Where Are My Keys?

A Survey of Matrix Cryptographic Key Infrastructure

Sumner Evans

21 September 2024

Beeper (Automattic)

Where Are My Keys?

A Survey of Matrix Cryptographic Key Infrastructure

Sumner Evans

21 September 2024

Beeper (Automattic)

1. Hello, my name is Sumner, I'm a software engineer at Automattic working on Beeper and today I'm going to be talking about cryptographic key infrastructure in Matrix.
2. End-to-end encryption is one of the things which brought me to Matrix, and I'm sure that it's one of the factors that brought many of you to Matrix as well.
3. However, Matrix's user experience with cryptography is often confusing. I mainly blame the other chat networks for their incompetence. Most other chat networks don't even provide any cryptographically-guaranteed security and privacy. Some networks provide encryption in a way that does not truly leave the user in control of their keys. Only a few networks (Signal) truly leave the user in control, and their UX is arguably worse than Matrix.
4. In this talk, my goal is to discuss the cryptographic key *infrastructure* in Matrix. What do I mean by "infrastructure"? I mean all of the features which support key sharing and identity verification, but don't actually themselves provide security. You can think of this talk as discussing the "UX layer of cryptography in Matrix". None of the things that I'm going to discuss are strictly necessary for ensuring secure E2EE communication, but without them, Matrix' UX would be horrible.

Why Cryptography?

Matrix uses cryptography for two main purposes:

1. **Message Security** — only the people who are part of the conversation should be allowed to view messages of the conversation.
2. **Identity** — verifying that a user or device is who they say they are.

1. As an additional benefit of how Matrix achieves this, encrypted messages cannot be tampered with by a man-in-the-middle actor without the receiving party knowing.
2. Note that one of the most important parts of identity this is verifying that our own devices are under our own control and we should allow our own clients to share keys to it.

Why Cryptography?

Matrix uses cryptography for two main purposes:

1. **Message Security** — only the people who are part of the conversation should be allowed to view messages of the conversation.
2. **Identity** — verifying that a user or device is who they say they are.

1. As an additional benefit of how Matrix achieves this, encrypted messages cannot be tampered with by a man-in-the-middle actor without the receiving party knowing.
2. Note that one of the most important parts of identity this is verifying that our own devices are under our own control and we should allow our own clients to share keys to it.

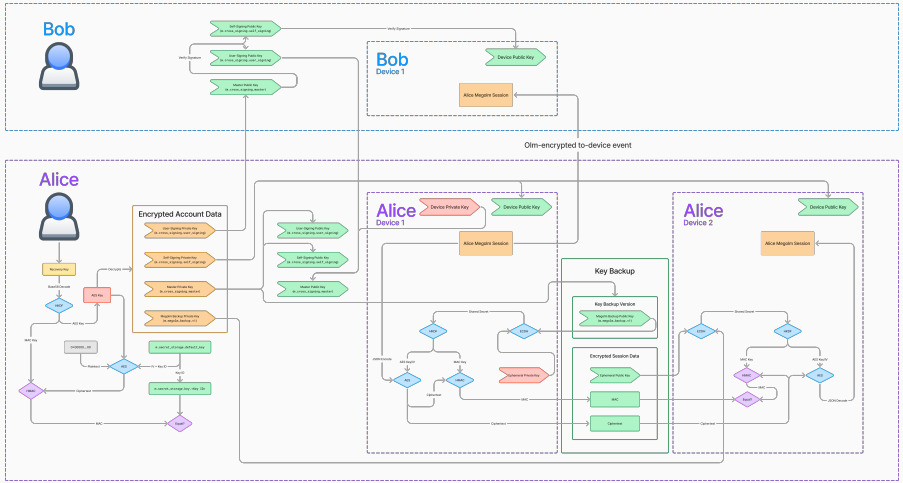
Why Cryptography?

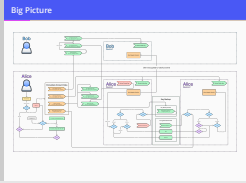
Matrix uses cryptography for two main purposes:

1. **Message Security** — only the people who are part of the conversation should be allowed to view messages of the conversation.
2. **Identity** — verifying that a user or device is who they say they are.

1. As an additional benefit of how Matrix achieves this, encrypted messages cannot be tampered with by a man-in-the-middle actor without the receiving party knowing.
2. Note that one of the most important parts of identity this is verifying that our own devices are under our own control and we should allow our own clients to share keys to it.

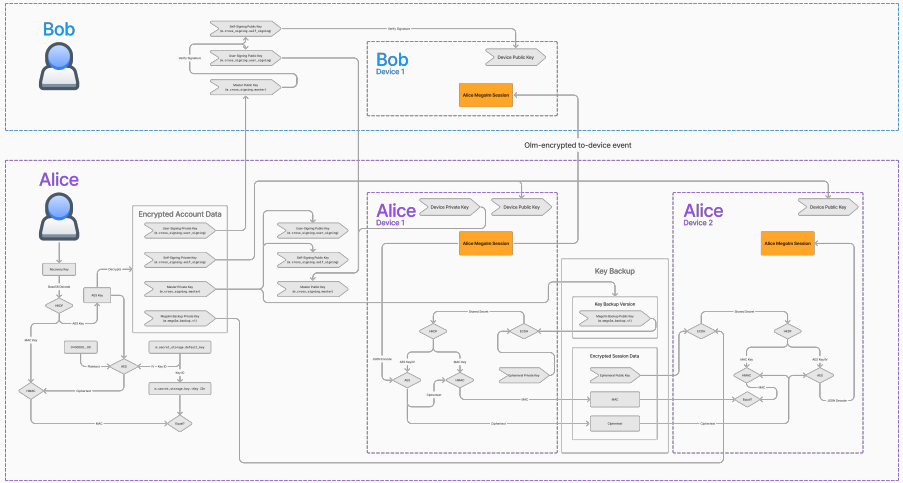
Big Picture

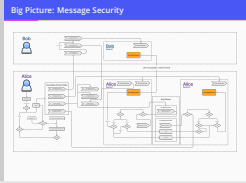




1. This diagram represents all of the infrastructure in Matrix for providing those core features of message security and identity verification.
2. I know, it's pretty overwhelming. But don't worry, we are going to go step-by-step through this, at the end of the talk you should have an understanding of what each part of this diagram means.
3. Let's start by orienting ourselves to the big picture of this diagram, then we will take a short detour into a few core cryptography concepts required to understand the diagram, and then we will break down the diagram into manageable pieces. And at the end of the talk hopefully you have a complete understanding of Matrix cryptographic key infrastructure.
4. You can see that there are two users represented in the diagram: Bob on the top and Alice on the bottom. The diagram is focused on how the Megolm session created by Alice Device 1 is shared to Bob and to Alice's Device 2.
5. The diagram is color-coded.
    - Red nodes represent data that does not leave the device.
    - Green nodes represent data is public and can be shared with the server and other users.
    - Orange nodes represent data that can be shared with trusted parties, or with members of the same Matrix room.
    - Blue and purple nodes represent cryptographic operations.

