IT University of Copenhagen

(master thesis)

Information-Flow Secure Programming on Matrix: A Case Study

Authors:
Ans Uddin anud@itu.dk

Supervisor:
Willard Rafnsson

Co-supervisor:
Carsten Schürmann

March, 2019

Contents

1 Introduction

1.1 Data privacy and protection

With GDPR becoming effective in 2018 the focus on data privacy is at its peak. Privacy violation is when sensitive data is exposed to unauthorized actors[3]. OWASP top ten ranks *sensitive data exposure* as 3rd biggest security threat[35].

Recent cases of data leakage has put more attention on data privacy and protection. Some cases are due to poor security measures and could arguably have been prevented. Examples of cases are:

- The infamous Facebook - Cambridge Analytica scandal. Third parties were able to collect data through Facebook Login API.
- Google Plus leak. 500.000 users private data was exposed to third parties through APIs[6].
- Medicaid leak. A medical assistant had accessed patients' health records and exchanged mails with another employee containing the patients' private data[14].

The cases above failed to achieve end-to-end security and the improper handling of sensitive data could have been prevented with appropriate security policies and enforcement technique that enforces these policies.

There is more awareness on how applications deal with data. This add extra concern to the programmer and the application about how sensitive data is handled and protected.

The well-known security enforcement techniques like access controls, firewalls and encryption are inadequate alone and does not ensure end-to-end security[39].

1.2 Information Flow Control

There exist useful security enforcement mechanisms for protecting confidential information such as firewalls, encryption and access control. However, these mechanisms each have their drawbacks.

- *Access* control prevents unauthorized access to information but once access is granted there is no guarantee how that confidential information is handled.

- *Firewall* limits communication from the outside hence isolate and protect information. Yet the firewall have no way of telling if the communication going through violates confidentiality.
- *Encryption* secures information on a channel with only the endpoints being able to access that information. However there is no assurance that once the data is decrypted that the confidentiality of that information is ensured.

The mechanisms mentioned above all have in common that they lack control of how the information flows. Information-flow security aims at protecting confidentiality and integrity of information by enforcing security policies. Information-Flow Control allows the programmer to define and enforce policies in a language-based way[39].

1.3 Matrix

Matrix is an open standard protocol for messaging over HTTP and synchronizing data. Matrix provides secure real-time communication over a decentralized federated network. Matrix secures data by providing end-to-end encryption.

Matrix cover use cases such as instant messaging, VoIP, Internet of Things communication and is generally applicable anywhere for subscribing and publishing data over standard HTTP API.

The fragmentation of IP communication is the problem Matrix essentially wants to solve. Making calls and messages between users needless of which app they use. However they define their longer term goal as *"to act as a generic HTTP messaging and data synchronisation protocol for the whole web"*[10].

1.4 The case study

The goal of the case study is to make secure implementation of a prototype using Information-Flow Control. The case study will use Matrix as the communication channel and strengthen the security at the endpoints using IFC.

1.4.1 Journal system

The prototype implements a journal system and is loosely based on the Danish E-journal system.

Medical privacy is a well-known issue[38]. Sensitive data about patients needs to be handled carefully. In Denmark patients have access to their medical records through E-journal[4]. A patient's journal on E-journal is available for up to 90.000 different medical employees[1].

There are clear policies about who and under what conditions should access a journal. It is legally required that an employee accessing the journal must have the patient in care and that the lookup must be relevant for the employee. Safety measures have been applied through logging and audit trails with random sampling checks however they do not prevent access to journals. Any medical employee can access the patient journal and even if prevention mechanism were established there would be no limitation to what a medical employee could see once access was granted [2][9].

The mechanisms in the current journal system might restrain malicious intend. However it does not guarantee prevention of unintentional access or disclosure of information[25]. What is missing is the enforcement of secure information flow policies. Unintentional access or disclosure of information can be prevented by enforcing policies that define secure information flow.

The prototype will model a simplified scenario of hospitals with different actors accessing a patient journal. The bulk of information on the journal system is extracted from newspaper articles hence there is a high uncertainty of how the system really works. Therefore many assumptions are made about the current system when programming the prototype.

1.4.2 Scope

The objective of the project is to do a secure implementation of the prototype described above. Secure exchange of patient journal is ensured using Matrix and the endpoints are secured using IFC.

A successful project is one that fulfills these criteria:

- Evaluation of Matrix security model
- Survey of IFC tools and selection of tool.
- Implement a prototype distributed system running on Matrix, using the chosen tools
- Demonstrate increased security guarantee with Matrix and IFC

1.4.3 Why Matrix?

In the Digital Strategy 2016-2020 the Danish Agency of Digitisation defines initiative 7.2 as *"Common standards for secure exchange of information"*. The large number of software systems in the Danish public sector has created a need for an uniform way of exchanging data across different application in a secure manner[45].

The initiative has similarities to the issue Matrix is trying to solve with fragmented IP communication. With Matrix security guarantees and their long term goal as a generic HTTP messaging protocol there is a strong case for using Matrix as a communication channel in this case study.

1.5 Method

1.6 Threat model

The threat model is defined in the context of confidentiality and integrity.

- The adversary has the ability to observe information sent over the network.
- The adversary can generate input to the system .
- The adversary can observe public output.

1.7 Contribution

The contributions to the field are the findings of secure implementation using Paragon and how they compare to similar findings from secure implementation with JIF.

The thesis also contributes with the interface created between Paragon and Matrix making it possible to develop other secure applications on top of secure communication channel Matrix provides.

1.8 Structure of thesis

Chapter 2 sets the foundation for the thesis and introduces relevant information and background. Chapters 3 analyzes the Matrix security model and survey IFC tools. Chapter **??** goes in depth with design of the solution. The results are then presented and discussed in Chapter **??**. The thesis is wrapped up in the conclusion section Chapter 6.

1.9 Summary

There is more awareness on how applications deal with data. This add extra concern to the programmer and the application about how sensitive data is handled and protected. Encryption is an obvious way of ensuring confidentiality; Matrix is a tool that provides end-to-end encryption. However there is no guarantee of encryption of the data is decrypted. Information flow control is such mechanism that enforces security policies throughout the system.

2 Background

This chapter builds the foundation for the thesis introducing relevant concepts and disciplines for the thesis.

Section 3.1 provides an evaluation of the Matrix security model. The security model is evaluated in the context of a secure messaging system. The paper *SoK: Secure Messaging* describes a evaluation framework for evaluating secure messaging systems. They define several security properties related to such systems [46] which will be presented.

All secure messaging systems with end-to-end encryption are based on the Double Ratchet algorithm from the Signal Protocol which will also be described.

Finally the architecture of Matrix and concepts related to Information-Flow Control will be introduced.

2.1 Information security

Information security is the discipline of protecting information. The key principles in information security are expressed through the CIA model. For a system to be secure these principles should be guaranteed [30].

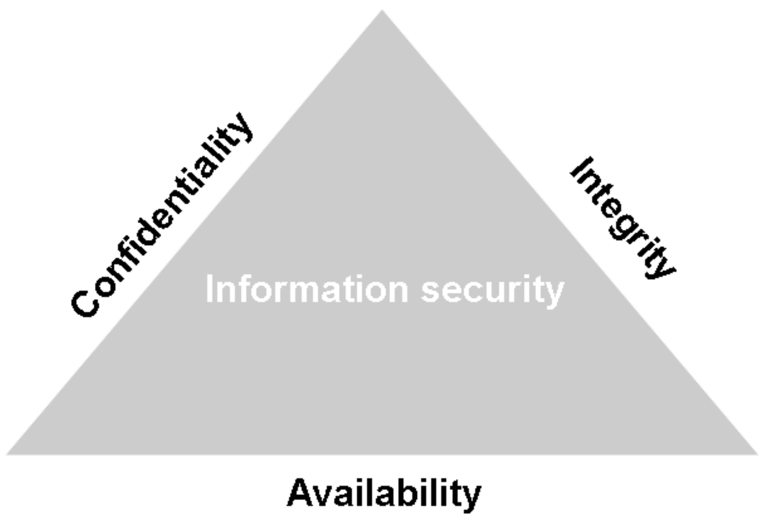


Figure 2.1: CIA triad

Confidentiality Confidentiality is keeping information secret from unauthorized people. This is a major goal in information security. Encryption and access control are common ways of ensuring confidentiality [30].

In a secure messaging system confidentiality would be guaranteed if the message being sent is only readable by the recipient and no one else [46].

Integrity Integrity is providing that information is unaltered and can only be changed by authorized people. If information is intercepted and changed during transit it would be a violation of integrity [30]. More specifically for a secure messaging it would mean that no altered message is accepted by the recipient [46].

Availability Making sure that information is accessible to authorized people is the goal of availability. Denial of Service attack [1] are common attacks targeting availability.

Availability is generally more related to the system being available where the information itself plays a minor role [30].

Depending on the type of system other properties must be satisfied as well.

2.1.1 Security properties

The goal in a secure messaging system is to protect the messages being sent. The following properties are related to protecting messages.

Authentication When a message is received the participant can verify that the message was sent from the actual sender. Furthermore a participant will receive evidence from a participant in a conversation that hey hold a known long-term secret.

Perfect Forward Secrecy If all keys are compromised than the decryption of any previously sent message should not be possible. Hence all previous messages would be secure however all future messages would be insecure

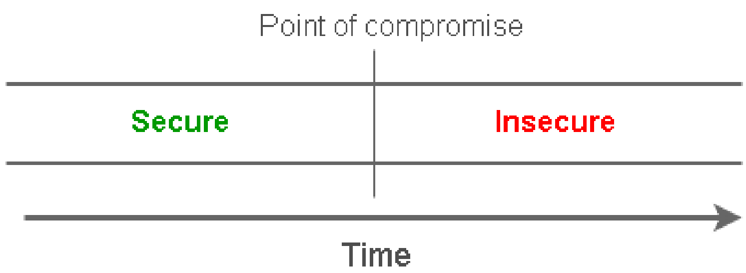


Figure 2.2: Forward secrecy

Backward secrecy If all keys are compromised than the decryption of *future* messages should be possible. This property also goes by the names *future secrecy* and *post compromise security*.

[1] https://en.wikipedia.org/wiki/Denial-of-service_attack

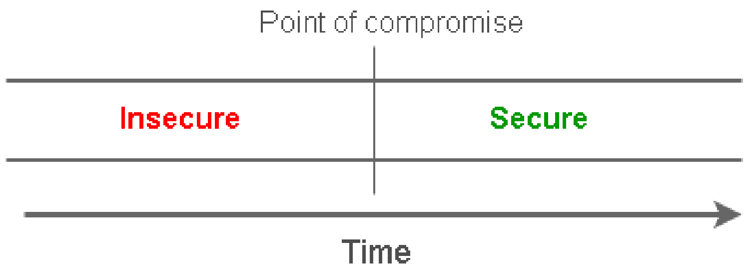


Figure 2.3: Backward secrecy

2.1.1.1 Other security properties

Participant Consistency Whenever a message is accepted by a participant all participants are guaranteed to have identical view of the participant list.

Destination Validation When a participant receives a message it can be verified that the participant was the intended recipient.

Anonymity Preserving The anonymity of the participants should be preserved and not linking any key identifiers.

Speaker Consistency There is consensus among the participants on the sequence of messages they receive by each participant. There might be a mechanism for checking consistency whenever a message is sent or after it has been received.

Causality Preserving Messages must not be displayed before the message that originally precedes it has been displayed.

Global Transcript A global order where all messages are viewed in the same order for all participants.

Deniability Deniability is a property where other participants cannot confirm that the message being sent was from the sender. Yet during the conversation there will be assurance for the recipient that the message being sent was authentic and sent by the sender [46].

- *Message Unlinkability:* A deniability property that gives no guarantees that if a participant sent a message that other messages was sent by that participant as well.
- *Message Repudiation:* It can not be proved that a message was authored by a participant given the conversation transcript and all cryptographic key material.
- *Participant Repudiation:* It can not be proved that a participant was in a group conversation without his conversation transcript and cryptographic key material.

