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

Abstract

The thesis is a case study of secure implementation using the Information-Flow Control tool Paragon on top of the Matrix protocol. The implementation is a prototype inspired by Danish patient journals systems which requires end-to-end security. End-to-end security is the guarantee of confidentiality and integrity throughout the system. Matrix is a secure communication protocol that ensures confidentiality and integrity of information through end-to-end encryption. However encryption provides no such guarantees once the information is decrypted at the endpoints. Information-Flow Control is such a mechanism that can ensure confidentiality and integrity at the endpoints by enforcing security policies. Paragon is a Java based programming language with the ability to define and enforce security policies.

The thesis contributes with a interface between Paragon and Matrix allowing implementations with end-to-end security on top of Matrix.

Acknowledgements

Contents

List of Figures

List of Tables

1 Introduction

1.1 Information Privacy and Protection

With GDPR becoming effective in 2018, the focus on information privacy is at its peak. Privacy violation is when sensitive information is exposed to an unauthorized actor; like how Cambridge Analytica was able to gain sensitive information about Facebook users without their permission or when sensitive information about halv a million Google Plus users was leaked [5] [8] [4]. Both information leaks were due to insecure implementation of API's. These cases have made people more aware of how application deals with information. Unsurprisingly OWASP top ten ranks *sensitive data exposure* as the third biggest security threat[37].

The mentioned cases failed to achieve *end-to-end security*; assurance of *confidentiality* and *integrity* throughout the system [41]. The improper handling of sensitive information could have been prevented with appropriate security policies and enforcement technique that enforces these policies.

1.2 Information-Flow Control

Secure information flow means that only authorized flow of information is allowed [25]. Secure information flow in a system guarantees confidentiality and integrity of information throughout the system [28]. Consider the following insecure flow with variables holding *public* and *secret* information:

```
public = secret;
```

Here the secret information is leaked by directly assigning it to a publicly observable variable. This is an example of *explicit flow* of information [28]. Information-Flow Control can prevent such flows by allowing the programmer to define and enforce policies in a language-based way[41].

The security enforcement mechanisms such as firewalls, encryption and access control all protect confidential information but is unable to control how the information flows:

- *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.

These are all useful mechanisms but fall short in terms of end-to-end security.

1.3 Matrix

Matrix provides an encryption mechanism for secure messaging and synchronizing data over HTTP. Matrix is an open standard protocol that enables secure decentralized communication over a *federation* of servers. The secure communication is provided by an end-to-end encryption mechanism ensuring confidentiality and integrity of information while it is stored at the servers and is at transit. However as mentioned earlier it cannot control the flow of information and is vulnerable to leaks at the endpoints.

Why Matrix? 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"*[12].

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[47].

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.

1.4 The Case Study

The goal of the case study is to make a prototype secure implementation of a distributed medical journal system, using a combination of Matrix and IFC. We will use a protocol called *Matrix* as the communication channel and strengthen the security at the endpoints using Information-Flow Control.

1.4.1 Journal System

Medical privacy is a well-known issue[40]. Sensitive data about patients needs to be handled carefully. In Denmark patients have access to their medical records through E-journal[6]. A patient's journal on E-journal is available for up to 90.000 different medical employees[1].

There are clear policies about who should access a journal and under what conditions . 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][11].

The mechanisms in the current journal system might restrain malicious intent. However it does not guarantee prevention of unintentional access or disclosure of information[27]. 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.

1.4.2 Objectives

The objective of the project is to do a secure implementation of the prototype inspired by the Danish journal systems. Secure exchange of patient journals 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

1.5 Method

The research method used is a case study. A case study allow us investigate the how and why of a case [50]. By investigating the following we can achieve the objectives described previously:

- Why there exist a shortcoming in the Matrix security model.
- What it takes to improve the security guarantee of Matrix.
- How the security guarantees can be improved using the Information-Flow Contol tool Paragon.

1.6 Contribution

The thesis contributes with an interface created between the Information-Flow Control language Paragon and the secure communication protocol Matrix. The contribution enables developing other secure applications on top of the Matrix protocol. Another contribution is the findings of secure implementation using Paragon.

1.7 Structure of thesis

Chapter 2 presents the evaluation of the Matrix security model. Chapters 3 presents the analysis and selection of Information-Flow Control tool. Chapter 4 describe the design choices and implementation details for the prototype. With Chapter 5 and 6 wrap up the thesis with discussion and conclusion.

1.8 Summary

This section introduces the scope of the thesis and what is expected of the thesis. We achieve end-to-end security through secure information flow. Information-Flow Control is a mechanism for enforcing secure information policies in a system. The objective of the thesis is to evaluate the security model of Matrix, survey and select an Information-Flow Control tools, implement a prototype using the tool and demonstrate the improvements to the Matrix security model.

2 Evaluation of Matrix Security model

This chapter consists of two parts. The first part will provide an evaluation of the Matrix security and relies on the paper *SoK: Secure Messaging* [48] and *The Olm Cryptographic Review* by NCC Group [36].

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

Section 2.3 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 [48] 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 [32].

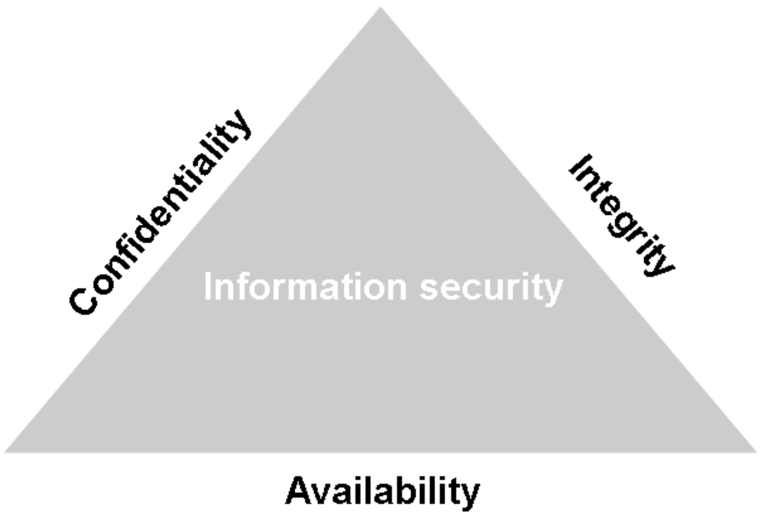


Figure 2.1: CIA triad

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 [32].

In a secure messaging system confidentiality would be guaranteed if the message being sent is only readable by the recipient and no one else [48].

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 [32]. More specifically for a secure messaging it would mean that no altered message is accepted by the recipient [48].

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 [32].

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 used to evaluate Matrix in section 2.3.

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.

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

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

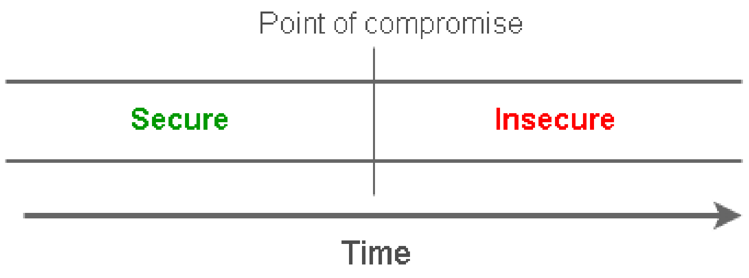


Figure 2.2: Forward secrecy

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.*

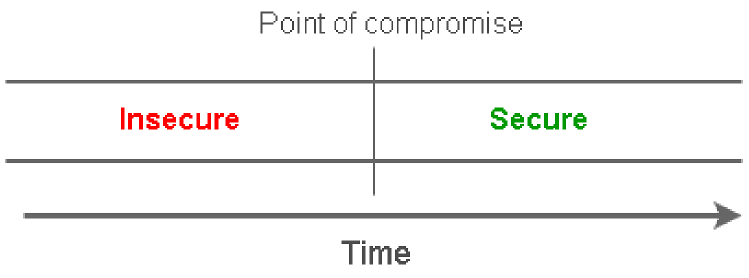


Figure 2.3: Backward secrecy

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 [48].

- *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.

2.2 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 2.4 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.

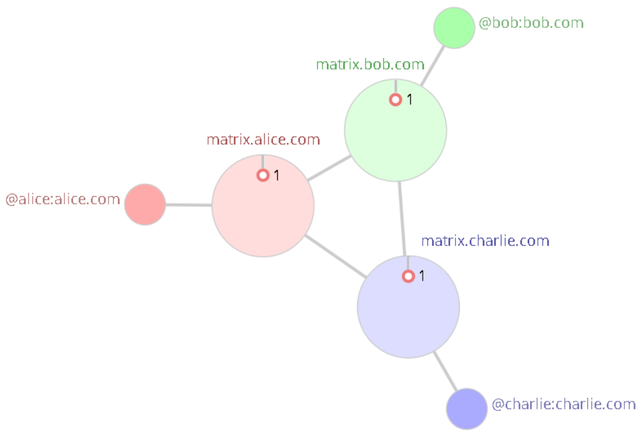


Figure 2.4: Matrix [13].

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 2.5.