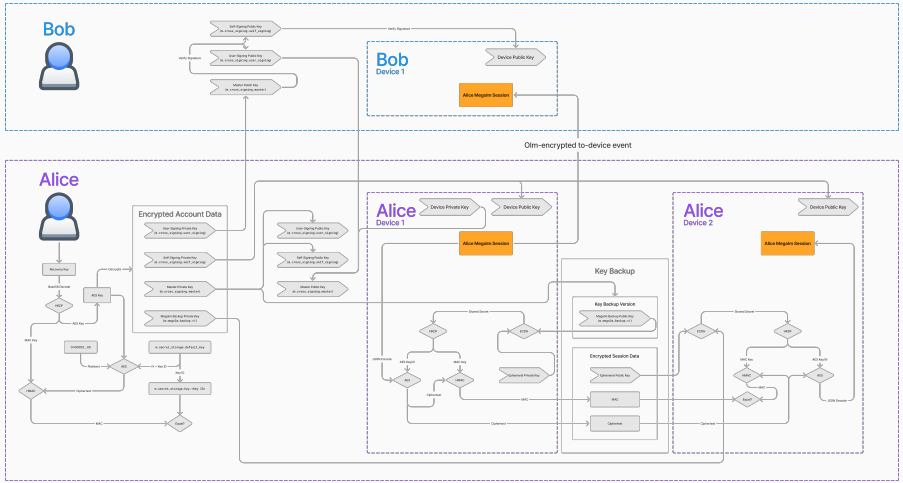
Big Picture: Message Security

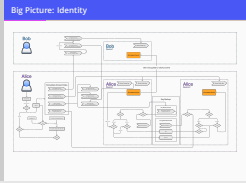




1. It's important that we don't loose sight of the reason for all of this infrastructure. In orange, we have the core of Matrix security: the Megolm session.
2. We aren't going to discuss this in detail today. I wrote an article about Megolm which you can find on my blog if you want to learn more. I'll provide a link at the end of the talk.
3. But for now, the only thing you need to know about it is that it provides end-to-end encryption for messages and needs to be shared with devices that Alice wants to be able to read her messages. So it needs to be shared to
   - the devices of other users in the same Matrix room (in this case Bob)
   - but her device also needs to share the session with her other devices.
4. All of the rest of the infrastructure in this diagram is to facilitate transferring that Megolm session, or verifying that a device should in fact have access to that Megolm session.

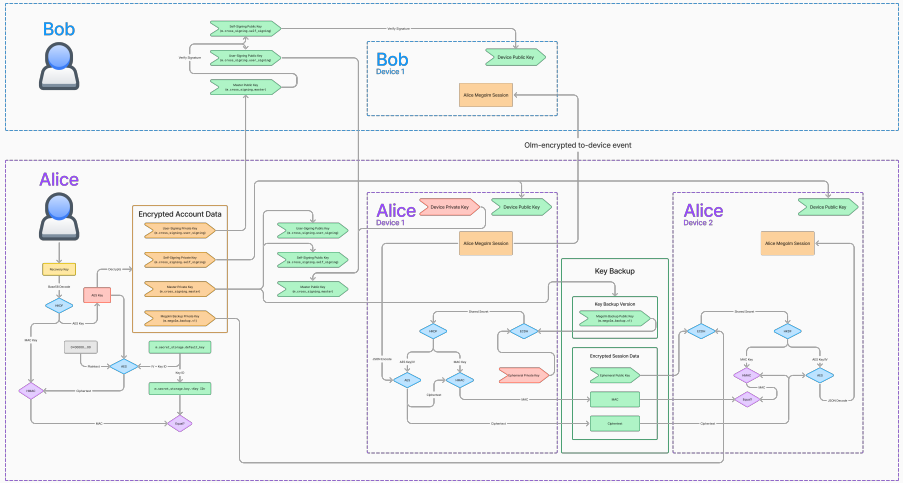
Big Picture: Identity

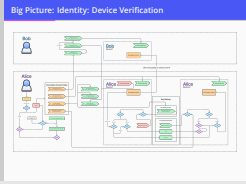




1. Let's move on to identity. The highlighted parts of the diagram provide a cryptographic way to verify that a device belongs to a particular user.
2. There are actually two pieces here...

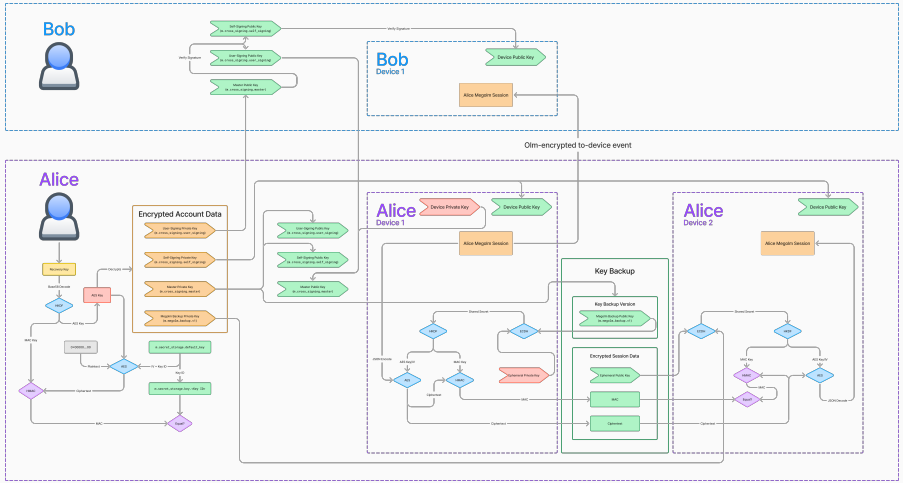
Big Picture: Identity: Device Verification

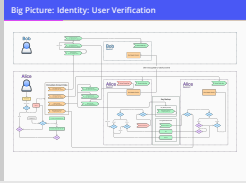




1. And here we have the infrastructure necessary for determining if we trust another device for our own user.

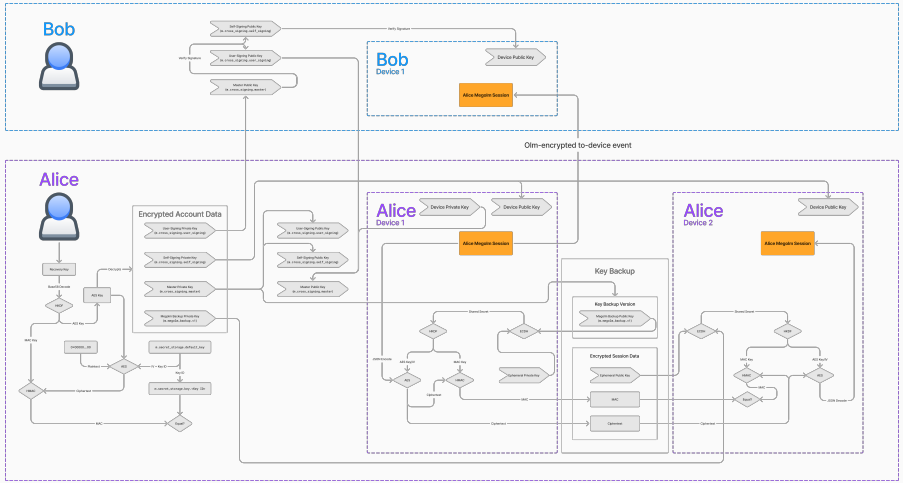
Big Picture: Identity: User Verification