The following properties are also defined in the paper *SoK: Secure Messaging* but are less relevant for security.

Group

- *Computational Equality:* The computational load is equal for all participants.
- *Trust Equality:* There is equal trust among all participants.
- *Subgroup messaging:* In the same conversation a participant can send messages to a subset of the participants.
- *Contractible Membership:* When a participant leaves a conversation the protocol does not need to restart.
- *Expandable Membership:* When a participant joins a conversation the protocol does not need to restart.

Adoption

- *Out-of-Order Resilient:* Messages received out-of-order should be accessible when received.
- *Dropped Message Resilient:* On a unreliable network messages might be dropped in transit however it should not prevent decryption of future messages.
- *Asynchronous:* Messages can be sent securely to recipients while they are offline.
- *Multi-Device Support:* A participant can have multiple devices in a conversation and each device must be synchronized and should have the same historical conversation view
- *No Additional Service:* There is no requirement of additional infrastructure being setup other than the participants.

2.1.2 Concepts

2.1.2.1 Diffie-Hellman Key Exchange

Diffie-Hellman is a key exchange protocol to establish a shared secret over an insecure channel. Public information is send over an insecure and using asymmetric keys two parties can derive the same shared key.

The first step is to agree on some public values. Either of the parties start the protocol by picking a large prime p and a integer g then the values are sent over the insecure channel.

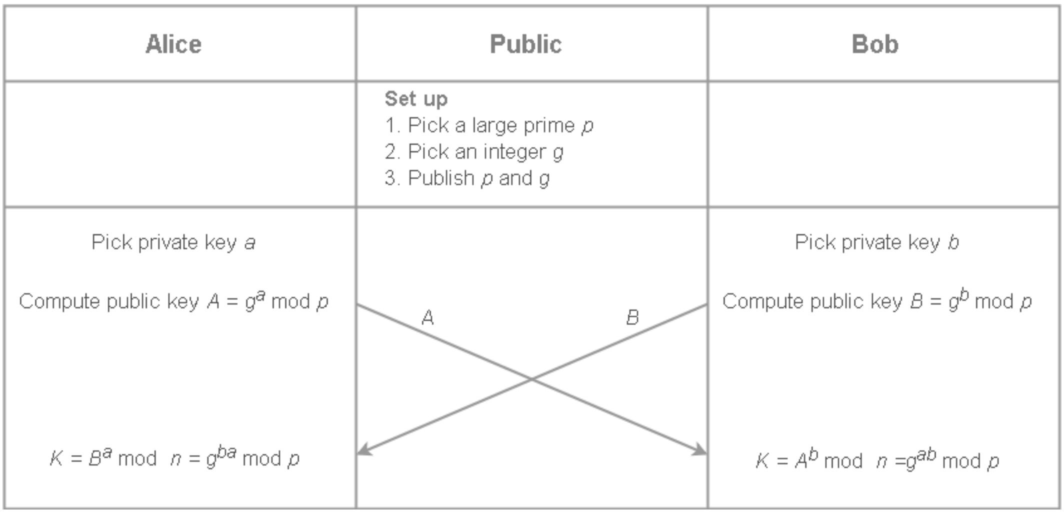


Figure 2.4: Simple Diffie-Hellman Key Exchange

Alice and Bob will each pick a private key value and compute a public key value. The computed key value is sent over the insecure channel and Alice and Bob will both perform the same computation as previously [37].

The Diffie-Hellman is vulnerable to man in the middle attack since there is no authentication taking place. There exist a solution to this problem using asymmetric key pairs and signing the messages being sent [37].

2.1.2.2 Key Derivation function

Assume a secret key is established between two parties and is used to encrypt messages and exchange them over an insecure channel. An adversary listening might store all the messages being send even though he is not able to read them. However at some point he manages to compromise the secret key hence being able to decrypting every message ever sent.

To overcome the above scenario ephemeral keys are used. Such keys are short lived and are discarded after use.

New secret keys can be generated using a *Key Derivation Function* (KDF). A KDF is a one way function that derives one or more randomized secret keys based on a secret key (or multuple) and optionally some input value [37]. Figure 2.5 illustrates this.

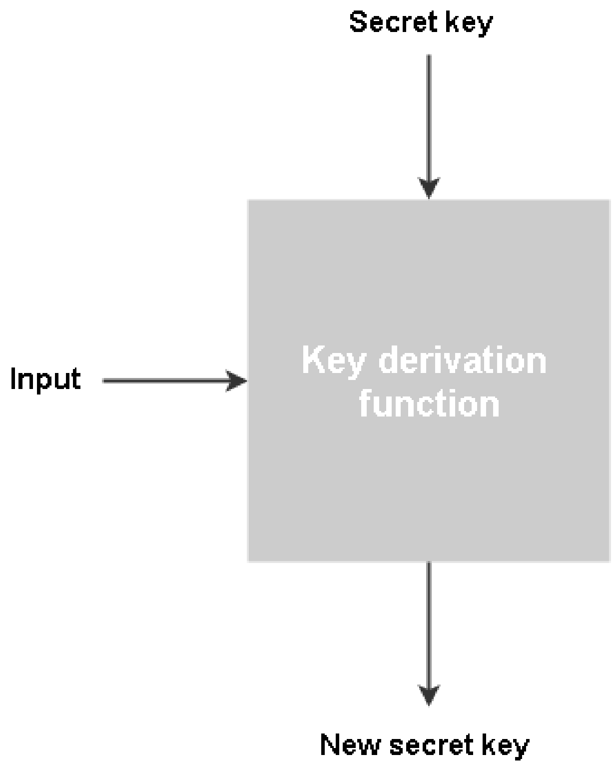


Figure 2.5: Key derivation function

A core concept in the Double Ratchet algorithm is KDF chains which build a chain of secret keys using KDF [36].

2.1.2.3 Signal Protocol

The Signal Protocol provides end-to-end encryption and was developed in 2013 and was first introduced in the app TextSecure[2].

The Signal Protocol consists of two parts; The *Triple Diffie-Hellman protocol* (TripleDH) and the Double Ratchet algorithm.

Triple Diffie-Hellman protocol

Before the Double Ratchet algorithm can be used the two parties communicating need to agree on a shared secret key. In the Signal protocol this is achieved with Triple Diffie-Hellman protocol *(TripleDH)*.

The Triple Diffie-Hellman protocol is a *key agreement protocol.* It involves a server and two parties; Alice and Bob.

The TripleDH protocol is characterized by three phases:

1. *Publishing keys:* A identity key and several prekeys belonging to Bob is published by him to a server.

[2] https://en.wikipedia.org/wiki/TextSecure

2. *Sending initial message:* Alice sends an initial message to Bob. A prekey bundle is obtained by Alice from the server in order to send an initial message to Bob.

3. *Receiving initial message:* Alice's message is received and processed by Bob.

Publishing keys Bob needs to register a *prekey bundle* to the server if he wants Alice to be able to send him messages. Alice will likewise have registered a prekey bundle so anyone can to anyone wants to start a message conversation with her. The prekey bundle exists of:

- Identity key IK_B. This key is only published once by Bob.
- Signed prekey SPK_B. This key is reuploaded again after some period of time (eg. after each week or each month).
- Prekey signature $Sig(IK_B, Encode(SPK_B))$. This key is also reuploaded again like the signed prekey.
- Set of one-time prekeys $(OPK_B{}^1, OPK_B{}^2, OPK_B{}^3, ...)$. These keys are uploaded by Bob occasionally. Bob is informed by the server when there are few one-time prekeys left.

  To ensure forward secrecy the private key of the one-time prekeys are deleted once Bob received messages that uses them. The signed prekey is deleted as well. However Bob might hold on to it for some time to get the messages that was delayed.

Sending initial message Alice retrieves Bobs public keys from the server. She receives one of Bob's single one-time prekey. The server deletes the one-time prekey that was send. It might be the case that all the one-time prekeys at the server has been used [29].

Alice verifies the prekey signature if the verification fails the protocol is aborted or else the following public keys are provided to generate a shared secret:

- Identity key IK_A. Her own identity key.
- Ephemeral key EK_A. The public key from a generated ephemeral key pair.

To generate the shared secret the following calculations are made:

$$DH_1 = DH(IK_A, SPK_B)$$
$$DH_2 = DH(EK_A, IK_B)$$
$$DH_3 = DH(EK_A, SPK_B)$$
$$DH_4 = DH(EK_A, OPK_B)$$
$$SK = KDF(DH_1||DH_2||DH_3||DH_4)$$

There are performed atleast three Diffie-Hellman where DH_4 is optional depending on if the server had more one-time prekeys.

Authentication is provided by DH_1 and DH_2 while DH_3 and DH_4 provides forward secrecy.

The figure 2.1.2.3 illustrates the calculations.

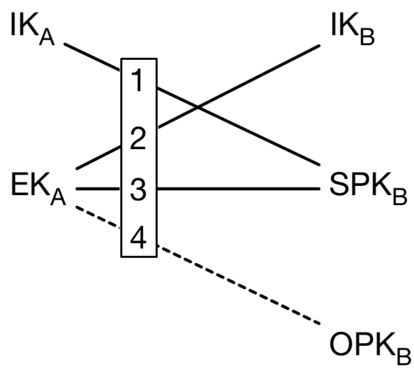


Figure 2.6: Calculations of DH1-DH4 [29].

Alice then deletes the DH values and her private key associated to the ephemeral key to uphold forward secrecy and sends the initial message to Bob which consists of [29]:

- Cypher text with AEAD encryption [3] with *associated data* using her own and Bob's identity keys.
- Her identity key IK_A.
- Her ephemeral key EK_A.
- Some identifiers to which of Bob's prekeys she used.

Receiving initial message When Bob receives Alice's initial message he performs the exact same calculations in phase two and derives the same shared secret key *SK*.

Bob then decrypts the cypher text with the shared key and *associated data* using his own and Alice's identity keys. If the decryption fails the protocol is aborted and *SK* is deleted. Otherwise the protocol is complete and Bob deletes the one-time private prekey. The shared secret key can then be used for the Double Ratchet algorithm [29].

Double Ratchet algorithm

After a shared secret key has been established the Double Ratchet algorithm can then be used to send and receive encrypted messages.

Each party has three chains; root chain, sender chain and receive chain. The chains are KDF chains and will take two keys as input (a KDF chain key and some other input key) and output new two keys (a new KDF chain key and some other output key). The KDF chain is illustrated in figure **??**.

The algorithm has a *Diffie-Hellman ratchet* step and *symmetric ratchet* step and the chains are used across both steps.

- *Diffie-Hellman ratchet:* The parties exchanges new Diffie-Hellman public keys with the messages being sent. New secrets are then derived using

[3] https://en.wikipedia.org/wiki/Authenticated_encryption

Diffie-Hellman (DH). The secret that DH outputs is used as input to the root chain. The root chain then output new chain keys for the receiving and sending chains.

- *Symmetric ratchet:* The sending and receiving chains uses the chain keys derived from the root chain and for each message sent and received the chains are advanced. The output from the receiving and sending chains are keys for encrypting or decrypting messages.

Symmetric ratchet The symmetric ratchet provides message key through the receiving and sending chains. A message key is used for encryption or decryption of a message.

In the symmetric ratchet a single ratchet step is the calculation of the next key chain and message key. The inputs are the current chain key and a constant. Figure 2.7 illustrates two steps in a symmetric ratchet.

Forward secrecy is provided since KDF is a one-way function and it is not possible to go backward and get the input chain key from the output chain key. However since the other input is simply a constant all future keys chain keys and message keys can be derived from an older chain key more specifically there is lack of backward secrecy.

