Secure policy-based communication for IoT devices on LAN, as implemented by Trifork

Michael Messell

Kongens Lyngby 2016

Technical University of Denmark
Department of Applied Mathematics and Computer Science
Richard Petersens Plads, building 324,
2800 Kongens Lyngby, Denmark
Phone +45 4525 3031
compute@compute.dtu.dk
www.compute.dtu.dk

Summary

English

The goal of this thesis is to establish a secure line of communication for IoT devices. The IoT devices are *CPEs*, as used by Trifork for their Secure Device Grid. A *CPE* device is placed in customers' homes and acts as a control unit for a home automation system. By connecting *CPEs*, different home automation systems would be able to communicate. This would give customers the opportunity to make automation processes across multiple home automation systems, instead of them only working within their own home automation system. The thesis investigates state of the art cryptographic techniques, and the existing system of Trifork, to gain knowledge of the techniques required to design and implement a secure line of communication. Protocol requirements are defined based on Trifork's security goals and the Dolev-Yao attack model. Two protocols are considered to facilitate secure communication between home automation systems: the ISO/IEC 11770-2 Key Establishment Mechanism 6 (ISO-6), relying on symmetric encryption; and the Station-to-Station protocol, relying on asymmetric encryption. Efficiency tests results in the selection of ISO-6 for the secure protocol. A prototype using the ISO-6 protocol is created, where the Networking and Cryptography library (NaCl) is used for cryptographic computations. Finally the implemented version of the ISO-6 protocol is tested using the OFMC model checker. OFMC doesn't find any attacks for the implemented version of the ISO-6 protocol.

Danish

Målet for denne afhandling er at etablere sikker kommunikation mellem IoT enheder. Disse IoT enheder er *CPEer*, som bruges af Trifork med deres Secure Device Grid. Enhederne er placeret i kunders hjem, hvor de fungerer som kontrolenheder for automatiseringssystemer i hjemmet. Hvis man forbinder flere *CPEer*, giver det mulighed for kommunikation på tværs automatiseringssystemer. Det giver kunder mulighed for at automatisere på tværs af automatiseringssystemer, i stedet for at de enkelt automatiseringssystemer kun arbejder inden for deres egets systems begrænsninger. Afhandlingen undersøger relevante kryptografiske teknikker, Triforks eksisterende system og bruge det som afsæt for designet og implementationen af den sikre protokol. Kravene for protokollen er defineret ud fra Triforks sikkerhedskrav og med udgangspunkt i en Dolev-Yao angrebsmodel. To protokoller er undersøgt som protokol mellem *CPEer*: ISO/IEC 11770-2 Key Establishment Mechanism 6 (ISO-6), som benytter sig af symmetrisk kryptering; og Station-to-Station protocol, som benytter sig af asymmetrisk kryptering. Test af protokollernes effektivitet afslører ISO-6, som den bedre protokol. Der er lavet en prototype af Triforks system som benytter sig af ISO-6 protokollen til at kommunikere sikkert. Her benyttes Networking and Cryptography library (NaCl), som bibliotek til at udføre kryptografiske beregninger. Test af den implementerede protocol med OFMC afslører af intet angreb er fundet for den implementerede version af ISO-6 protokollen.

Preface

This thesis was prepared at DTU Compute in fulfilment of the requirements for acquiring an M.Sc. in Engineering.

I would like to thank Associate Professor Christian Damsgaard Jensen for giving counsel and help seeing things from a different perspective. Thank you!

The thesis is created in cooperation with Trifork. Trifork has an issue with an already existing product: the Secure Device Grid. The Secure Device Grid is a safe scalable infrastructure, which is used for NAT relaying of devices. Specifically connecting smartphones with IoT devices for home automation systems. The two devices are paired with each other using the Secure Device Grid, which also relay all future traffic between the two.