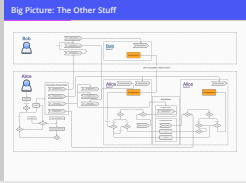




1. And here we have the infrastructure necessary for determining if we trust another user and their devices.

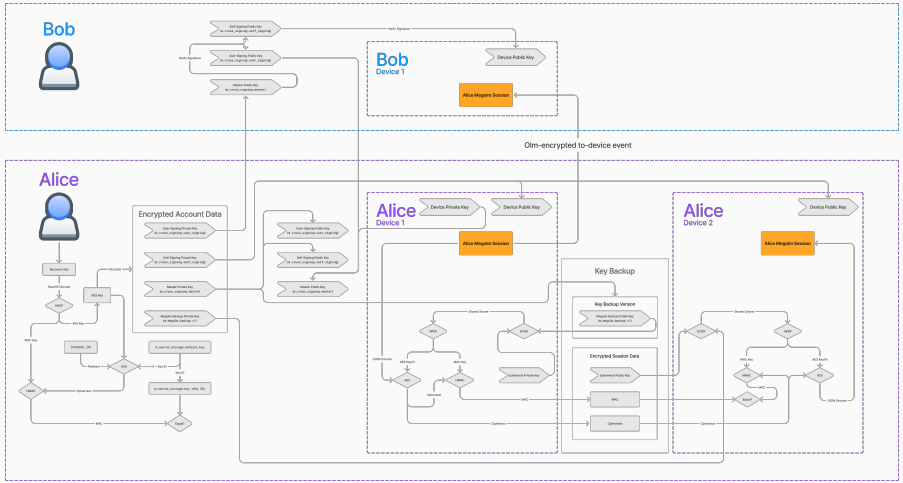
Big Picture: The Other Stuff

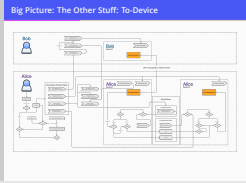




1. So what is all of the other stuff? That is the **infrastructure** for sharing the Megolm session around to other devices and users.

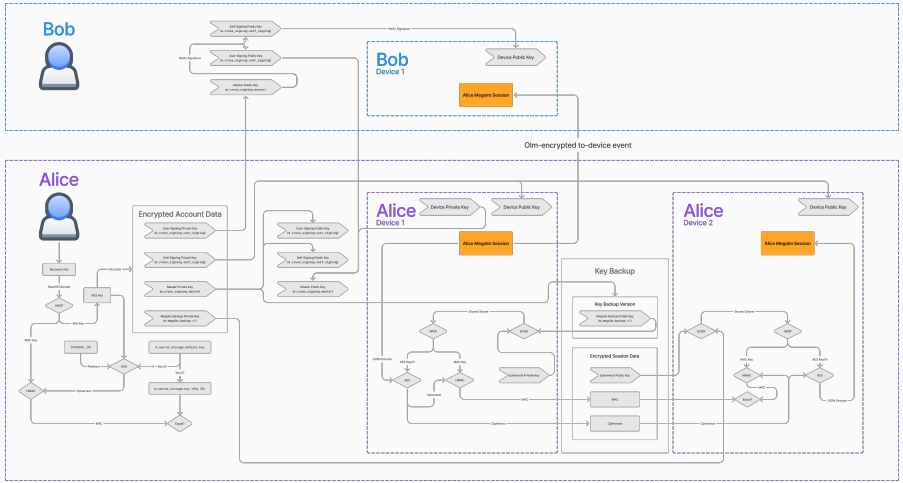
Big Picture: The Other Stuff: To-Device

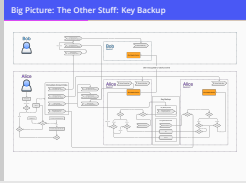




1. For example, in this arrow represents sending the Megolm session via Olm-encrypted to-device messages.

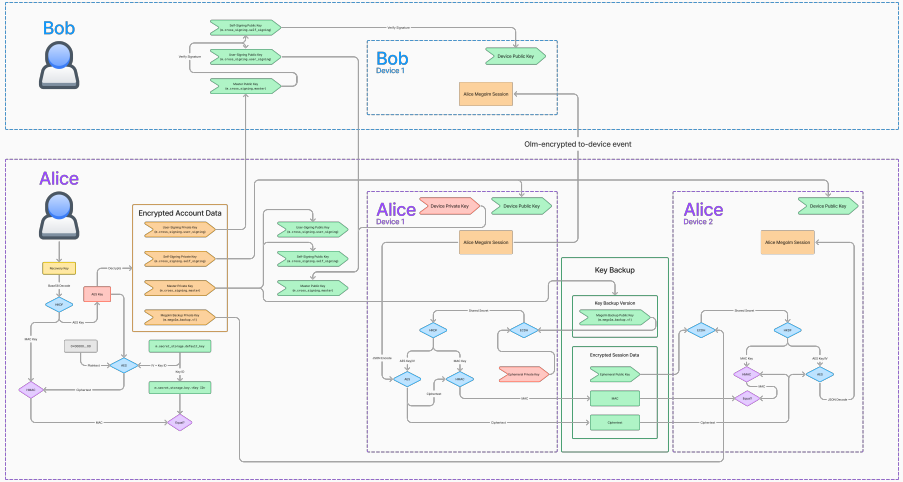
Big Picture: The Other Stuff: Key Backup

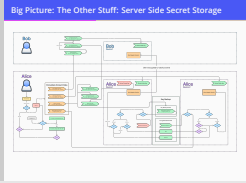




1. This lower section of the diagram represents key backup which allows you to backup your keys to the server and restore from your other devices.

Big Picture: The Other Stuff: Server Side Secret Storage





1. Lastly, on the left we have the infrastructure necessary for storing secrets on the server encrypted by a recovery code.

Cryptography Crash Course

Cryptography Crash Course

Before we dive deeper into the details of the diagram, we need to discuss some basic cryptography primitives.

I will try and explain these in simple terms. It's not going to be mathematically rigorous, but will focus on the **functionality** that each cryptographic primitive provides.

Encryption: Symmetric vs Asymmetric

There are two main categories of encryption schemes:

- **Symmetric** — both **the encryptor and the decryptor share the same key** and that key is used in both the encryption and decryption of the message
- **Asymmetric** — **the encryptor needs the public key, and the decryptor needs the private key** and the encryptor encrypts the message with the public key, and the private key is required to decrypt the message

Encryption: Symmetric vs Asymmetric

There are two main categories of encryption schemes:

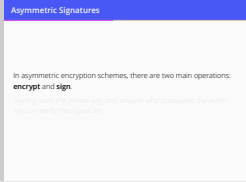
- **Symmetric** — both **the encryptor and the decryptor share the same key** and that key is used in both the encryption and decryption of the message
- **Asymmetric** — **the encryptor needs the public key, and the decryptor needs the private key** and the encryptor encrypts the message with the public key, and the private key is required to decrypt the message

Asymmetric Signatures

In asymmetric encryption schemes, there are two main operations: **encrypt** and **sign**.

Signing uses the *private* key, and anyone who possesses the *public* key can verify the signature.

1. We discussed encryption already.
2. You use the public key to encrypt a message, and the private key to decrypt, but...



Asymmetric Signatures

In asymmetric encryption schemes, there are two main operations: **encrypt** and **sign**.

Signing uses the *private* key, and anyone who possesses the *public* key can verify the signature.

Asymmetric Signatures

In asymmetric encryption schemes, there are two main operations: **encrypt** and **sign**.

Signing uses the *private* key, and anyone who possesses the *public* key can verify the signature.

1. We discussed encryption already.
2. You use the public key to encrypt a message, and the private key to decrypt, but...