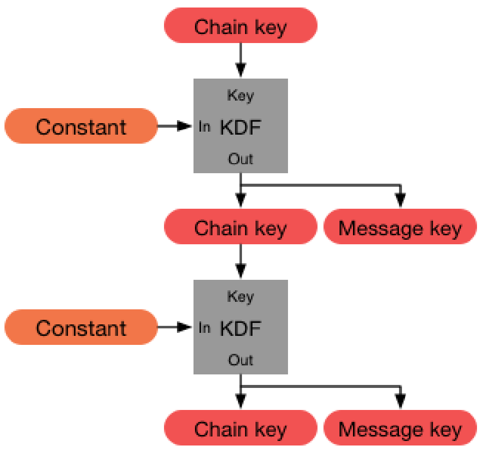


Figure 2.7: Symmetric key ratchet [36].

Diffie-Hellman ratchet Double ratchet provides backward secrecy by coming the Symmetric ratchet with a Diffie-Hellman ratchet hence the name *Double Ratchet*.

Every message from either party begins with a header which contains the sender's current ratchet public key. Whenever a new ratchet public key is received a new ratchet key pair is generated; a secret is derived through Diffie-Hellman with the input being the received ratchet public key and the ratchet private key from the new generated key pair.

Alice starts a conversation with Bob and uses his published public key as a ratchet public key. Alice then generates a new ratchet key pair and derives a shared secret key using Diffie-Hellman and would that as input to her *sending chain*. Alice then sends her new ratchet public key to Bob. At the receiving end Bob derives the same shared secret which would be the input to his *receiving chain*. Alice's sending chain and Bob's receiving chain share the same secret hence he can derive the message key and decrypt the message sent from Alice. When Bob sends a reply to Alice he would generate a new ratchet key pair and derive a new secret which would be input to his *sending chain*.

Figure 2.8 shows an ongoing message exchange with new secrets being derived and the sending and receiving chains being advanced.

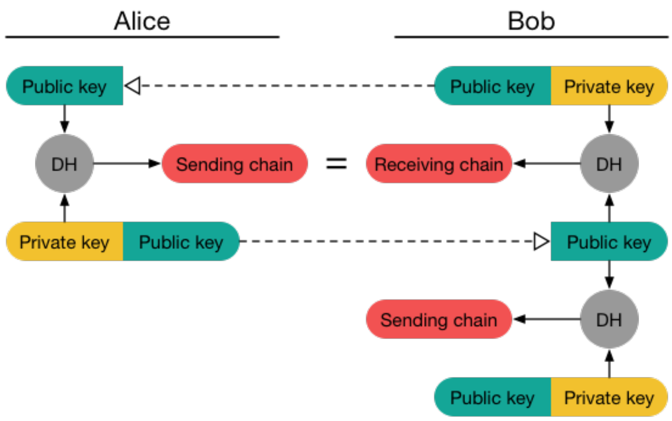


Figure 2.8: Diffie-Hellman ratchet [36].

When Alice receive the reply from Bob she would perform the exact same steps. This ultimately results in a continous loop of generating new ratchet key pairs and using Diffie-Hellman to derive the same shared secret key. A continuation is shown in figure 2.9

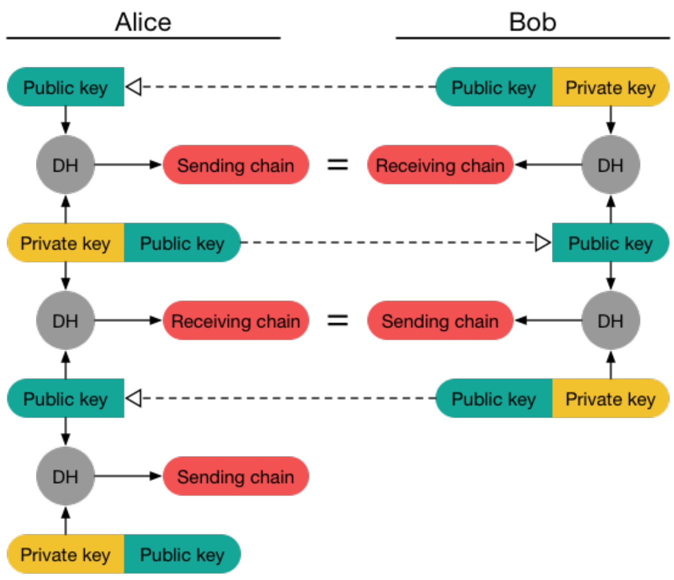


Figure 2.9: Continuation of Diffie-Hellman ratchet [36].

As mentioned in the beginning of the Double Ratchet section the Diffie-Hellman ratchet does have a root chain which would provide inputs to the sending and receiving chains. A more correct view of the process in Diffie-Hellman is shown in figure 2.10.

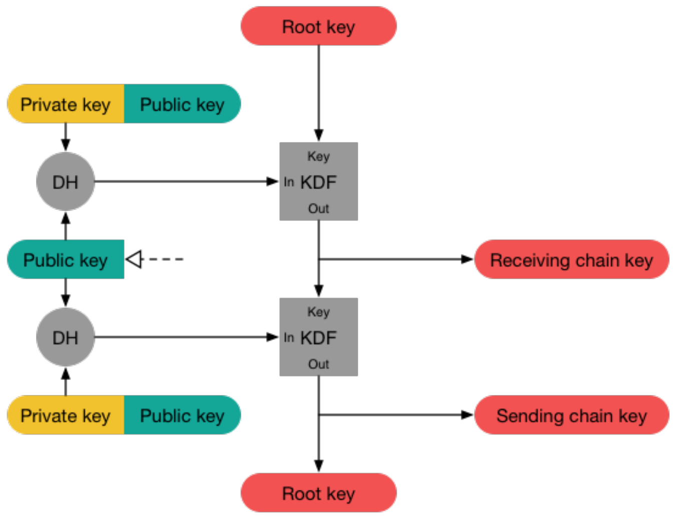


Figure 2.10: Diffie-Hellman ratchet 7 [36].

Double ratchet

When combining Diffie-Hellman ratchet and symmetric-key ratchet the result is the Double ratchet.

- When sending or receiving a message the corresponding message key is derived by performing a symmetric-key ratchet step.
- Upon receiving a new ratchet public key the Diffie-Hellman ratchet step performed right before the symmetric-key ratchet step with the goal of replacing old chain keys with new ones.

Assume that the message exchanged is a continuation from the TripleDH key exchange described in section 2.1.2.3. Alice had sent an initial message. The initial ratchet public key would be Bob's signed prekey SPK_B and the new ratchet key pair would be the Alice's ephemeral key pair that she generated. Alice calculated a shared secret which is the *root key*. She then generates a new ratchet key pair and takes the output from Diffie-Hellman and use it as input for the *root chain*. The root chain then outputs a new root key RK and a sending chain key CK.

The figure **??** depicts this with a view of Alice's chains.

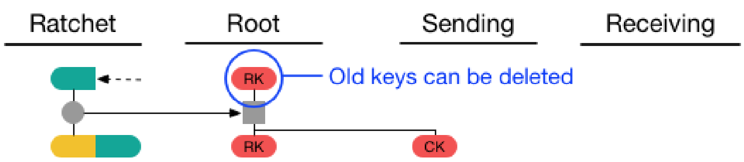


Figure 2.11: Double ratchet 1 [36].

When Alice then sends a message *A1* the symmetric-key ratchet step will return a new chain key and a message key. The message can then be encrypted with the message key.

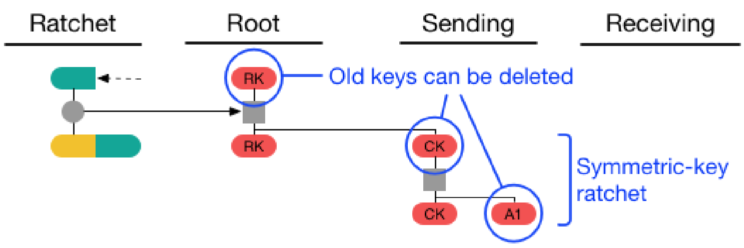


Figure 2.12: Double ratchet 2 [36].

Next Alice receives a message *B1* from Bob. The message header contains a new ratchet public key and a Diffie-Hellman ratchet step is performed. New sending and receiving chain keys are derived and followed by a symmetric-key ratchet step to derive the receiving message key to decrypt the message.

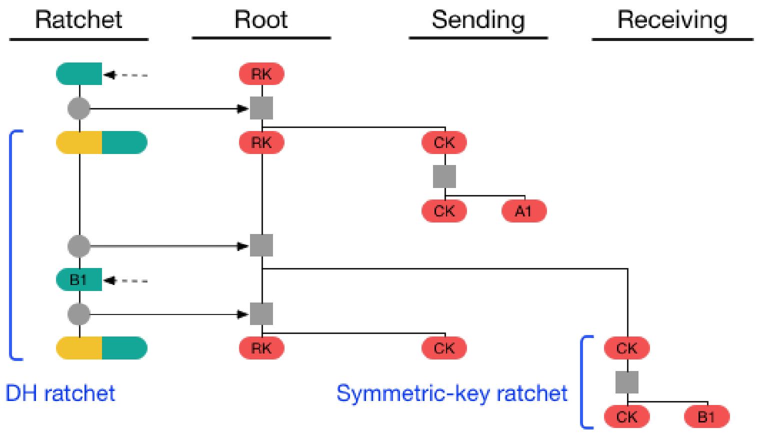


Figure 2.13: Double ratchet 3 [36].

2.2 Matrix

Matrix is an open standard protocol for messaging over HTTP and synchronizing data. Matrix provides secure real-time communication over a decentralized federated network with eventual consistency. Matrix cover use cases such as instant messaging, VoIP, Internet of Things communication and is generally applicable anywhere for subscribing and publishing data over standard HTTP API.

2.2.1 How does it work?

Matrix defines a conceptual place *room* where data can be published and subscribed to. A room is shared and replicated among multiple *homeservers*. The example shown in figure 2.14 is a conversation in a room between three clients on different homeservers.

In the example Alice starts by sending a message to the room. When Alice sends a message it is send to her homeserver. Each homeserver stores messages in a *directed acyclic graph* called an *event graph*. The message send by Alice is added to the event graph and is linked to most recent message(s) in the graph. The message is then signed with the signtatures of all previous messages by Alice's homeserver in order to make it tamper-proof. Finally the homeserver relays the message to the Bob's and Charlie's homeservers.

Upon receiving the message the other homeservers validates the message and then adds it to their own event graph. The message now persists in Bob's and Charlie's servers and can be retrieved from Bob's and Charlie's clients.

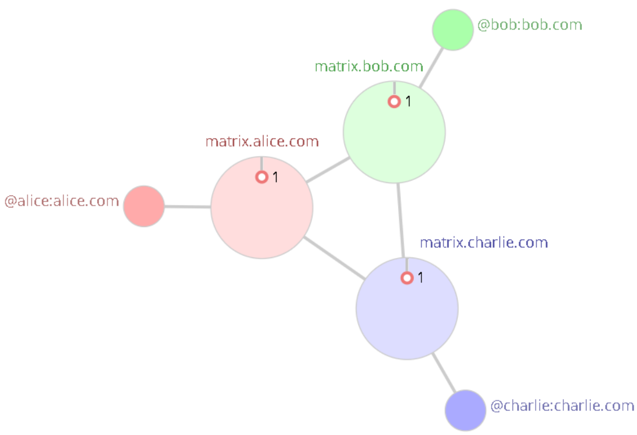


Figure 2.14: Matrix [11].

Bob replies to Alice's message and is first send to the homeserver and linked to Alice's message in the event graph. At the same time Charlie also send a reply and an inconsistency occurs as depicted in figure 2.15.

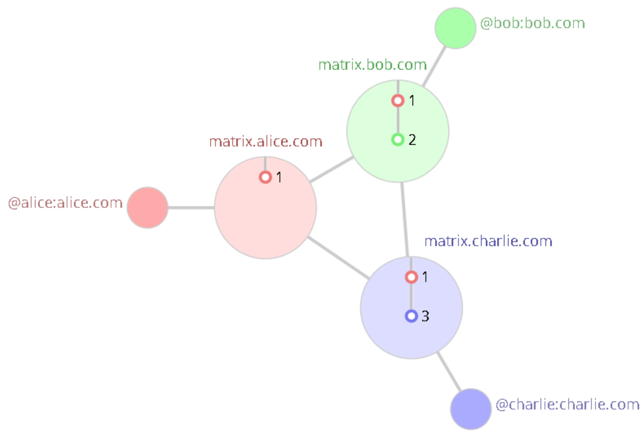


Figure 2.15: Matrix [11].

