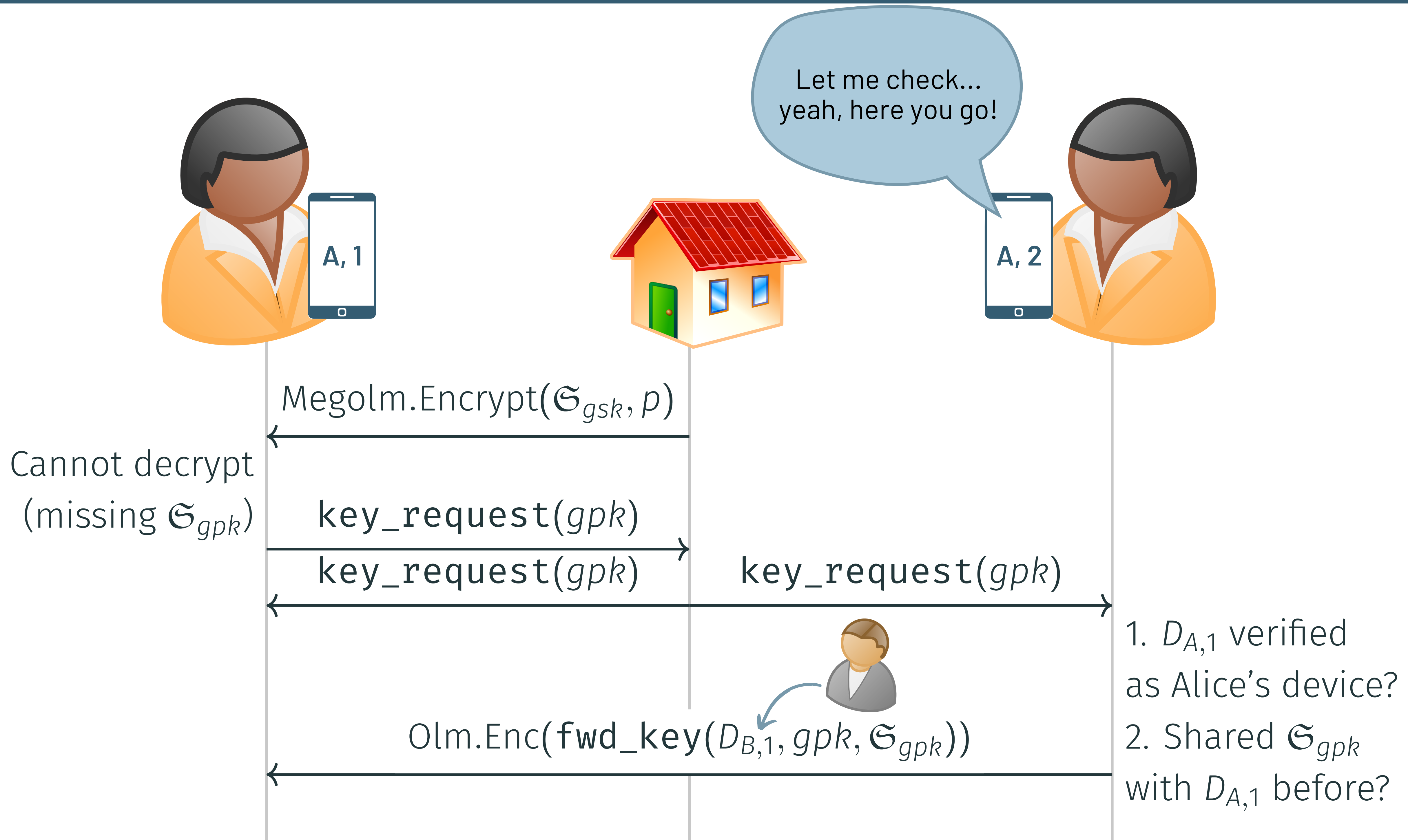
Key Request Protocol

3. Another device (possibly) shares their copy of the keys.



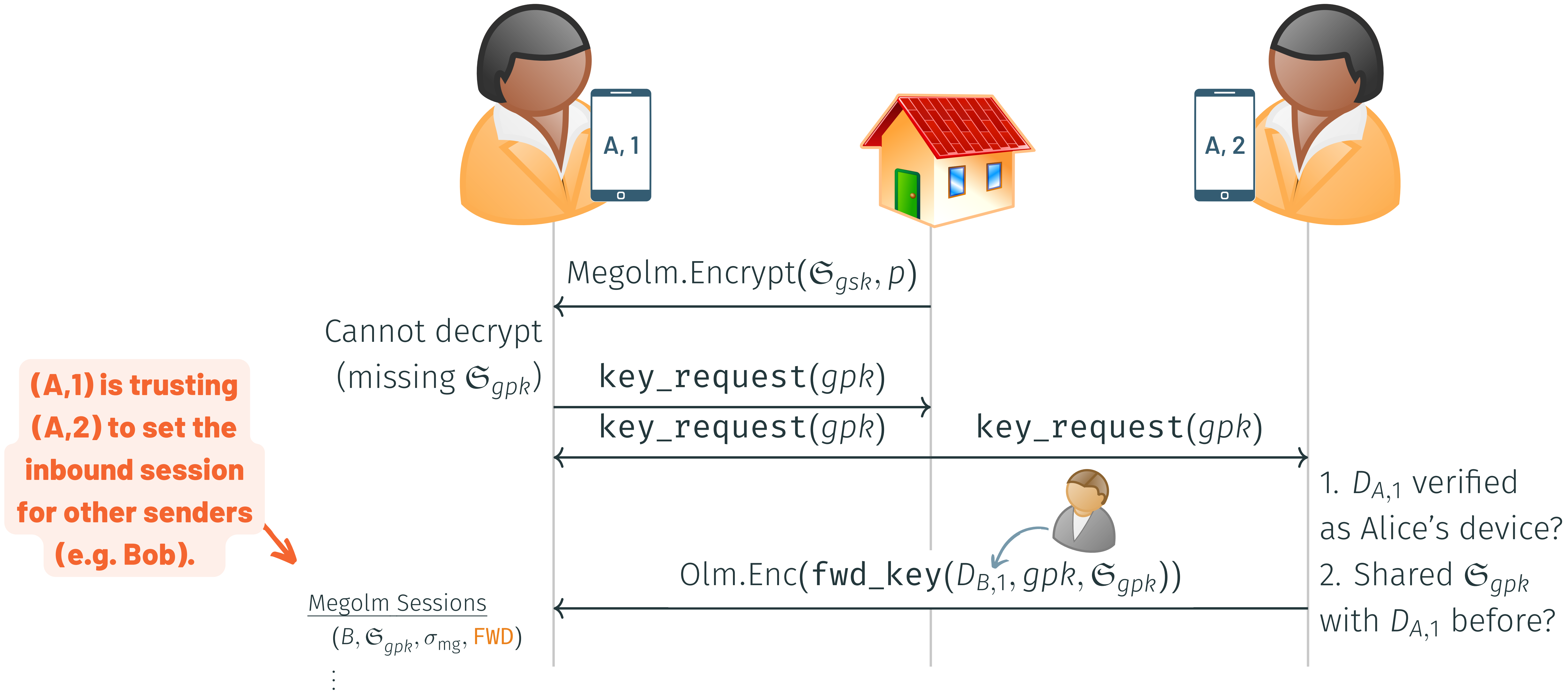
Key Request Protocol

3. Another device (possibly) shares their copy of the keys.



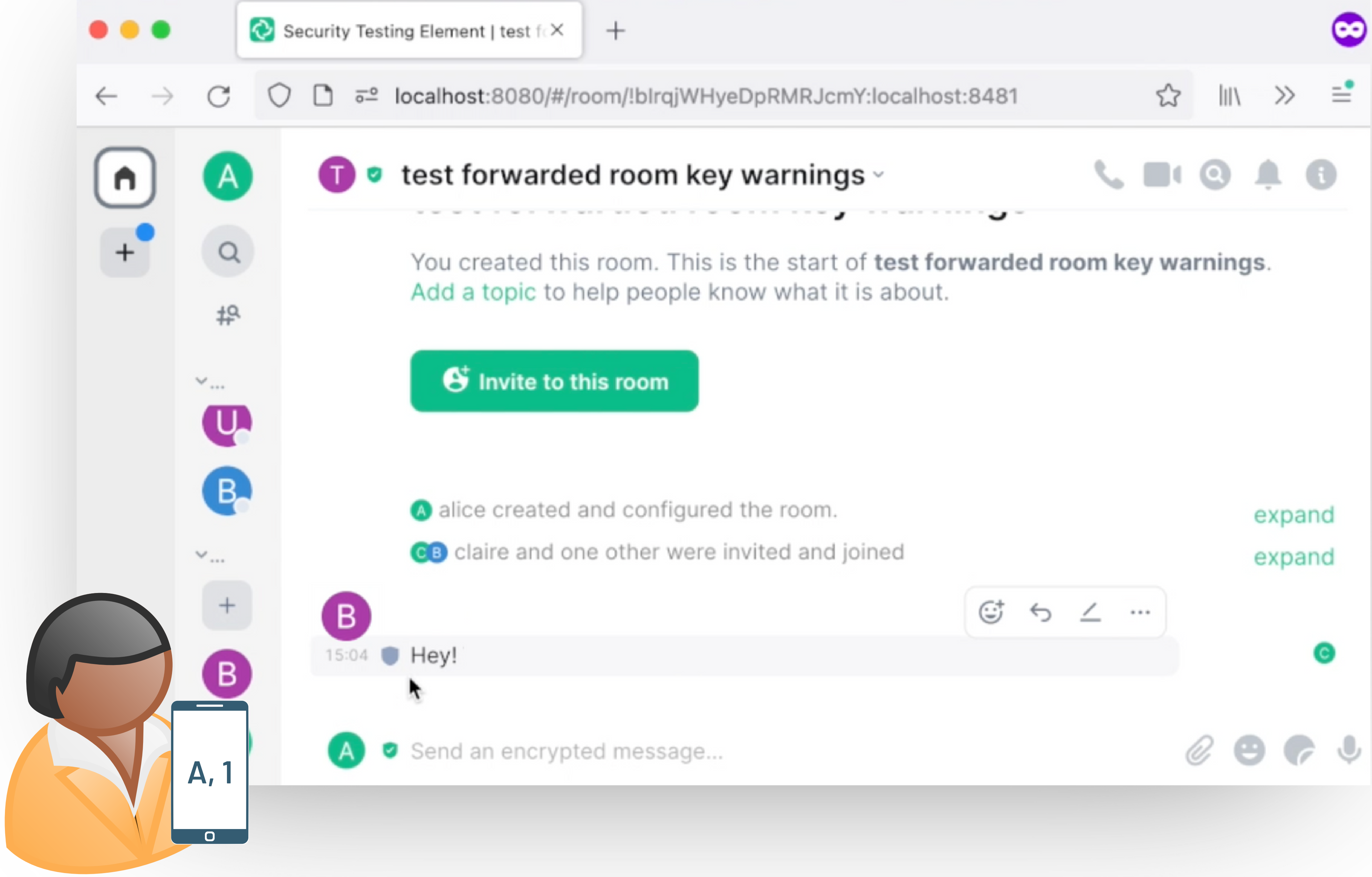
Key Request Protocol

4. (Possibly) accept shared keys then decrypt message.



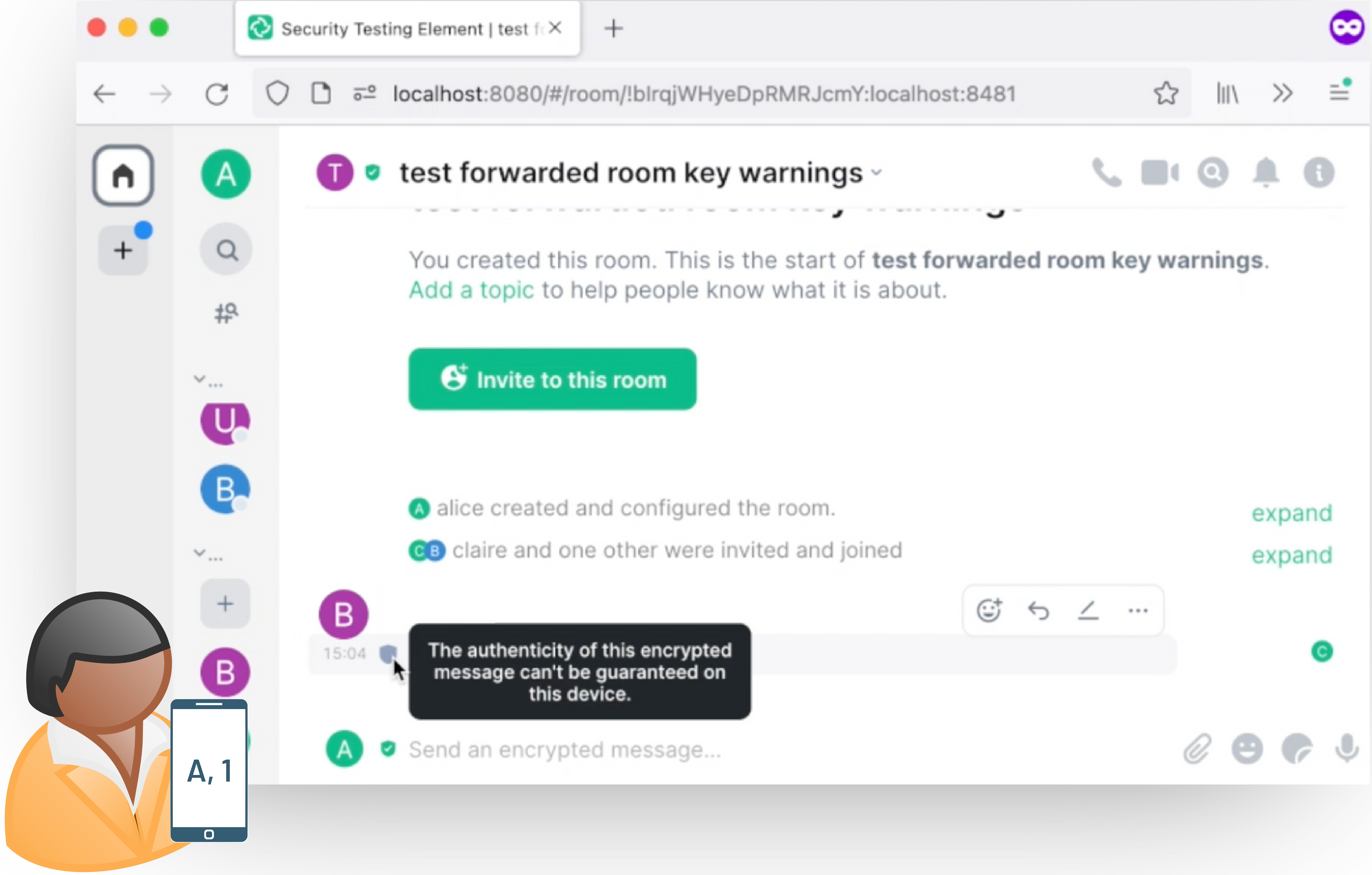
Key Request Protocol

4. (Possibly) accept shared keys then decrypt message.

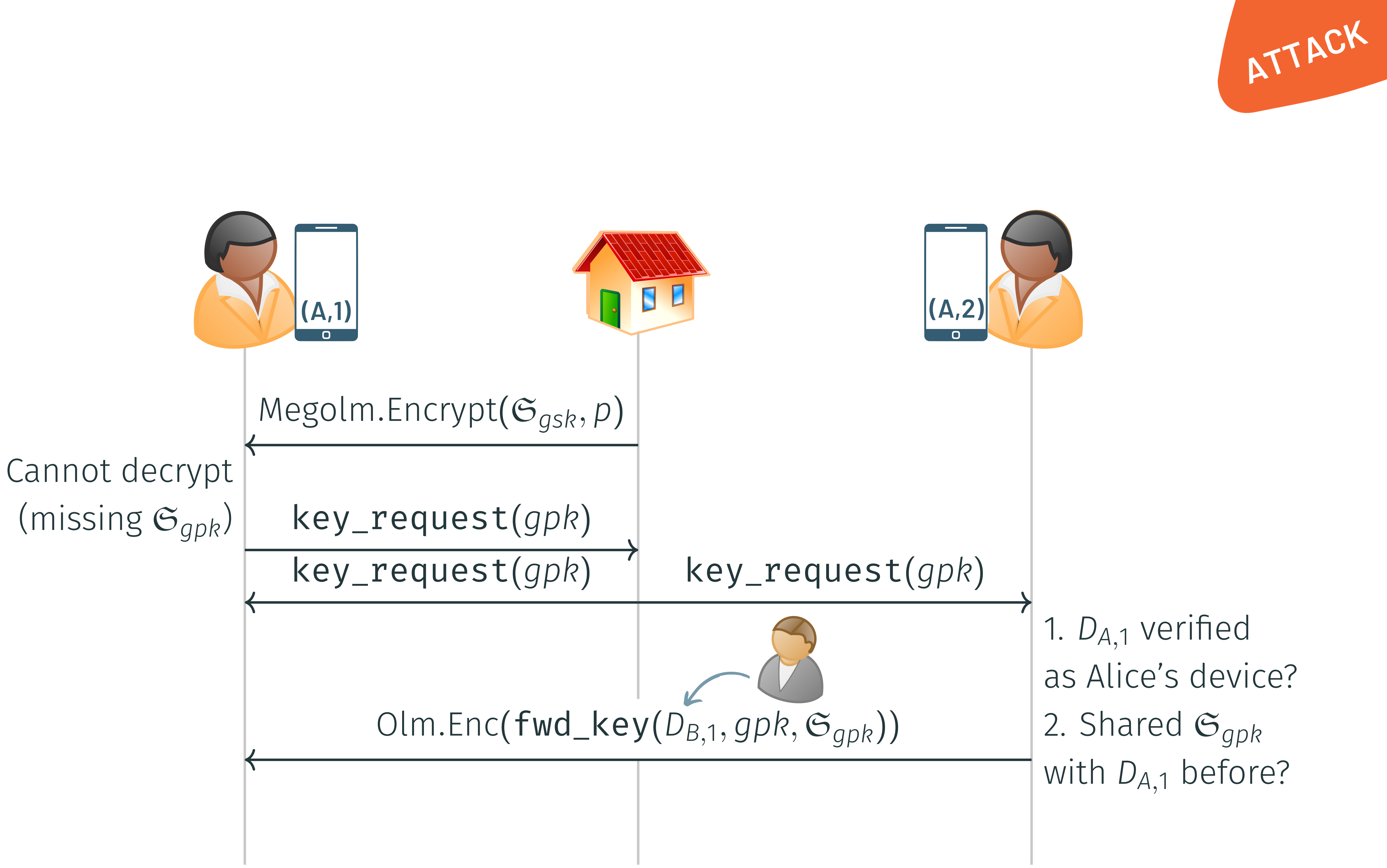


Key Request Protocol

4. (Possibly) accept shared keys then decrypt message.