Hashes and HMAC

A **cryptographic hash function** is a one-directional function which takes an arbitrarily large set of data and produces a unique fixed-size output (called the hash).

Given the same data, a hash function will always return the same output.

This allows us to verify that the data did not change in transit (for example, by a malicious actor).

Hashes are vulnerable to **metadata attacks**. To prevent these, we use HMAC which adds a secret key to the hash.

Hashes and HMAC

A **cryptographic hash function** is a one-directional function which takes an arbitrarily large set of data and produces a unique fixed-size output (called the hash).

1. If you hash the same message multiple times, you will receive the same value, and an attacker could use this information to deduce the frequency of certain messages being sent.
2. How the key is added is an implementation detail that is not relevant to your understanding of what functionality HMAC provides.

Hashes and HMAC

A **cryptographic hash function** is a one-directional function which takes an arbitrarily large set of data and produces a unique fixed-size output (called the hash).

Given the same data, a hash function will always return the same output.

This allows us to verify that the data did not change in transit (for example, by a malicious actor).

Hashes are vulnerable to **metadata attacks**. To prevent these, we use HMAC which adds a secret key to the hash.

Hashes and HMAC

A **cryptographic hash function** is a one-directional function which takes an arbitrarily large set of data and produces a unique fixed-size output (called the hash).

Given the same data, a hash function will always return the same output.

This allows us to verify that the data did not change in transit (for example, by a malicious actor).

Hashes are vulnerable to **metadata attacks**. To prevent these, we use HMAC which adds a secret key to the hash.

1. If you hash the same message multiple times, you will receive the same value, and an attacker could use this information to deduce the frequency of certain messages being sent.
2. How the key is added is an implementation detail that is not relevant to your understanding of what functionality HMAC provides.

Hashes and HMAC

A **cryptographic hash function** is a one-directional function which takes an arbitrarily large set of data and produces a unique fixed-size output (called the hash).

Given the same data, a hash function will always return the same output.

This allows us to verify that the data did not change in transit (for example, by a malicious actor).

Hashes are vulnerable to **metadata attacks**. To prevent these, we use HMAC which adds a secret key to the hash.

Hashes and HMAC

A **cryptographic hash function** is a one-directional function which takes an arbitrarily large set of data and produces a unique fixed-size output (called the hash).

Given the same data, a hash function will always return the same output.

This allows us to verify that the data did not change in transit (for example, by a malicious actor).

Hashes are vulnerable to **metadata attacks**. To prevent these, we use HMAC which adds a secret key to the hash.

1. If you hash the same message multiple times, you will receive the same value, and an attacker could use this information to deduce the frequency of certain messages being sent.
2. How the key is added is an implementation detail that is not relevant to your understanding of what functionality HMAC provides.

Hashes and HMAC

A **cryptographic hash function** is a one-directional function which takes an arbitrarily large set of data and produces a unique fixed-size output (called the hash).

Given the same data, a hash function will always return the same output.

This allows us to verify that the data did not change in transit (for example, by a malicious actor).

Hashes are vulnerable to **metadata attacks**. To prevent these, we use HMAC which adds a secret key to the hash.

Hashes and HMAC

A **cryptographic hash function** is a one-directional function which takes an arbitrarily large set of data and produces a unique fixed-size output (called the hash).

Given the same data, a hash function will always return the same output.

This allows us to verify that the data did not change in transit (for example, by a malicious actor).

Hashes are vulnerable to **metadata attacks**. To prevent these, we use HMAC which adds a secret key to the hash.

1. If you hash the same message multiple times, you will receive the same value, and an attacker could use this information to deduce the frequency of certain messages being sent.
2. How the key is added is an implementation detail that is not relevant to your understanding of what functionality HMAC provides.

Key-Derivation Functions (HKDF)

Sometimes, we want to turn a small key into a larger key (or set of larger keys).

Key-Derivation Functions (KDFs) are used to do this.

Matrix uses HKDF which uses HMAC for the key derivation process.

Key-Derivation Functions (HKDF)
Sometimes, we want to turn a small key into a larger key (or set of larger keys).
Key-Derivation Functions (KDFs) are used to do this.
Matrix uses HKDF which uses HMAC for the key derivation process.

1. For example, we might want to "stretch" a 32-byte shared secret into a 32-byte AES key, a 16-byte AES IV, and a 32-byte HMAC key.

Key-Derivation Functions (HKDF)

Sometimes, we want to turn a small key into a larger key (or set of larger keys).

Key-Derivation Functions (KDFs) are used to do this.

Matrix uses HKDF which uses HMAC for the key derivation process.

Key-Derivation Functions (HKDF)

Sometimes, we want to turn a small key into a larger key (or set of larger keys).

Key-Derivation Functions (KDFs) are used to do this.

Matrix uses HKDF which uses HMAC for the key derivation process.

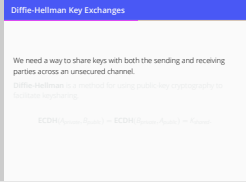
1. For example, we might want to "stretch" a 32-byte shared secret into a 32-byte AES key, a 16-byte AES IV, and a 32-byte HMAC key.

Diffie-Hellman Key Exchanges

We need a way to share keys with both the sending and receiving parties across an unsecured channel.

Diffie-Hellman is a method for using public-key cryptography to facilitate keysharing.

$$\mathbf{ECDH}(A_{private}, B_{public}) = \mathbf{ECDH}(B_{private}, A_{public}) = K_{shared}.$$



1. For example, we may need to share a MAC or AES key. This could be done in-person, but that is very impractical.
   That's where the Diffie-Hellman (DH) Key Exchange method comes in.
2. Since Matrix uses elliptic-curve cryptography, the specific variant of Diffie-Hellman that Matrix uses is ECDH (the elliptic curve variant).
   I'm not going to discuss the actual mathematical mechanism behind ECDH as it's quite complex and not relevant to understanding how Matrix uses ECDH. However, it is essential to understand the main feature it provides:
3. In this equation, $A_{private}$ and A_{public} and $B_{private}$ and B_{public} are related to one another.
4. The key here is that we will get the same value out of ECDH *regardless of which private key you have*. You do need the other public key, but those are public and can be shared freely.
5. If A_{public} and $B_{private}$ are both available, then K_{shared} can be recovered. This is true even if $A_{private}$ has been discarded.

Diffie-Hellman Key Exchanges

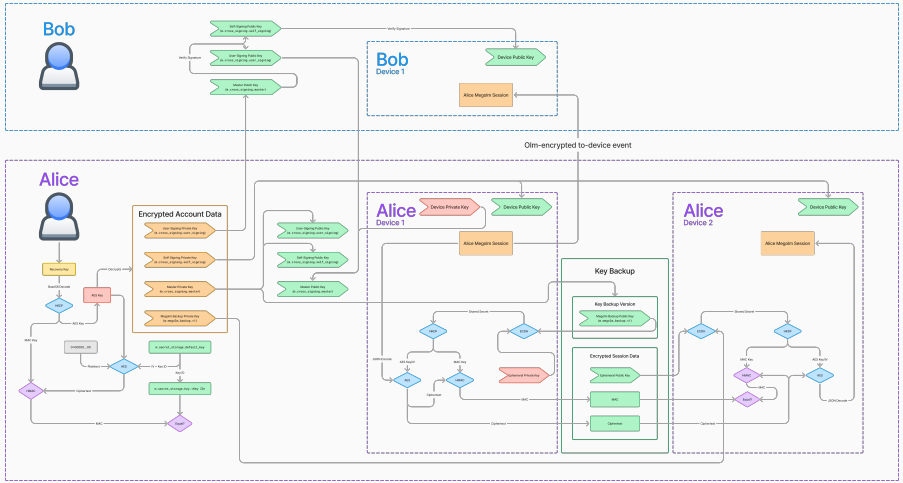
We need a way to share keys with both the sending and receiving parties across an unsecured channel.