Bob's message arrives first to Alice and Charlie's homeservers. Alice adds the message to her own event graph and has a view that is consistent with Bob's. Charlie adds the message to the event graph; Bob's message precedes Alice's message hence message 2 and 3 are linked to message 1 shown in figure 2.16;

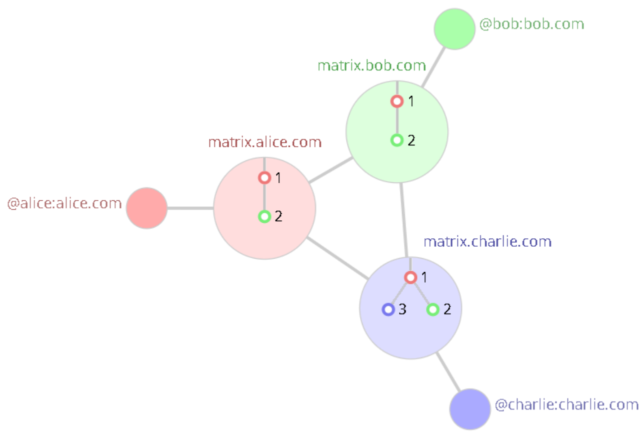


Figure 2.16: Matrix [11].

The other homeservers then receive Charlie's message and is added to their event graph. The room is yet again in sync and all the homeservers have a consistent view of the room.

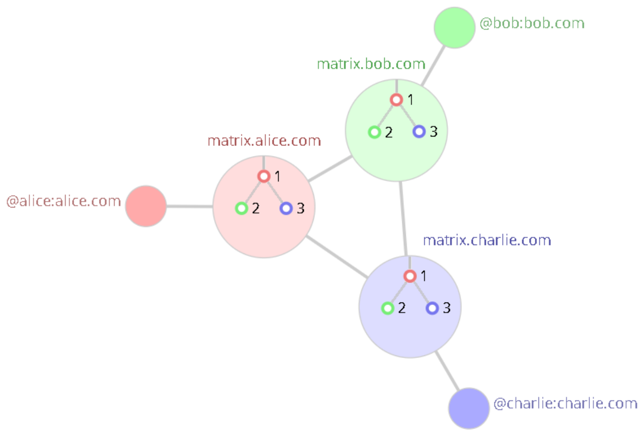


Figure 2.17: Matrix [11].

Alice sends a new message and is linked to the most recent unlinked objects (both Bob's and Charlie's). The message is then relayed to the other homeservers. The split in the event graph is merged. This example also shows how Matrix provides eventual consistency.

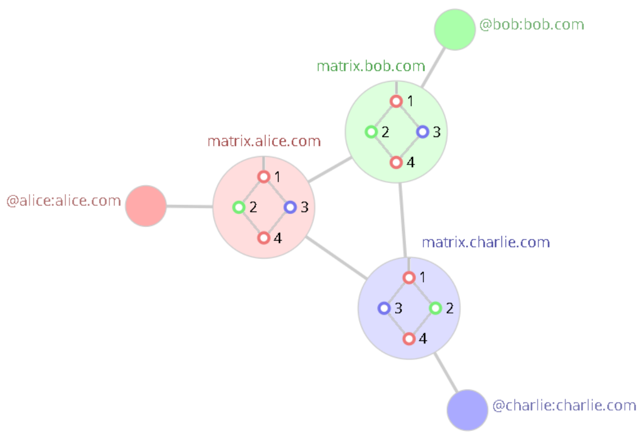


Figure 2.18: Matrix [11].

2.2.2 Architecture

The previous section introduced the notion of a *room* which clients sends messages to. The messages being in Matrix are actually JSON objects called *events*. The events can be of any kind of structure hence it is not limited to messaging. Events are stored at homeservers and the communication history for a room is modelled using *directed acyclic graph* called *event graphs*. Matrix provides a specification for Client-Server API which is used for sending and synchronizing events between the client and its belonging homeserver. Matrix also provides specification for a Server-Server API which synchronizes data among homeservers with eventual consistency. The synchronization process between homeservers is defined by the term *Federation* [12].

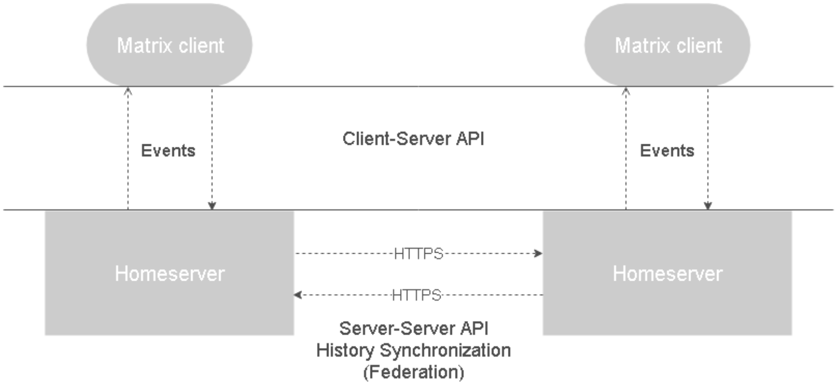


Figure 2.19: Matrix conceptual architecture [12].

As previously mentioned a room is a conceptual place for sending and receiving events. Each room is uniquely identified by a *Room ID*. The figure 2.20 shows how events are send and received from a room.

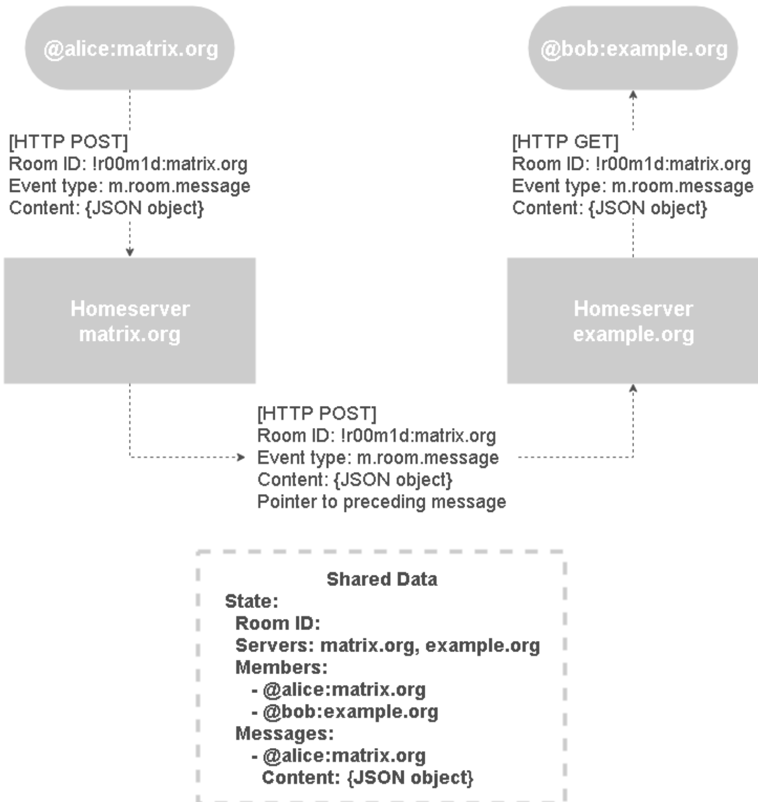


Figure 2.20: Matrix conceptual architecture [12].

2.2.2.1 Event

Every data exchange in Matrix is an event. Events are used to express state changes in a room as well as messages being sent. An event has a type used to differ between what kind of event is being sent and what kind of data it might hold. Matrix comes with reserved name spaces; *m.room.message* is a type for instant messaging. The following is an example of a event of type *m.room.message* [12].

```
{
  "content": {
    "body": "Hello world!",
    "msgtype": "m.text"
  },
  "room_id": "!wfgy43Sg4a:matrix.org",
  "sender": "@bob:matrix.org",
  "event_id": "$asfDuShaf7Gafaw:matrix.org",
  "type": "m.room.message"
}
```

The specification is open for defining custom types which can express any kind of data one might want to exchange [15].

2.2.3 Matrix specification

Matrix has two main API specifications; Client/Server API and Federated API. Any Matrix SDK implements the API defined at the Client/Server API specification. If a custom homeserver was to be developed from scratch it would have to conform to the Federated API to be able to be a part of Matrix.

Client/Server API For clients to send and receive messages the Client/Server API is used. The API mainly provides specification for:

- Sending and receiving messages
- Configure rooms
- Synchronize historical conversation

There exist several SDKs implementing the API.

2.2.4 End-to-end Encryption

Security is a high priority for Matrix design. Especially with the decentralized architecture with data being replicated over a federation federation of servers. Matrix provides security guarantees through end-to-end encryption using Olm and Megolm cryptographic ratchet. Olm is based on the Double Ratchet algorithm and Megolm is an extension for secure group communication. Olm and Megolm are examined in section 3.1.3.

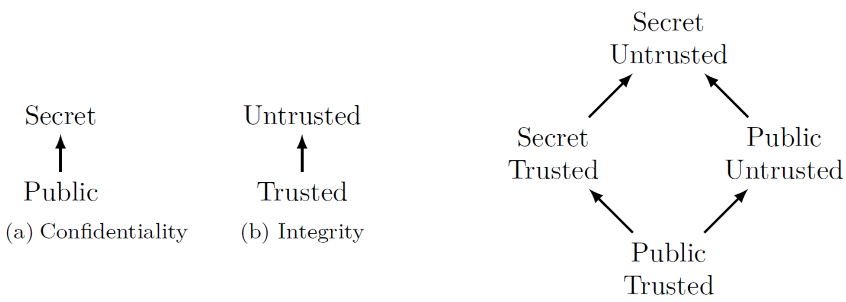
2.3 Information Flow Control

In the beginning of this chapter the security properties confidentiality and integrity were mentioned which are major security goals for a system. If there is *secure information flow* throughout the system then confidentiality and integrity is achieved [26]. Secure information flow means that only authorized flow of information is allowed [23]. There are two aspects to secure information flow; the reading of the information (confidentiality) and the writing of the information (integrity) [26].

Information Flow Control is a *security mechanism*[4] for achieving secure information flow. Information Flow Control is a language-based security technique that can enforce defined *security policies*[5] concerning confidentiality and integrity of data. It enables us to express where information may flow to and under what conditions.

2.3.1 Lattice model

By classifying information with *security levels* we can express where information may flow. Consider the following confidentiality policy; we classify *secret* as secret information, and classify *public* as public information. For the two secuirty levels it holds that flow from public to public is allowed, public to secret is allowed, and secret to secret is allowed. For preserving confidentiality flow from secret to public is under no circumstances allowed. This can be expressed as $public \leq secret$ and gives us a simple lattice[6] structure where information flows upwards shown in figure 2.21a [43].



(a) Confidentiality (b) Integrity (c) Product Lattice

Figure 2.21: Lattice model for confidentiality and integrity [18]

Likewise a policy for integrity can be specified where information from an *untrusted* source must not flow to information with a security level of *trusted* which is somewhat opposite of the confidentiality policy. Figure 2.21b illustrates an integrity lattice. By combining the two we get a more complex lattice show in figure 2.21c.

[4]"A security mechanism is a method, tool, or procedure for enforcing a security policy" [19]
[5]"A security policy is a statement of what is, and what is not, allowed." [19]
[6]https://en.wikipedia.org/wiki/Lattice_(order)

2.3.2 Explicit and implicit flow

The problem with mainstream programming languages is that they are unable to enforce defined policies such as the confidentiality and integrity policies described in the previous section. Consider the following flow:

```
public = secret;
```

This is an *explicit flow* and is where the *secret* value is directly copied into the *public* value which obviously is a violation of the confidentiality policy and is an insecure flow of information [26].