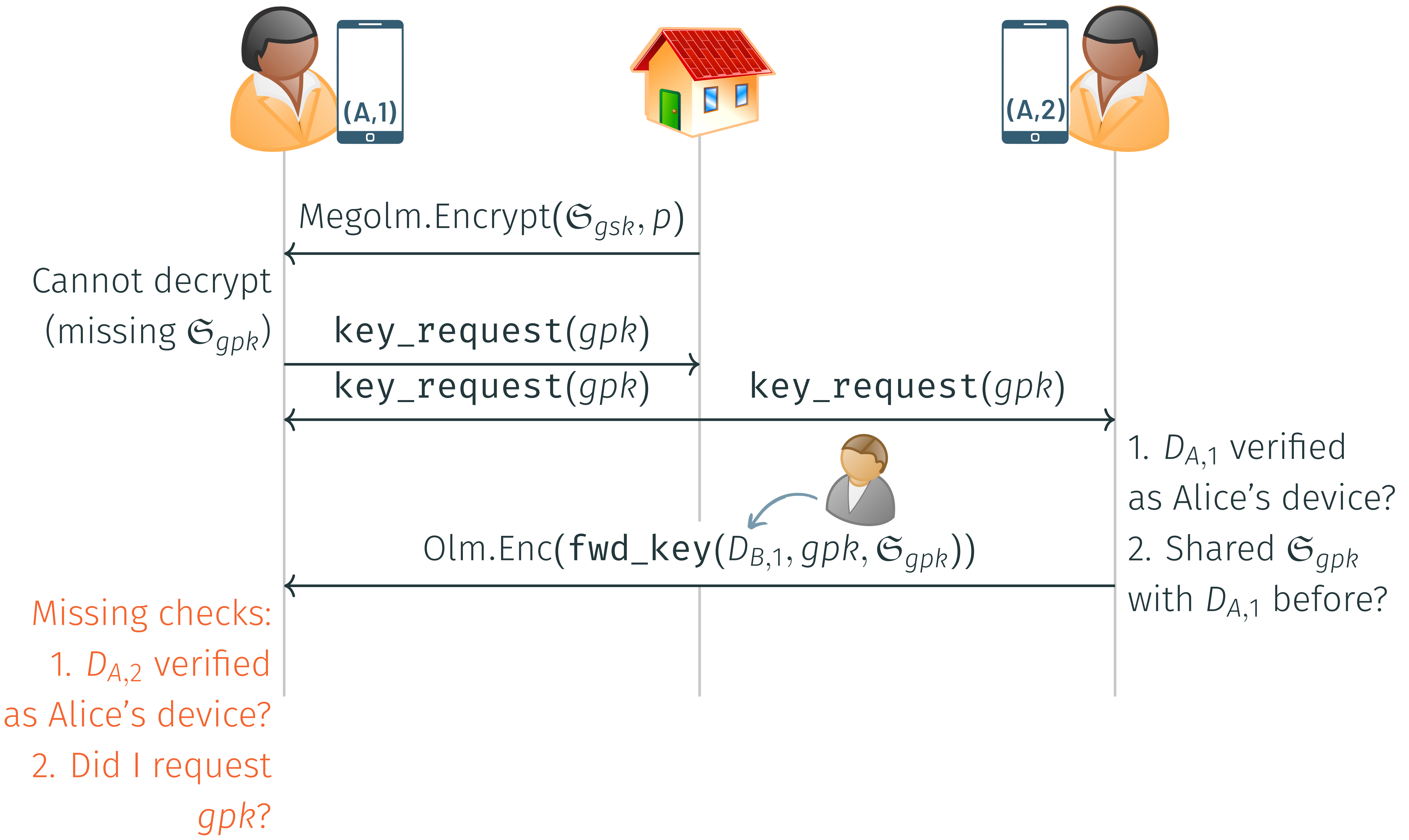
Impersonation through key sharing



ATTACK

Impersonation through key sharing

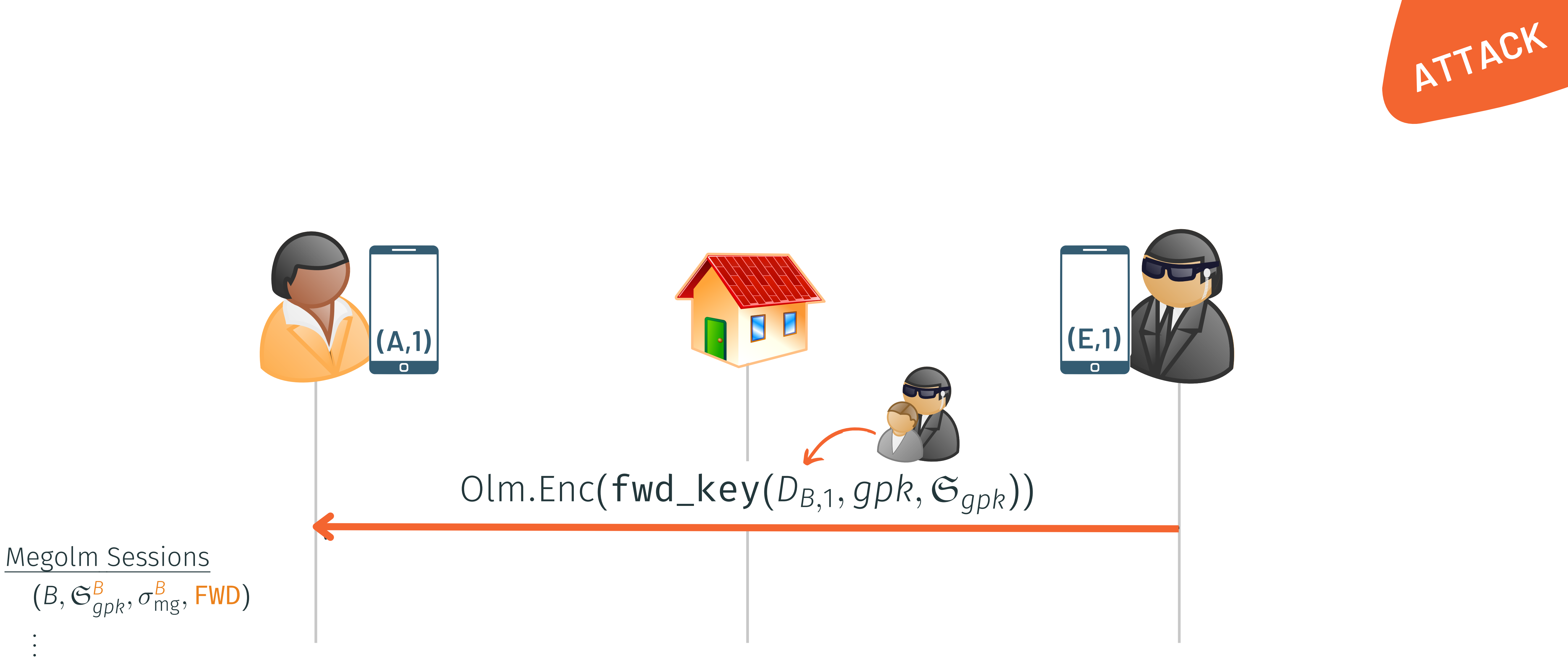
- Missing checks on the *receiving side*.



Impersonation through key sharing

- Missing checks on the *receiving side*.
- Allows attackers to inject an inbound Megolm session as if it were someone else's.

↪ Impersonation *

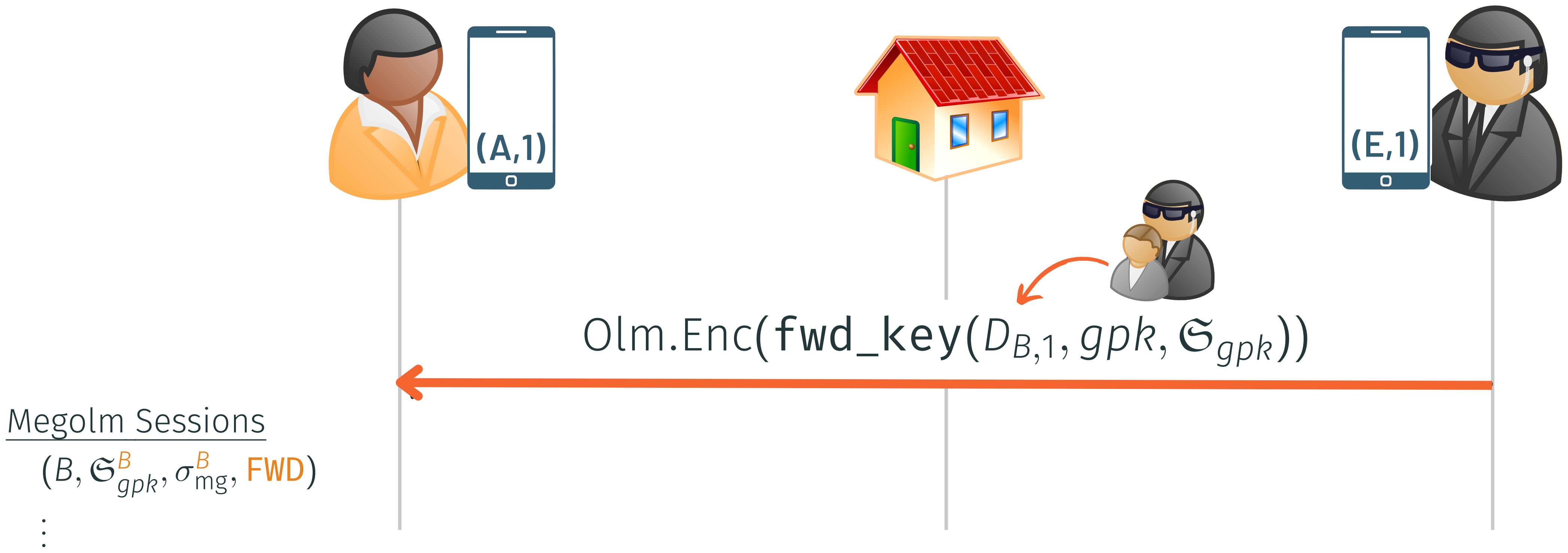


ATTACK

Impersonation through key sharing

- Missing checks on the *receiving side*.
- Allows attackers to inject an inbound Megolm session as if it were someone else's.

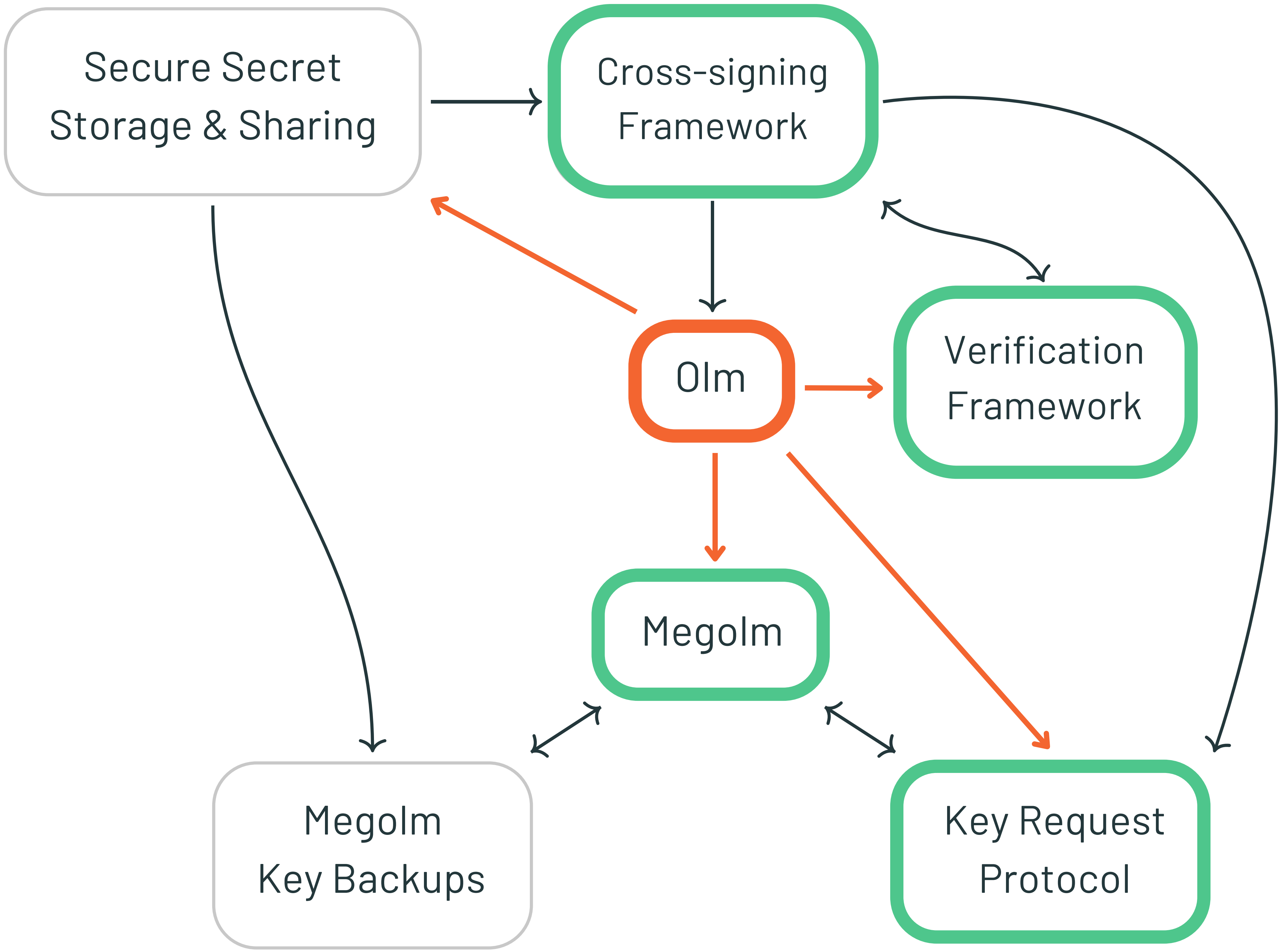
↪ Impersonation *



* The authenticity of this encrypted message can't be guaranteed on this device.

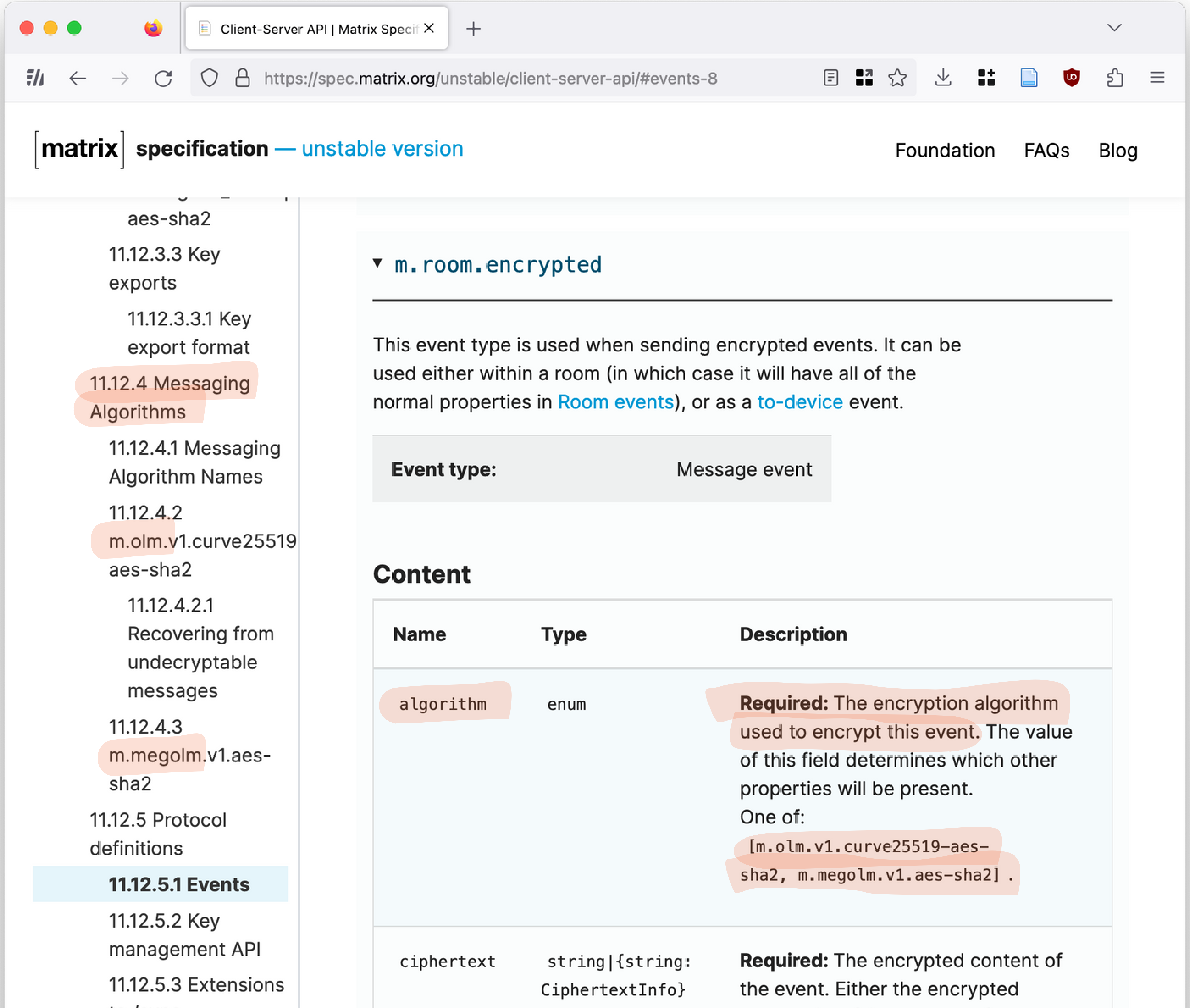
Modelling Olm's Composition

How to differentiate messages from different protocols and purposes?



Modelling Olm's Composition

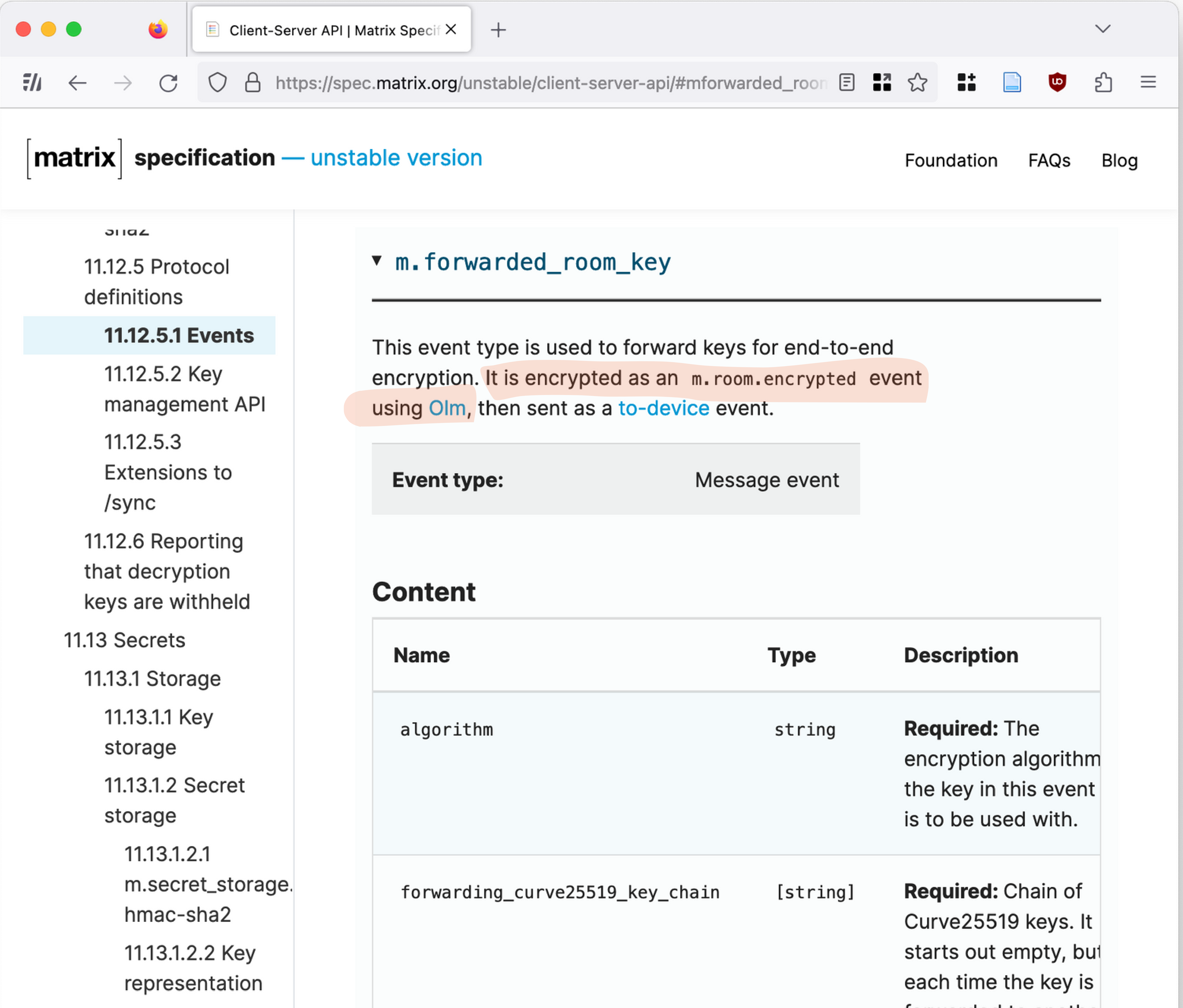
Olm and Megolm are *generally* interchangable in the specification.



Modelling Olm's Composition

Olm and Megolm are *generally* interchangable in the specification.

With some special casing.