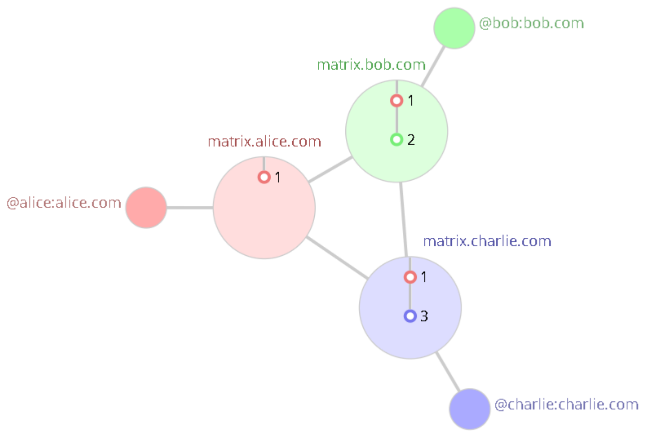


Figure 2.5: Matrix [13].

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 2.6;

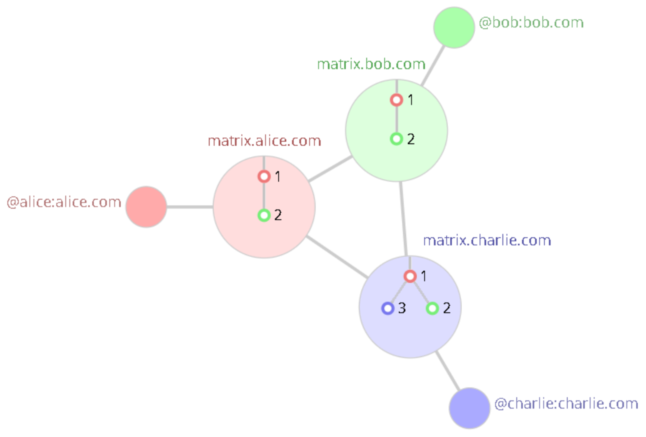


Figure 2.6: Matrix [13].

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.

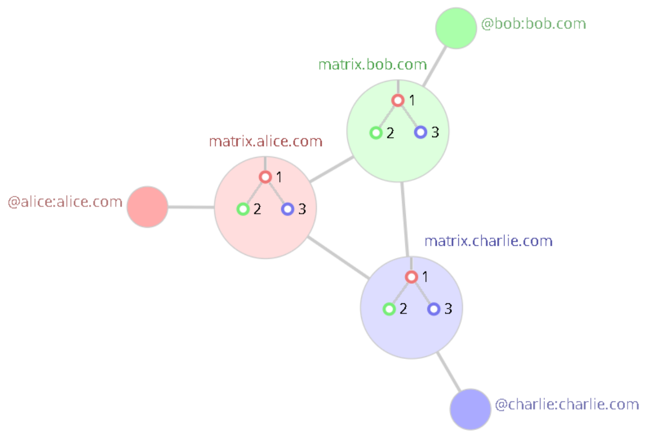


Figure 2.7: Matrix [13].

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.

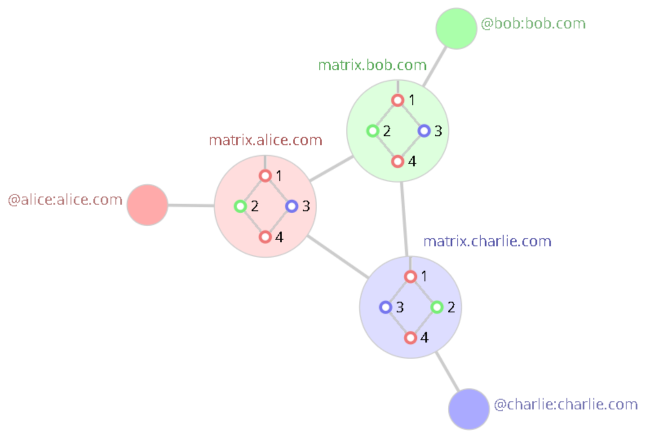


Figure 2.8: Matrix [13].

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* [14].

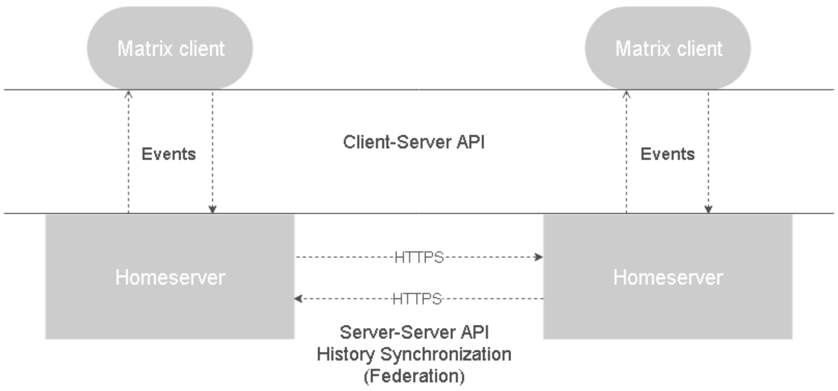


Figure 2.9: Matrix conceptual architecture [14].

As previously mentioned a room is a conceptual place for sending and receiving events with the room data being distributed over each homeserver. Each room is uniquely identified by a *Room ID*. The figure 2.10 shows how events are send and received from a room.

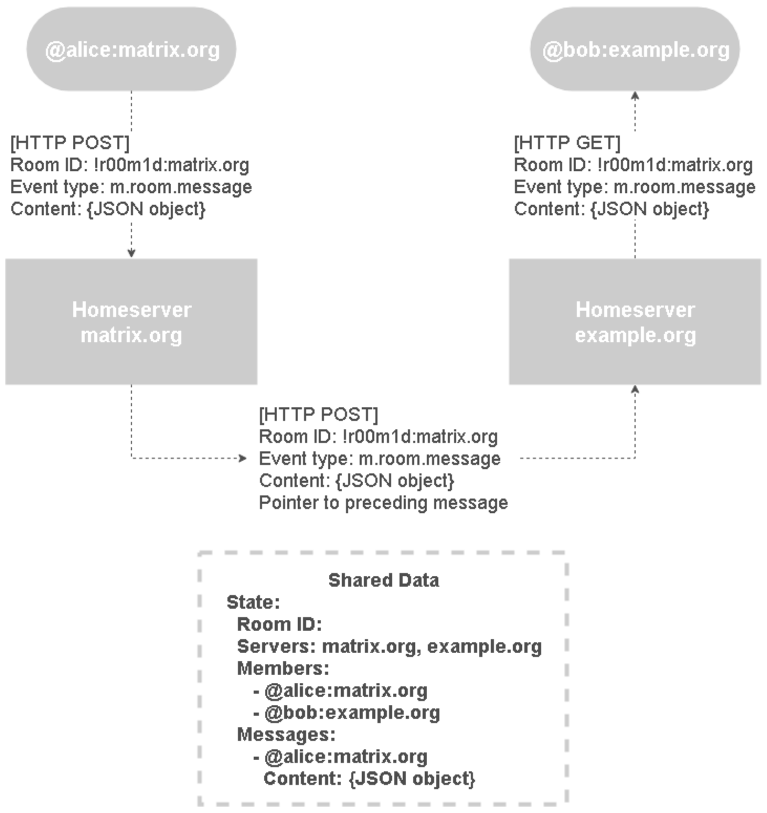


Figure 2.10: Matrix conceptual architecture [14].