Another example of insecure information flow is *implicit flow*:

```
public = false;
if secret then public = true
```

In this example the *secret* value affects the control flow and some information is leaked about the *secret* value. Hence this also violates the confidentiality policy. When secret values can affect the control flow there is an implicit flow. An implicit flow is a type of *covert channel* [39].

2.3.3 Covert channels

The explicit and implicit flows can also be considered as *channels* that signal information [28] [39]. *Covert channels* are channels that are not meant to signal information but somehow leaks information [28]. The most prominent covert channels are:

- *Implicit channels* leaks information through the path the program takes in the control flow.
- *Termination channels* leaks information by considering if a program terminates.
- *Timing channels* leaks information by considering when an action occurs or how much time a program takes.

Other channel are *probabilistic channels*, *resource exhaustion channels*, and *power channels*. It is not necessarily all covert channels that are of concern and in the end depends on what is observable by an adversary [39].

2.3.4 Noninterference

Secure information flow can be expressed by the concept of *noninterference*. The notion of noninterference is that someone observing the public input and output of a system should not be able to learn anything about the secret input of the system. If the secret input interferes with the public output then there is information leak. The figure 2.22 illustrates noninterference [26].

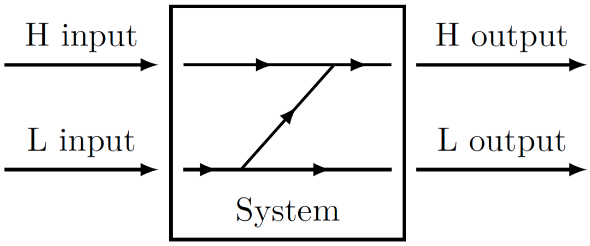


Figure 2.22: Noninterference [18]

The figure illustrates that the *L output* (public) should be independent of the *H input* (secret). In other words if a program is executed with fixed public input but with different secret input; then there should not be any changes to the public output [26].

Depending on what an observer is capable of observing; different variations of noninterference can be expressed. Two of such variants are *termination-insensitive* and *termination-sensitive*.

Termination-insensitive noninterference Termination-insensitive noninterference would guarantee that when a program terminates; the program's public output remains unaffected of the secret input. However it is possible to leak information through termination channels. By observing termination; 1 bit leaks can be achieved for each run of the program [26]:

```
if (secret == public) then while (true) do skip done
```

If the program does terminate then the observer has learned that the secret is not equal to the public value hence 1 bit of information is leaked.

Termination-sensitive noninterference The alternative is the termination-sensitive noninterference variant. The public output is independent of the secret input. Furthermore if the program is run with secret input s_1 and it terminates; then the program run with secret input s_2 would terminate as well. Hence if s_1 did not terminate then s_2 would not either [26].

This variant of noninterference would be achieved by not allowing secret variables as guards in loops and if-else-than statements. This variant would be more restrictive and disallow flows as such [26]:

```
while (secret == true) do ... done
```

2.3.5 Declassification

Noninterference is a very strict policy and not practical. Consider the classic example of a user logging into a system which violates noninterference. It is unavoidable that a login attempt gives some information away about the password. If the password is incorrect the login fails and partial information is given away. There is no problem with the above and is regarded as secure. Another example is sending encrypted data over an insecure channel; secret input is encrypted and the cipher text would then be send over an insecure channel. However this goes against noninterference since the secret input clearly interferes with public output [26].

Declassifying of information is necessary. Taking some specific information from a higher security class and changing it to a lower security classification is declassification. When declassifying information the following four dimensions needs to be addressed [41]:

- *What* information should be declassified. We would like to specify what information is released.
- *Who* can declassify the information. By specifying exactly who can release information it rules out that someone unspecified can make unintended leaks through declassification.
- *Where* the information can flow. The dimension describes which security levels the declassified information may flow to and also where in the code the declassification occurs.
- *When* the declassification occurs. The release of information is only allowed relative to some even e.g. only after a purchase can a software key be released.

2.3.5.1 Decentralized label model

Decentralized Label Model allows us to define flow policies and address declassification of information in a program. A policy is defined by adding *labels* to a value. A value can hold information for different *principals* called *owners*. A label L specifies an *owners* set *owners(L)*. Each owner can allow a list of principals called readers *readers(L,O)* that the information may be released to. *Effective readers* are readers that all owners agree on the information can be released to and is essentially those where information can flow to. An example of a label is $\boldsymbol{\{o_1{:}r_1,r_2;\ o_2{:}r_2,r_3\}}$; where o_1 and o_2 are the different owners with their specified readers [32]. The example L has the following:

$$owners(L) = \{o_1, o_2\}$$
$$readers(L, o_1) = \{r_1, r_2\}$$
$$readers(L, o_2) = \{r_2, r_3\}$$
$$effectiveReaders(L) = \{r_2\}$$

Such labeling makes it possible for each owner to have an independent flow policy and give control of where the information may flow. Declassification is possible if the program detects it is running as the authority of one of the owners [32] [31]. This is known as the *principal hierarchy* and can allow a principal to act for another principal.

2.3.6 Information-flow enforcement

There exist two general techniques for enforcing secure information flow; *static analysis* through a type system and *dynamic analysis* through a monitor. Both techniques give the assurance of termination-insensitive noninterference [40]. The static analysis have the advantage that it does not have the runtime overhead as the dynamic analysis while the dynamic analysis is more permissive [40].

2.3.6.1 Static enforcement

Enforcement of information-flow through static analysis is done using type systems. A simple type system is presented and shown in figure 2.23.

$$pc \vdash \texttt{skip} \qquad \frac{lev(e) \sqsubseteq \Gamma(x) \qquad pc \sqsubseteq \Gamma(x)}{pc \vdash x := e} \qquad \frac{pc \vdash c_1 \qquad pc \vdash c_2}{pc \vdash c_1 ; c_2}$$

$$\frac{lev(e) \sqcup pc \vdash c_1 \qquad lev(e) \sqcup pc \vdash c_2}{pc \vdash \texttt{if}\ e\ \texttt{then}\ c_1\ \texttt{else}\ c_2} \qquad \frac{lev(e) \sqcup pc \vdash c}{pc \vdash \texttt{while}\ e\ \texttt{do}\ c}$$

Figure 2.23: Typing rules [40]

We assume that the security lattice has the levels L for low (public) and H for high (secret). Γ denotes a typing environment that takes a variable x and maps it to a security level. The function *lev(e)* takes an expression e and returns a security level; if e holds a high variable then H is returned or else L is returned. The security level of the context is kept tracked of by the program counter *pc*. The typing judgment for commands is denoted by $pc \vdash c$ [40].

The typing rule $pc \vdash x := e$ is for assignment; it prevent assignment of a expression that holds a high variable to a low variable [40]. Hence the explicit flow would be detected by the type system:

```
low = high
```

Furthermore implicit flow are detectable through the program counter *pc*. If there is a high guard then the *pc* has the security level H and would expect assignments of high variables hence preventing low assignments [40]. The implicit flow would be detected by the typing system as well:

```
if high then low = true else low = false
```

2.3.6.2 Dynamic enforcement

Dynamic analysis is provided with *monitors*. Monitoring can only consider one path when running. This has led to the belief that the dynamic approach falls short compared to the static enforcement. However both dynamic enforcement and static enforcement achieve termination-insensitive noninterference hence they give the same security assurance [40].

$$st \xrightarrow{nop} st \qquad \frac{lev(e) \sqsubseteq \Gamma(x) \qquad lev(st) \sqsubseteq \Gamma(x)}{st \xrightarrow{a(x,e)} st} \qquad st \xrightarrow{b(e)} lev(e) : st \qquad hd : st \xrightarrow{f} st$$

Figure 2.24: Monitoring rules [40]

The figure 2.24 shows monitoring rules for a monitor. Programs generates events and the monitor chooses to accept an event or block it by halting. Security levels are tracked by stack *st* which is called a *monitor configuration* and serves the same purpose as the program counter; keeping track of the current security context. The monitor can execute following *events*:

- *nop event:* signals a skip and the monitor always this event. It does not change the state of the monitor.
- *assignment event a(x,e):* assigns the value of expression e to variable x. It does not change the state of the monitor but the monitor has the following two conditions:
    1. $lev(e) \sqsubseteq \Gamma(x)$: the expression e's security level is equal or lower than variable x's security level.
    2. $lev(st) \sqsubseteq \Gamma(x)$: the highest security level in the stack is equal or lower than variable x's security level
- *branching event b(e)*: branches on e and changes the state of the monitor. The security level of e is pushed on the stack.
- *event f*: signals are loop or if-else statement has finished evaluating. The security level pushed to the stack *st* would be popped when evaluation is finished.

The first condition for the assignment event prevents explicit flows. The second condition prevents implicit flows as such:

```
if high then low=true else low=false
```

The mechanism related to the branching event aids in preventing implicit flows by pushing the security level of *st* to the stack. Now consider the following code:

```
if high then low=true else skip
```

If high is *false* then the code would continue however if high is *true* the execution would be stopped. Hence 1 bit would be leaked and is acceptable in context of termination-insensitive noninterference.

2.4 Summary

The background section introduces security properties necessary for a secure system. Several other security properties are described relevant for a secure messaging system which is relevant for the section 3.1 where Matrix security is evaluated. This section also presented a description of the Signal Protocol necessary for the evaluation as well. Furthermore Matrix and its architecture was described and finally the basic concepts of Information-Flow Control was presented.

3 Analysis

This chapter consists of two parts. The first part will provide an evaluation of the Matrix security model and relies on the paper *SoK: Secure Messaging* [46] and *The Olm Cryptographic Review* by NCC Group [34].

The second part provides a preliminary analysis of the IFC tools, the selection of Paragon and the rationale behind it, and a further analysis of the selected tool Paragon.

3.1 Evaluation of Matrix security model

The security of matrix will be evaluated in the context of secure messaging. An evaluation framework has been proposed in the paper *SoK: Secure messaging* which the evaluation will be loosely based on.

The evaluation framework covers several areas with *conversation security* being the most relevant for this evaluation. The area *conversation security* describes three categories; *Security and Privacy*, *Adoption*, and *Group Chat*. Obviously the most relevant category for the evaluation is *Security and Privacy*

3.1.1 Threat model

For secure messaging the evaluation framework defines a threat model with three types of adversaries. Note that an adversary can be of several types:

- *Local adversary:* The adversary is in control of the local network.
- *Global adversary:* The adversary is in control of great portions of the Internet
- *Service providers:* A potential adversary for messaging systems with centralized infrastructure.

In the messaging system the adversary may be a participant with the following properties:

- An adversary can start a conversation.
- An adversary can send messages.
- An adversary can perform any other action that a participant is capable of.

Furthermore it is assumed that the system's endpoints are secure [46].
This evaluation will inherit the described threat model.

3.1.2 The Signal Protocol

Matrix provides end-to-end encryption by using the Olm and Megolm library with the former being an implementation of the Double Ratchet algorithm also known as the Signal Protocol, and the latter being the algorithm used for group chat.

Olm is used for securely exchanging message keys/session keys during group chat and is vital part of the end-to-end encryption in Matrix.

Before the Matrix protocol is evaluated the Signal Protocol will be considered. The Signal Protocol is described in section xx.

Section xx provides a list of security properties relevant for *conversation security*. These security properties is used for evaluating a secure messaging protocol such as the Signal Protocol.

The table below shows an evaluation of the Signal Protocol (previously known as TextSecure) [46].