Modelling Olm's Composition

How to differentiate messages from different protocols and purposes?

$\text{Matrix.Decrypt}(st_{mt}, type, alg, c)$

// Decrypt ciphertext

if $(type = \texttt{m.room.encrypted}) \land (alg = \texttt{megolm})$ **then**

// Select correct Megolm session for decryption

$\mathfrak{S}'_{gpk}, ipk_{snd}, type, m \leftarrow \text{Megolm.Decrypt}(\mathfrak{S}_{gpk}, c)$

elseif $(type = \texttt{m.room.encrypted}) \land (alg = \texttt{olm})$ **then**

// Select correct Olm session for decryption

$st_{olm}, ipk_{snd}, type, m \leftarrow \text{Olm.Dec}(st_{olm}, c)$

// Handle plaintext

if $(type = \texttt{m.room.message})$ **then**

// Handle room message

elseif $(type = \texttt{m.room\_key})$ **then**

// Handle initial Megolm session distribution

elseif $(\ldots)$ **then**

// Handle other message types

Modelling Olm's Composition

How to differentiate messages from different protocols and purposes?

$\text{Matrix.Decrypt}(st_{mt}, type, alg, c)$

// Decrypt ciphertext

if $(type = \texttt{m.room.encrypted}) \wedge (alg = \texttt{megolm})$ **then**

// Select correct Megolm session for decryption

$\mathfrak{S}'_{gpk}, ipk_{snd}, type, m \leftarrow \text{Megolm.Decrypt}(\mathfrak{S}_{gpk}, c)$

elseif $(type = \texttt{m.room.encrypted}) \wedge (alg = \texttt{olm})$ **then**

// Select correct Olm session for decryption

$st_{olm}, ipk_{snd}, type, m \leftarrow \text{Olm.Dec}(st_{olm}, c)$

// Handle plaintext

if $(type = \texttt{m.room.message})$ **then**

// Handle room message

elseif $(type = \texttt{m.room\_key})$ **then**

// Handle initial Megolm session distribution

elseif $(\ldots)$ **then**

// Handle other message types

Modelling Olm's Composition

How to differentiate messages from different protocols and purposes?

$\mathsf{Matrix.Decrypt}(st_{mt}, type, alg, c)$

// Decrypt ciphertext

if $(type = \texttt{m.room.encrypted}) \wedge (alg = \texttt{megolm})$ then

// Select correct Megolm session for decryption

$\mathfrak{S}'_{gpk}, ipk_{snd}, type, m \leftarrow \mathsf{Megolm.Decrypt}(\mathfrak{S}_{gpk}, c)$

elseif $(type = \texttt{m.room.encrypted}) \wedge (alg = \texttt{olm})$ then

// Select correct Olm session for decryption

$st_{olm}, ipk_{snd}, type, m \leftarrow \mathsf{Olm.Dec}(st_{olm}, c)$

// Handle plaintext

if $(type = \texttt{m.room.message})$ then

// Handle room message

elseif $(type = \texttt{m.room\_key})$ then

// Handle initial Megolm session distribution

elseif $(\ldots)$ then

// Handle other message types

Determine if message was encrypted by checking if ipk_snd has been set.

Modelling Olm's Composition

How to differentiate messages from different protocols and purposes?

$\mathsf{Matrix.Decrypt}(st_{mt}, type, alg, c)$

// Decrypt ciphertext

if $(type = \texttt{m.room.encrypted}) \wedge (alg = \texttt{megolm})$ **then**

  // Select correct Megolm session for decryption

  $\mathfrak{S}'_{gpk}, ipk_{snd}, type, m \leftarrow \mathsf{Megolm.Decrypt}(\mathfrak{S}_{gpk}, c)$

elseif $(type = \texttt{m.room.encrypted}) \wedge (alg = \texttt{olm})$ **then**

  // Select correct Olm session for decryption

  $st_{olm}, ipk_{snd}, type, m \leftarrow \mathsf{Olm.Dec}(st_{olm}, c)$

// Handle plaintext

if $(type = \texttt{m.room.message})$ **then**

  // Handle room message

elseif $(type = \texttt{m.room\_key})$ **then**

  // Handle initial Megolm session distribution

elseif $(\ldots)$ **then**

  // Handle other message types

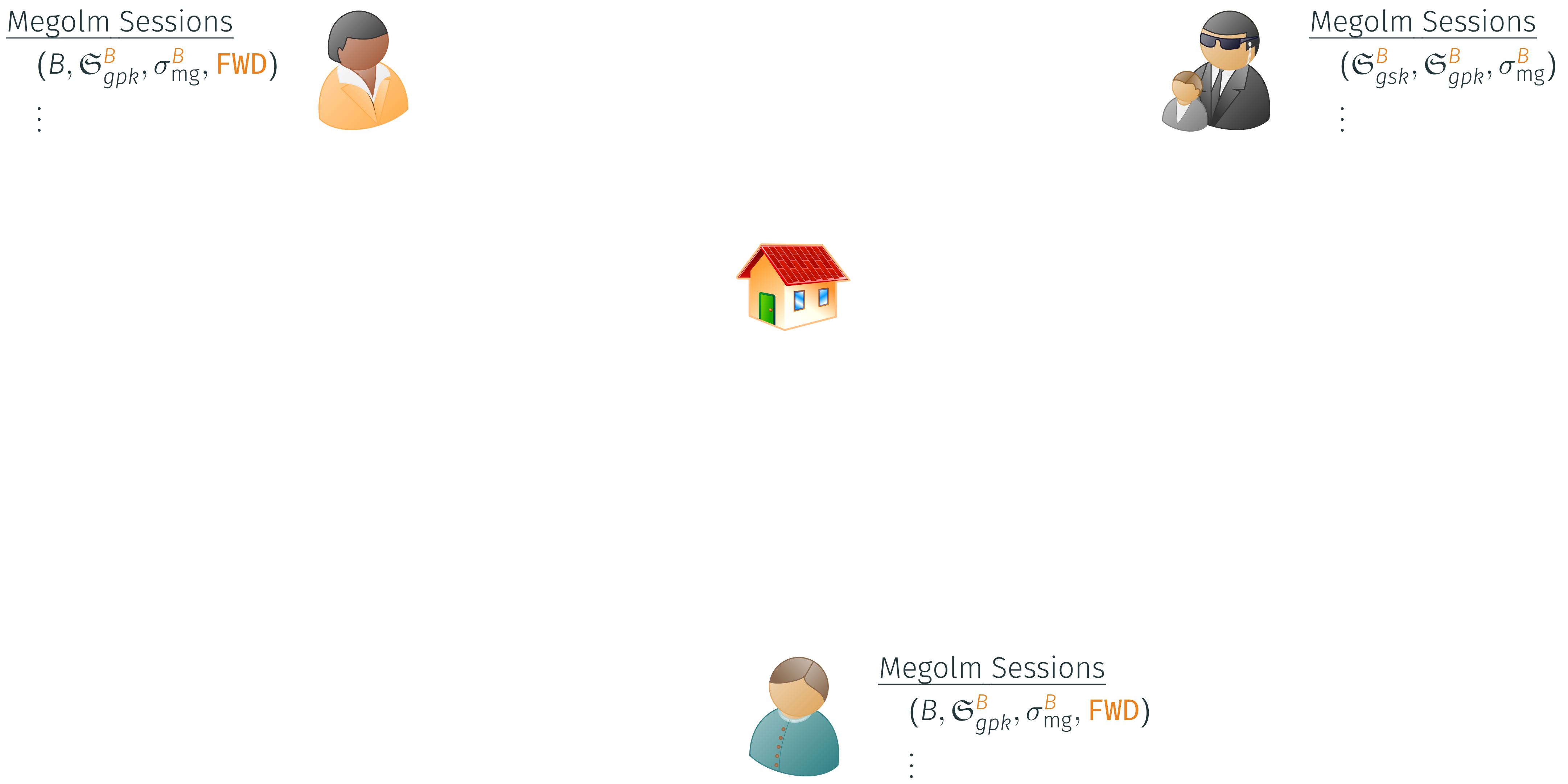
But never check which algorithm was used for encryption!

ATTACK

Olm/Megolm Protocol Confusion

Keys sent as part of Megolm's *initial key share* are *fully trusted.*
But they are sent over Olm.

Can we share a session key over Megolm?

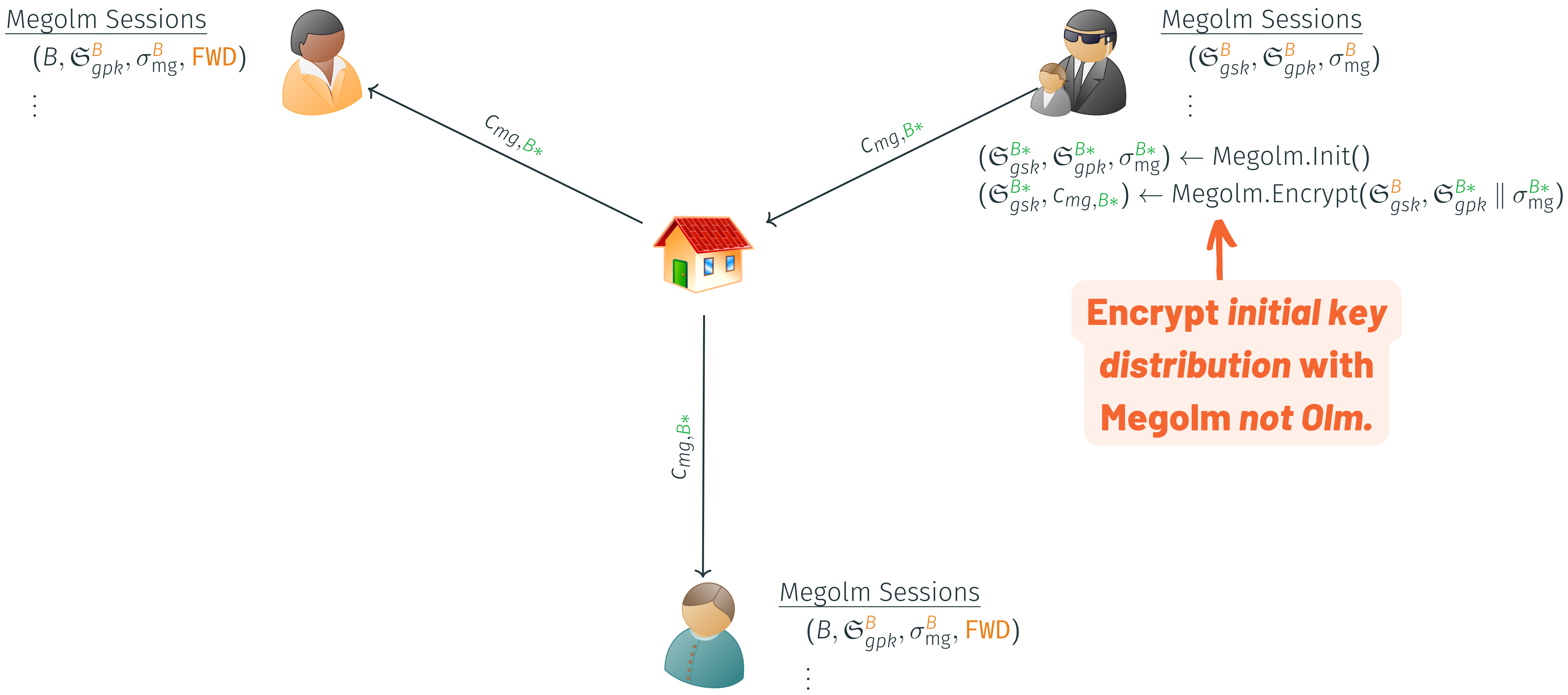


Olm/Megolm Protocol Confusion

Keys sent as part of Megolm's *initial key share* are *fully trusted*.
But they are sent over Olm.

Can we share a session key over Megolm? Yes

ATTACK



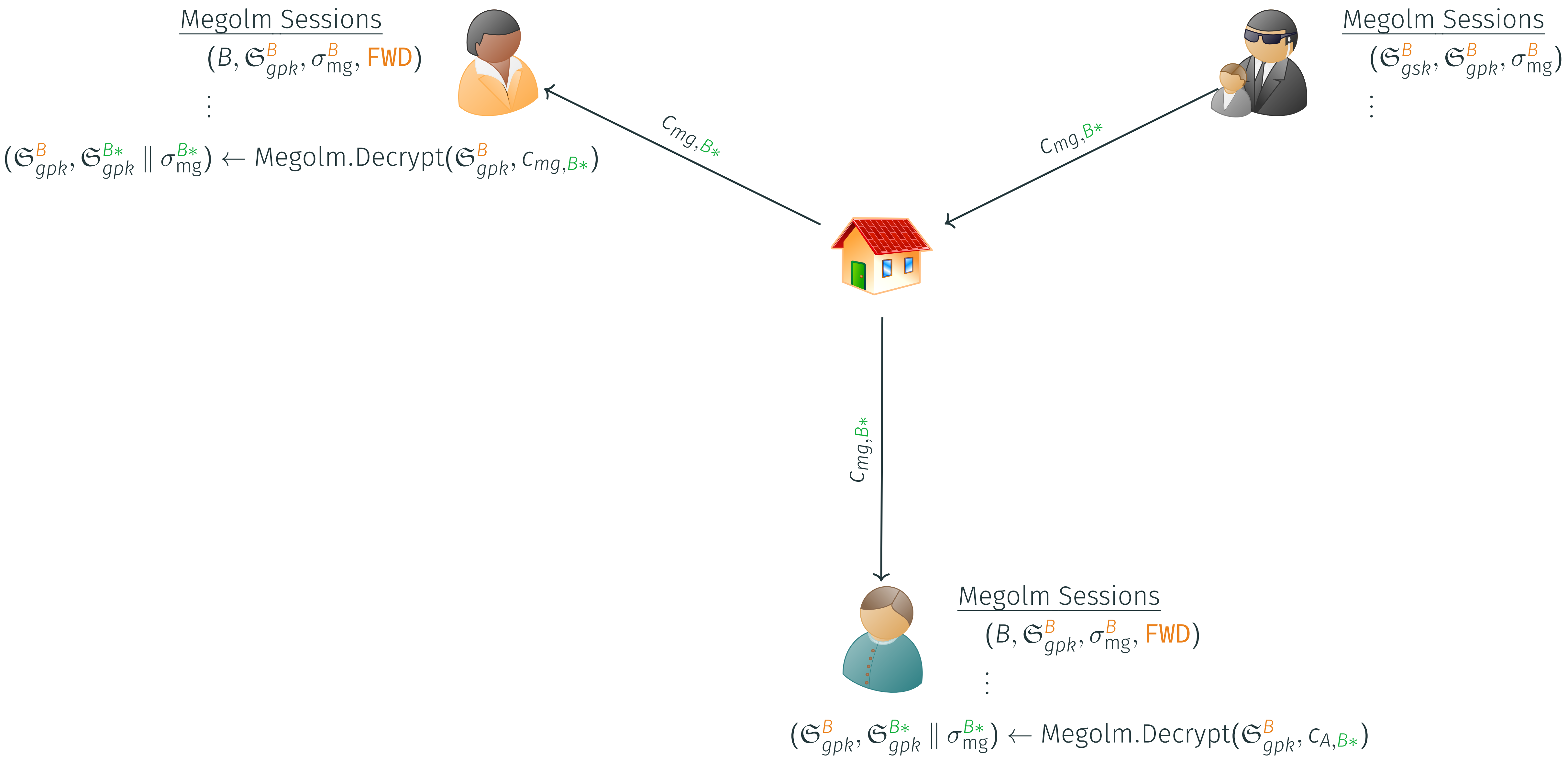
Encrypt *initial key distribution* with *Megolm not Olm.*

ATTACK

Olm/Megolm Protocol Confusion

Keys sent as part of Megolm's *initial key share* are *fully trusted*.
But they are sent over Olm.

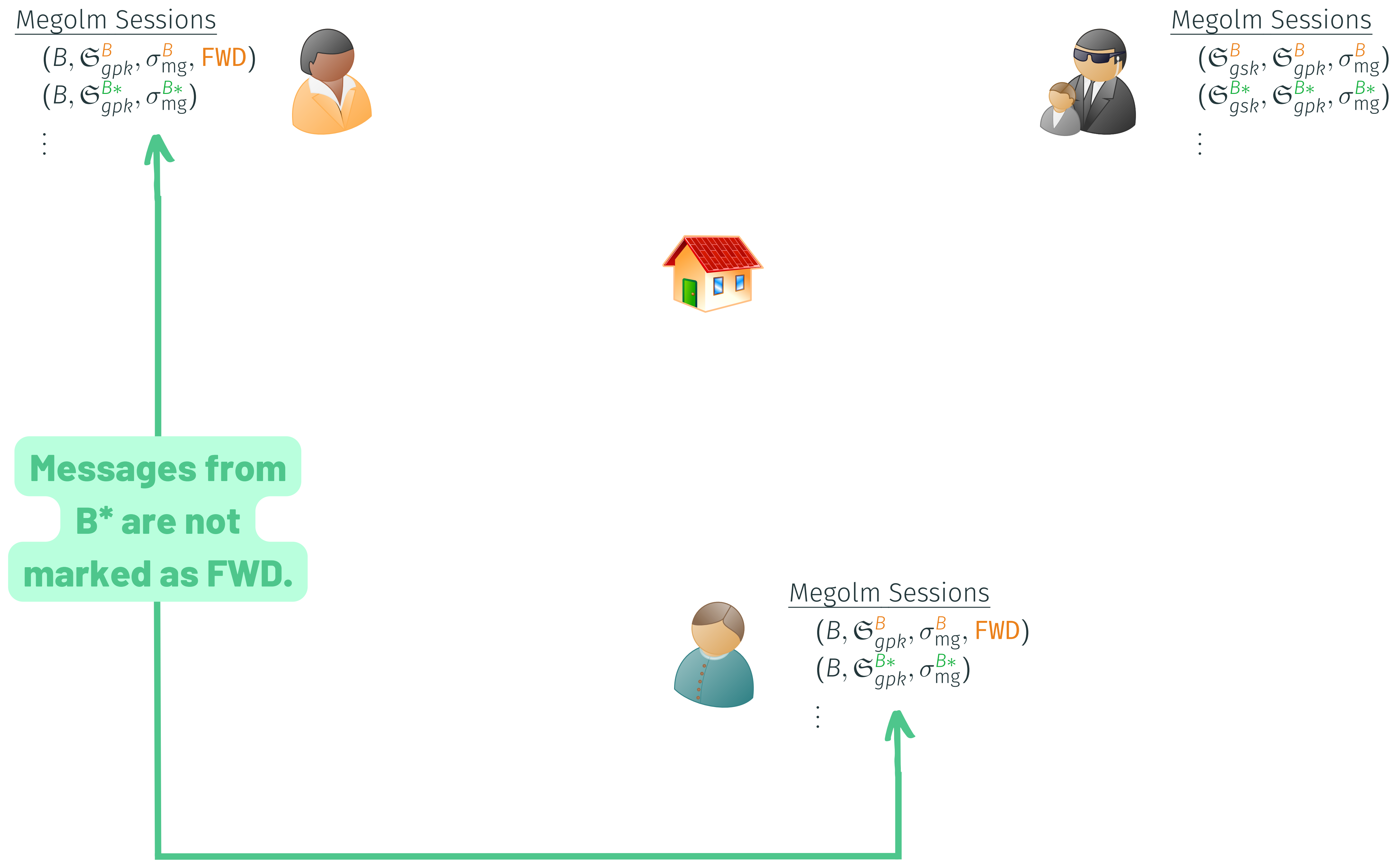
Can we share a session key over Megolm? ***Yes***



Olm/Megolm Protocol Confusion

Keys sent as part of Megolm's *initial key share* are *fully trusted*.
But they are sent over Olm.