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* [14].

```
{
    "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 [17].

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 2.3.3.

2.2.5 Concepts

2.2.5.1 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.

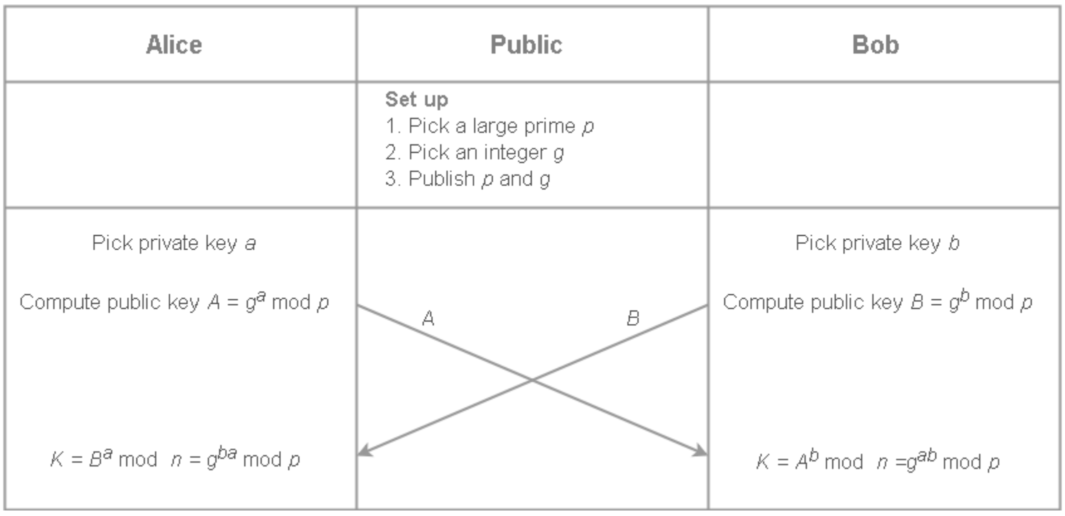

Alice
Public
Bob
Set up
1. Pick a large prime p
2. Pick an integer g
3. Publish p and g
Pick private key a
Compute public key $A = g^a \bmod p$
Pick private key b
Compute public key $B = g^b \bmod p$
A
B
$K = B^a \bmod n = g^{ba} \bmod p$
$K = A^b \bmod n = g^{ab} \bmod p$

Figure 2.11: 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 [39].

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 [39].

2.2.5.2 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 [39]. Figure 2.12 illustrates this.

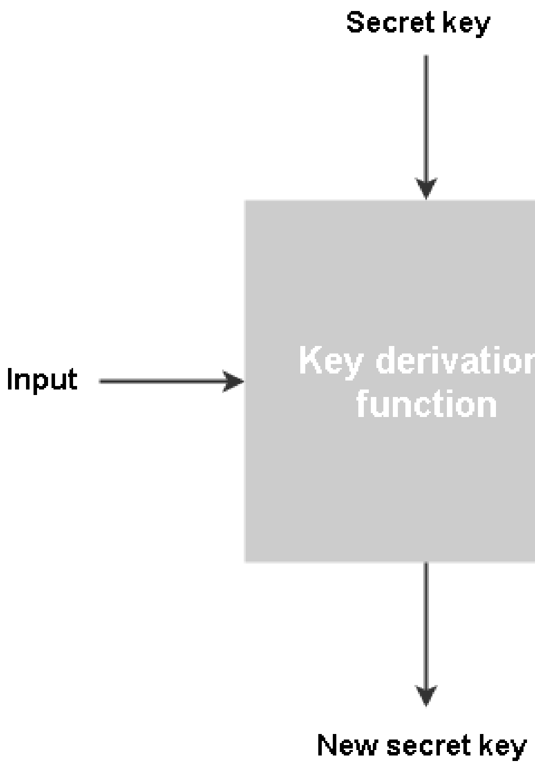


Figure 2.12: Key derivation function

A core concept in the Double Ratchet algorithm is KDF chains which build a chain of secret keys using KDF [38].

2.2.5.3 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.

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 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 2.13 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.

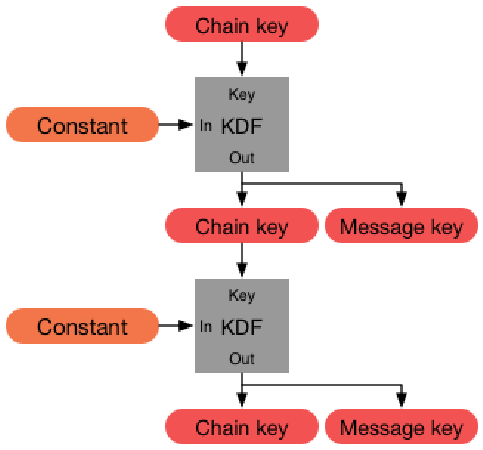


Figure 2.13: Symmetric key ratchet [38].

Diffie-Hellman ratchet Double ratchet provides backward secrecy by comining 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 2.14 shows an ongoing message exchange with new secrets being derived and the sending and receiving chains being advanced.

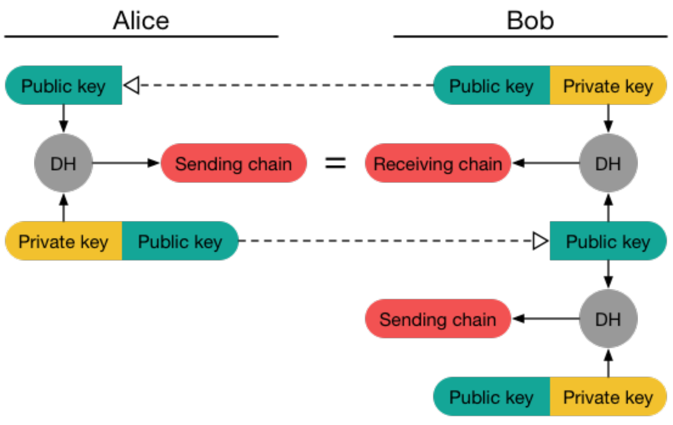


Figure 2.14: Diffie-Hellman ratchet [38].

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 2.15

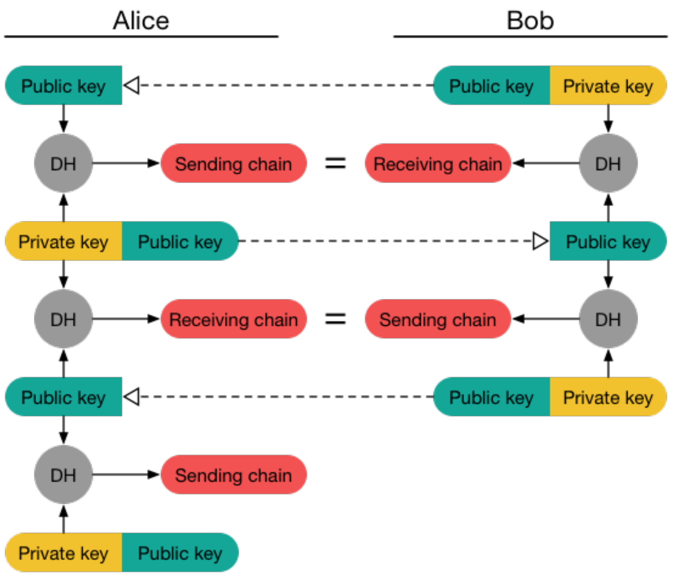


Figure 2.15: Continuation of Diffie-Hellman ratchet [38].

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 2.16.

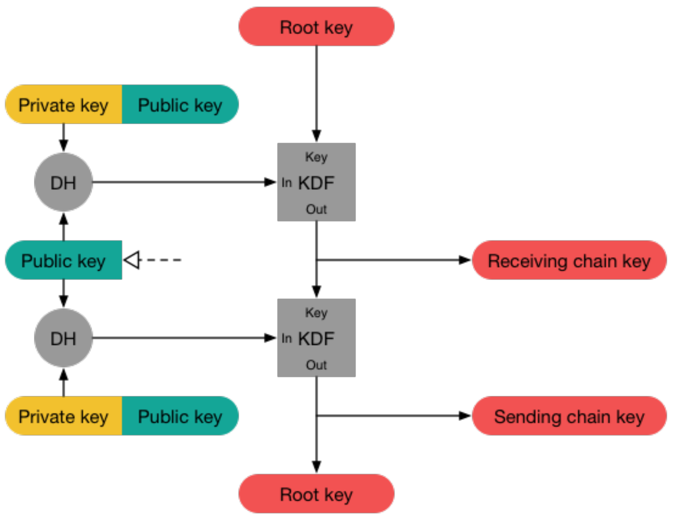


Figure 2.16: Diffie-Hellman ratchet 7 [38].

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 2.2.5.3. 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.

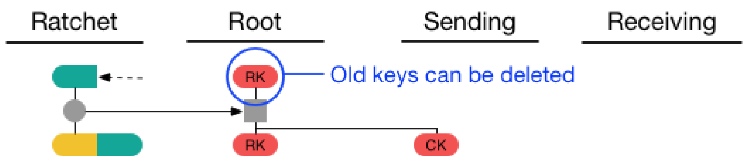


Figure 2.17: Double ratchet 1 [38].

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.

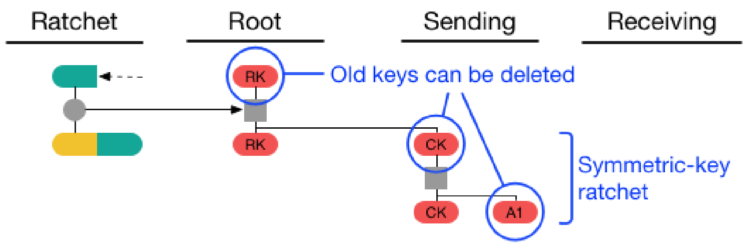


Figure 2.18: Double ratchet 2 [38].

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.

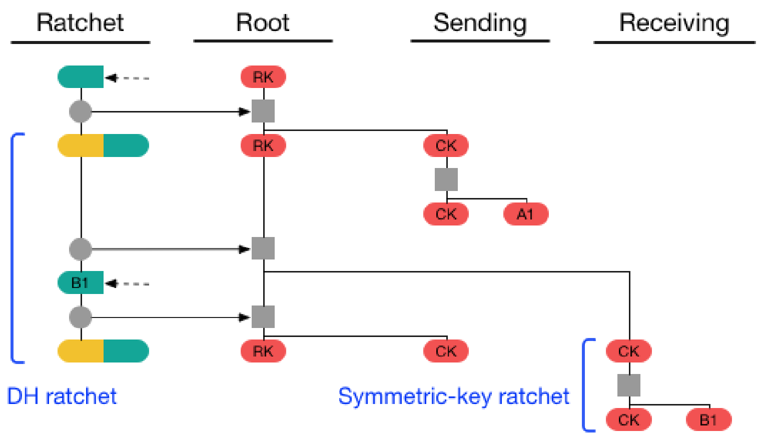


Figure 2.19: Double ratchet 3 [38].

2.3 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*

2.3.1 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 [48].
This evaluation will inherit the described threat model.

2.3.2 The Signal Protocol

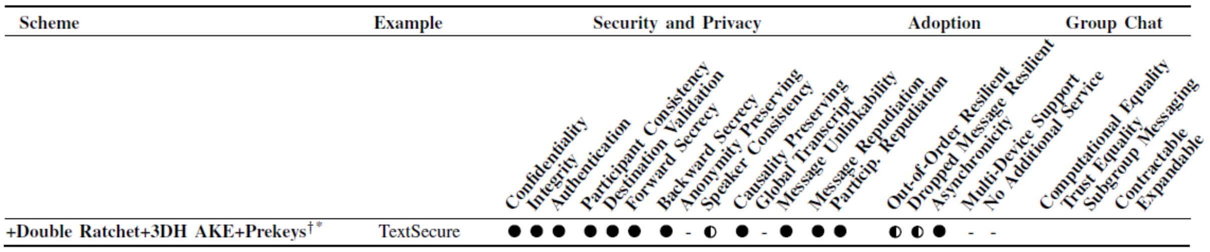
Matrix provides end-to-end encryption by using the Olm and Megolm library with the former being an implementation of the Double Ratchet algorithm also known as the Signal Protocol, and the latter being the algorithm used for group chat.

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) [48].



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$^{\dagger *}$	TextSecure	●	●	●	●	●	●	●	-	◐	●	-	●	●	●	◐	◐	●	-	-					

Figure 2.20: Evaluation of Signal (TextSecure) [48].

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 [48].

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 [48].

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 [48].
- ***Destination validation***. The Deffie-Hellman ratchet provides this property since the recipients public key is used to generate the chain key [48].

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 [24].

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.

2.3.3 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 [48].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 2.21 illustrates the setup with three participants.

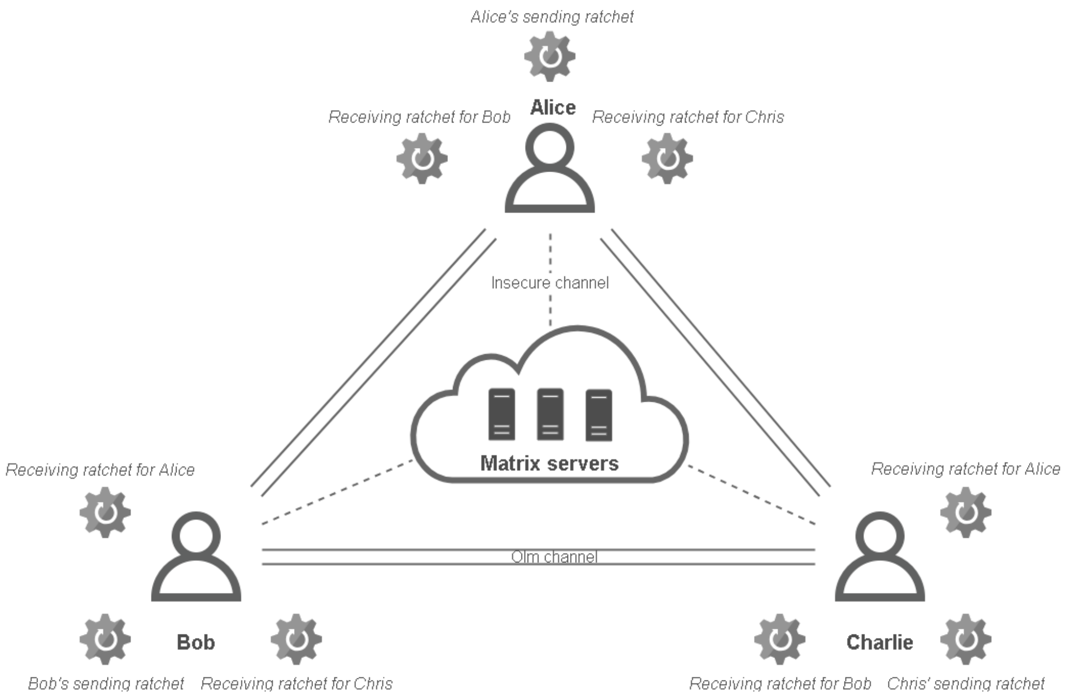

Alice's sending ratchet
Receiving ratchet for Bob
Alice
Receiving ratchet for Chris
Insecure channel
Matrix servers
Receiving ratchet for Alice