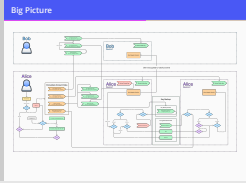
Diffie-Hellman is a method for using public-key cryptography to facilitate keysharing.

$$\mathbf{ECDH}(A_{private}, B_{public}) = \mathbf{ECDH}(B_{private}, A_{public}) = K_{shared}$$

1. For example, we may need to share a MAC or AES key. This could be done in-person, but that is very impractical.
   That's where the Diffie-Hellman (DH) Key Exchange method comes in.
2. Since Matrix uses elliptic-curve cryptography, the specific variant of Diffie-Hellman that Matrix uses is ECDH (the elliptic curve variant).
   I'm not going to discuss the actual mathematical mechanism behind ECDH as it's quite complex and not relevant to understanding how Matrix uses ECDH. However, it is essential to understand the main feature it provides:
3. In this equation, $A_{private}$ and A_{public} and $B_{private}$ and B_{public} are related to one another.
4. The key here is that we will get the same value out of ECDH *regardless of which private key you have*. You do need the other public key, but those are public and can be shared freely.
5. If A_{public} and $B_{private}$ are both available, then K_{shared} can be recovered. This is true even if $A_{private}$ has been discarded.

Big Picture



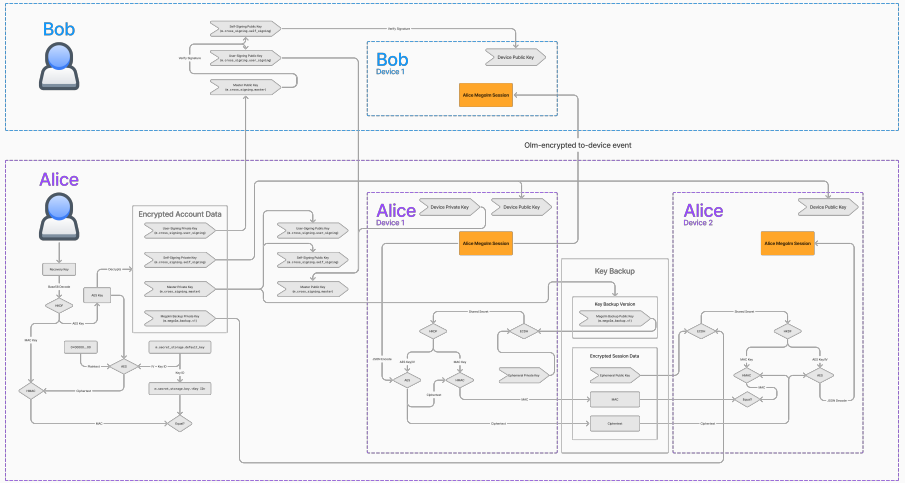


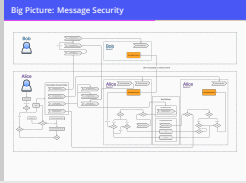
1. Let's go back to the big picture now. Recall that the blue and purple nodes represent cryptographic operations.
   All of these are one of the operations that we discussed. Now we are going to talk about how they are applied to Matrix in the context of this diagram.

Sharing Keys

We're going to start with key sharing. How do we get keys from one device to another?

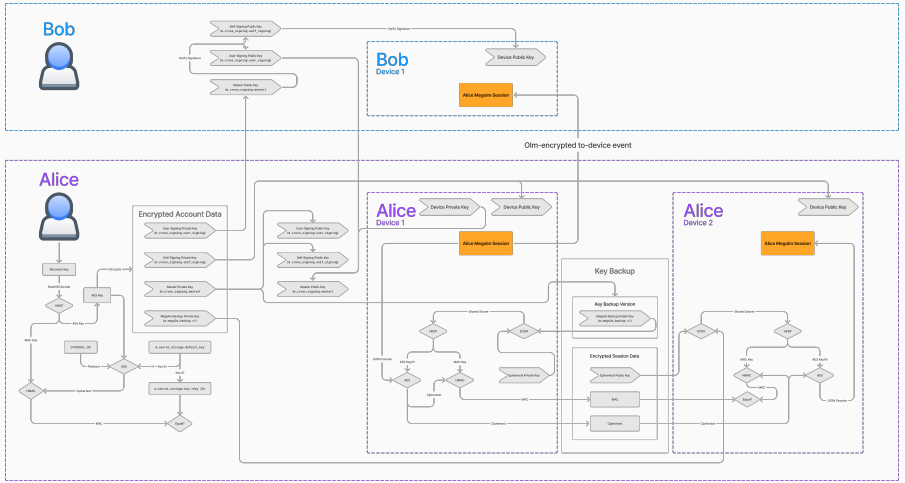
Big Picture: Message Security

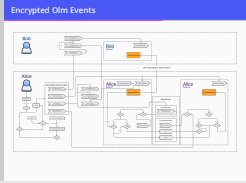




1. Remember, what the key that we want to share is the Megolm key. That's what encrypts the messages.
2. There are two ways to share these: encrypted olm events and key backup.

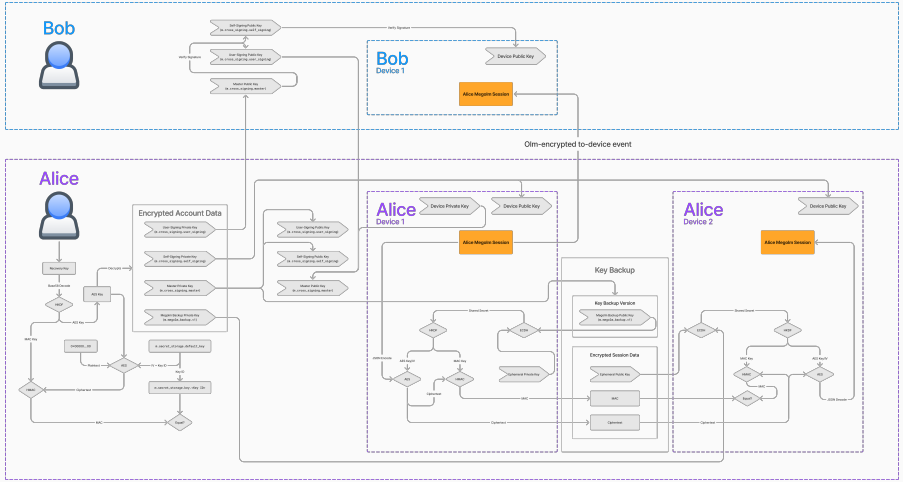
Encrypted Olm Events

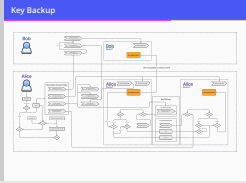




1. Encrypted olm events are represented by these arrows. They are sent via to-device messages which allow you to send messages to specific devices (rather than rooms).
2. I'm not going to discuss how Olm encryption works. It's already been covered many times since it's basically just the Signal double-ratchet algorithm.
3. For our purposes, it's sufficient to know that we can send keys securely via to-device events via Olm. We can also request keys from our own **verified** devices. We will talk about how we know a device is verified later.
4. This seems great, why do we have anything else?
5. Well, new logins are the issue. Say Alice just logged in on Device 2 and finished verification.
   - If Device 1 is *online*, she can send key requests, but there's likely going to be a ton of requests due to all of the keys in all of the rooms to her Device 1. This will make Device 1 do a lot of work sending back all the keys.
   - If Device 1 is ***off****line*, her send key requests won't be answered.