Can we share a session key over Megolm? Yes



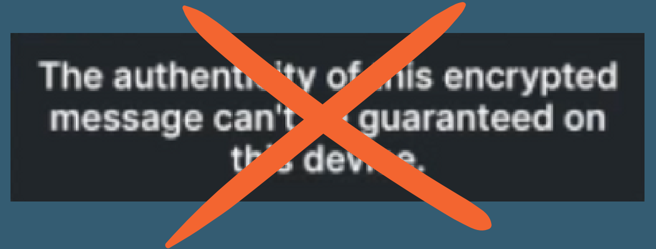
Olm/Megolm Protocol Confusion

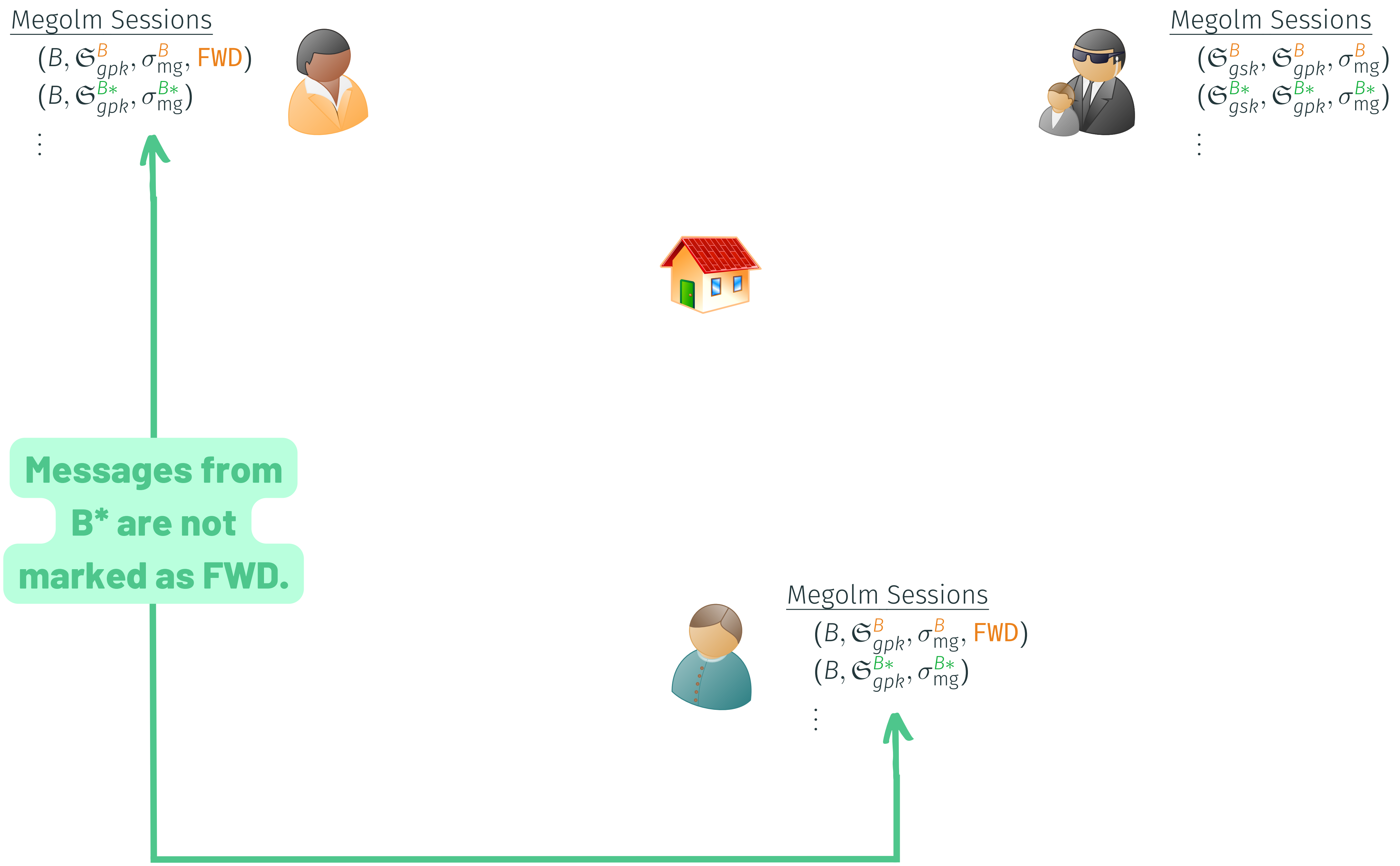
ATTACK

Keys sent as part of Megolm's *initial key share* are *fully trusted*.
But they are sent over Olm.

Can we share a session key over Megolm? Yes

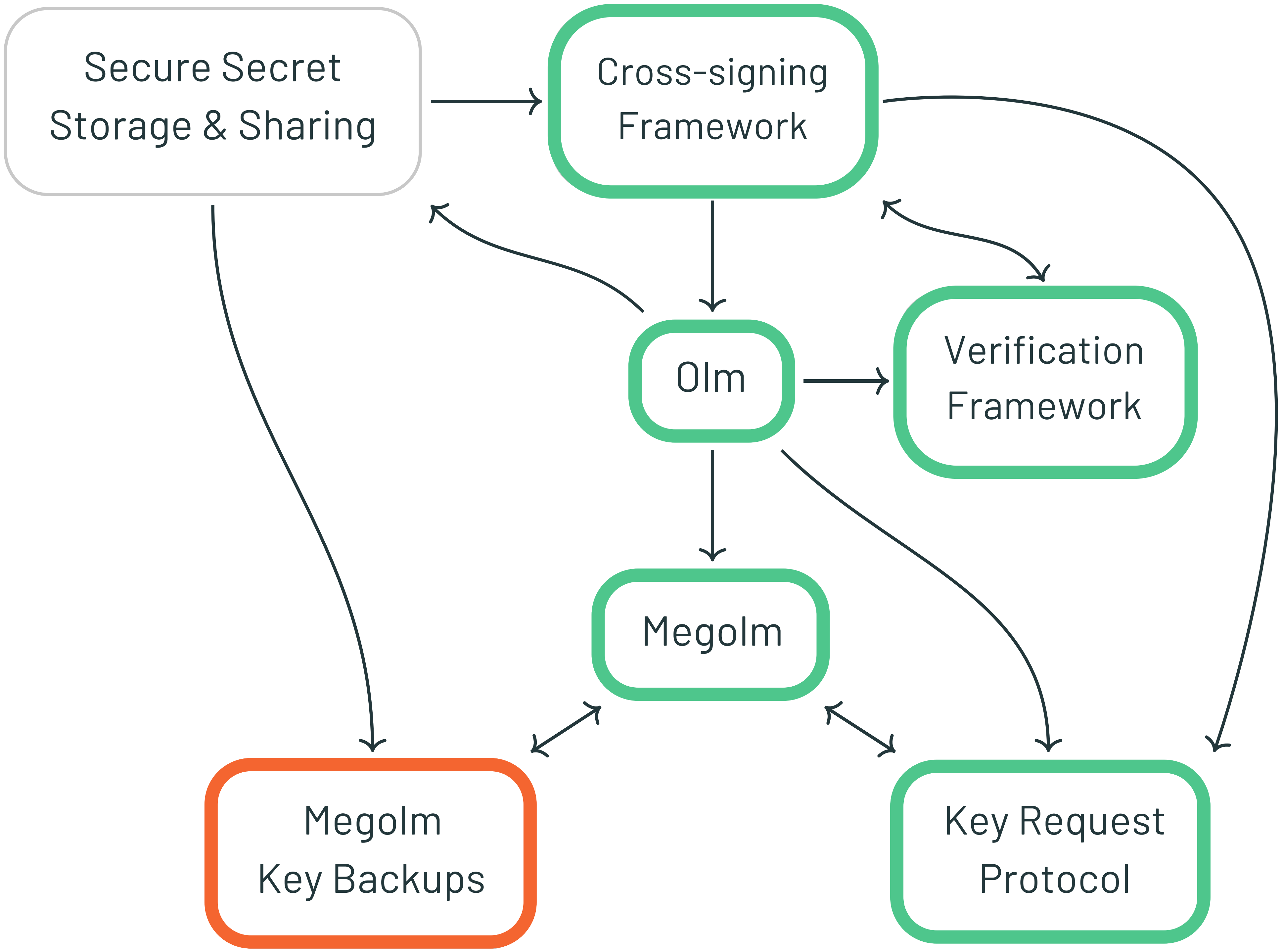
↪ Impersonation *(no caveat)*





Megolm Key Backups

- Asynchronous alternative to Key Request protocol.
- Inbound Megolm sessions are encrypted and saved to the homeserver.
- Encrypt using a secret key shared between a user's devices.



Megolm Key Backups

1. Setup

$bsk, bpk \leftarrow \text{X25519.KGen}(1^n)$

$\texttt{auth\_data}(bpk)$ →

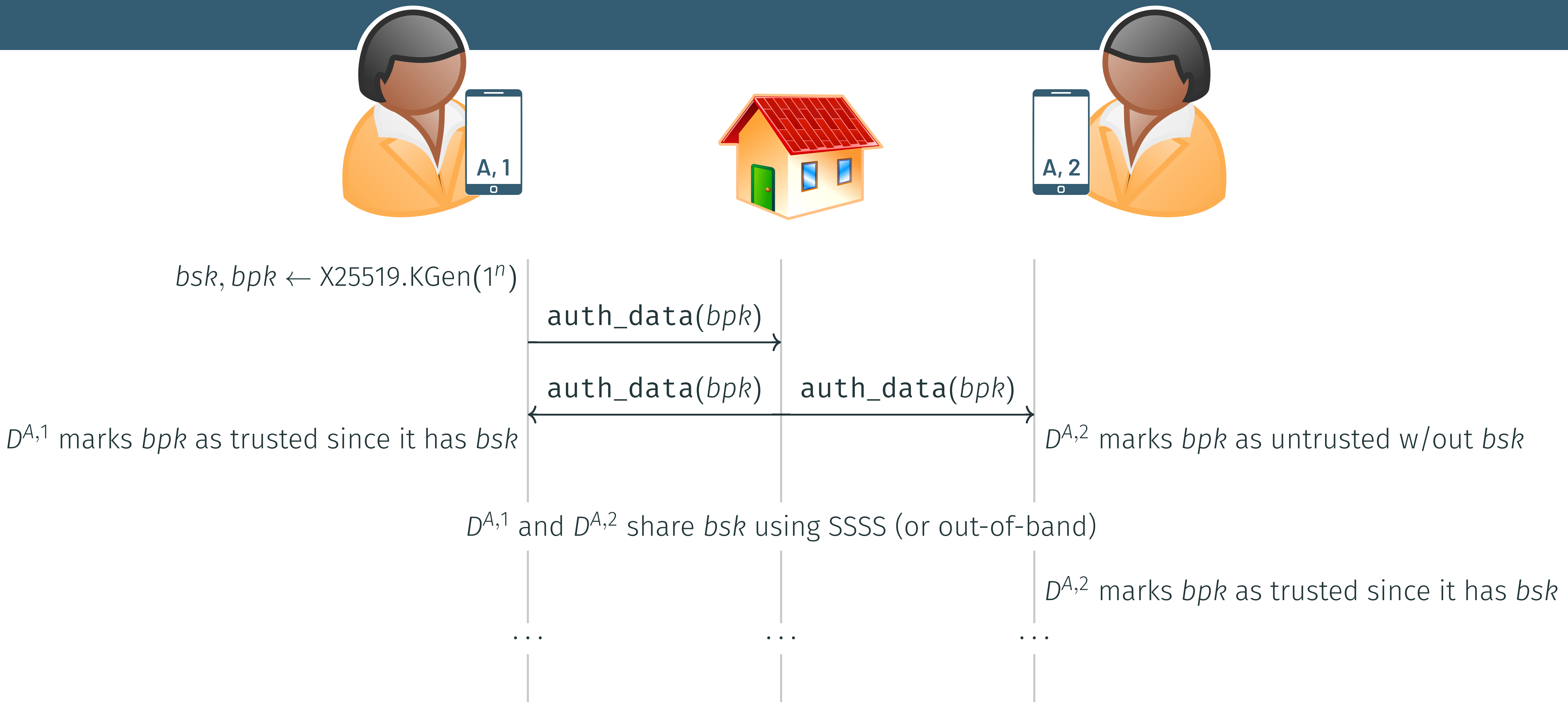
← $\texttt{auth\_data}(bpk)$ | $\texttt{auth\_data}(bpk)$ →

$D^{A,1}$ marks bpk as trusted since it has bsk

$D^{A,2}$ marks bpk as untrusted w/out bsk

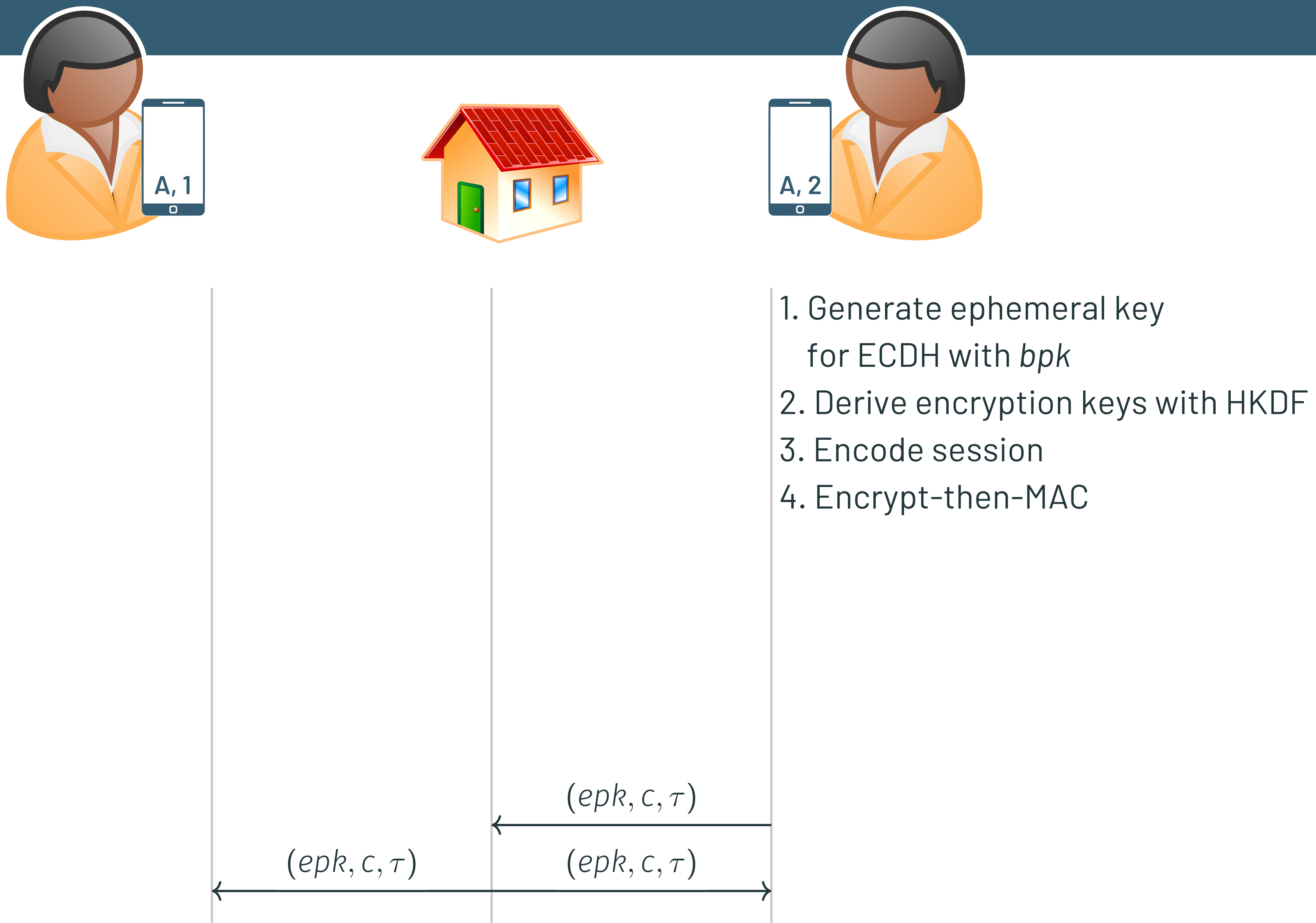
Megolm Key Backups

1. Setup



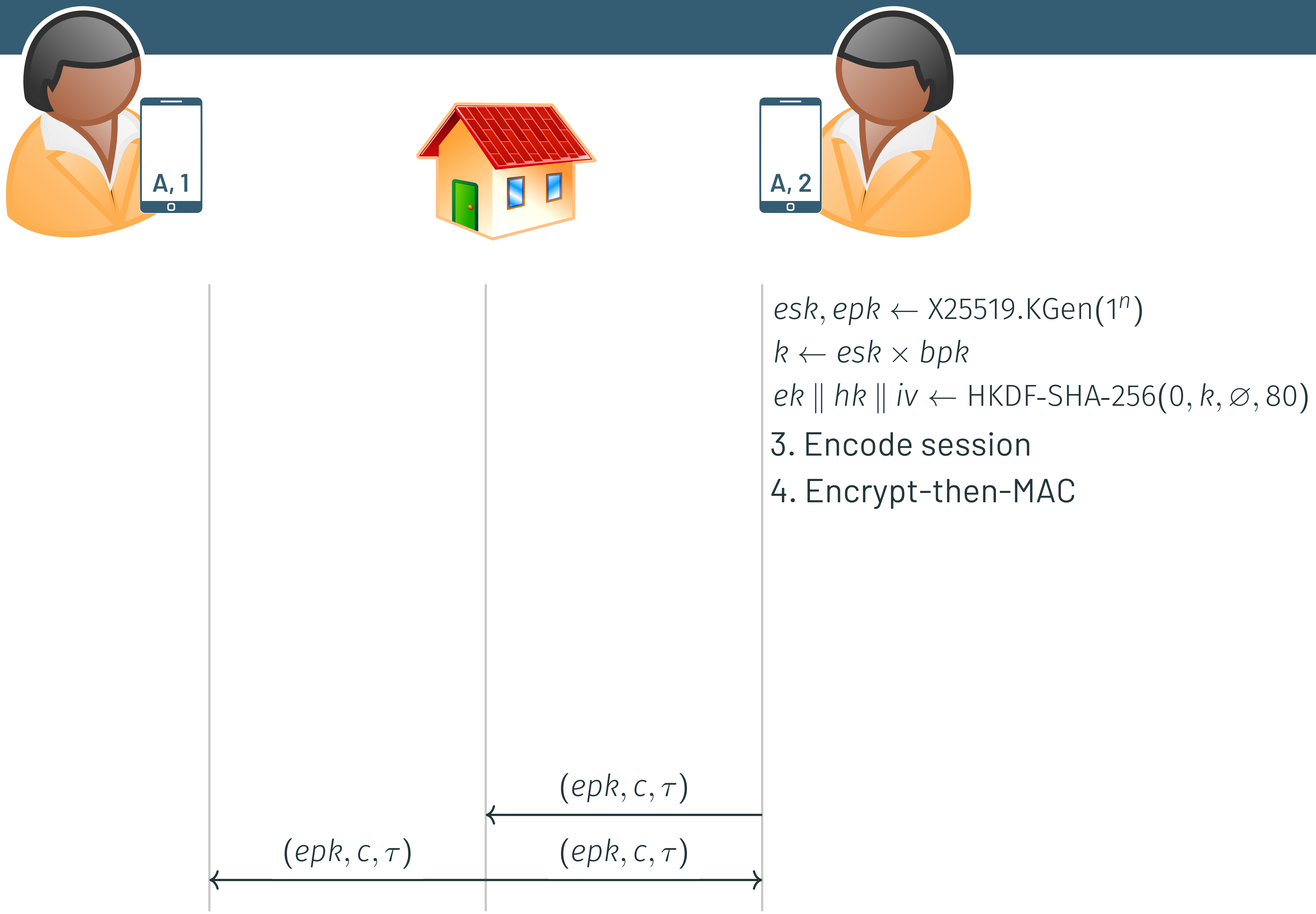
Megolm Key Backups

2. Backup



Megolm Key Backups

2. Backup



Megolm Key Backups

2. Backup

$$esk, epk \leftarrow \text{X25519.KGen}(1^n)$$
$$k \leftarrow esk \times bpk$$
$$ek \,\|\, hk \,\|\, iv \leftarrow \text{HKDF-SHA-256}(0, k, \varnothing, 80)$$
$$m \leftarrow \texttt{session\_data}($$
$$\quad \text{alg} = \texttt{megolm},$$
$$\quad \text{id} = gpk,$$
$$\quad \text{session} = \mathfrak{S}_{gpk},$$
$$\quad \text{dpk} = dpk_{\text{snd}},$$
$$\quad \text{ipk} = ipk_{\text{snd}},$$
$$\quad \text{fwd} = dpk_{A,2} \,\|\, \texttt{FWD})$$
$$c \leftarrow \text{AES-CBC}(ek, iv, m)$$
$$\tau \leftarrow \text{HMAC-SHA-256}(hk, c)$$