Scheme	Example	Security and Privacy														Adoption					Group Chat				
		Confidentiality	Integrity	Authentication	Participant Consistency	Destination Validation	Forward Secrecy	Backward Secrecy	Anonymity Preserving	Speaker Consistency	Causality Preserving	Global Transcript	Message Unlinkability	Message Repudiation	Particip. Repudiation	Out-of-Order Resilient	Dropped Message Resilient	Asynchronicity	Multi-Device Support	No Additional Service	Computational Equality	Trust Equality	Subgroup Messaging	Contractable	Expandable
+Double Ratchet+3DH AKE+Prekeys†*	TextSecure	●	●	●	●	●	●	●	-	◐	●	-	●	●	●	◐	◐	●	-	-					

Figure 3.1: Evaluation of Signal (TextSecure) [46].

Confidentiality When a message is sent using the Signal Protocol then only the intended recipient can read the message. The senders sending ratchet and receivers receiving ratchet will derive the same message key hence only the two parties will be able to encrypt the messages.

Integrity The receiver will only accept a message if it is successfully decrypted hence if in transit a message was modified then the message would be rejected.

Authentication The decryption of a message also gives authentication guarantees since only the intended recipient could compute the message key.

Forward secrecy The symmetric ratchet ensures forward secrecy. If a chain session key is compromised then the previous keys can not be generated since the ratchet is one way cryptographic hash function hence secrecy is provided for all previous send messages.

Backward secrecy Diffie-Hellman ratchet have the self-healing property and will generate a new chain session key for the symmetric ratchet hence if a chain key is compromised then secrecy for future messages is still provided because a new chain ratchet key will be generated.

Anonymity preserving Anonymity preservation is lost in the Signal Protocol since the initial key agreement requires long-term public keys hence making them observable during Triple-DH. However ***participant consistency*** is provided by Triple-DH [46].

Speaker consistency This property is partially provided through the key evolution of the ratchets. If a message is dropped then it is not possible to generate message keys for future messages. This also makes the protocol have the property ***Causality Preserving*** and partially have the property ***Dropped message resilence***. It will also not go unnoticed if a message is received out of order since this will result in the message's key being an unexpected key. Hence the recipient have to store expired keys to decrypt delayed messages. This makes the property ***Out-of-order resilient*** only partially provided [46].

Global transcript In an asynchrounous messaging protocol there is no global transcript. Both participants have to be online to receive messages hence the participants will not have all the messages if one of them is offline. This is a result of having the ***Asynchronicity*** property.

Deniability properties Since the ratchet session keys are used for encrypting messages and not the long-term public keys the properties ***Message unlinkability*** and ***Message repudiation*** are provided.

Other properties

- ***Participant repudiation***. Triple-DH achieves full participant repudiation since anyone can forge a transcript between any two participants [46].
- ***Destination validation***. The Deffie-Hellman ratchet provides this property since the recipients public key is used to generate the chain key [46].

The evaluation shows that several security properties are provided with the important ones being confidentiality, integrity, authentication, forward secrecy, backward secrecy.

Furthermore a formal analysis have been made on the Signal Protocol that proves the protocol is free from any major flaws and it satisfy the following security properties; confidentiality, authentication and secrecy [22].

Application variants

The Signal Protocol is a secure messaging protocol and have been extensively studied including proof that the standard security properties are assured.

The Olm library used by Matrix is a variant of the Signal Protocol. There is no implementation analysis of the Olm library hence there is no guarantee that all the security properties defined in xx is inherited by Olm. Nevertheless it is assumed that Olm inherits the above properties.

The further evaluation relies upon the the security assessment of Matrix.

3.1.3 Matrix protocol

As described in section xx *rooms* are a fundamental part of Matrix' architecture. There can be multiple participants in a room hence the support for secure group conversation is required.

Olm (and the Signal Protocol it is based on) is ideally meant for two party communication. Group conversation could be supported with a näive variant of Olm. In a group with N participants each participant would establish a secure Olm session with every other participant. When a message is send each message would then have to be encrypted N times. This solution would scale poorly if N was a large number. This was the motivation for introducing Megolm.

Megolm

Megolm is a multicast encryption solution [46].Each sender has a sender ratchet (Megolm Ratchet). Each recipient has a corresponding receiving ratchet for each sender. So if there are N participants in a group then each participant will have N-1 receiving ratchets. Figure 3.2 illustrates the setup with three participants.

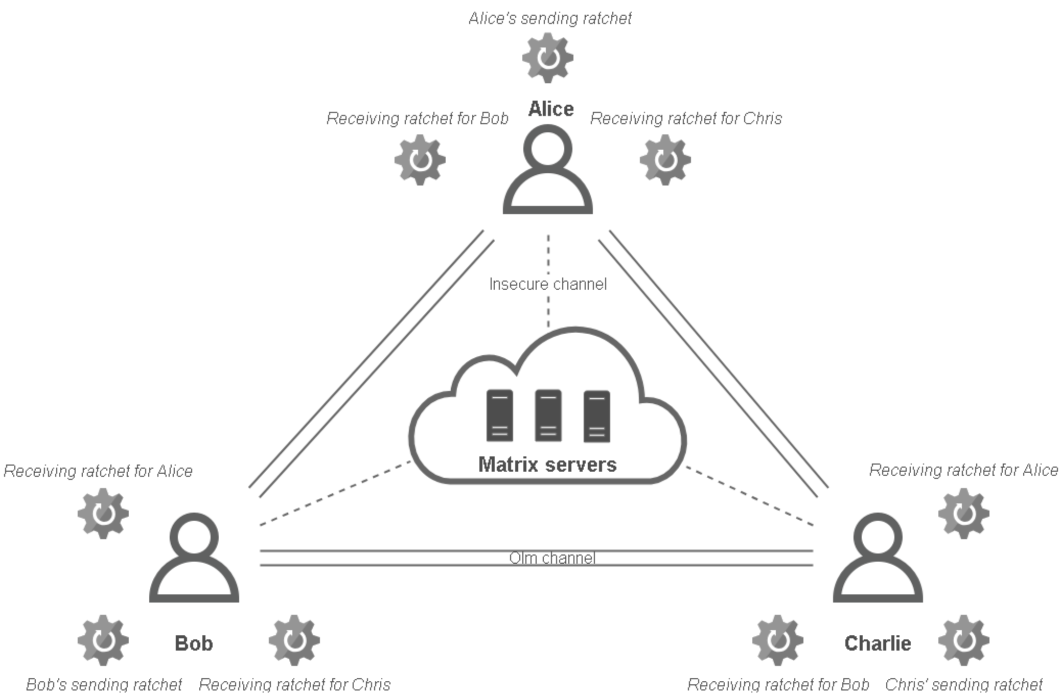


Figure 3.2: Conceptual model of Megolm with three participants.

When a session is started a sender will send his initial ratchet key to each recipient, so that the sender ratchet and each recipients ratchet are in sync. This key exchange happens over a secure communication channel (Olm). Furthermore there is send N-1 initial messages when a session is initiated. Until a new session is started no further session keys are exchanged and the corresponding message keys are generated by incrementing the ratchet.

When a sender sends a message a message key is generated from the ratchet key and the message is encrypted using that message key. The message will be signed so the recipient will know which sender the message is from and which ratchet to utilize. The message is then send to the server which relays the message to all recipients over an insecure channel. When they receive the message the same message key is generated using the corresponding receiver ratchet and the message is decrypted.

When a new participant joins the latest ratchet key would then be shared by each participant over Olm (or an earlier one if he should have access to historical conversation).

When a participant leaves a new session would be initiated yielding in refreshing the ratchet keys hence not making it possible for that ex-participant to decrypt any further messages.

The Matrix Protocol will be evaluated in the context of Megolm. The evaluation of the Matrix protocol heavily relies on the security assessment by NCC.

3.1.3.1 Evaluation

The Matrix Protocol provides several security properties shown in the table xx.

It is worth mentioning that there is a trade-off between security and usability which must be decided at application layer. The most secure configuration would come at the cost of usability and performance.

- *Usability.* From a users point of view it would be nice to have the possibility to load historical conversation instead of having to keep full history locally. Matrix supports multiple devices and if a participant adds another device at some later point it makes sense to load the participants historical conversation into the device. From a security perspective this would mean that the *initial ratchet state* is stored and is send to the new device so every message key can be generated. This certainly goes against the principle of forward secrecy. The most secure configuration would not store the *initial ratchet state* hence satisfy forward secrecy thus disable the described usability feature [34] [**?**].

- *Performance.* When a megolm session is initialized there is an initial burst of messages to exchange the initial ratchet key which is then stored in a *initial ratchet state* value at each recipient. If this key is compromised then any future key can be generated for that session. To satisfy backward secrecy this would mean initiating a new session for each message which would trigger a burst of messages to exchange the ratchet key [34] [**?**]. This would scale poorly for a large group or when sending large-sized messages.

Protocol	Security and Privacy													Adoption					Group Chat				
	Confidentiality	Integrity	Authentication	Participant Consistency	Destination Validation	Forward Secrecy	Backward Secrecy	Anonymity Preserving	Speaker Consistency	Causality Preserving	Global Transcript	Message Unlinkability	Message Repudiation	Particip. Repudiation	Out-of-Order Resilient	Dropped Message Resilient	Asynchronicity	Multi-Device Support	No Additional Service	Computational Equality	Trust Equality	Subgroup Messaging	Contractable	Expandable
Matrix	●	●	●	●	●	◑	◑	-	●	●	-	●	●	●	●	●	●	●	-	●	●	●	●	●

● = provides property ◑ = partially provides property

Figure 3.3: Evaluation of Matrix Security.

Some of the security properties in the table are briefly examined.

Confidentiality When a message is send it is encrypted and can only be decrypted by the intended recipients who has the corresponding ratchet session key received over an Olm channel.

Integrity The receiver will only accept a message if it is successfully decrypted hence if in transit a message was modified then the message would be rejected.

Forward secrecy Each participant keeps a *initial ratchet state* which holds the earliest ratchet session key for a session. This clearly violates forward secrecy since every message can be decrypted if the *initial ratchet state* value is compromised. However it is a deliberate trade-off for usability to enable historical conversation and storing the value is optional. Since this is an optional feature the forward secrecy is partially provided [34].

Backward secrecy If a ratchet key is compromised then an adversary can generate every message key from that point on hence intercept any message that sender sends to the group. This can be prevented strictly by starting a new session with every send message however it would not be possible to keep conversation history (only locally when data is encrypted). Hence the property is only partially provided [34].

Speaker consistency There is no guarantee for speaker consistency. A well known problem of multi-cast encryption group chat is transcript inconsistency. A sender may send different messages to different recipients. However it requires that the server is in collusion with the sender. This also applies to ***Causality preserving*** [34].

Other properties

The multi-cast encryption design does not provide *participant consistency* [46].

The properties *Dropped message resilence* and *Out-of-order resilient* are provided by keeping track of ratchet indices.

Several properties are inherited from the secure key exchanging channel provided by Olm while other properties are inherited because of asynchronicity of the Megolm protocol.

- *Authentication* is provided by Olm since the ratchet session key is send to the recipient through an Olm channel or else the message key could not be derived.
- *Destination validation.* The ratchet session key is exchanged over a secure Olm channel hence only the intended recipient could decrypt it.
- *Anonymity preserving* is not provided since Olm requires the long-term public key in the initial key exchange.
- *Global transcript* is also not provided because of the asynchronous nature of the Megolm protocol.
- *Asynchronicity* is obviously provided.
- *Deniability* properties are inherited from Olm as well.

All properties related to group chat are also provided. Although they are additional features and not related to security.

Other findings

Message Replays Matrix allows decryption of a message multiple times hence it is vulnerable to replay attacks. Replay attacks are handled at the application layer. Whenever a message is decrypted a message index is generated and stored. If the exact message is decrypted again the same message index will be generated and can be compared to the stored message index making the replayed message invalid.

Unknown key-share attack The *Unknown key-share attack*[1] is a vulnerability found with a high risk in Megolm. The vulnerability is inherited from Olm and occurs after the initial message in Triple-DH.

The vulnerability has been mitigated at the application layer by providing a unique identifier for the sender and receiver into each message and then checking the values when decrypted [34].