Receiving ratchet for Alice
Olm channel
Bob
Charlie
Bob's sending ratchet
Receiving ratchet for Chris
Receiving ratchet for Bob
Chris' sending ratchet

Figure 2.21: 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.

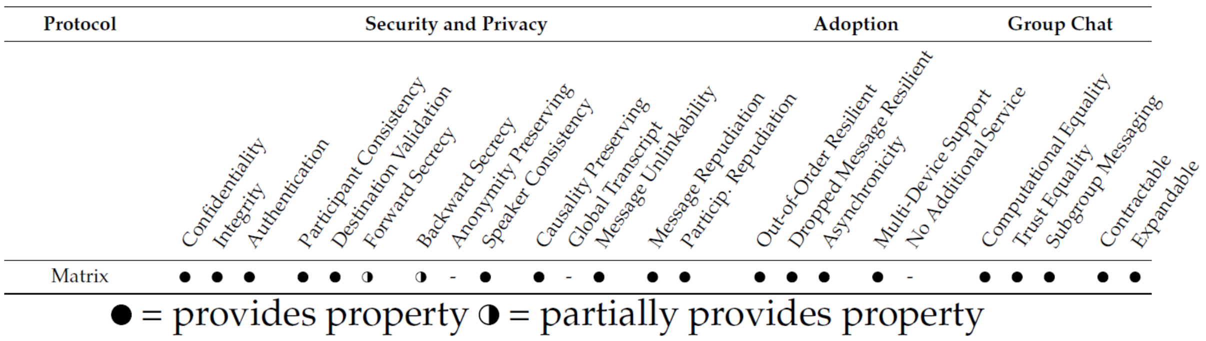
2.3.3.1 Evaluation

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 [36] [**?**].

- *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 [36] [**?**]. 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 2.22: 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 [36].

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 [36].

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*** [36].

Other properties

The multi-cast encryption design does not provide *participant consistency* [48].

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*[3] 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 [36].

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.

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

2.3.4 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.

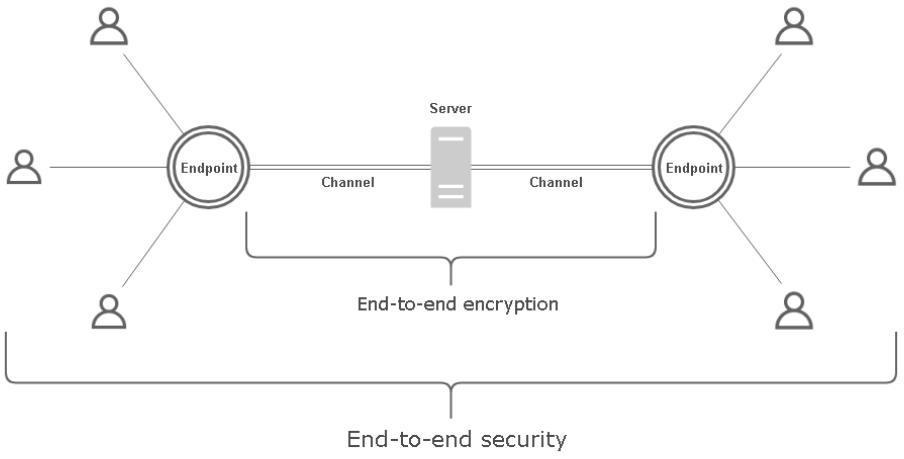


Figure 2.23: End-to-end security.

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 IoT 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 2.24 below.

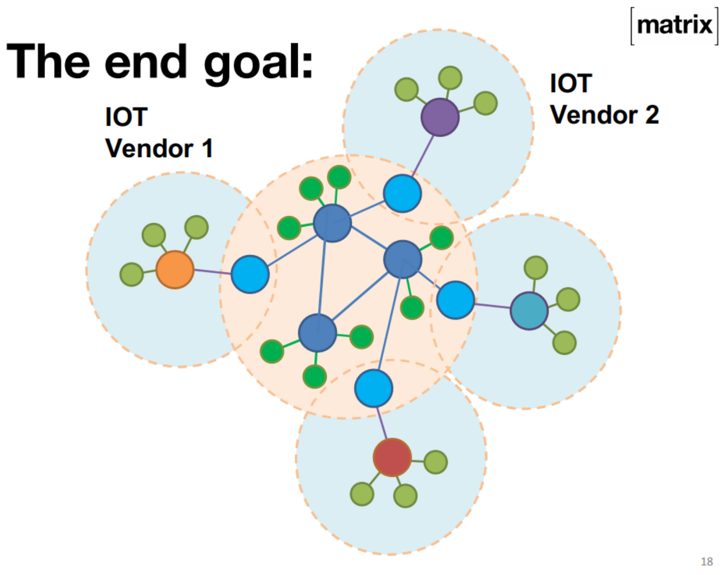


Figure 2.24: End goal for Matrix IoT

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.

2.3.5 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 Survey of Information-Flow Control Tools

3.1 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 [28]. Secure information flow means that only authorized flow of information is allowed [25]. There are two aspects to secure information flow; the reading of the information (confidentiality) and the writing of the information (integrity) [28].

Information Flow Control is a *security mechanism*[1] for achieving secure information flow. Information Flow Control is a language-based security technique that can enforce defined *security policies*[2] concerning confidentiality and integrity of data. It enables us to express where information may flow to and under what conditions.

3.1.1 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 security levels we express what information may flow where by classifying information with a security level, 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 we can formalize these flow constrains by means of a lattice[3] structure where information flows upwards shown in figure 3.1a [45].

[1]"A security mechanism is a method, tool, or procedure for enforcing a security policy" [21]
[2]"A security policy is a statement of what is, and what is not, allowed." [21]
[3]https://en.wikipedia.org/wiki/Lattice_(order)

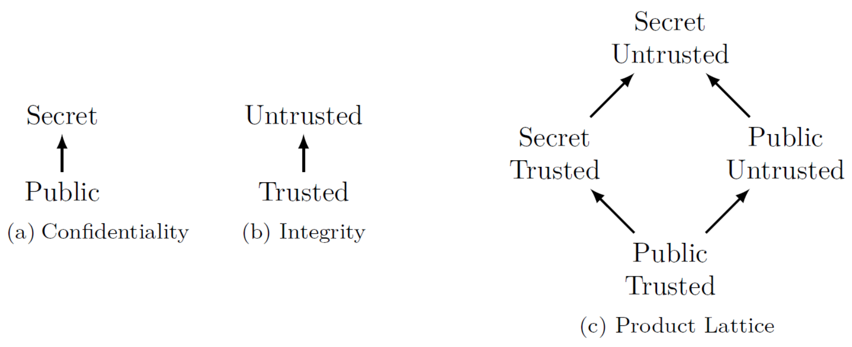


Figure 3.1: Lattice model for confidentiality and integrity [20]

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 exactly the opposite of the confidentiality policy. Figure 3.1b illustrates an integrity lattice. By combining the two we get a lattice show in figure 3.1c.

3.1.2 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* since 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 [28].

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 [41].

3.1.3 Covert channels

The explicit and implicit flows can also be considered as *channels* that signal information [30] [41]. *Covert channels* are channels that are not meant to signal information but somehow leaks information [30]. 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 [41].

3.1.4 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 and can pick the public input 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 3.2 illustrates noninterference [28].

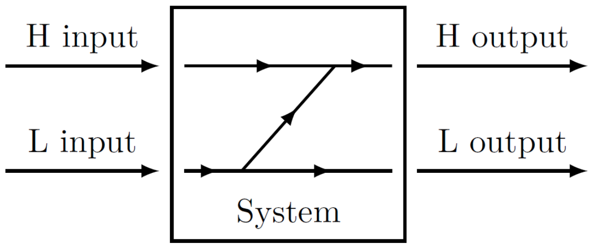


Figure 3.2: Noninterference [20]

The figure illustrates that the low confidentiality output *L* (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 [28].

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 [28]:

```
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 [28].

This variant of noninterference does not allow secret variables as guards in loops and if-else-than statements as long as those in no way affect possible future L output. This variant would be more restrictive and disallow flows as such [28]:

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

3.1.5 Declassification

Noninterference is a very strict policy and not practical. Consider the classic example of a user logging into a system. It is unavoidable that a login attempt gives some information away about whether or not the password combination was valid. 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 sent over an insecure channel. However this goes against noninterference since the secret input clearly interferes with public output [28].

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 [43]:

- *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.

3.1.5.1 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 [34]. 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 [34] [33]. This is known as the *principal hierarchy* and can allow a principal to act for another principal.

3.1.6 Information-flow enforcement

There exist two general techniques for enforcing secure information flow; *static analysis* through e.g a type system, and *dynamic analysis* through e.g. a monitor. Both techniques give the assurance of termination-insensitive noninterference [42]. 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 [42].

3.1.6.1 Static enforcement

Enforcement of information-flow through static analysis is done using type systems. A simple type system is presented and shown in figure 3.3.

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

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

Figure 3.3: Typing rules [42]

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 $pc \vdash c$ [42].

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 [42]. 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 [42]. The implicit flow would be detected by the typing system as well:

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

3.1.6.2 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 [42].

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

Figure 3.4: Monitoring rules [42]

The figure 3.4 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* generated by the program:

- *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, since whenever the program takes a step, the monitor must take a step; if the monitor cannot take a step, that halts the program. 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.

3.2 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 2.3 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.

3.3 Analysis of IFC Tools

Different Information-Flow Control tools were considered to be used in the case study. This section describes three of those tools; *Jif*, *Paragon* and *JSFlow*. This section further justifies the selection of Paragon.

3.3.1 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 3.1.5.1. Policies are defined conforming to the DLM. The policies are enforced at compile time by and Information-Flow Security type system 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 [9].

Each variable in Jif has a DLM policy associated with it. A policy is 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 "$\rightarrow$" expresses *Alice* controls the variable x and the variable can be read by *Bob*, the second part with "$\leftarrow$ expresses that bob can write to it [9].

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 [9].

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 [9] [46].

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[4] and parameterized classes[5].

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

3.3.2 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 takes a different approach to defining information flow policies; *Paralocks* [23] [44]. At the core, *Paralocks* is based on the concepts *actors* and *locks* [23]. 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 parameterized locks and *actor polymorphism* allows us to reason about all actors [44].

In Paragon policies are immutable and the following example shows how policies can be defined:

```
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 [49].

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 [23].

It is possible to support policies at runtime with locks [49]. 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 [49].

```
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 relevant features in paragon are policy inference and Java generics.

3.3.3 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 [10]. 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 [10]:

```
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 [29]:

```
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 [10]. 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.

3.3.4 Selection of IFC tool

The approach to selecting a tool is from a programmer's perspective with emphasis on choosing the right tool for developing rather than the tool that achieves the best security. The rationale behind this is that if the tool is too strict, it is impossible to write practical programs in it. The selection of the tool is based on different parameters that falls into two categories; *technical features* and *soft parameters*. The technical parameters are related to features that are necessary for developing the prototype while the soft parameters considers other aspects such as documentation, flexibility and permissiveness.

The technical features are relevant for applying the tool. The tool must have the following features;

- *Define policies:* It must be possible to express where information may flow.
- *Declassification:* Noninterference is too strict and the ability to declassify information is necessary.
- *Policy inference:* The tool can automatically infer labels to avoid retyping labels.
- *Support for external libraries:* An important feature since the tool should be applicable to a Matrix SDK library.
- *Run-time label checking* to enforce dynamic labeling.

The soft parameters are indirectly related to the tool. It can be an overhead developing if the program is to strict hence *permissiveness* is defined as a parameter. The parameter *flexibility* covers the flexibility and restrictiveness of the tool; it should be uncomplicated and straightforward work with the policy language. The final soft parameter is *documentation.*

The table below gives an overview of the three tools related to the parameters.

	Defining policies	Declassification	Run-time label checking	Policy inference	Support for external libraries	Documentation	Flexibility	Permissiveness
Jif	X	X	X	X	X	X		
Paragon	X	X	X	X	X		X	
JSFlow	X		X	x				X

Table 3.1: Comparison of IFC tools

JSFlow is still an immature tool and lack support important features. JSFlow does currently provide support for use with external libraries. It also lacks a mechanism for declassification. JSFlow would have been an interesting tool because of the permissiveness that comes with dynamic enforcement but also since the main Matrix SDK is for Javascript.

Jif and Paragon offer many of the same features with the fundamental difference being the policy language. The policy language of Paragon is expressive, flexible and intuitive. Paragon is less restrictive and more flexible regarding the policies it specifies [44]. The programming experience in Paragon is more as an extension to Java then a different programming language which is the experience you get when programming in Jif. Jif however has a more complete documentation and more code examples to showcase.

Another important aspect is the support for external libraries. Matrix can be used through client SDK's hence support for external libraries is necessary. Both Jif and Paragon supports this by allowing interacting with external Java classes. Both languages must specify a signature for the external Java class.

Based on the consideration and parameters Paragon has been selected as the tool used in the case study.

3.4 Summary

In this chapter the Matrix security model has been evaluated. Matrix provides end-to-end encryption and uses the Double Ratchet algorithm by Signal. To achieve end-to-end security the endpoints need to be secured as well [41] 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.

4 Implementation

This chapter describes the implementation of the prototype. The prototype is a system for sending and retrieving patient journals among different hospitals. The system relies on Matrix as the secure communication channel and storage.

4.1 Journal system

The journal system is inspired by Danish journal system as described in section 1. A journal system serves an important purpose by providing patient journals to different hospitals and clinics. If a patient arrives at the ER and the doctor cannot access the patient's journal then the treatment of the patient gets problematic. A doctor might miss out on important details about the patient or even worse prescribe medication that might give the patient an allergic reaction. The availability of a patient journal is a necessity however the number of medical employees that have access to such a journal has raised privacy concerns. Around 90.000 medical employees have access to patient journals. Consider the scenario where a patient gets referred to a physiotherapist with muscle pain. When the therapist opens the journal the full medical history will be present; if the patient had received psychiatric treatment those session would be readable too. 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 [2][11]. Yet a patient journal is accessible by a large number of unrelated medical employees with the only prevention mechanism being logging and audit trails.

The lack of secure information flow is evident and the prototype demonstrates how Information-Flow control can be leveraged to enforce security policies concerning the information. The journal system is a small distributed system where Hospitals can send, receive and store patient journals. Matrix provides the distributed structure and is responsible for securely storing and transmitting the journals. Paragon provides secure information flow at the endpoints hence providing end-to-end security. The following requirements are defined for the prototype:

- A patient journal contains low (public), medium (confidential) and high (secret) information.
- A patient journal is send and received securely over a channel.
- Hospitals have shared access to patient journals.
- A hospital has two actors: Doctor and Secretary.
- A secretary can only see low parts of the journal.

- A secretary can edit the public parts of a journal.
- A doctor can see the everything up to confidential information.
- A doctor must have the patient in care to gain the journal's secret information.
- A doctor can edit the public part and confidential parts of a patient journal.
- A doctor can only add to a secret fields in a patient journal if the patient has been referred to the doctor.