(epk, c, τ)

(epk, c, τ) (epk, c, τ)

Secure Secret Storage & Sharing

Backup, recover and share *user-level secrets.*

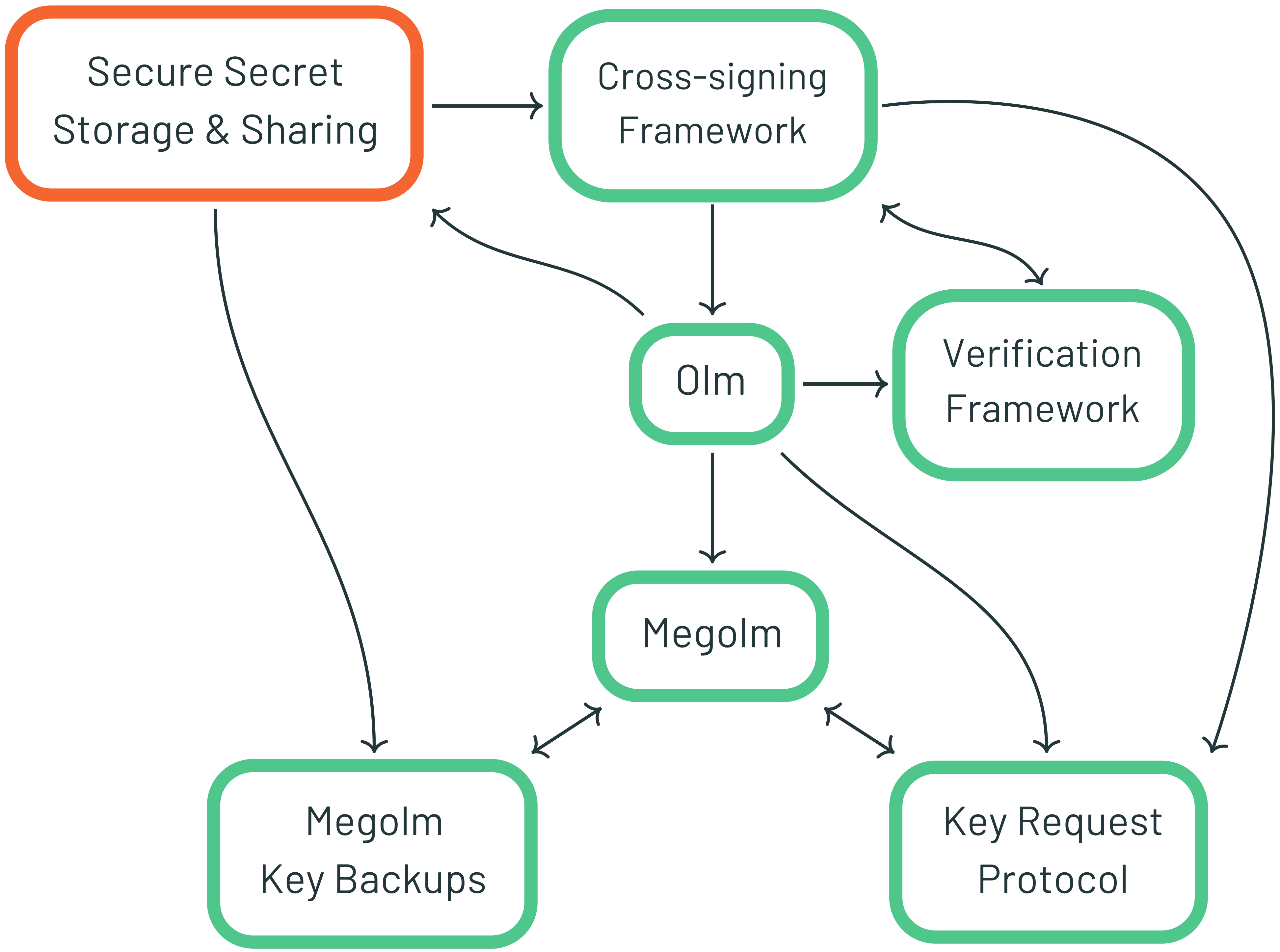
E.g. Alice's *(msk, usk, ssk).*

Secret Storage:

- Encrypt secrets and store on homeserver.
- Shared symmetric key (may be password-derived).

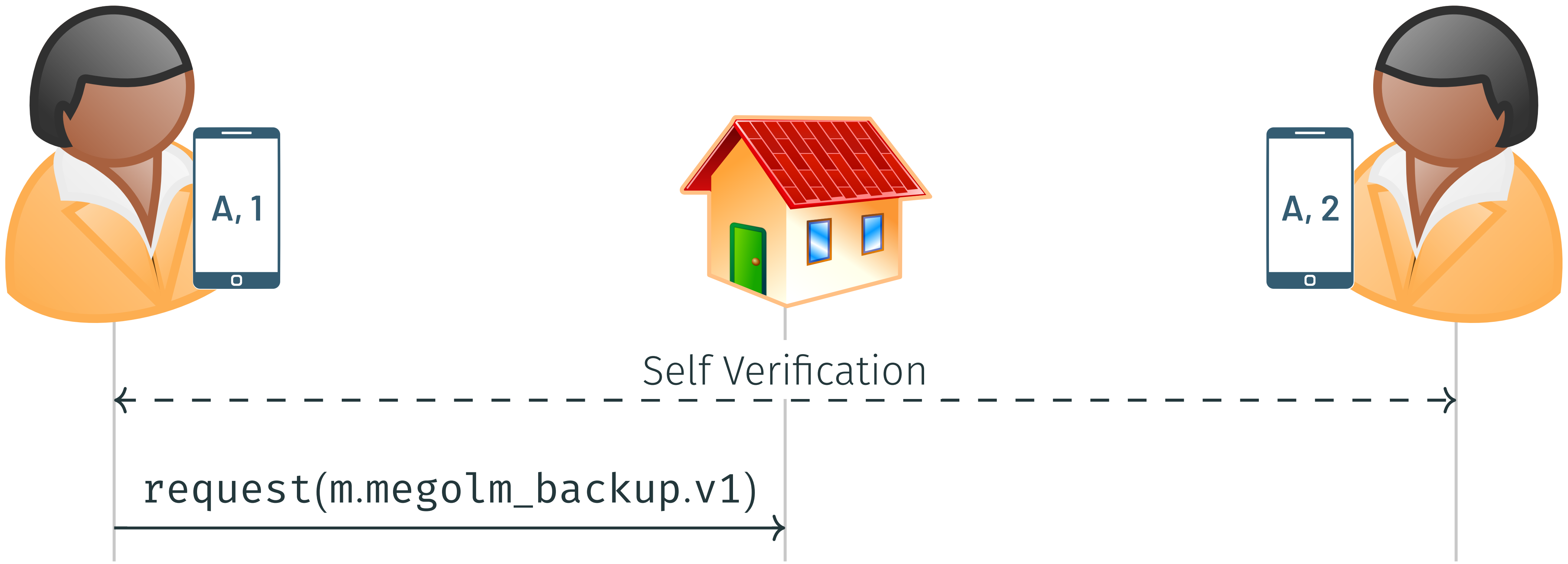
Secret Sharing:

- Use Olm to share secrets to newly verified devices.



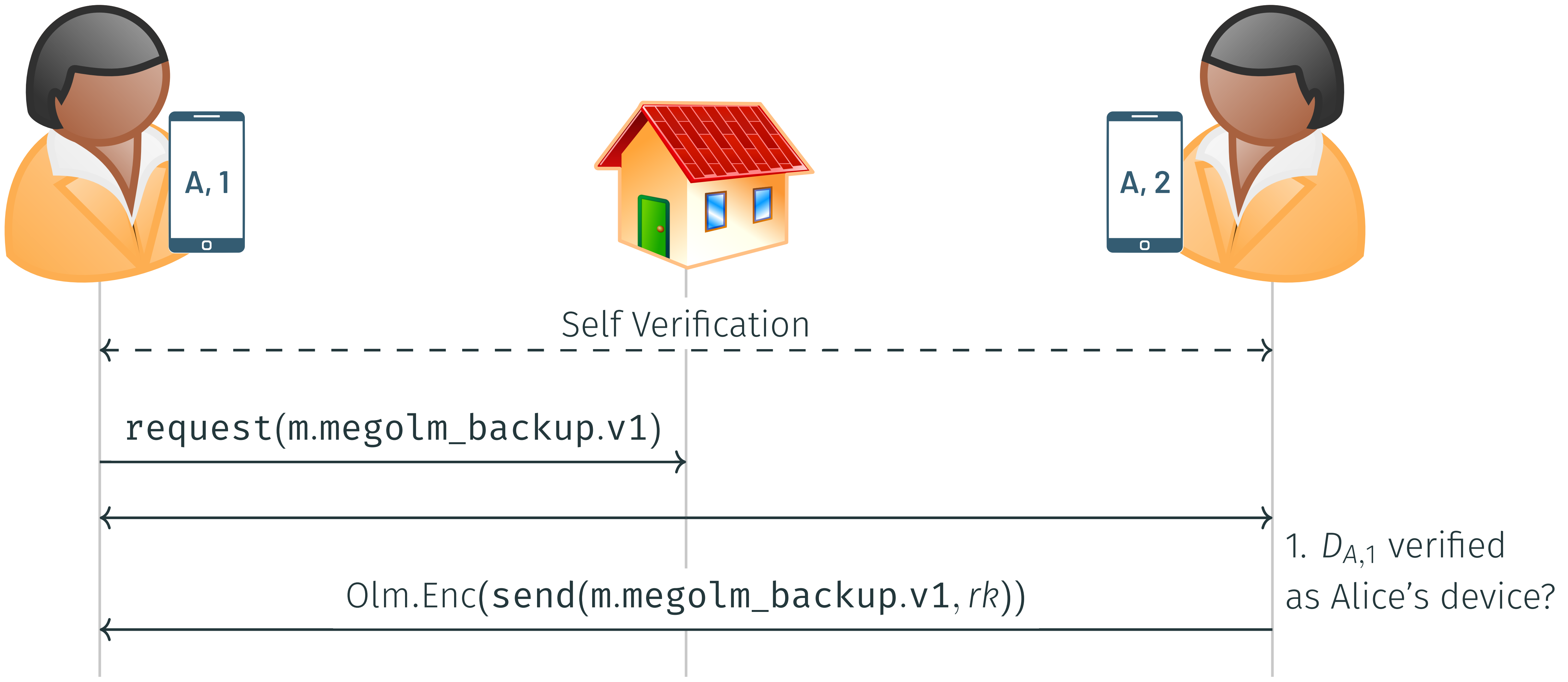
Secure Secret Sharing

Example: Sharing secret for Megolm Key Backups



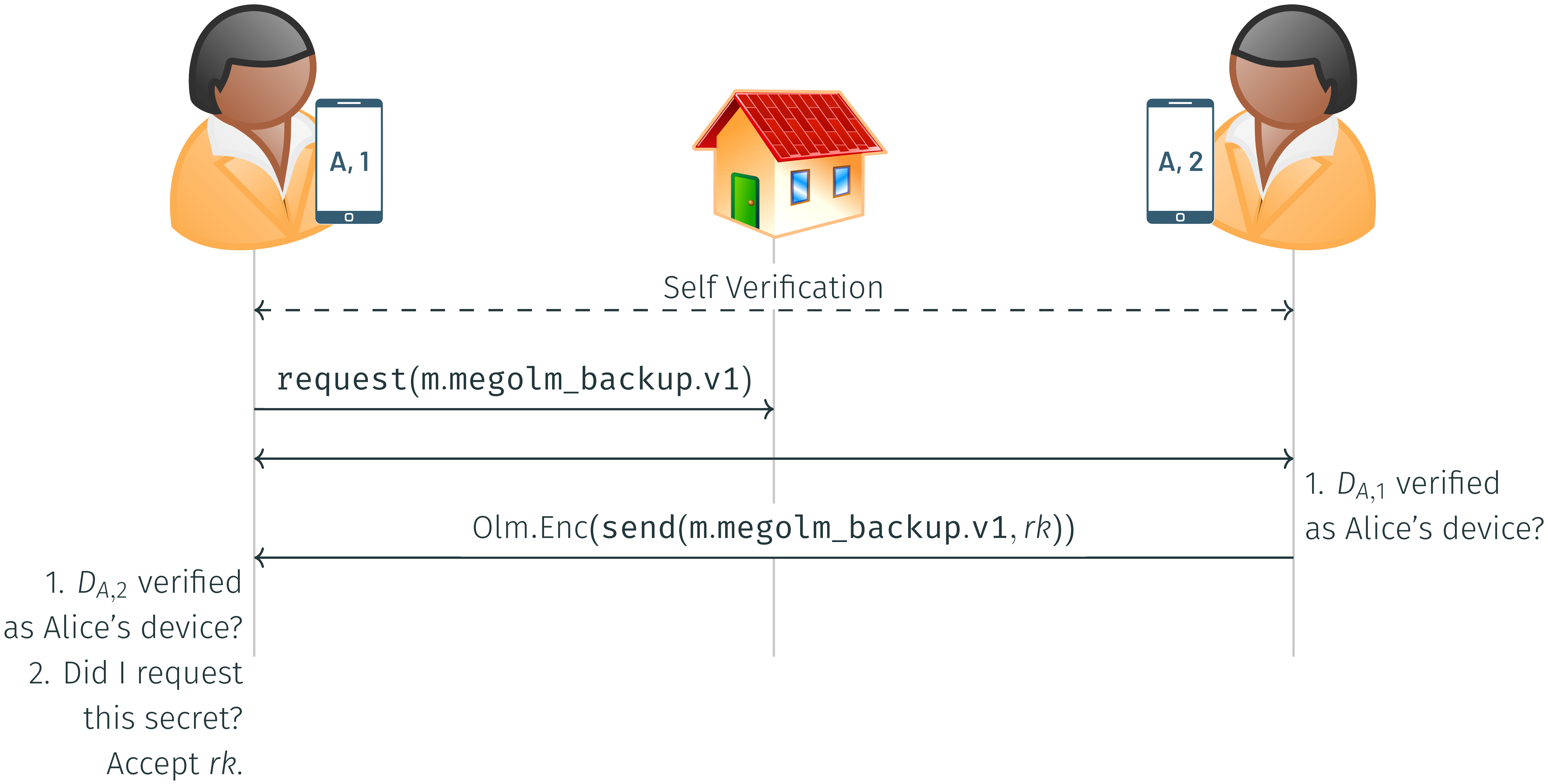
Secure Secret Sharing

Example: Sharing secret for Megolm Key Backups



Secure Secret Sharing

Example: Sharing secret for Megolm Key Backups



Adversary controlled backup keys

Since Secure Secret Sharing uses Olm to share secrets between devices (incl. Megolm backup keys)

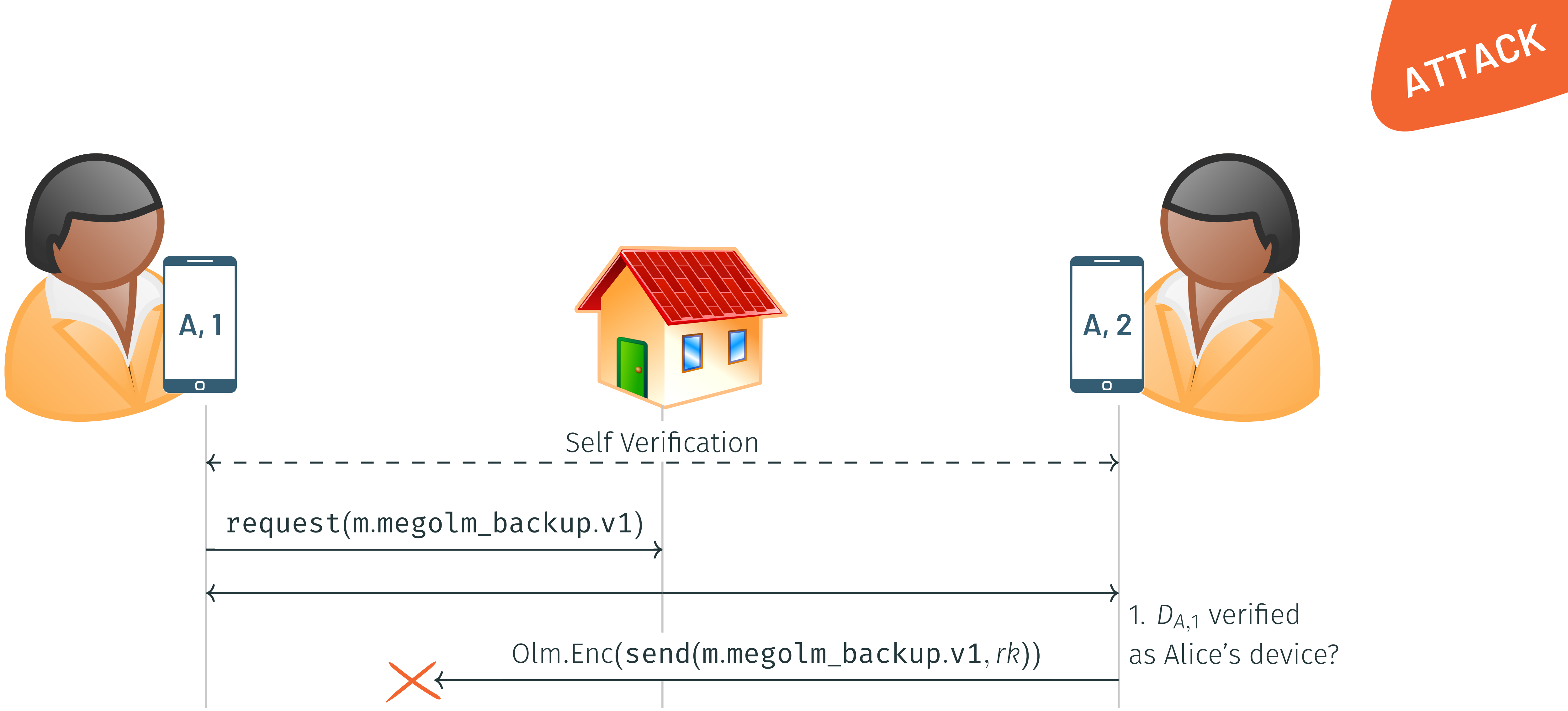
Can we pull off the same trick off again?

ATTACK

Adversary controlled backup keys

Since Secure Secret Sharing uses Olm to share secrets between devices (incl. Megolm backup keys)

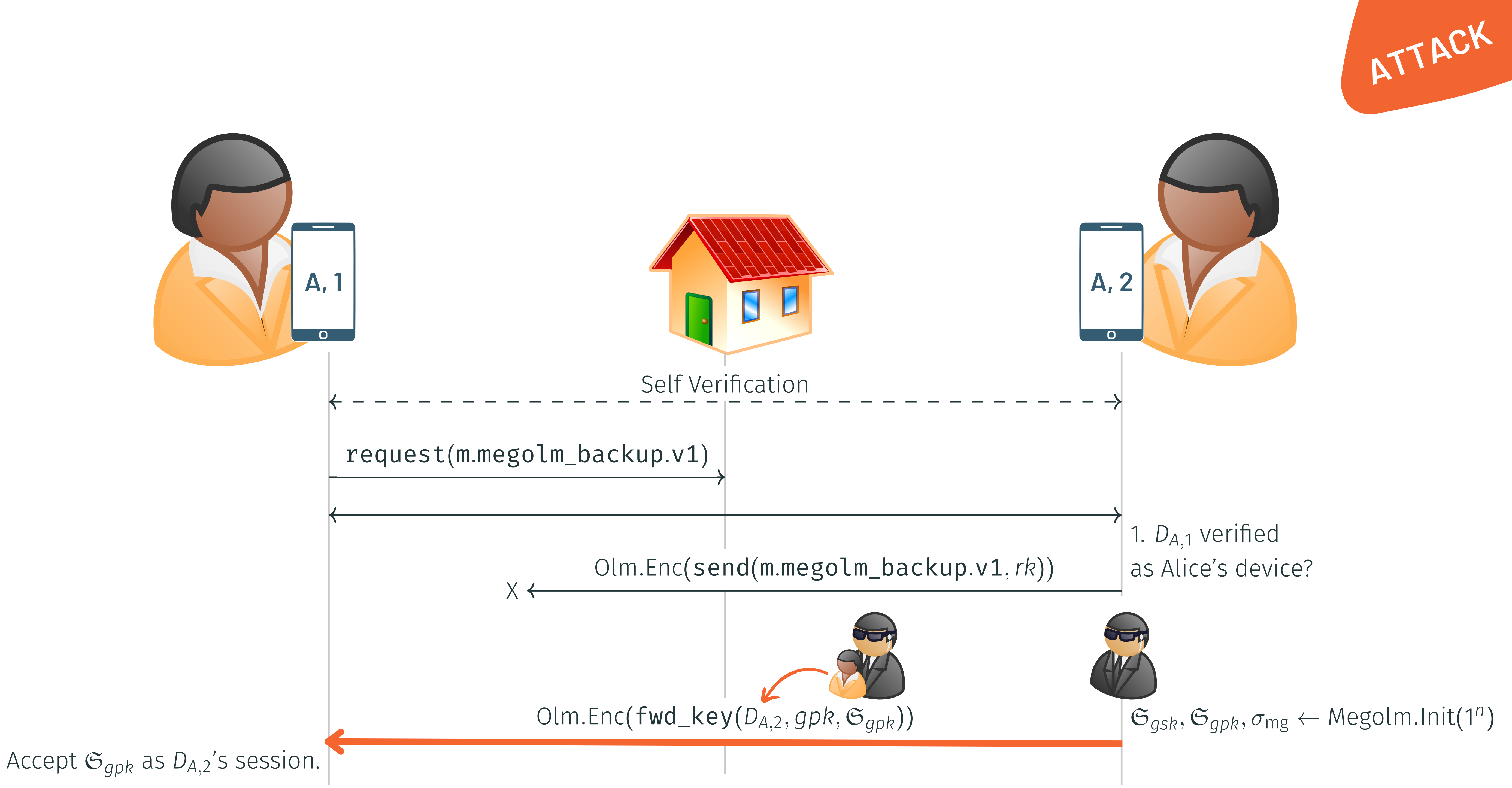
Can we pull off the same trick off again?