Trifork's challenge is homes containing more than one IoT devices, called *CPE*. A *CPE*is a control box for a home automation system, e.g.. an automated heating system. The issue is that a customer might have more than one *CPE* in its home, e.g. an automated heating system and a automated windows control system. These systems will each have a *CPE*, one for controlling the heating and one controlling the windows, and they will work as two individual home automation systems. Trifork would like to make the two home automation systems seem like one home automation system. This means that the two systems should be able to communicate, so that the customer will be able to have e.g. the heating turn off in a room when a window has been opened.

This means that the *CPEs* would have to communicate. This communication should be able to use the LAN with a point-to-point connection between the *CPEs*. The connection must be secure. It is important for data to be kept confidential, and that data exchanged between the *CPEs* can be trusted. The design of the solution should provide a secure line of communication for the *CPEs* to communicate using the LAN, and a prototype working independently of the existing Secure Device Grid.

Lyngby, 29-April-2016

Michael Messell
s140045

Notation for protocols

This description explains the protocol syntax use throughout the paper. First by explaining the different modules used in the protocol and ending with a protocol example, Protocol 1.

A is the identity of a protocol participant.

B is the identity of a protocol participant.

T is the trusted entity in the protocol.

I is the the identity of the intruder. The party trying to attack a protocol.

x, y, i indicates the private values selected in a Diffie-Hellman key exchange: $x \rightarrow A$, $y \rightarrow B$, $i \rightarrow I$.

p is a prime defining prime-order subgroup Z_p^*, and is used as modulus for Diffie-Hellman key exchange.

α^x is the public session key value of A and α is a primitive root of p.

xG is the public session key value of A in a Diffie-Hellman key exchange using elliptic curves.

pkA is the long term public key of A.

a is the long term private key of A.

K_{AB} is the long term shared key of A and B.

N_A is a nonce generated by A.

R_A is a random number generated by A and can be used as input for key generation.

$[message]$ indicates the hash of a message.

$sign_T(message)$ indicates that a message has been signed by the trusted party T.

$Cert_A$ is the certificate of A containing: $A, pkA, T, sign_T([A, pkA, T])$

$\{message\}K_{AB}$ is a symmetric encryption of message using the shared key of A and B.

$\{message\}K_{xy}$ is a symmetric encryption of message using the session key generate based on A's and B's private keys in a Diffie-Hellman key exchange.

$\{|message|\}K_{AB}$ is a symmetric encryption of message using the shared key of A and B including data integrity.

Protocols will be recognised throughout the thesis by the number just right to the description. For the protocol right below, the number 1.

$$\begin{aligned}
&1.\ A \rightarrow B : N_A \\
&2.\ B \rightarrow A : sign_B(N_A) \\
&3.\ A \rightarrow B : message
\end{aligned} \tag{1}$$

Contents

Chapter 1

Introduction

Internet of Things (IoT), is often defined as bringing physical objects online. IoT is merging the real world of physical objects with the virtual world of the internet. According to *Gartner*, IoT has been among the top 10 strategic technology trends for the last four years [Gara][Garb][Garc][Gard]. *Gartner* identifies IoT to have four basic usage models: manage, monitor, operate and extend. Four usage models that could basically include anything, but also four usage models that encourage automation of manual processes. This has made IoT popular in the market for home automation, where putting physical object on the LAN, has made it possible to make more flexible and cost effective systems that don't rely on wiring. Considering a heating system, being automated by relying on IoT. The customer would be able to operate the system, by setting a desired temperature for each room. IoT sensors could then monitor the rooms, and ensure that the desired temperature is maintained by regulating the IoT thermostats. The setup keeps the customer informed of the temperature for each room. This allows the customer to improve utilisation of heating resources, which means that the customer is able to better manage the heating of the home. Such a home automation system covers the three usage models: manage, monitor and operate; but not the extend usage model. This thesis addresses the extend usage model, as will become apparent later.