The following non-functional requirements are defined:

- Confidentiality: the system must ensure the confidentiality throughout the system according to the security policies at all times.
- Integrity: the system must ensure that only intended actors can modify the specific parts of a patient journal.

The current journal systems allows medical employees to access the journals from anywhere. The prototype makes the assumption that the system can only be used within hospitals. Furthermore access for patients to their journal is not supported.

4.1.1 System design

The system is a distributed system that allows multiple clients to send and retrieve patient journals. The system consists of two components; *Matrix* and the *client*. The Matrix component encapsulates the Matrix SDK and provides an interface to Paragon. The client component consumes that interface and can be considered as the endpoint in a communication channel. The client component provides secure information flow for data received through Matrix. Without the interface it would not be possible to achieve end-to-end security in the system. The component diagram in figure 4.1 depicts this.

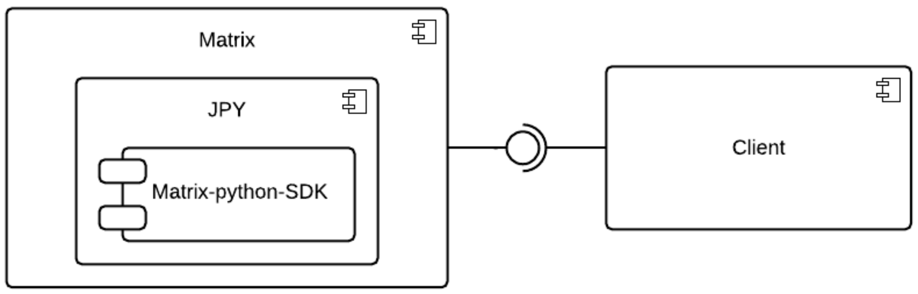


Figure 4.1: Component diagram for the system

Matrix provides several SDK libraries that client application can be build on top of. The most prominent is the Javascript Matrix SDK. It is the best maintained with the largest feature set however it is not compatible with the Java based Information-Flow Control tool Paragon. A Java Matrix SDK exists but it is an early alpha version with end-to-end encryption not implemented yet. The Python Matrix SDK is another major SDK with support for end-to-end encryption in beta. The Python Matrix SDK is used in the Matrix component through a Java-Python bridge *JPY* that can embed Python code in Java as shown in the Matrix component in figure 4.1

4.1.1.1 Matrix

Matrix is an important component in the system. It manages the transmission and storage of patient journals through rooms while managing end-to-end encryption. The encryption mechanism is automatically provided and the SDK manages the session keys under the hood. As described in section 2.2.2 a room is a conceptual place for sending an receiving events and events can be any of any structure. The event history in a rooms is replicated at each homeserver.

The following design choices and assumptions are made regarding Matrix and the system:

- A homeserver represents a hospital server that replicates the history of a patient journal.
- A room represents a single patient journal's version history.
- An event represents a patient journal.
- The latest event in a room is the global state of the patient journal.
- A hospital is represented by a single matrix user that participates in a room.
- A hospital's matrix user is used by doctors and secretaries to access patient journals.

The room can be considerably large since many different types of hospitals needs access to a patient journal. This puts a lot of responsibility on securing the endpoints but also adds concern to who controls the rooms and how hospitals are added. It is assumed that a central authority would be managing all room whom all participants in the room trust. That authority would be the government which are responsible for creating and managing the rooms. Only the authority can invite and remove Hospitals from a room. Hospitals can only join a room if they have been invited.

4.1.1.2 Client

A client represents a hospital with a set of employees that can view some journals. The hospital subscribes to rooms in Matrix that the employees can fetch. The client is a simple console application and is preconfigured to run as a hospital. During program start a list of employees is presented. It is assumed that by selecting an employee the user login as that employee. The user is then presented a list of patients that can be selected. The list is provided by the hospital that keeps track of all journals in a *map* with *SSN* (Social security number) as the key and *matrix room ID* as the value. When the user selects a journal; it is first retrieved from Matrix and the user then receives the journal. It is determined what tasks the user can perform and depending on the user's role the journal can be partially or fully accessible

Figure 4.2 depicts the class diagram for the system.

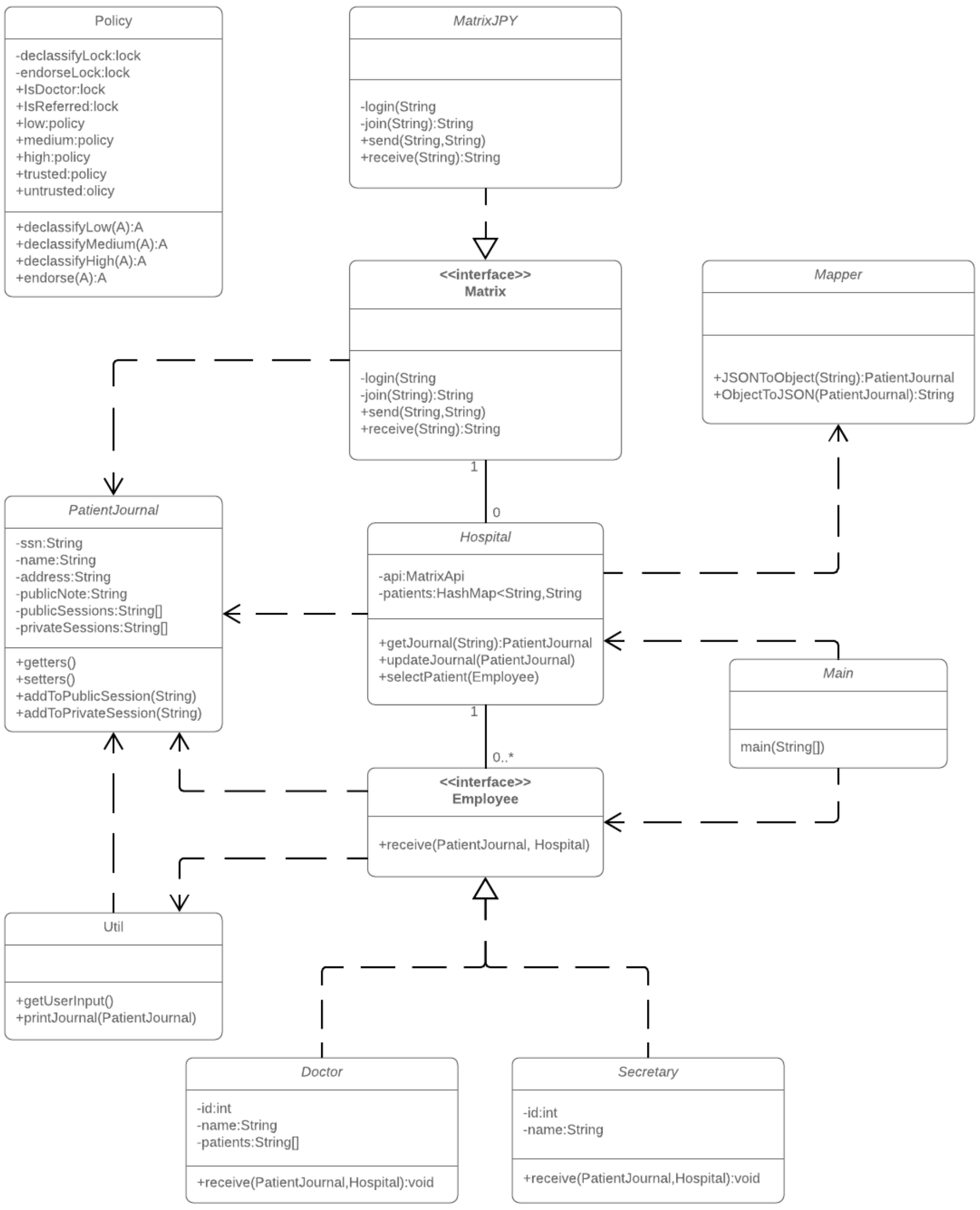


Figure 4.2: Class diagram for the Journal system

4.1.1.3 Limitations

A design problem is the concurrent writes to a journal from multiple hospital. Before writing to a journal; the latest version of the journal in Matrix is first retrieved and then the writes are appended to the journal. However multiple hospitals might have retrieved the latest journal and different doctors might have committed changes to the journal and send it to Matrix hence one of the writes would be lost since only the latest journal is retrieved. A solution to this would be extending the rooms with version control functionality that would track and combine the changes made to a journal.

4.2 Paragon implementation

4.2.1 Matrix interface

The Matrix interface was briefly described in section 4.1.1. The interface is provided through Java using the foreign language binder JPY. JPY is used to leverage the Matrix Python SDK thus make it possible to use Matrix with Java/-Paragon. The interface binds the most important methods in the the Matrix Python SDK like *send*, *retrieve*, *login*, *joinRoom* etc. Without the interface it would not be possible to use Matrix with Paragon. To be able to use the interface; a Paragon *interface* file must be provided. In fact a interface file has to be specified to use any external Java class. The *MatrixApi* interface file is specified as such:

```
public native interface Matrix<policy p> {
  !p void send(?p String message, ?p String room);
  ?p String retrieve(?p String room);
  ?p String objectToJSON(?p PatientJournalD object);
  ?p PatientJournalD JSONToObject(?p String json);
}
```

The interface file describes that the *MatrixApi* class takes a type argument *policy* which specifies what policy the API should enforce. Note that Paragon cannot assert if the external class will respect the policies.

The annotation *?* and *!* represents *read* and *write* effects, and will be explained in the next sections.