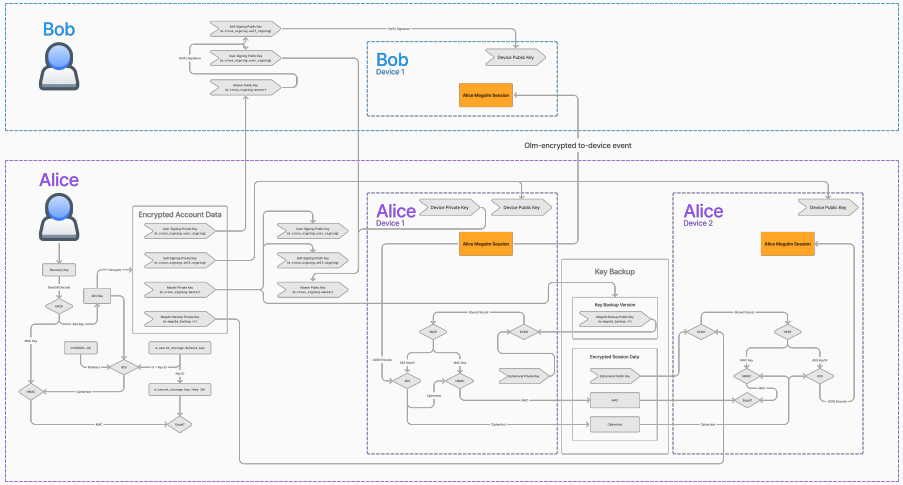
Key Backup



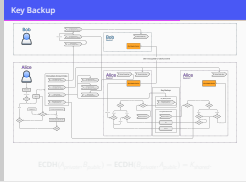


1. That's where **key backup** comes into play. Key backup allows us to store keys on the server, and restore them from our other devices without the other devices needing to be online.
2. Let's zoom in and see what's going on.

Key Backup

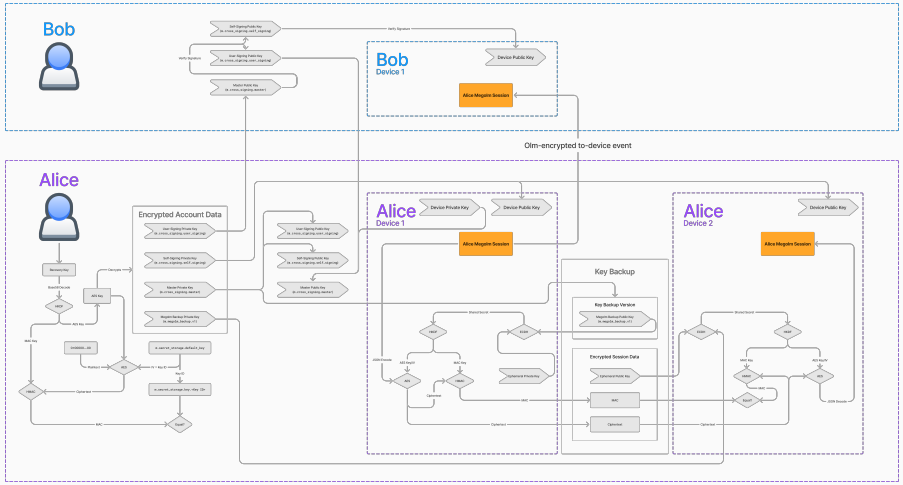


$$\mathbf{ECDH}(A_{private}, B_{public}) = \mathbf{ECDH}(B_{private}, A_{public}) = K_{shared}.$$

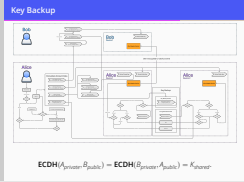


1. In the middle here we have this green “key backup” thing which is stored on the server. We’re trying to get the Megolm key from Alice’s Device 1 to her Device 2, so left to right.
2. There are two pieces to key backup:
   - the version which includes a public key for the backup.
   - the encrypted session data (there are many of these in a backup (one per megolm session stored)).
3. The first thing to note is that AES is used on both sides to do the encryption and decryption, so only the encrypted version is stored on the server. But, where do we get the key and IV from?
4. Well, we get it from HKDF. So, the next question is where do we get the key for HKDF?
5. That comes from a call to ECDH.
6. Note that all this up to ECDH is **the same on both the encrypting and decrypting sides!**
7. Recall that ECDH requires a public key and a private key, so where do we get those?
8. This is where the sides diverge.
9. Recall that we need two keypairs for ECDH.
   - The first keypair is the Megolm backup keypair.
   - The second keypair is the Epemeral keypair.
10. We can use either private key and the other keypair’s public key and get the same value out of ECDH.
    - The encrypting side gets its private key from this *ephemeral keypair*. It’s ephemeral because the private part can be discarded immediately after the encryption is done.
      Thus, it uses the Megolm backup public key as its public key.
    - The decrypting side gets its private key from the Megolm backup private key.
      Thus, it uses the ephemeral public key as its public key.
11. **Critically** you must have the Megolm backup private key to decrypt the key backup. We will discuss how that is stored later.

Key Backup



$$\mathbf{ECDH}(A_{private}, B_{public}) = \mathbf{ECDH}(B_{private}, A_{public}) = K_{shared}$$



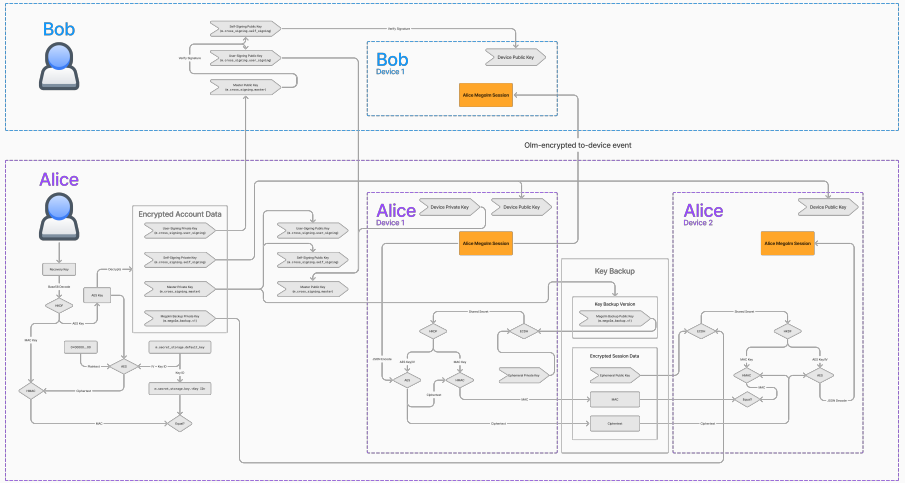
1. In the middle here we have this green “key backup” thing which is stored on the server. We’re trying to get the Megolm key from Alice’s Device 1 to her Device 2, so left to right.
2. There are two pieces to key backup:
   - the version which includes a public key for the backup.
   - the encrypted session data (there are many of these in a backup (one per megolm session stored)).
3. The first thing to note is that AES is used on both sides to do the encryption and decryption, so only the encrypted version is stored on the server. But, where do we get the key and IV from?
4. Well, we get it from HKDF. So, the next question is where do we get the key for HKDF?
5. That comes from a call to ECDH.
6. Note that all this up to ECDH is **the same on both the encrypting and decrypting sides!**
7. Recall that ECDH requires a public key and a private key, so where do we get those?
8. This is where the sides diverge.
9. Recall that we need two keypairs for ECDH.
   - The first keypair is the Megolm backup keypair.
   - The second keypair is the Epemeral keypair.
10. We can use either private key and the other keypair’s public key and get the same value out of ECDH.
    - The encrypting side gets its private key from this *ephemeral keypair*. It’s ephemeral because the private part can be discarded immediately after the encryption is done.
      Thus, it uses the Megolm backup public key as its public key.
    - The decrypting side gets its private key from the Megolm backup private key.
      Thus, it uses the ephemeral public key as its public key.
11. **Critically** you must have the Megolm backup private key to decrypt the key backup. We will discuss how that is stored later.