Adversary controlled backup keys

Since Secure Secret Sharing uses Olm to share secrets between devices (incl. Megolm backup keys)

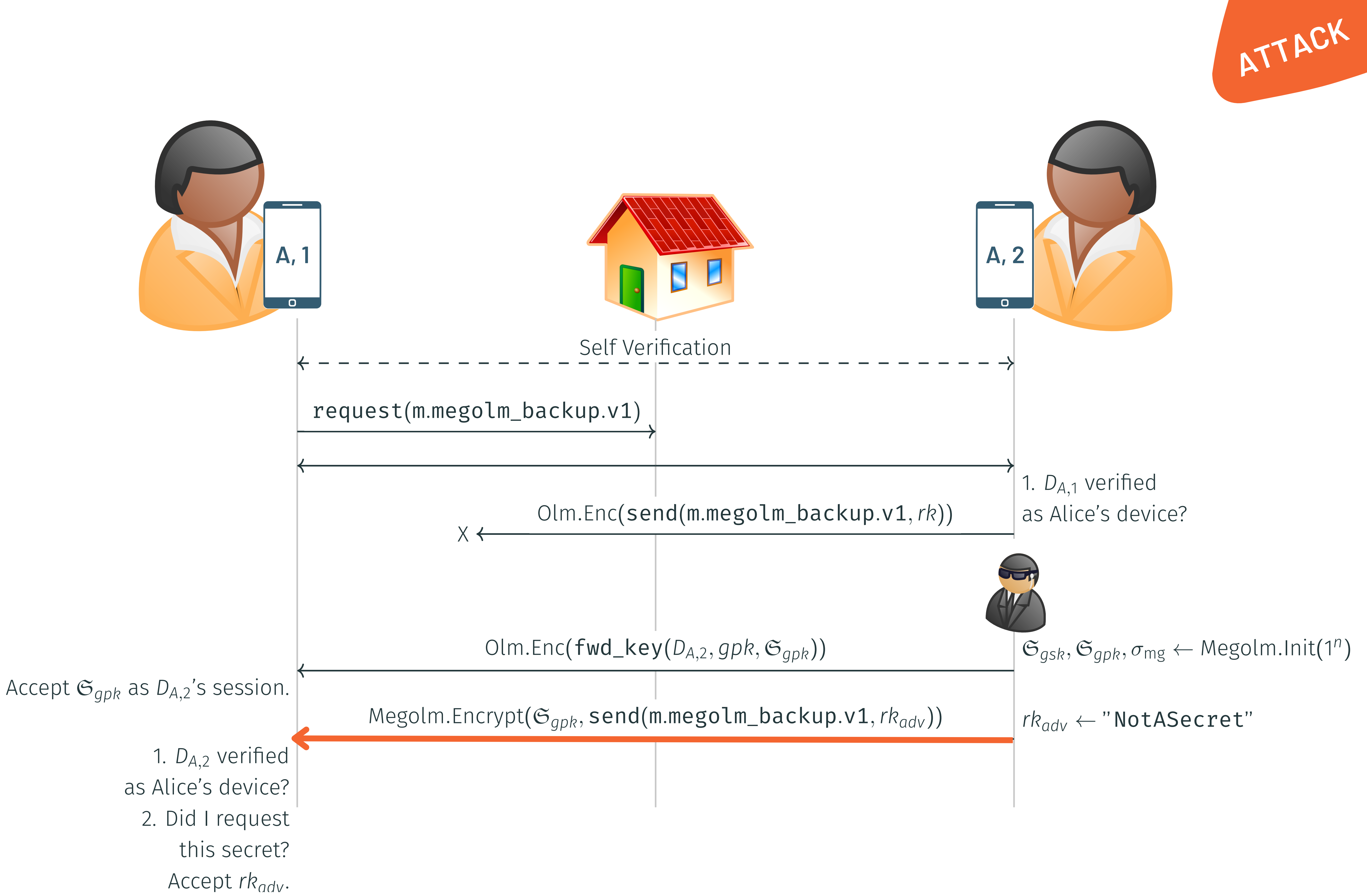
Can we pull off the same trick off again?



Adversary controlled backup keys

Since Secure Secret Sharing uses Olm to share secrets between devices (incl. Megolm backup keys)

Can we pull off the same trick off again?



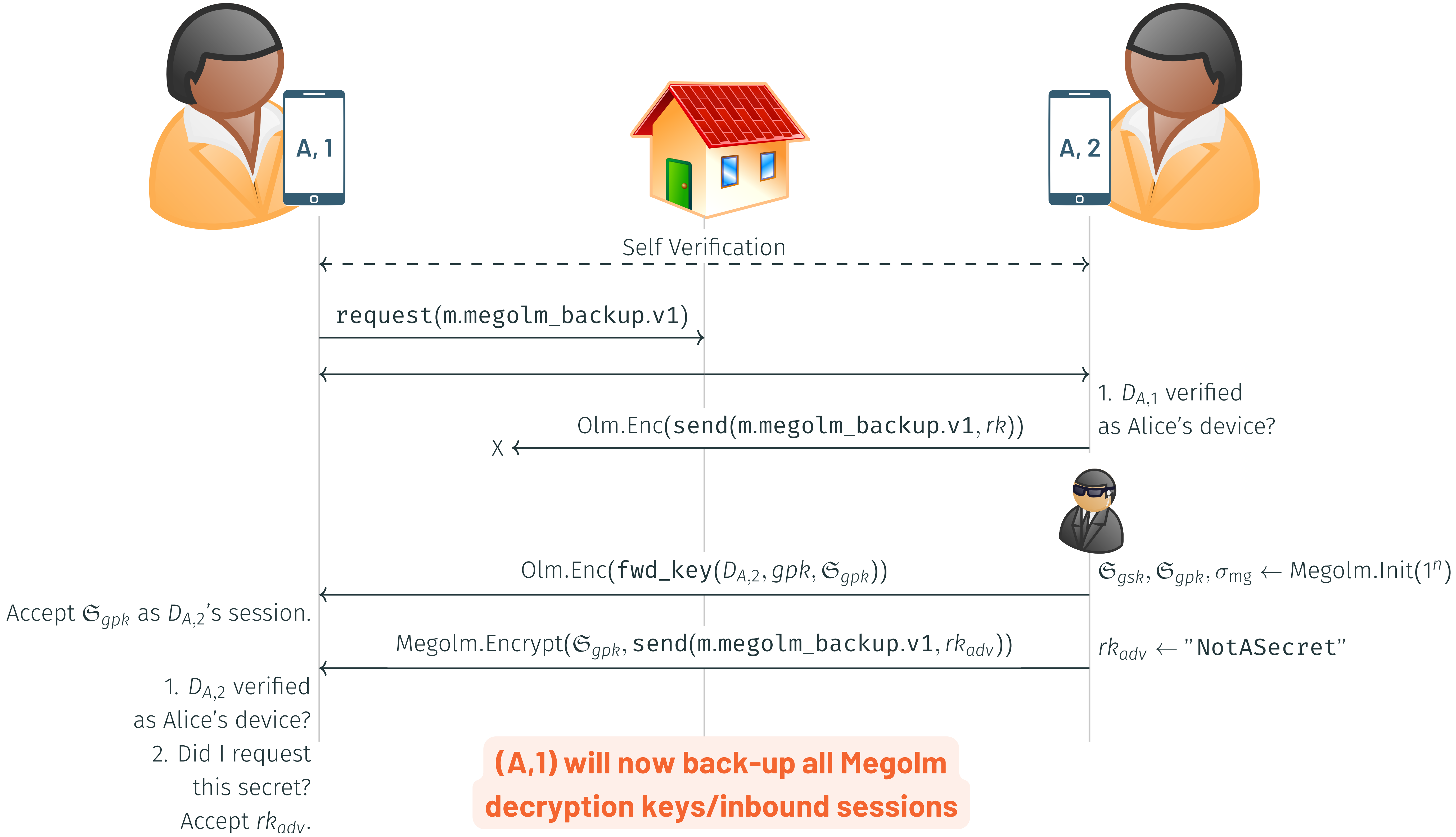
ATTACK

Adversary controlled backup keys

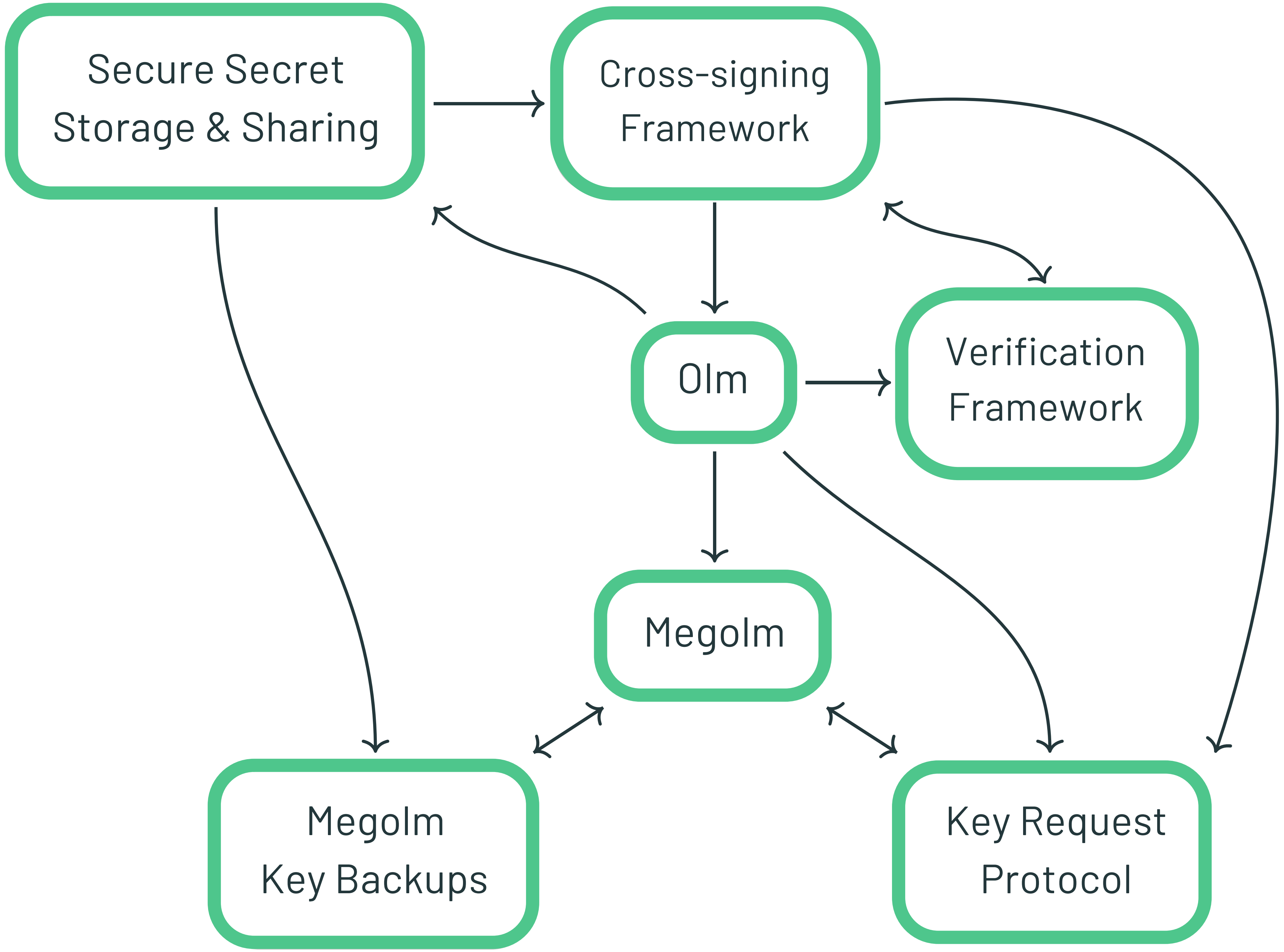
Since Secure Secret Sharing uses Olm to share secrets between devices (incl. Megolm backup keys)