4.2.2 Policies

A *policy-first* approach is used when implementing the prototype. The first step is to carefully specify the policies for the system. A journal has several parts that may only be obtained by appropriate users; the policies should uphold that. The policies are defined in the class *Policy* and used throughout the system.

For confidentiality three policies are defined with *low* being the most liberal, *High* being the most strict and *medium* in the middle. The figure 4.3 depicts a lattice for the three policies where information can only flow upwards. In the figure *public* represents low, *confidential* represents medium and *secret* represents high.

Based on the requirements the information labeled with secret should only be obtainable by the Doctor that the patient has been referred to. There could be defined a similar independent secret labels e.g. for information that only the patient's psychiatrist could view.

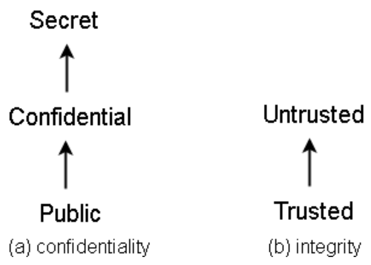


Figure 4.3: Lattice in the system

The lattice in figure 4.3b illustrates the integrity policy and states that data labeled as *untrusted* can not flow to *trusted.* The trusted and untrusted labels serves the purpose of disallowing flows to the journal.

The fields in the PatientJournal class are decorated with a combination of confidentiality labels. The *product lattice* is illustrated in the figure 4.4.

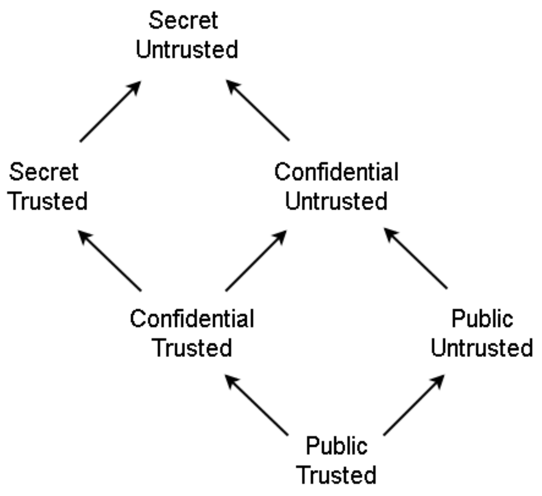


Figure 4.4: Product lattice

The next section will show how the policies have been defined and used in paragon.

4.2.2.1 Defining policies

In Paragon a policy defines what actor information can flow to and under what conditions [49]. Any object can be used as an actor. The confidentiality policies in the prototype uses *Doctor* and *Secretary* as actors.

The policies *low*, *medium* and *high* are defined as this:

```
policy low = { Doctor d: ;Secretary s:};
policy medium = { Doctor d:};
policy high = { Doctor d: IsReferred(d)};
```

Here policy *low* is less restrictive than *medium* since a variable labeled with the policy low can be viewed by the any*Doctor* or *Secretary* actor whereas a variable labeled with *medium* can only be viewed by any *Doctor* actor. The policy *high* is a more interesting policy. It is more restrictive since it can only flow to a specific *Doctor* if the lock *IsReferred(d)* is open. Policy *high* is an example of a dynamic policy which uses a parameterized unary lock.

The following are integrity policies defined as follows:

```
Object untrustedObserver = new Object();
Object trustedObserver = new Object();
policy untrusted = { untrustedObserver :; };
policy trusted = { untrustedObserver:
                 ; trustedObserver: };
```

The actors used here are *untrustedObserver* and *trustedObserver*. The policy *trusted* can be viewed by both observers while *untrusted* can only be viewed by *untrustedObserver*. This captures integrity since a variable labeled with *untrusted* can not flow to *trusted*. Note that actors used here are instances of objects and using them as actors gives the policies a special property of being combinable with other policies [49].

4.2.2.2 Using policies

A policy is used as a label on variables. To express concerns for both confidentiality and integrity the policies described previously can be combined and labeled on variables. Fields in the class *PatientJournal* are defined as:

```
?(Policy.low + Policy.trusted)
private String publicNote;

?(Policy.medium + Policy.trusted)
private String[] publicSessions;

?(Policy.high   + Policy.trusted)
private String[] privateSessions;
```

The *PatientJournal* class has methods that adds a session to *publicSession* or *privateSession*. The class also implements simple getter and setter methods. Common for these methods are they must specify the *read* or *write* effect for the method.

Read effects The read effect specifies what the information policy is. It has already been introduced when labeling the fields in the *PatientJournal* class. When decorating a method with a read effect signature it simply tells what the policy is for the returned type. Paragon ensures that the policy returned must respect the policies of the field or parameters that are in the context of the method. Through read effects explicit flows are captured across methods and fields. This is an example of how a simple *get()* method is defined in *PatientJournal*.

```
?(Policy.medium + Policy.trusted)
public  String[] getPublicSessions(){
    return publicSessions;
}
```

If the read effect instead was *?(Policy.low+Policy.trusted)* then Paragon would have caught it since the field *publicSession* has a more restrictive policy. Note that is a read effect is not specified then Paragon sets the read effect as *?(Object x:)* (the least restrictive policy in Paragon).

Write effects In Paragon the write effect prevents implicit flows. The write effect specifies what context the method can be called in. The context would have to at least as restrictive as the method write effect. The write effect for a method is defined like this:

```
!(Policy.low + Policy.trusted)
public void setPublicNote(
    ?(Policy.low + Policy.trusted) String note){
    this.publicNote = note;
}
```

This method has the write effect *!(Policy.lowD+Policy.trusted)*. Now if this method was called in a method that has the write effect *!(Policy.highD+Policy.trusted)* then there would be an implicit flow and Paragon would detect it.

Read and write effects are important aspect of Paragon since they ensure secure information flow across methods and fields.

4.2.3 Locks

The integrity policy ensures that no information can not flow from untrusted source to variables labeled with *trusted*. However we also want to specify which actors are allowed to change information. A secretary should only be able to edit the field *publicNote* in *PatientJournal*. A doctor should be able to add sessions to *publicSessions* but should only be able to add sessions to *privateSessions* if the patient is referred to the Doctor. This has been achieved through locks. The lock *IsReferred* was introduced when defining the *high* policy. The lock takes a *Doctor* as parameter and becomes open for the doctor. Another parameterized lock is the*IsDoctor* lock that opens during program start if the user is a doctor. The locks are accompanied with 0-ary locks *ReferredLock* and *DoctorLock*. The locks are defined as the following:

```
public static ?(lowD+trusted) lock IsReferred(Doctor);
public static ?(lowD+trusted) lock ReferredLock;
public static ?(lowD+trusted) lock IsDoctor(Employee);
public static ?(lowD+trusted) lock DoctorLock;
```

The locks *ReferredLock* and *DoctorLock* are used on methods with a special annotation that specifies that the method can only be called if the lock is open. By using lock combined with the annotation the requirement for modifying information can be fulfilled. The class *PatientJournal* provides two methods for adding sessions with the annotation:

```
~Policy.DoctorLock
public !(Policy.mediumD + Policy.trusted)
void addToPublicSessions(
    ?(Policy.mediumD + Policy.trusted) String session){
    // Add sessions
}

~Policy.Referred
public !(Policy.highD + Policy.trusted)
void addToPrivateSessions(
    ?(Policy.highD + Policy.trusted) String session){
    // Add sessions
}
```

The lock *DoctorLock* is opened when *IsDoctor* is opened and the lock *Referred-Lock* opens when *IsReferred* is open.

The Paragon compiler might not be able to infer if the lock is opened through compilation. Hence the lock must be checked at run-time and used like this:

```
if(Policy.IsReferred(self)){
    journal.addToPrivateSession(session);
}
```

4.2.4 Declassification

A journal system must be able to view the patient journals through some output channel. Furthermore the system must take user input through an input channel to edit a journal. Hence it is necessary to *declassify* information when using an output channel and *endorse* information when using an input channel. The system provides mechanism for achieving this. We need to revisit the policy definitions to make this possible.

The policy for *System.out* is the least restrictive defined as *?Object a:*. Hence any flow using the defined policies would be rejected. Declassification of information must be done for allowing flow to *System.out*. To achieve this we have to extend the policies with a lock *declassifyLock*:

```
private lock declassifyLock;
policy low = { Doctor d:
             ; Secretary s:
             ; Object x: declassifyLock};

policy medium = { Doctor d:
                ; Object x: declassifyLock};

policy high = { Doctor d: IsReferred(d)
              ; Object x: declassifyLock};
```

After the modification the policy now states that a variable labeled with the policy can flow to any actor if the *declassifyLock* is open. Thus we can specify a method that takes some value as input, opens the lock and returns the value with the least restrictive policy:

```
?bottom
public static <A>
A declassifyLow(?(bottom*low) A x){
    open declassifyLock {
        return x;
    }
}
```

The method expects a parameter with a policy that is at least as restrictive as the policy *low* and *bottom* (variable name for Object x:). The method opens the lock and returns the variable and has the read effect *?bottom*. We provide similar declassify methods for *low* and *high*. However it is an issue that a secretary could call declassify methods for *medium* and *high*. We can use the annotation seen in the previous section to overcome this so only the appropriate actors can declassify:

```
~Referred
?high
public static <A>
A declassifyHigh(?(high*low) A x){
    open declassifyLock {
        return x;
    }
}
```

The approach for endorsement is similar with an *endorseLock* added to the *untrusted policy* and then specifying a method that can endorse a variable.