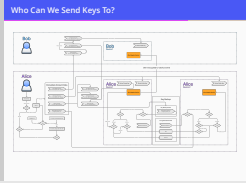
Device Verification

Device Verification

Now, let's discuss device verification.

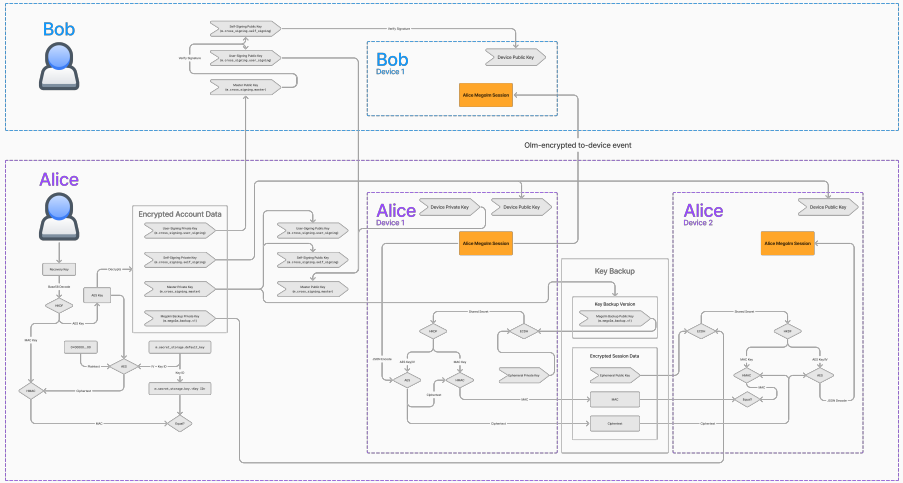
Who Can We Send Keys To?

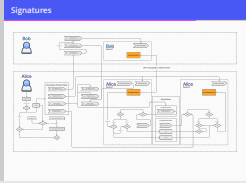




1. Recall how I said that we only want to forward room keys to our **verified** devices?
2. Now, we are going to discuss how verification status is determined.
3. The answer is...

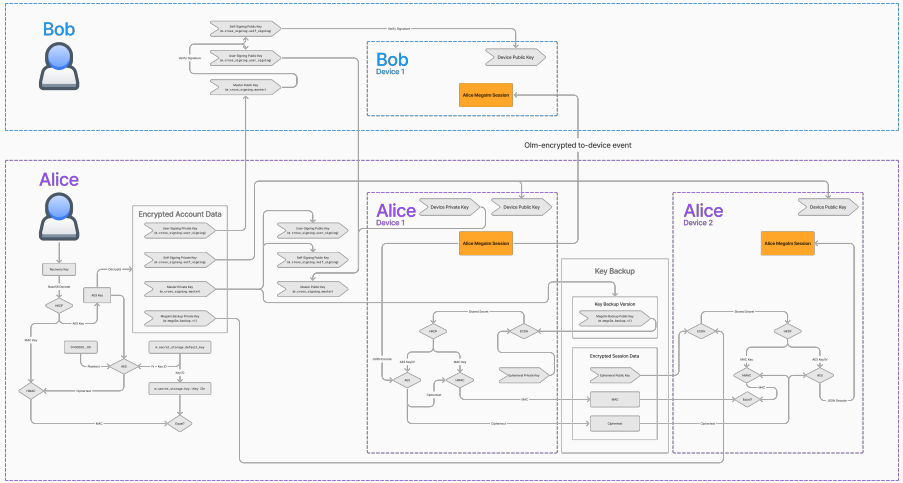
Signatures

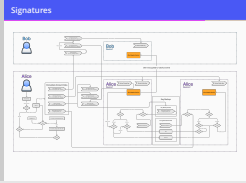




1. **Signatures!**
2. Let's zoom in on this part.

Signatures

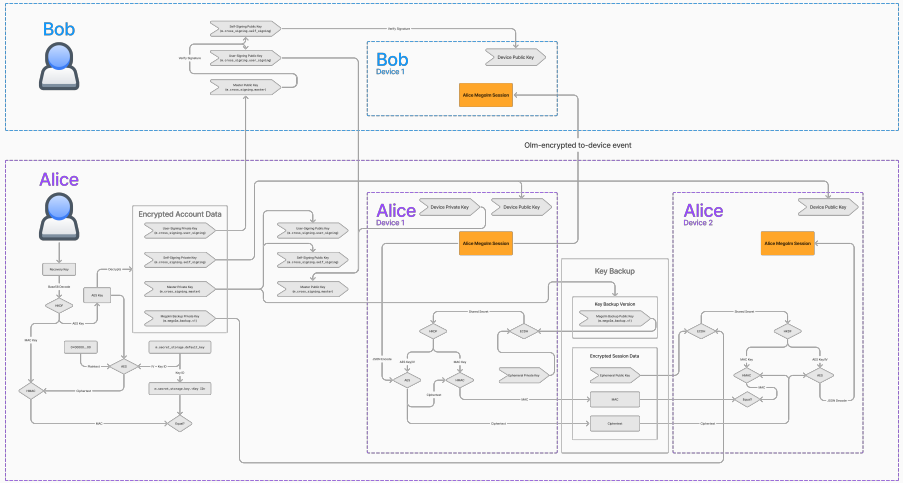


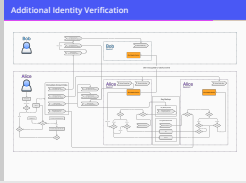


1. Remember, signatures use the *private* key, and anyone who possesses the *public* key can verify the signature.
2. Often, we use "trust" terminology. When we "trust" a key, we have a signature (or chain of signatures) that sign the public key.
3. Ultimately, the key we want to trust is the other device key.
4. We can use our device key to directly generate the signature for the other device key.
   But that is inconvenient. When we log in a new device, we will have to create a signature from all our other devices for that new device. And that device will have to create a signature for all of the existing devices!
5. So, we introduce a new user-wide key called the "self-signing key" because it signs our own device keys.
6. We use the self-signing key to sign the device keys but how do we know if we should trust the self-signing key?
7. That's where the **master key** comes in. The master key signs the self-signing key.
8. We then trust the master key by signing it with our device private key.
9. So the **chain-of-trust** is device private key → master public key which corresponds to its private key → self-signing public key which corresponds to its private key → device public key.
10. This allows us to trust a single key (in this case, the master key) and then trust all of our own devices.