Can we pull off the same trick off again? *Yes*

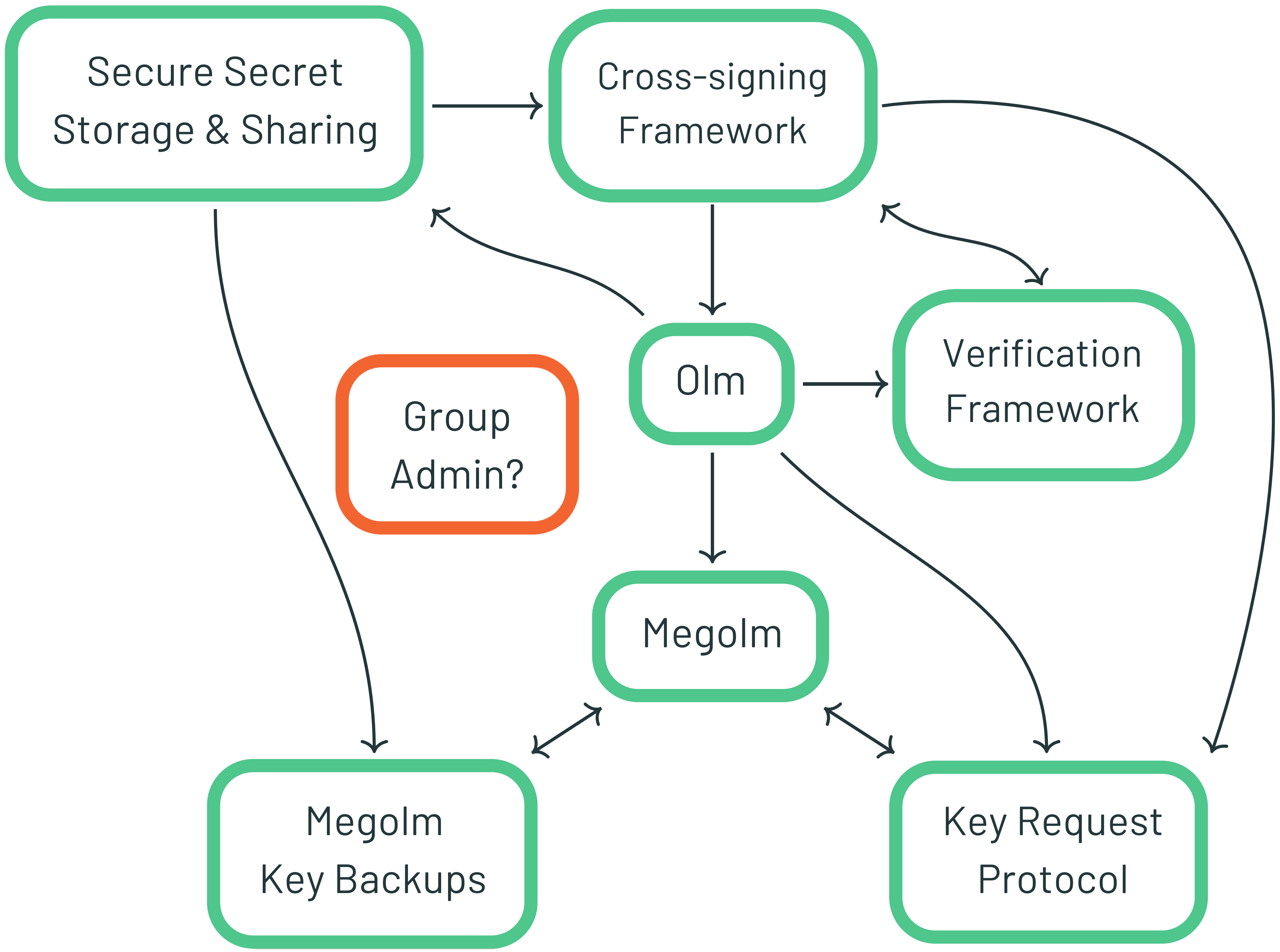
↪ Confidentiality break



Modelling Matrix & Finding Attacks

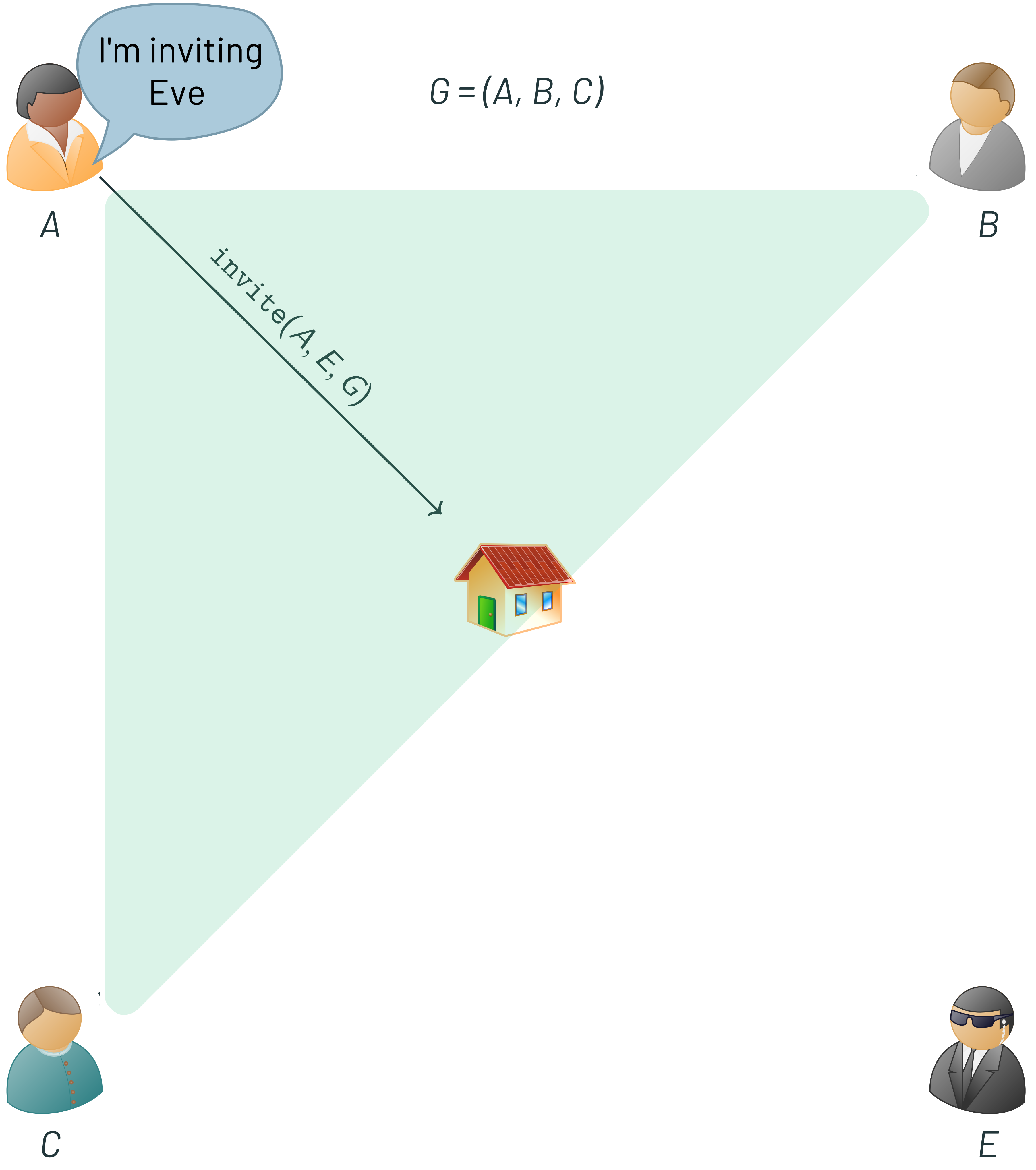


Group Administration



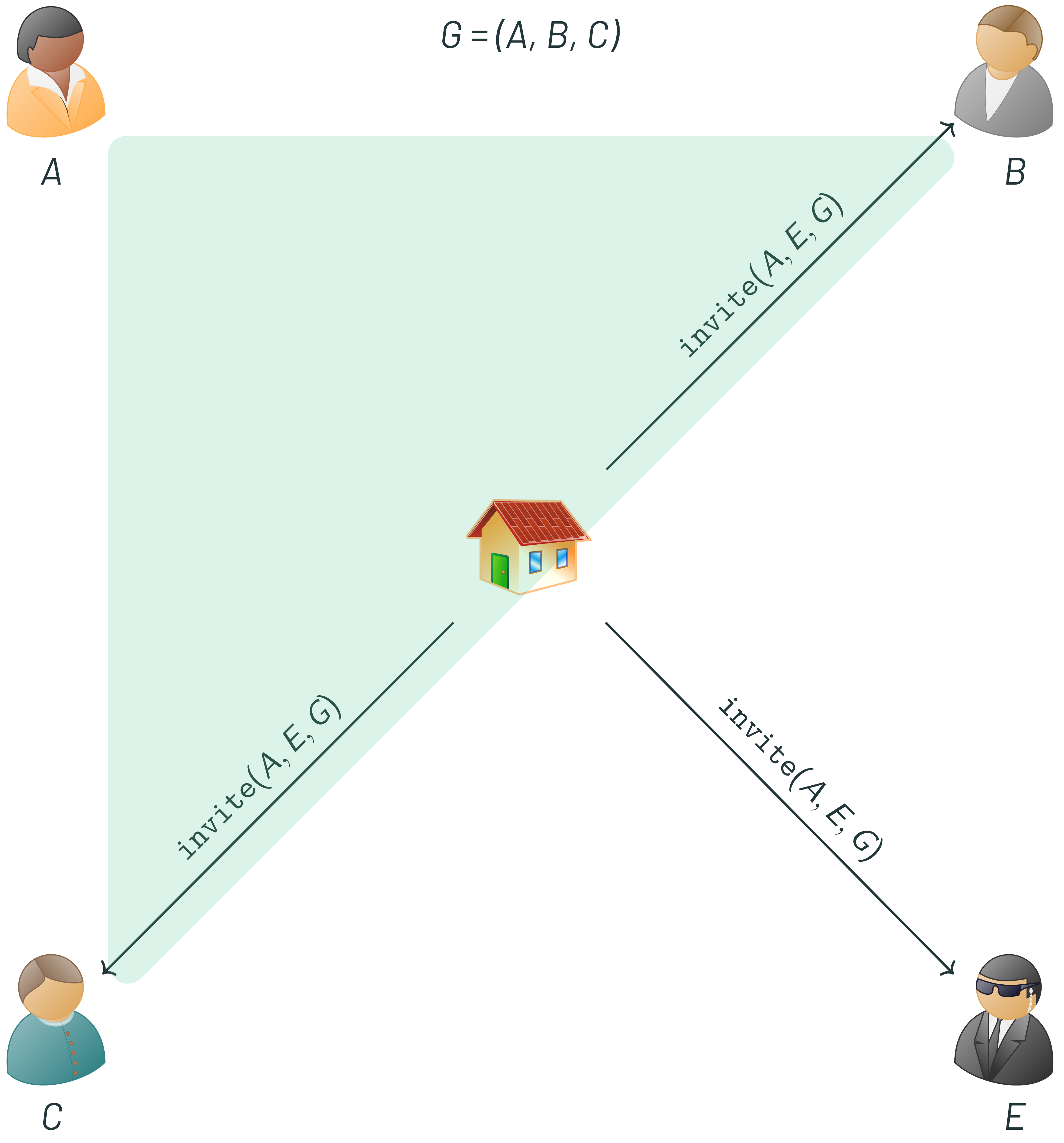
Group Administration

- Group membership is managed through events.



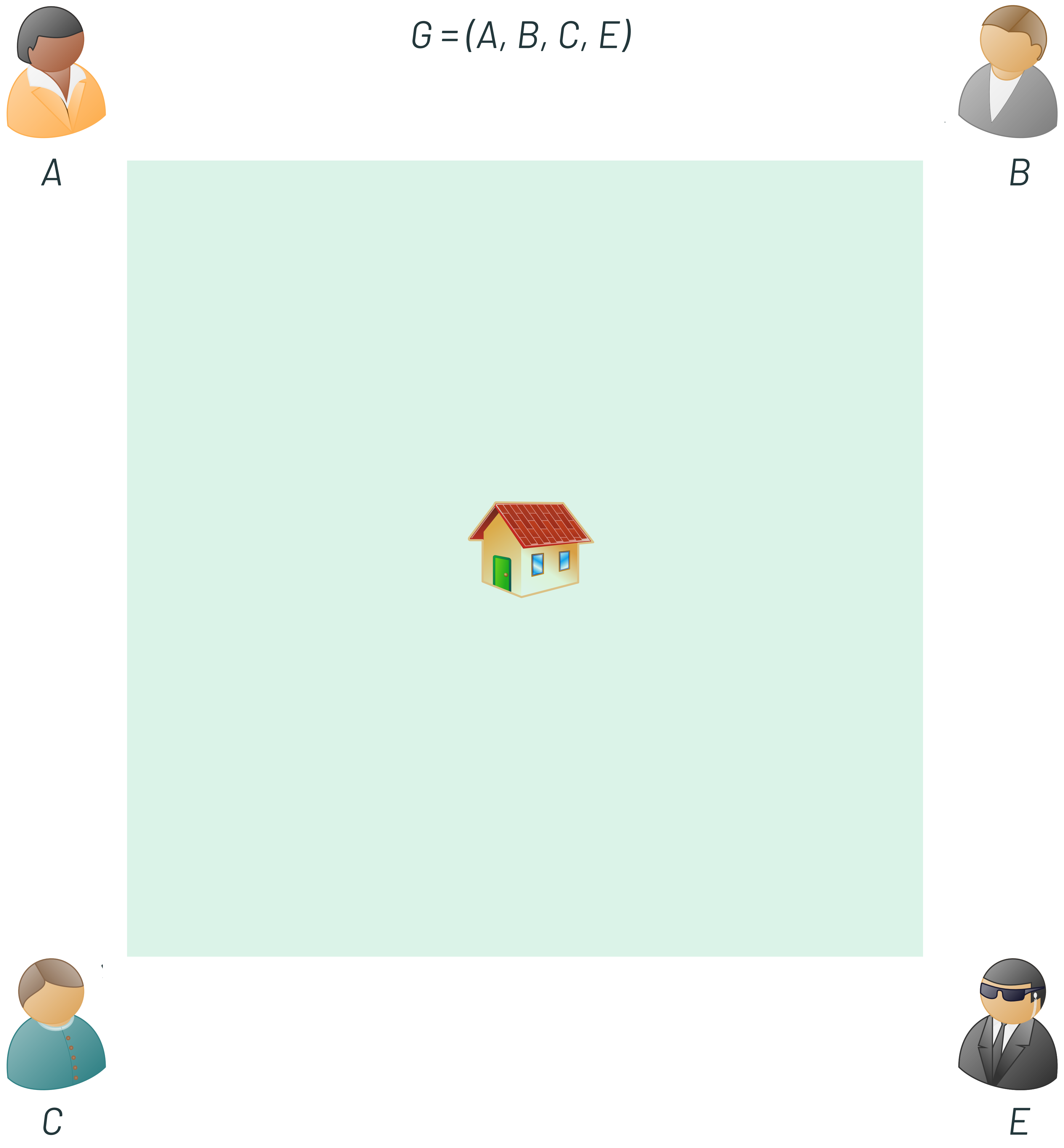
Group Administration

- Group membership is managed through events.
- Like messages in the room.



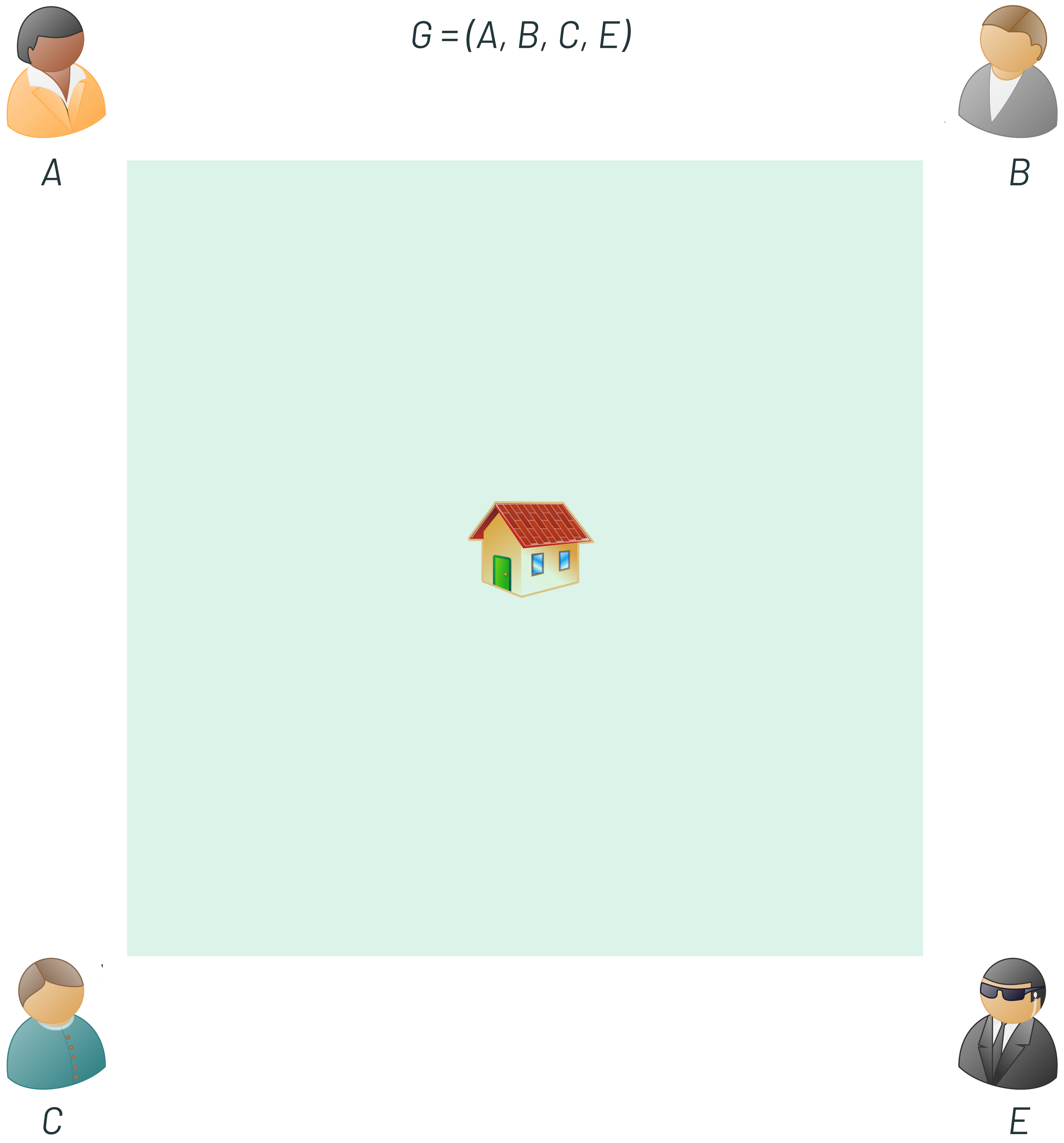
Group Administration

- Group membership is managed through events.
- Like messages in the room.



Group Administration

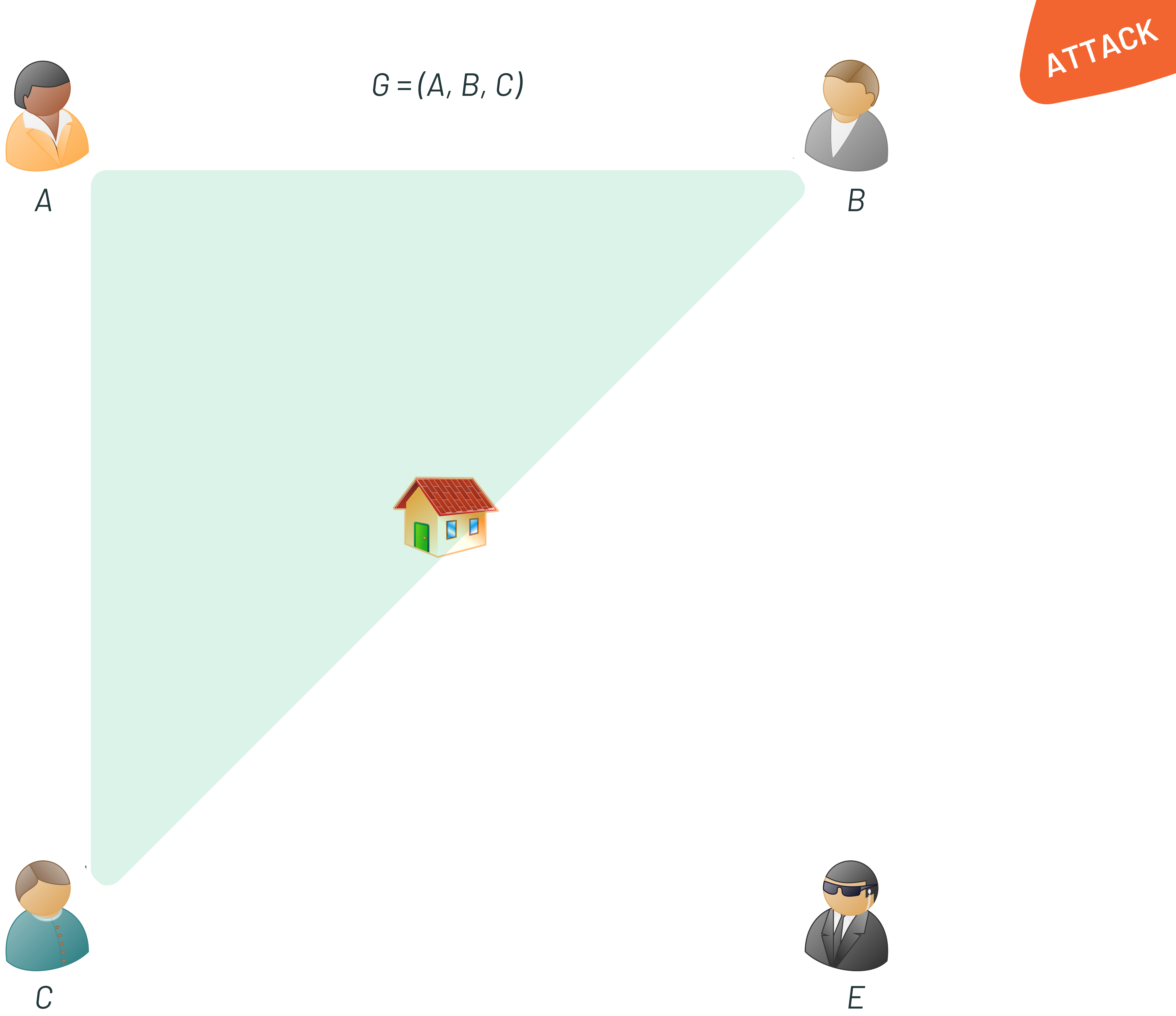
- Group membership is managed through events.
- Like messages in the room.
- These events are *unauthenticated*.



Server Control of Group Membership

Aim: add a server-controlled user to the group attack.

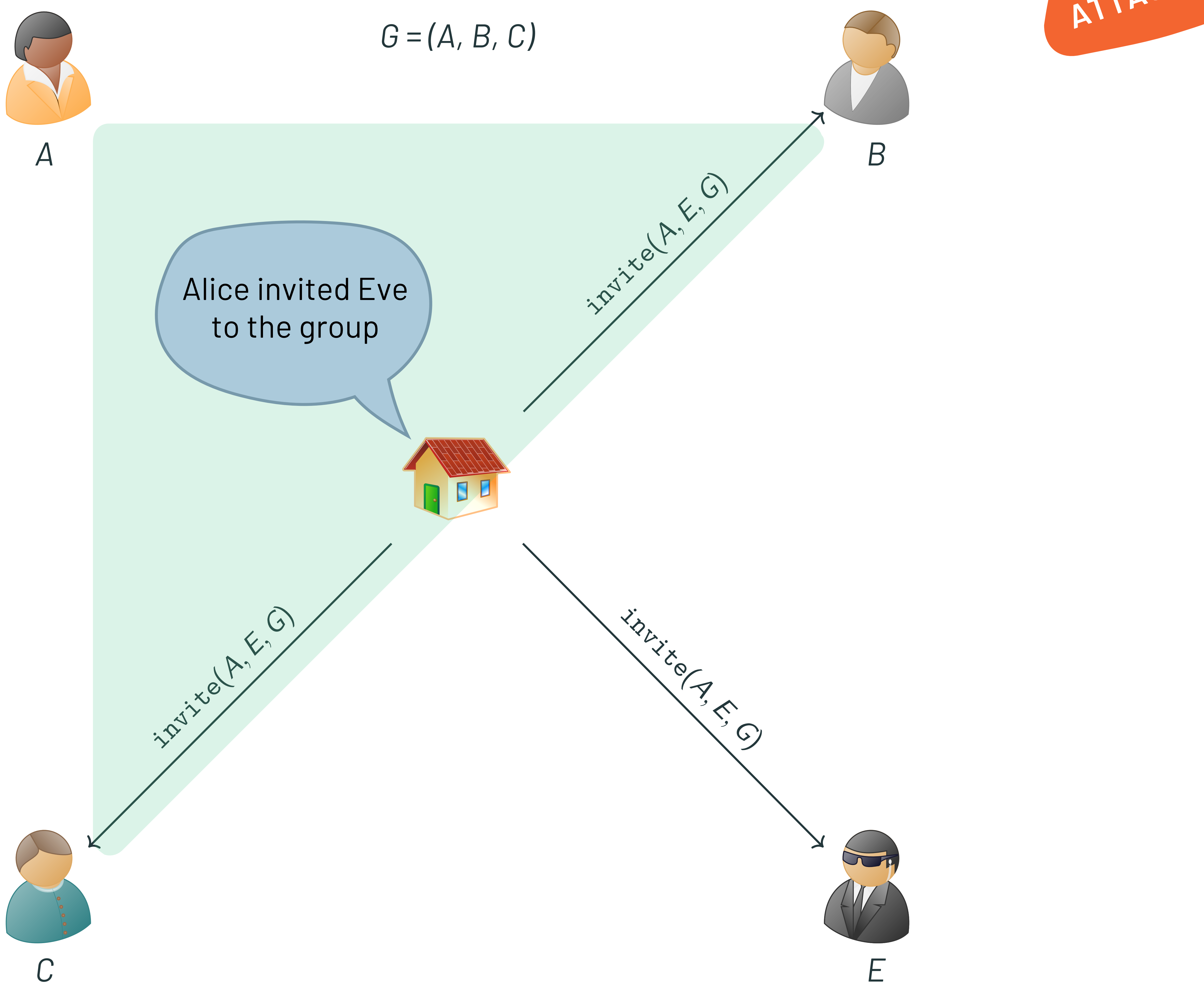
Can the server forge group invites?