Declassification and endorsement are methods that should be applied with great care. It should be considered who can declassify, what is declassified, where the declassification occurs and when it can occur. The table below gives an overview of this.

	Who	What	Where	When
Declassification	Anyone	Journal.name	Util.printJournal()	Printing the journal
	Anyone	Journal.ssn	Util.printJournal()	Printing the journal
	Anyone	Journal.address	Util.printJournal()	Printing the journal
	Anyone	Journal.publicNote	Util.printJournal()	Printing the journal
	Doctor	Journal.publicSessions	Util.printJournal()	Printing the journal
	Doctor (Referred)	Journal.privateSessions	Util.printJournal()	Printing the journal

Endorsement	Anyone	User input	secretary.receive()
			doctor.receive()
				After prompting for input

Table 4.1: Overview of declassification

4.2.5 Limitations

4.2.5.1 Matrix interface

When a journal is passed to the interface the encryption is not performed until the Python code is executed. Hence there exists a layer between Paragon and Matrix where the data is unencrypted and where Paragon policies cannot be enforced.

This could be solved by applying encryption before passing data on to the interface and have assurance of confidentiality throughout the system.

4.2.5.2 Concurrency

Concurrency is unsupported in Paragon and has been identified as an area for future works [49]. Concurrency might be an issue for locks that shares the state.

4.2.5.3 Exception handling

Another limitation of the prototype is that exceptions are unexplored. Any exception is a potential channel for implicit flows. The tools provided by Paragon such as read effects, write effects and locks can be used to properly handle exception.

4.3 Summary

The prototype implements a distributed journal system. The journals in the system must be stored and transmitted securely and once received at the endpoints confidentiality and integrity must still be preserved. The system uses Paragon to express and enforce security policies at the endpoints and Matrix to transmit the journals. The system design has been presented with implementation specific details related to Paragon.

5 Discussion

The previous section applied Paragon as a programming language and demonstrated how policies can be defined and enforced. Hence achieving stronger security guarantees. This chapter evaluate the programming experience in Paragon with Matrix and considers what Information-Flow Control offers to the prototype.

5.1 Paragon

Programming in a Information-Flow Control can be a challenging task. Paragon provides secure information flow by defining policies and labeling variables and methods. It forces the programmer to change its mindset and have more concern about the flow of information in the system.

5.1.1 Defining policies

The policy language in Paragon is flexible and expressive. The same policy can be expressed in several ways. Consider the policies for high and low:

```
Object observer = new Object();
Object highObserver = new Object();

policy lowA = {Object x:};
policy highA = {:};

policy lowB = {Object x:};
policy highB = {observer:};

policy lowC = {Secretary s:; Doctor d:};
policy highC = {Doctor d:};
```

Each encoding of the low and high policies has some differences e.g. *lowA* has the least restrictive policy while *lowC* allows flows to any doctor or secretary. Even though *lowC* is low it is encoded in such way that it cannot flow to an output channel without declassification. There are even more possibilities when introducing locks as seen in the prototype. The prototype relies heavily on locks as for enabling declassification and endorsement of information. Another important usage of locks are to ensure that method can only be called depending on the state of the lock. This is how it is guaranteed that a secretary cannot add to *privateSessions* or *publicSession*. This guarantee could also have been achieved by extending the integrity policies and use write effects to control what context methods can be called.

Since policies can be defined in many ways; the task of defining policies should be done with care and consideration for what information flows should be captured by the policies.

5.1.2 Pros

Some of the benefits of Paragon has already been mentioned in section ??. The main benefits are the flexibility of the policies and how policies can be combined. Another advantage is how programmer-friendly it is. Paragon might seem alien at first glance but once the Paragon concepts of policies, locks, write and read effect become clear the experience is more like programming in Java with some extra decorators for methods and fields.

5.1.3 Cons

There are some overheads related to the Paragon compiler. Some of them are listed below:

- If Class A depends on Class B then Class B must be compiled first otherwise the compilation will fail. This becomes problematic when several classes might depend on each other.
- Generally good error messaging but at times very vague.
- Lack of support for common Java features such as the class *java.lang.Class*, *foreach* loops.
- Poor support for inheritance. An interface with method definitions that has a return type will for some reason give compile errors. Only interfaces void methods could be compiled. *@Override* annotation is not supported.
- No support for *foreach* loops.
- The compilation time from Paragon to Java is slow. The prototype is a rather small application and still takes up to 1 minute to compile.
- That Paragon program could successfully compile but errors would be detected by the Java compiler.

These shortcomings makes it tedious

5.1.4 Matrix

In the evaluation of Matrix security we found that Matrix is capable of having several security properties such as confidentiality, integrity, forward and backward secrecy which is achieved through the end-to-end encryption.

Matrix primary use case is as a secure messaging protocol however it has a wide range of use cases. If Matrix is used as a secure communication channel like in the use case for IoT described in section 2.3.4, then the end-to-end encryption is not enough to achieve confidentiality and integrity throughout the system.

Information-Flow Control is a mechanism that aid in achieves stronger security guarantee.

The interface between Matrix and Paragon makes it possible to combine the end-to-end encryption security guarantees with policy enforcement through Paragon hence achieving end-to-end security. The interface provides few basics methods from Matrix Python SDK that were necessary for the prototype. It is challenging to setup and use the foreign binding tool JPY. However once becoming comfortable with JPY the interface can easily be extended with more methods from the Matrix Python SDK.

Due to the lack of concurrency support in Paragon it is not possible to implement secure messaging applications since such application must listen to events in a different thread.

5.2 The Policy system

The prototype displays major improvements in terms of confidentiality and integrity in the system. The prototype is secure by construction and gives a guarantee that the defined policies about the information is enforced. Consider the Java code where a secretary handles a patient journal:

```java
public class PatientJournal{

private String publicNote;
private String[] publicSessions; // Unaccessible to
    secretary
private String[] privateSessions; // Unaccessible to
    secretary

}

public class Secretary {

public void receive(PatientJoural journal){
// Perform task on journal
}
```

The Java compiler would be helpless in detecting if the secret parts of the journal are unintentionally accessed or modified. In the prototype such unintentional access or modification of information would be detected immediately by the Paragon compiler. This is a strong guarantee that Information-Flow Control tools offer.

The prototype also demonstrate improvement to an actual problem in the current journal systems. The section xx describes how the only mechanism is auditing and logging trail and clearly a mechanism for enforcing policies would be of great benefit.

The prototype is by no means a full-fledged journal system and has some obvious limits. The prototype lacks support for concurrency and assumes that only a single user can use it at a time. The prototype does not handle the issue with concurrent writes described 4.1.1.3 which would be insufficient in a real journal system. Despite the limitations of the prototype; improvements to the overall security guarantees have been demonstrated by combining Matrix' end-to-end security with Paragon's enforcement of security policies to ensure end-to-end security.

5.3 Summary

Paragon provides a flexible and expressive way of defining policies. Paragon can be used on top of Matrix through the provided interface. End-to-end security is achieved by extending the security guarantees from Matrix' end-to-end encryption with Paragon's enforcement of specified policies. This is demonstrated by implementing a prototype inspired by the Danish journal system.

6 Conclusion

The criterias of a successful project described in section ?? are presented below:

- Evaluation of Matrix security model
- Survey IFC tools to improve the Matrix security model
- Implement a prototype distributed system running on Matrix
- Demonstrate improvements to the Matrix security model

Matrix was evaluated in terms of secure messaging systems. The evaluation found that the end-to-end encryption provided by Matrix is sufficient and several security properties are ensured. However information leak can occur at the endpoints and additional mechanisms must be applied. Information-Flow Control is such mechanism that can prevent leaks at endpoints hence provide stronger security guarantees.

For improving the security guarantees Matrix can be combined with a Information-Flow Control tool. The tools Jif, Paragon and JSFlow was considered. Paragon was selected for the flexibility and expressiveness for defining policies.

A prototype inspired by the Danish journal system has been implemented using Paragon on top of Matrix. To be able to use Matrix with Paragon an API for Matrix has been provided. The API is the main technical contribution of the thesis.

Improvements have been demonstrated how by providing enforcement of security policies at the endpoints hence achieving end-to-end security.

Appendix

The source code for the prototype can be found at *https://github.com/Ansuddin/thesis-code-handin*

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] Etablering af ét nationalt system til elektronisk patientjournal (EPJ). URL https://www.borgerforslag.dk/se-og-stoet-forslag/?Id=FT-01508.

[4] The Facebook and Cambridge Analytica scandal, explained with a simple diagram - Vox. URL https://www.vox.com/policy-and-politics/2018/3/23/17151916/facebook-cambridge-analytica-trump-diagram.

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

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

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

[8] 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.

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

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

[11] 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/.

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

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

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

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

[16] 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/`.

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

[18] 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.

[19] 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.

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

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

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

[23] 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.

[24] 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.

[25] 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`.

[26] 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`.

[27] 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`.

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

[29] 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`.

[30] 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/`.

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

[32] 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`.

[33] 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/`.

[34] 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`.

[35] 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.

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

[37] 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.

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

[39] 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`.

[40] 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/`.

[41] 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.

[42] 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.

[43] 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.

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

[45] 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`.

[46] 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.

[47] 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`.

[48] 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.

[49] 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`.

[50] R. K. Yin. *Case study research: Design and methods (5th ed.).* 2014. ISBN 9781452242569.