Recent research

The way backward secrecy would be provided in Matrix is computationally expensive. Recent research has proposed solutions with early implementations for these problems with IETF leading the research on the standard on *Messaging Layer Security*. Matrix has expressed awareness of the protocol and a possibility of adaption in the future.

[1] https://en.wikipedia.org/wiki/Unknown_key-share_attack

3.1.4 End-to-end security

Section xx describes Matrix long-term goal as being a generic HTTP messaging API. It could be utilized for any kind of data exchange in a system or between multiple systems.

A system using the Matrix Protocol for exchanging data would benefit from the security properties found in the evaluation yet the system-wide security or end-to-end security would be incomplete an further measures must be taken. Such system might demand confidentiality and integrity throughout the system yet the system as a whole would have a different threat model than the one described for a secure messaging system hence no guarantee of confidentiality or integrity beyond the endpoints in end-to-end encryption.

The following figure depicts how end-to-end encryption might be inadequate in such system.

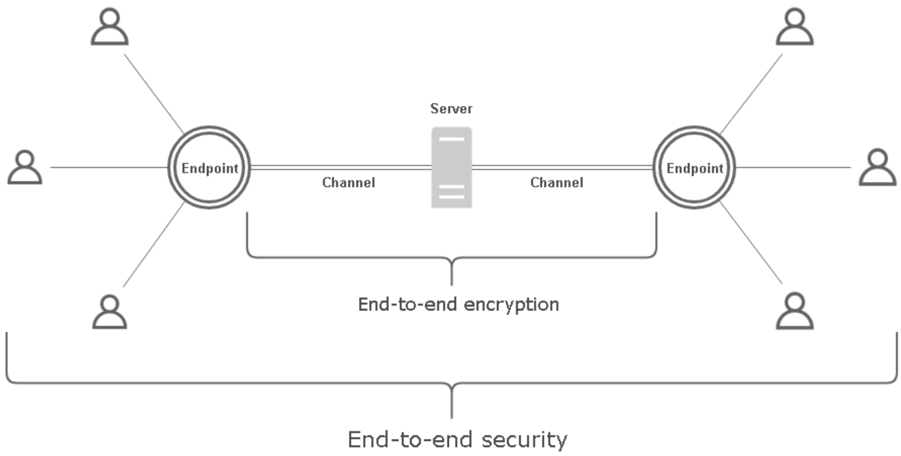


Figure 3.4: End-to-end security.

As the system depicts there might be several principals accessing the endpoint. Each principal could retrieve some information possibly protected with access control. Assume that the information resting at the endpoint is of confidential nature; access could still be granted with no respect of the confidentially of that information. There clearly lack a mechanism of specifying what information is confidential or public and where it may flow under what conditions.

Matrix identifies IoT as another use case. A person can have several devices for health tracking, entertainment and so on. The data from the devices are send to vendors - a device might send data to several vendors. Ultimately this give a fragmentation of the person's own data with it being placed at several vendors data back-end. Matrix proposes a solution where all the device data for a person is synchronized and persisted on Matrix. Vendors would be connected to Matrix. This is depicted in figure 3.5 below.

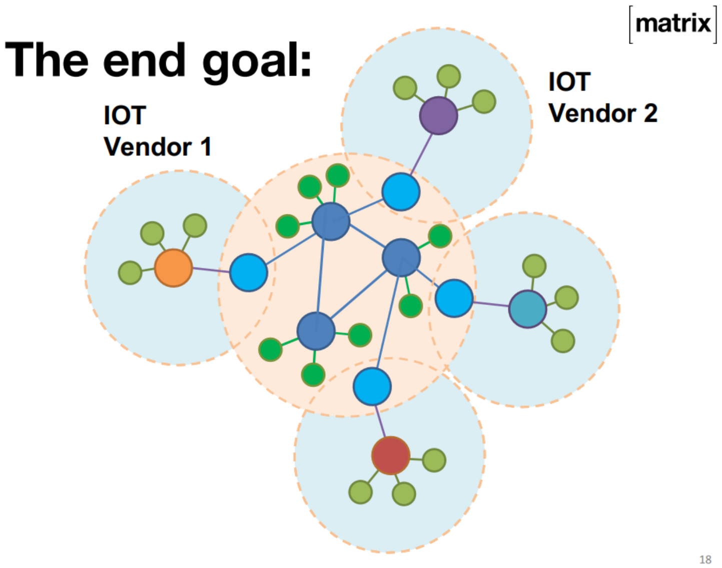


Figure 3.5: End goal for Matrix IoT

The data flowing from the sensors to Matrix might be of sensitive nature or the owner might only allow some data to flow to some vendors under specific conditions. This issue related to confidentiality is not addressed by Matrix.

3.1.5 Evaluation summary

In this section an evaluation of Matrix security was presented. Several security properties are a part of Matrix security model with forward secrecy and backward secrecy being provided depending on the Matrix configuration.

End-to-end encryption is not the end of security. Other security measures must be taken to provide confidentiality and integrity. Information Flow Control is such measure and the next section will present a survey on Information Flow Control.

3.2 Analysis of IFC Tools

Having established that end-to-end encryption is not the end of security we will now look at Information-Flow Control tools.

The approach to the section will be from a programmers perspective hence usability and practicality will play a role in choosing an IFC tool.

3.2.1 Jif

Jif is a Java based security-typed programming language that provides information flow control through static enforcement. Jif implements the Decentralized Label Model (DLM) described in section 2.3.5.1. Policies are defined conforming to the DLM. The policies are enforced at compile time with support for enforcement of dynamic policies at runtime. If a Jif program adheres to the specified policies the Jif compiler then compiles it to a secure Java program [7].

A policy is defined by labels and are associated with program variables. A policy can specify multiple principles that have readers or writers. A policy is defined as:

```
int {Alice→Bob; Alice←Bob} x;
```

The policy specifies two things; that first part with "→" expresses *Alice* controls the variable x and the variable can be read by *Bob*, the second part with "← expresses that bob can write to it [7].

The following code shows another example:

```
int {Alice} x;
int {Alice→Bob} y;
x = y; // OK
y = x; // BAD
```

Variable x has a policy that *Alice* controls with no readers. The variable y is owned by *Alice* with *Bob* being able to read hence the label is less restrictive than x's label. The compiler allows the $x = y$ since *Bob* is specified as a reader for x. However the expression $y = x$ is illegal since x has a stronger policy than y and the explicit flow is caught [7].

A important feature for security-typed languages are declassification. Non-interference is too strict for practical programs thus it is necessary to declassify information at times. The following example has two variables with different labels. Variable x is more restrictive than y; that has *Bob* as a reader. The example depicts an implicit flow:

```
void implicitFlow(){
  int {Alice→} x;
  int {Alice→Bob} y;
  if (x == 1) {
    // pc has label {Alice→}
    y = 0; // BAD
  }
}
```

When the code branches on the if statement the pc has the label *{Alice→}* and the expression $y = 0$ becomes illegal since it has the label *{Alice→Bob}* which is less restrictive than the label pc is holding. If for some reason we would want the expression to become valid we would have to declassify it:

```
void declassificationExample() where authority(Alice) {
  int{Alice→} x;
  int{Alice→Bob} y;
   // PC has label {}
  if (x == 1) {
     // PC has label {Alice→}
    declassify({Alice→} to {Alice→Bob}) {
      y = 0; // OK
    }
  }
}
```

To be able to declassify it must be through the authority of the owner. This has to be specified at the method definition. When the code branches on the if statement the program counter *pc* has the label *{Alice→}* which can then be declassified to a label as restrictive as *y*'s label [7] [44].

Another interesting feature is dynamic labels. Jif provides a run-time library which compares labels at runtime using a syntax that resembles if-statements.

```
void m(int{*lbl} i, label{} lbl) {
  int{Alice→} x;
  if (lbl <= new label {Alice→}) {
    x = i; // OK, since {*lbl} <= {Alice→}
  }
  else {
    x = 0;
  }
}
```

The parameter variable *i* has the label held by the label *lbl* which would be resolved at runtime. Since the variable *x* has label *{Alice→}* we can allow a flow to that variable as long it is less or equally restrictive. The static analysis of the program will pass and the program will be able to compile.

Other relevant features that Jif supports are label inference[2], parameterized classes[3] and polymorphism [4].

3.2.2 Paragon

Paragon is a programming language that extends Java with the ability to express security policies for data. Paragon has similar characteristics to Jif and essentially solving the same problem. Paragon have a different approach to defining information flow policies; *Paralocks* [21] [42]. At the core *Paralocks* is based on the concepts; *actors* and *locks* [21]. An actor is a user with some role. For information can flow to an actor there might be a condition that states; that the actor must be of a specific role. These conditions are represented by boolean variables *locks* and can be modified throughout program execution. These locks are called *parameterized locks* since they are parameterized over actors. Policies are specified by

[2] http://www.cs.cornell.edu/jif/doc/jif-3.3.0/language.html#inference
[3] http://www.cs.cornell.edu/jif/doc/jif-3.3.0/language.html#parameterized-classes
[4] http://www.cs.cornell.edu/jif/doc/jif-3.3.0/language.html#label-polymorphism

parameterized locks and *actor polymorphism* allows us to reason about all actors [42].

In Paragon policies are immutable and are defined as such:

```
public static final policy low  = { Object x: };
public static final policy high = { };
```

Policies for *low* and *high* can encoded in different ways. The encoding specifies the most liberal policy that anyone can see the data that has the policy of *low*. The actor is any *Object x* hence anyone can read. The encoding for *high* is the most strict policy and specifies that actors so no one can see the data.

To support a simple declassification mechanism a lock would have to be introduced and the *high* policy definition would have to be redefined:

```
private lock Declassify;
public static final policy low  = { Object x: };
public static final policy high = { Object x: Declassify};
```

The *high* policy now specifies that information can only be read if the *declassify lock* is open. The following method can now allow declassification:

```
public static ?low int declassify(?high int x){
  open Declassify { return x; }
}
```

The method is a custom declassification method for variables with type *int*. By using Java generics the same method could be used for any type. The *declassify()* method takes a parameter that has the policy *high* and returns the parameter with the lower policy *low*. The method opens the lock hence allowing the value of x to be read and returned.

```
publicVariable = declassify(secretVariable); // OK
```

The example illustrates a simple declassify method however it can be called by anyone. To ensure that only those with the right authority can call it the example could easily be extended with another lock [21].

It is possible to support policies at runtime with locks. Suppose we have a customer who buys some software. The software keys should only be given when the customer has paid. We define the following:

```
public static lock Paid;
?{customer: } String customerData
?{customer: Paid} String softwareKey
```

When the the customer's payment is processed the lock *Paid* should only be open if the payment is successful.

```
public void processPayment() {
  // customer pays for item
  if (paymentSuccessful) { open Paid; } else { ... }
}
```

It is not possible for the compiler to learn the state of the lock *Paid*. By using the lock in a conditional statement the lock will be checked at runtime.

```
processPayment();
if (Paid) { customerData = softwareKey; } else { ... }
```

Paragon is a powerful tool for Information-Flow Control and has an interesting policy language with support for expressive dynamic policies. Other features supported by Paragon are policy inference,

3.2.3 JSFlow

JSFlow is a tool for tracking information flow in JavaScript web applications. This tool is not a programming language as the two previously described tool but a JavaScript interpreter that supports full non-strict ECMA-262 [8]. JSFlow enforces secure information flow through dynamic analysis and can detect explicit and implicit flows. JSFlow uses a program counter *pc* to track the security context. JSFlow defines two built-in security levels; *public* and *secret*, it also supports custom security labels on values. JSFlow allows pure explicit flows by upgrading the security label of the variable being assigned to [8]:

```
high = lbl(true);
low = high;
```

The variable *high* is assigned a secret value denoted by the *lbl* function. When *high* is assigned to *low* the security label is upgraded for low. JSFlow prevents implicit flow through *no sensitive update* that under secret control disallows changes to security labels [27]:

```
high = lbl(true);
if(high){
  l = true;
}
```

The execution would halt for the above code since no sensitive update is allowed.