Server Control of Group Membership

Aim: add a server-controlled user to the group attack.

Can the server forge group invites?

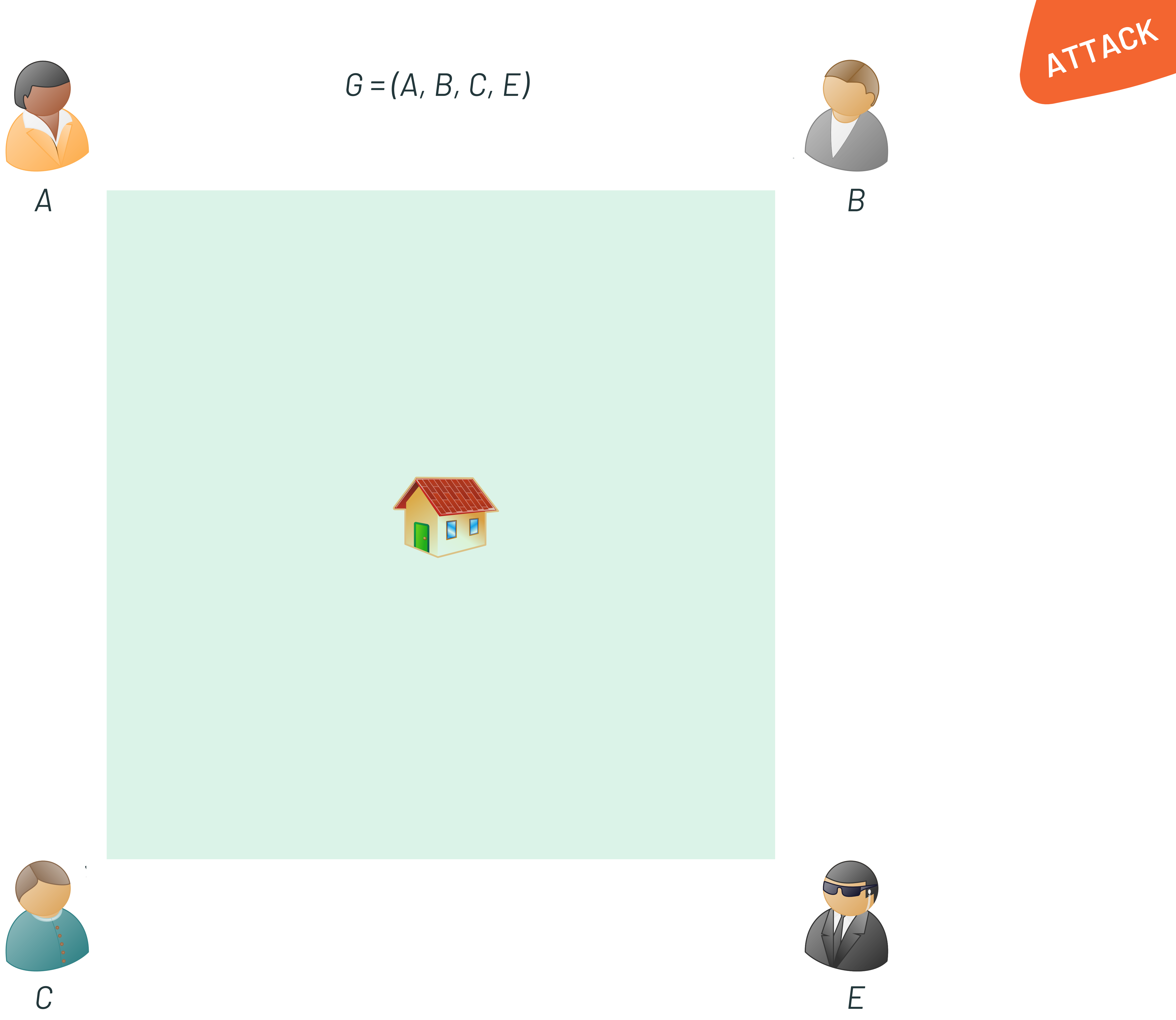


Server Control of Group Membership

Aim: add a server-controlled user to the group attack.

Can the server forge group invites? ***Yes***

Confidentiality break *



* detectable in the interface
* for future messages only

Server Control of Group Membership

Common issue among real-world group messaging protocols.

More is Less: both Signal and WhatsApp were vulnerable to a *burgle-into-the-group* attack.

2018 IEEE European Symposium on Security and Privacy

More is Less: On the End-to-End Security of Group Chats in Signal, WhatsApp, and Threema

Paul Rösler, Christian Mainka, Jörg Schwenk
{paul.roesler, christian.mainka, joerg.schwenk}@rub.de
Horst Görtz Institute for IT Security
Chair for Network and Data Security
Ruhr-University Bochum

***Abstract*—Secure instant messaging is utilized in two variants: one-to-one communication and group communication. While the first variant has received much attention lately (Frosch et al., EuroS&P16; Cohn-Gordon et al., EuroS&P17; Kobeissi et al., EuroS&P17), little is known about the cryptographic mechanisms and security guarantees of secure group communication in instant messaging.**

To approach an investigation of group instant messaging protocols, we first provide a comprehensive and realistic security model. This model combines security and reliability goals from various related literature to capture relevant properties for communication in dynamic groups. Thereby the definitions consider their satisfiability with respect to the instant delivery of messages. To show its applicability, we analyze three widely used real-world protocols: Signal, WhatsApp, and Threema. By applying our model, we reveal several shortcomings with respect to the security definition. Therefore we propose generic countermeasures to enhance the protocols regarding the required security and reliability goals. Our systematic analysis reveals that (1) the *communications' integrity* – represented by the integrity of all exchanged messages – and (2) the *groups' closeness* – represented by the members' ability of managing

a list of their members. Additionally, meta information is attached to groups, for example, a group title. Depending on the IM application and its underlying protocol, groups are administrated by selected users or all group members.

With the revelation of mass surveillance activities by intelligence agencies, new IM applications incorporating end-to-end encryption launched, as well es established IM applications added encryption to their protocols to protect the communication towards the message delivering servers. Hence analyses, investigating these protocols, also include malicious server-based attacks [3, 7].

In contrast to open standardized communication protocols like Extensible Messaging and Presence Protocol (XMPP) or Internet Relay Chat (IRC), most IM protocols are centralized such that users of each application can only communicate among one another. As a result, a user cannot choose the most trustworthy provider but needs to fully trust the one provider that develops both, protocol and application.

End-to-end encryption is the major security feature of secure instant messaging protocols for protecting the protocol security when considering malicious server-based attacks.

Server Control of Group Membership

Common issue among real-world group messaging protocols.

Matrix has a fix in progress!

Attacks Summary

1

COMPLETE BREAK

User/Device Confusion in Out-of-Band Verification

2

AUTHENTICATION BREAK *

Impersonation through Key Sharing

3

CONFIDENTIALITY BREAK

Adversary Controlled Megolm Backup Key

4

AUTHENTICATION BREAK

Protocol Confusion

5

TRIVIAL CONFIDENTIALITY BREAK *

Server Control of Group Membership

6

(THEORETICAL)

IND-CCA BREAK

* Detectable in the user interface.

Attacks Summary

1

COMPLETE BREAK

User/Device Confusion in Out-of-Band Verification

2

AUTHENTICATION BREAK *

Impersonation through Key Sharing

6

(THEORETICAL)

IND-CCA BREAK

3

CONFIDENTIALITY BREAK

Adversary Controlled Megolm Backup Key

4

AUTHENTICATION BREAK

Protocol Confusion

5

TRIVIAL CONFIDENTIALITY BREAK *

Server Control of Group Membership

* Detectable in the user interface.

Attacks Summary

1

COMPLETE BREAK

User/Device Confusion in Out-of-Band Verification

2

AUTHENTICATION BREAK *

Impersonation through Key Sharing

6

(THEORETICAL)

IND-CCA BREAK

3

CONFIDENTIALITY BREAK

Adversary Controlled Megolm Backup Key

4

AUTHENTICATION BREAK

Protocol Confusion

5

TRIVIAL CONFIDENTIALITY BREAK *

Server Control of Group Membership

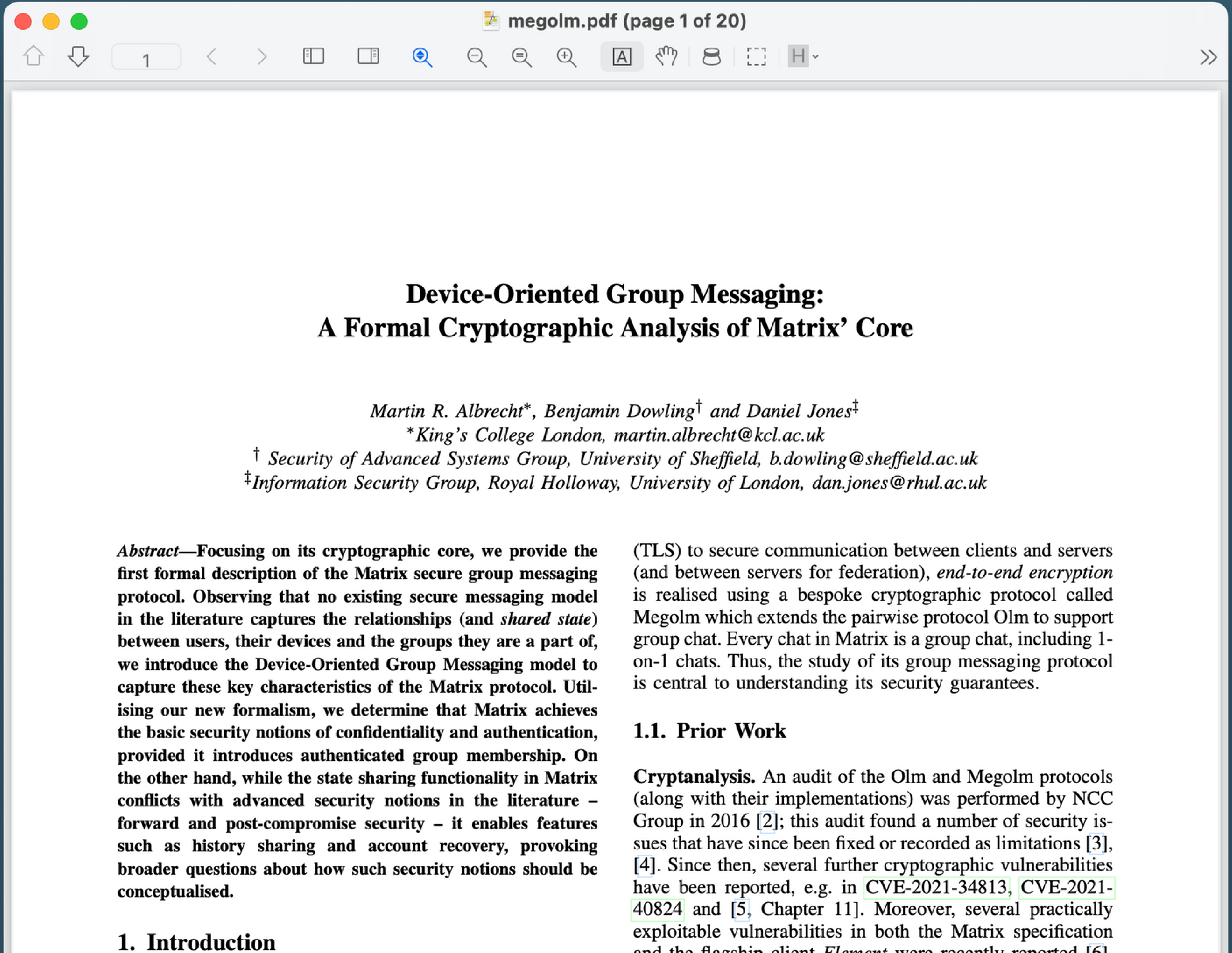
WIP

* Detectable in the user interface.

Modelling Matrix

Completed formalisation of Matrix' cryptographic core.

Security analysis and proof focuses on subset aiming to capture how the security of messages is affected by device-to-device interactions and state sharing.



megolm.pdf (page 1 of 20)

Device-Oriented Group Messaging: A Formal Cryptographic Analysis of Matrix' Core

Martin R. Albrecht, Benjamin Dowling† and Daniel Jones‡*

**King's College London, martin.albrecht@kcl.ac.uk*

† Security of Advanced Systems Group, University of Sheffield, b.dowling@sheffield.ac.uk

‡Information Security Group, Royal Holloway, University of London, dan.jones@rhul.ac.uk

***Abstract*—Focusing on its cryptographic core, we provide the first formal description of the Matrix secure group messaging protocol. Observing that no existing secure messaging model in the literature captures the relationships (and *shared state*) between users, their devices and the groups they are a part of, we introduce the Device-Oriented Group Messaging model to capture these key characteristics of the Matrix protocol. Utilising our new formalism, we determine that Matrix achieves the basic security notions of confidentiality and authentication, provided it introduces authenticated group membership. On the other hand, while the state sharing functionality in Matrix conflicts with advanced security notions in the literature – forward and post-compromise security – it enables features such as history sharing and account recovery, provoking broader questions about how such security notions should be conceptualised.**

1. Introduction

(TLS) to secure communication between clients and servers (and between servers for federation), *end-to-end encryption* is realised using a bespoke cryptographic protocol called Megolm which extends the pairwise protocol Olm to support group chat. Every chat in Matrix is a group chat, including 1-on-1 chats. Thus, the study of its group messaging protocol is central to understanding its security guarantees.

1.1. Prior Work

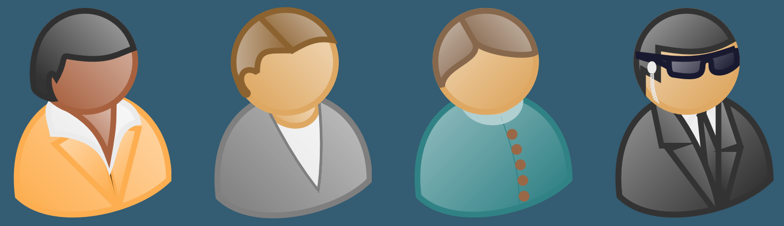
Cryptanalysis. An audit of the Olm and Megolm protocols (along with their implementations) was performed by NCC Group in 2016 [2]; this audit found a number of security issues that have since been fixed or recorded as limitations [3], [4]. Since then, several further cryptographic vulnerabilities have been reported, e.g. in CVE-2021-34813, CVE-2021-40824 and [5, Chapter 11]. Moreover, several practically exploitable vulnerabilities in both the Matrix specification

Interested?

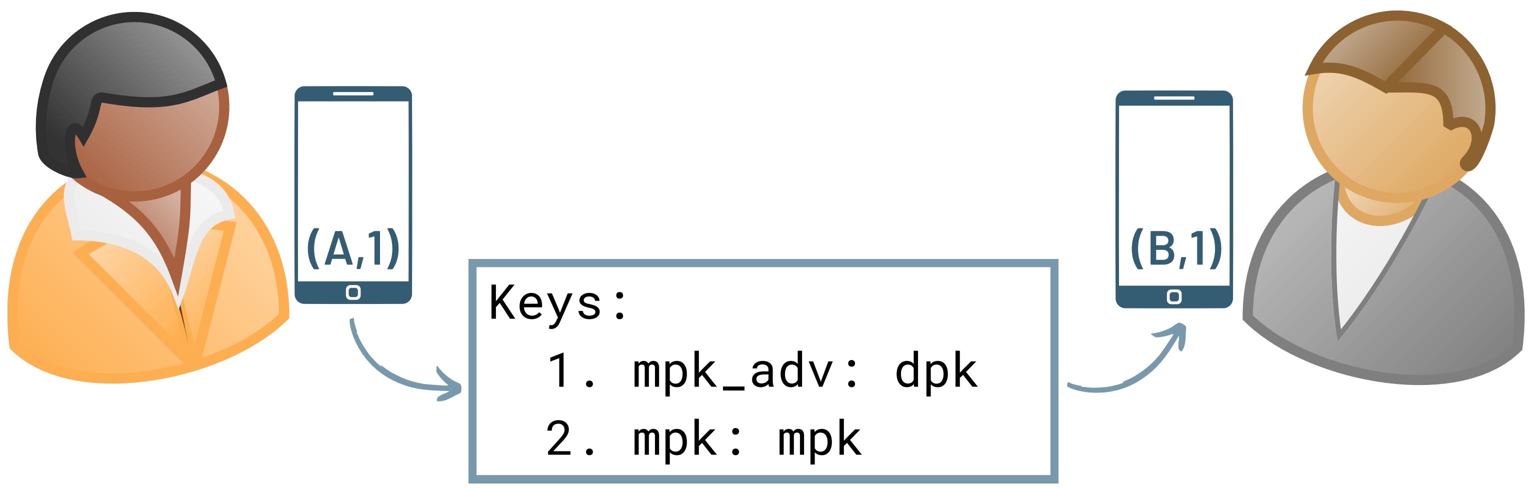
Find our paper at https://ia.cr/2023/485.

Keep an eye out for the follow-up modelling paper!

Image Credits

- Home from Nuvola Icon set by David Vignoni (LGPL) 
- tikzpeople package by Nils Fleischhacker 
- Matrix screenshots from Matrix.org
- Element screeenshots from element.io

SAS Attack



1. $\mathsf{SAS.SendMAC}(A, D_{A,i}, B, D_{B,j}, mpk, dpk, k, t)$

$c \leftarrow$ `"MATRIX_KEY_VERIFICATION_MAC"` $\| A \| D_{A,i} \| B \| D_{B,j} \| t$

$id_{\text{dev}} \leftarrow$ `"ed25519:"` $\| D_{A,i}$

$mac_{\text{dev}} \leftarrow \mathsf{SAS.CalcMAC}(k, dpk, c \| id_{\text{dev}})$

$id_{\text{cs}} \leftarrow$ `"ed25519:"` $\| mpk$

$mac_{\text{cs}} \leftarrow \mathsf{SAS.CalcMAC}(k, mpk, c \| id_{\text{cs}})$

$ms \leftarrow ((id_{\text{dev}}, mac_{\text{dev}}), (id_{\text{cs}}, mac_{\text{cs}}))$

$ks \leftarrow \mathsf{SAS.CalcMAC}(k, \mathsf{sort}(id_{\text{dev}}, id_{\text{cs}}), c \|$ `"KEY_IDS"`$)$

return (ms, ks)

2. $\mathsf{SAS.VerifyMAC}(A, D_{A,i}, B, D_{B,j}, mac, k, t)$

$(id_{\text{dev}}, mac_{\text{dev}}), (id_{\text{cs}}, mac_{\text{cs}}), ks \leftarrow mac$

$c \leftarrow$ `"MATRIX_KEY_VERIFICATION_MAC"` $\| B \| D_{B,j} \| A \| D_{A,i} \| t$

$ks' \leftarrow \mathsf{SAS.CalcMAC}(k, \mathsf{sort}(id_{\text{dev}}, id_{\text{cs}}), c \|$ `"KEY_IDS"`$)$

assert $ks' = ks$

$v \leftarrow \emptyset$

for (id, mac) **in** $((id_{\text{dev}}, mac_{\text{dev}}), (id_{\text{cs}}, mac_{\text{cs}}))$

  `"ed25519:"` $\| D_{B,j} \leftarrow id$

  // Check if this is a device verification request

  $dpk \leftarrow \mathsf{HS.QueryKey}($`"dpk"`$, B, D_{B,j})$

  if $dpk \neq \perp$ **then**

    $D \leftarrow x$

    if $mac = \mathsf{SAS.CalcMAC}(k, dpk, c \| id)$ **then**

      $v \leftarrow v \cup \{(B, D)\}$

  // Check if this is a cross-signing verification request

  elseif $(x = \mathsf{HS.QueryKey}($`"mpk"`$, B)$

      $\cap\ mac = \mathsf{SAS.CalcMAC}(k, x, c \| id))$ **then**

    $mpk \leftarrow x$

    $v \leftarrow v \cup \{(B, mpk)\}$

return v

3. $\mathsf{SAS.SignDevice}(A, D_{A,i})$

// Check whether $D_{A,i}$ is a cross-signing identity

$mpk \leftarrow \mathsf{HS.QueryKey}($"`mpk`"$, A)$

if $D_{A,i} = mpk$ **then**

  return $\mathsf{UserVerified}(A, mpk)$

// Otherwise, $D_{A,i}$ refers to a device

else

  return $\mathsf{DeviceVerified}(A, D_{A,i})$