User Verification

Additional Identity Verification



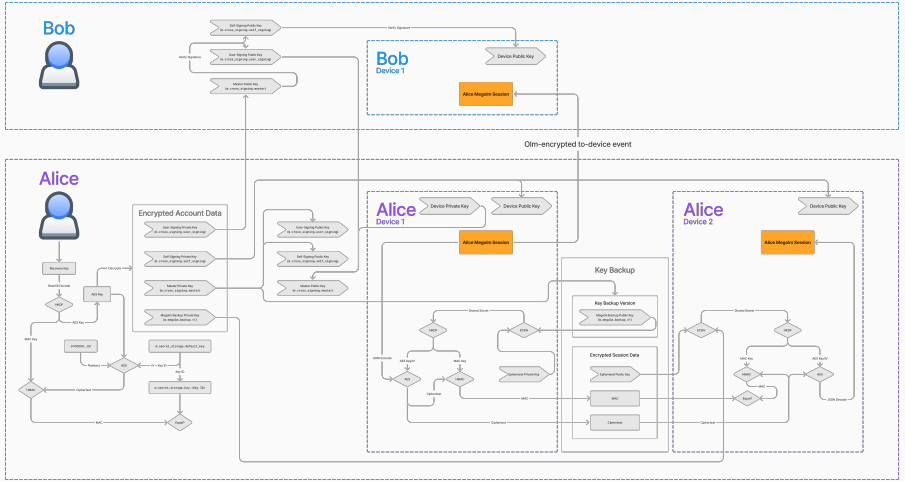


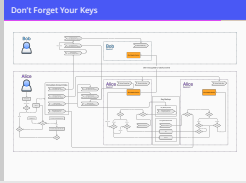
1. Sometimes, we want to trust a user so that we know that all of the devices on their account are under their control. If a new device is logged in, we will know if they control the device if it's signed.
2. They have already signed their devices with their self-signing key, which is itself signed by their master key. So, if we are able to trust their master key, we will have a chain of trust to all of their devices.
3. This means that we know that when we send the keys to their devices, they are actually under their control.
4. This is where the "user-signing key" comes into play. The user-signing key signs other users' master keys and is itself signed by our master key.
5. So the **chain-of-trust** is device private key → master public key which corresponds to its private key → user-signing public key which corresponds to its private key → other user's public master key which corresponds to its private key → other users self-signing key which corresponds to its private key → other users device public keys.

Server Side Secret Storage (SSSS)

Server Side Secret Storage (SSSS)

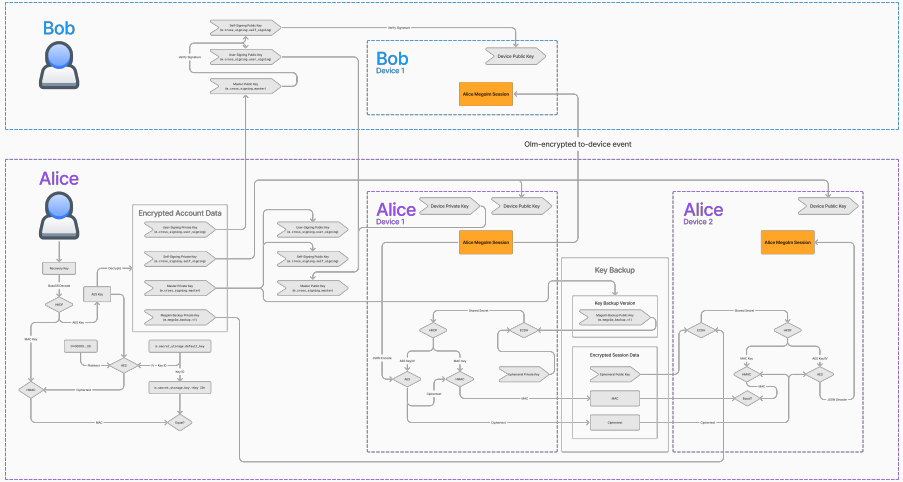
Don’t Forget Your Keys

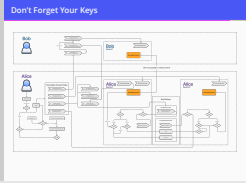




1. You’ve probably noticed that there’s a lot of private keys that need to be stored for all of this to work properly. The keys are stored each of your devices and can be shared with your other verified devices using encrypted to-device events.
2. But what if you sign out of all of your devices or lose access to them?
3. That’s where server-side secret storage comes in: it allows you to store your keys encrypted within account data on the server.

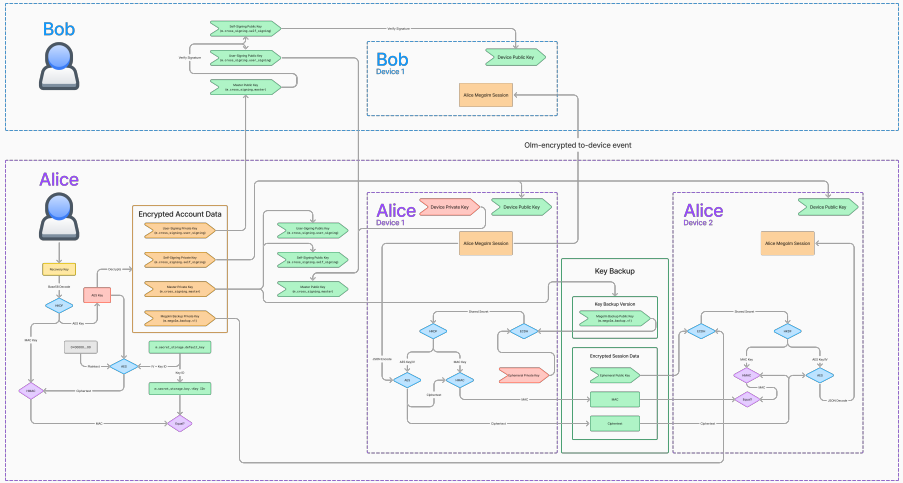
Don’t Forget Your Keys

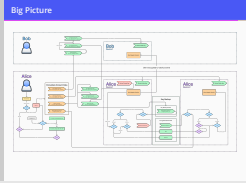




1. The key that they are encrypted using is basically the recovery key. There is an HKDF transformation which gives you the actual key, but it’s basically just your recovery key that unlocks the encrypted account data.
2. This bottom part isn’t actually strictly necessary. It allows you to verify that your recovery key is correct. If you want the details here, you can read the blog post associated with this talk.

Big Picture





1. Let's go back to the overview.
2. We've talked about each piece of this puzzle.
   - We talked about to-device events
   - We talked about key backup
   - We talked about self-signing of devices
   - We talked about signing of other users
   - And then we talked about server-side secret storage
3. I hope that this presentation has helped you understand how it fits together.
4. My goal is to convince people that Matrix cryptography is not scary. It's complex, but not inaccessible.
5. If you have access to all of the underlying cryptography primitives, all of this is something that any security-conscious programmer could implement.
6. You almost certainly should not implement the cryptography primitives yourself, but composing them together is not that bad.
7. And with that, I'll open it up to questions

Thank You for Listening!

Questions?

Thank You for Listening!
Questions?