As mentioned before security labels can be assigned to variables. In the following examples *l*, *m* and *h* is defined with different labels.

```
var l = lbl(10, 'low');
var m = lbl(15, 'mid');
var h = lbl(20, 'mid', 'high');
if(m === 15) {
  h=m; // OK
  l=m; // BAD
}
```

JSFlows uses a subset lattice hence *m* can flow to *h* since *m*'s label is a subset of *h*'s labels [8]. The assignment *l=h* would halt since the *m*'s labels is not a subset of *l*'s labels.

JSFlow is an exciting tool for dynamic Information-Flow Control. However the JSFlow is still immature and does not support important features such as declassification.

3.2.4 Selection of IFC tool

The selection of the IFC tool used for developing the prototype is based the following defined parameters. The selection of IFC tool put emphasis on the practical usage in combination with Matrix.

3.2.4.1 Decision matrix

Parameters: Hard parameters are related to language features. Soft parameters are related to outside the language and that are relevant for a programmer. Hard parameters: declassification, parameterized class

Soft parameters: simplicity, documentation, support for external libraries, programmer-friendly

3.2.4.2 Comparison

3.2.5 IFC tools summary

3.3 Summary

In this chapter the Matrix security model has been evaluated. Matrix provides end-to-end security and uses the Double Ratchet algorithm by Signal. The evaluation found that there are no major flaws in the design. To achieve end-to-end security the endpoints need to be secured as well [39] this leads us to the chapter's second part. The chapter analyzed information-flow control tools and justifies the selection of Paragon which the prototype is programmed in.

Bibliography

[1] Op imod 90.000 ansatte kan kigge i din journal - Indland. URL https://jyllands-posten.dk/indland/ECE6715461/op-imod-90000-ansatte-kan-kigge-i-din-journal/.

[2] Adgang til sundhedsdata - sundhed.dk. URL https://www.sundhed.dk/borger/service/om-sundheddk/om-portalen/datasikkerhed/andres-dataadgang/adgang-til-sundhedsdata/.

[3] CWE - CWE-359: Exposure of Private Information ('Privacy Violation') (3.2). URL https://cwe.mitre.org/data/definitions/359.html.

[4] Journal fra sygehus. URL https://www.sundhed.dk/borger/min-side/min-sundhedsjournal/journal-fra-sygehus/.

[5] Principles for a More Informed Exceptional Access Debate - Lawfare. URL https://www.lawfareblog.com/principles-more-informed-exceptional-access-debate.

[6] Google Plus Will Be Shut Down After User Information Was Exposed - The New York Times. URL https://www.nytimes.com/2018/10/08/technology/google-plus-security-disclosure.html.

[7] Jif Reference manual. URL http://www.cs.cornell.edu/jif/doc/jif-3.3.0/manual.html.

[8] jsflow. URL http://www.jsflow.net/.

[9] Kontrol af opslag - sundhed.dk. URL https://www.sundhed.dk/borger/service/om-sundheddk/om-portalen/datasikkerhed/portalens-beskyttelse-af-data/kontrol-af-opslag-e-journal/.

[10] FAQ | Matrix.org, . URL https://matrix.org/docs/guides/faq.

[11] Home | Matrix.org, . URL https://matrix.org/.

[12] Matrix Specification, . URL https://matrix.org/docs/spec/{#}architecture.

[13] Video: IoT through Matrix | Matrix.org, . URL https://matrix.org/blog/2015/04/08/video-iot-through-matrix/.

[14] 91,000 state Medicaid clients warned of data breach | The Seattle Times. URL https://www.seattletimes.com/seattle-news/health/91000-state-medicaid-clients-warned-of-data-breach/.

[15] Client-Server API. URL `https://matrix.org/docs/spec/client{_}server/r0.4.0.html{#}types-of-room-events`.

[16] Asian Askarov and Andrei Sabelfeld. Security-typed languages for implementation of cryptographic protocols: A case study. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 3679 LNCS:197–221, 2005. ISSN 03029743. doi: 10.1007/11555827_12.

[17] Aslan Askarov, Sebastian Hunt, Andrei Sabelfeld, and David Sands. Termination-insensitive noninterference leaks more than just a bit. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 5283 LNCS:333–348, 2008. ISSN 03029743. doi: 10.1007/978-3-540-88313-5-22.

[18] Musard Balliu. *Logics for Information Flow Security : From Specification to Verification Doctoral Thesis in Computer Science.* 2014. ISBN 9789175952598.

[19] Matt Bishop. *Introduction to Computer Security.* Addison-Wesley, 2004. ISBN 0321247442.

[20] Matt Bishop. *Introduction to computer security.* Addison-Wesley, Boston, 2005. ISBN 978-0321247445.

[21] Niklas Broberg and David Sands. Paralocks: role-based information flow control and beyond. *Proceedings of the 37th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, (Section 4):431–444, 2010. ISSN 0362-1340. doi: 10.1145/1707801.1706349.

[22] Katriel Cohn-Gordon, Cas Cremers, Benjamin Dowling, Luke Garratt, and Douglas Stebila. A Formal Security Analysis of the Signal Messaging Protocol. *Proceedings - 2nd IEEE European Symposium on Security and Privacy, EuroS and P 2017*, (November):451–466, 2017. ISSN 13484214. doi: 10.1109/EuroSP.2017.27.

[23] Dorothy E. Denning. A lattice model of secure information flow. *Communications of the ACM*, 19(5):236–243, 1976. ISSN 0001-0782. doi: 10.1145/360051.360056. URL `http://dl.acm.org/citation.cfm?id=360056`.

[24] Tilman Frosch, Christian Mainka, Christoph Bader, Florian Bergsma, Jörg Schwenk, and Thorsten Holz. How Secure is TextSecure? Technical report. URL `https://cryptome.org/2014/11/textsecure-secure.pdf`.

[25] Laurinda B. Harman, Cathy A. Flite, and Kesa Bond. Electronic Health Records: Privacy, Confidentiality, and Security. *Virtual Mentor*, 14(9):712–719, sep 2012. ISSN 1937-7010. doi: 10.1001/virtualmentor.2012.14.9.stas1-1209. URL `http://virtualmentor.ama-assn.org/2012/09/stas1-1209.html`.

[26] Daniel Hedin and Andrei Sabelfeld. A Perspective on Information-Flow Control. 2011.

[27] Daniel Hedin, Arnar Birgisson, Luciano Bello, and Andrei Sabelfeld. [103]JS-Flow_ Tracking information flow in JavaScript and its APIs. *Proceedings of the 29th Annual ACM Symposium on Applied Computing - SAC '14*, pages

1663–1671, 2014. ISSN 1069-417X. doi: 10.1145/2554850.2554909. URL http://dl.acm.org/citation.cfm?doid=2554850.2554909.

[28] Vineeth Kashyap, Ben Wiedermann, and Ben Hardekopf. Timing- and Termination-Sensitive Secure Information Flow: Exploring a New Approach. pages 413–428, 2011. doi: 10.1109/SP.2011.19. URL http://ieeexplore.ieee.org/document/5958043/.

[29] Moxie Marlinspike and Trevor Perrin. The X3DH Key Agreement Protocol. Technical report, 2016. URL https://signal.org/docs/specifications/x3dh/x3dh.pdf.

[30] Katina Michael. *The Basics of Information Security: Understanding the Fundamentals of InfoSec in Theory and Practice*, volume 31. 2012. ISBN 9781597496537. doi: 10.1016/j.cose.2012.03.005. URL http://linkinghub.elsevier.com/retrieve/pii/S0167404812000557.

[31] A.C. Myers and B. Liskov. Complete, safe information flow with decentralized labels. *Proceedings. 1998 IEEE Symposium on Security and Privacy (Cat. No.98CB36186)*, (May):186–197, 1998. ISSN 1081-6011. doi: 10.1109/SECPRI.1998.674834. URL http://ieeexplore.ieee.org/document/674834/.

[32] Andrew C. Myers and Barbara Liskov. A decentralized model for information flow control. *ACM SIGOPS Operating Systems Review*, 31(5):129–142, 1997. ISSN 01635980. doi: 10.1145/269005.266669. URL http://portal.acm.org/citation.cfm?doid=269005.266669.

[33] Andrew C. Myers and Barbara Liskov. Protecting privacy using the decentralized label model. *Foundations of Intrusion Tolerant Systems, OASIS 2003*, 1(212):89–116, 2003. ISSN 1049331X. doi: 10.1109/FITS.2003.1264929.

[34] NCC Group. Olm Cryptographic Review. (November):1–27, 2016. ISSN 0737-4038.

[35] P. D. Pacey and J. H. Purnell. OWASP Top 10 - 2017. *International Journal of Chemical Kinetics*, 4(6):657–666, 1972. ISSN 10974601. doi: 10.1002/kin.550040606.

[36] Trevor Perrin and Moxie Marlinspike. The Double Ratchet Algorithm. Technical report, 2016. URL https://signal.org/docs/specifications/doubleratchet/doubleratchet.pdf.

[37] Ralph Spencer Poore. Crypto' 101. *Information Systems Security*, (2), 2017. ISSN 1065-898X. doi: 10.1201/1086/43305.8.2.19990601/31062.6. URL http://www.tandfonline.com/doi/full/10.1201/1086/43305.8.2.19990601/31062.6.

[38] Fiza Abdul Rahim, Zuraini Ismail, and Ganthan Narayana Samy. Information privacy concerns in electronic healthcare records: A systematic literature review. In *2013 International Conference on Research and Innovation in Information Systems (ICRIIS)*, pages 504–509. IEEE, nov 2013. ISBN 978-1-4799-2487-5. doi: 10.1109/ICRIIS.2013.6716760. URL http://ieeexplore.ieee.org/document/6716760/.

[39] Andrei Sabelfeld and Andrew C. Myers. Language-based information-flow security. *IEEE Journal on Selected Areas in Communications*, 21(1):5–19, 2003. ISSN 07338716. doi: 10.1109/JSAC.2002.806121.

[40] Andrei Sabelfeld and Alejandro Russo. From dynamic to static and back: riding the roller coaster of Information-flow control research. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 5947 LNCS:352–365, 2010. ISSN 03029743. doi: 10.1007/978-3-642-11486-1_30.

[41] Andrei Sabelfeld and David Sands. Declassification: Dimensions and principles. *Journal of Computer Security*, 17(5):517–548, 2009. ISSN 0926227X. doi: 10.3233/JCS-2009-0352.

[42] David Sands and Bart Van Delft. Paragon for Practical Flow-Oriented Programming. pages 1–12, 2012.

[43] Geoffrey Smith. Principles of Secure Information Flow Analysis. *Malware Detection*, pages 291–307. ISSN 15682633. doi: 10.1007/978-0-387-44599-1_13. URL `http://link.springer.com/10.1007/978-0-387-44599-1{_}13`.

[44] Y. N. Srikant and Priti Shankar. *The Compiler Design Handbook: Optimizations and Machine Code Generation, Second Edition*. CRC Press, Inc., Boca Raton, FL, USA, 2nd edition, 2007. ISBN 142004382X, 9781420043822.

[45] The Government, Local Government Denmark, and Danish Regions. *The Digital Strategy - A stronger and more secure digital Denmark*. Agency for Digitisation, 2016. ISBN 9789400769250 | 9400769245 | 9789400769243. doi: 10.1007/978-94-007-6925-0_9. URL `https://en.digst.dk/media/14143/ds{_}singlepage{_}uk{_}web.pdf`.

[46] Nik Unger, Sergej Dechand, Joseph Bonneau, Sascha Fahl, Henning Perl, Ian Goldberg, and Matthew Smith. SoK: Secure Messaging. pages 232–249, 2015. doi: 10.1109/SP.2015.22.

[47] Bart van Delft, Niklas Broberg, and David Sands. Programming in Paragon. *Software Systems Safety*, 36:279–308, 2014. URL `http://dblp.uni-trier.de/db/series/natosec/natosec36.html{#}DelftBS14`.