Trifork works together with manufacturers to supply home automation systems, which offers similar services as the heating system. The manufacturers are responsible for the monitor process, and Trifork is responsible for the managing and the operation of the IoT devices. For a heating system this means that the manufacturer would regulate the room temperatures based on the rooms' temperature settings and the current temperature of the rooms. The monetising process is tied to the *event-condition-action* paradigm. An *event* is evaluated by the *condition*, which if true triggers the *action*. When related to a heating system with IoT devices, the events would be produced by sensors and the actions performed by thermostats. Events could simply be the current temperature, where the condition would assess if the temperature is within the defined range. Should the temperature be lower or higher than the defined range, the condition would trigger the thermostat to either increase or decrease the heating. In using the *event-condition-action* paradigm with IoT; It is easy to imagine sensor triggering events and actuators performing actions, but for the condition no such apparent solution presents itself. One option is to include the condition in the individual thermostats of the system. This approach turns out to be quite expensive, as each actuator would need to be provided with extensive resources. What is usually done, and what is also done in the cooperation between Trifork and manufacturers, is to have a centralised control unit, which evaluates events received from sensor and instructs actuators to perform accordingly. The control unit is also an IoT device. Having a centralised control unit, or a *Customer premises equipment (CPE)* as Trifork calls them, has an added benefit. When Trifork implements management and operation, they don't need to communicate directly with all sensors and actuators. They instead rely on the control unit to relay the secure communication to and from sensors and actuators.

Home automation systems, such as the heating system, have already been implemented. Further cooperations with other manufacturers are investigated. Working with different manufacturers, when designing automated home systems, results in having multiple control units, as manufacturers are interested in having their own control unit. This somewhat cripples the automation process, as the heating system would be able to only react to events from the heating system. This mean that if you have two home automation systems, you would have two control units, that wouldn't be able to react to each others' events, even if this might be relevant to the customer.

This thesis suggests a solution for a policy-based home automation system, where multiple control units are available. More specifically, the thesis concentrate on how events from a sensor belonging to one home automation system may result in an action for an actuator belonging to another home automation system. The challenge is that the current home automation systems don't support this form of secure communication. To solve this, the current policy-based system is formalised and extended to include multiple control units.

A protocol is designed, which facilitates the secure communication between home automations systems. The protocol supports confidentiality, authentication and data integrity, which are needed for this system to be considered secure. The protocol facilitates the secure distribution of a sensor event from one home automation system to an actuator of another home automation system.

This protocol is implemented. It is implemented separately from the actual system of Trifork (Secure Device Grid) as a prototype. It is implemented using virtual linux machines, which acts as the IoT devices: control units (*CPEs*), sensors and actuators. The security of the protocol designs are evaluated using a model checker (*OFMC*), to ensure that designs doesn't introduce security flaws. The overall flow of the implemented protocol is evaluated using unit tests.

Working with the challenges faced by Trifork, has given me a chance take a deep dive into modern cryptographic techniques. I have learned that a lot of thought goes into defining the requirements for a secure protocol, as you will have to consider which type of attacker you are protecting your system against. The design reflects the requirements and setup of your system. The protocol used could then be based on either symmetric- or asymmetric encryption or both depending on these requirements. The symmetric encryption usually turns out to be much more efficient than asymmetric encryption, on the other hand symmetric encryption doesn't scale as well and doesn't provide non-repudiation. The two techniques are therefor often combined, to get the best of both encryption schemes. The final design takes all these things into consideration before settling for the best design for that system. For this system the symmetric encryption scheme turned out to be the better fit.

The thesis starts of by explaining the *Motivation* of Trifork and how it's relevant from an academic stand point. The motivation is then refined into a more tangible *Goal* to fulfil. The *State of the art* of cryptographic techniques is then investigated, to know which techniques are need to design a secure protocol and to give a base to take more responsible choices for the secure protocol. The goal is then further refined into *Protocol requirements*, which should be fulfilled by the secure protocol. The *Protocol design* then decides on a design for the secure protocol, which fulfils the protocol requirements and also provides an efficient solution for the existing system's setup. The *System design* takes a wider look at how the system for the prototype is designed: How is the secure protocol triggered and what is the effect of this trigger. The *Implementation* looks at how the secure protocol is implemented, which cryptographic techniques are used, and how are they used with the implementation of the protocol design. The *Evaluation* explains the tests made to insecure that the works and is in fact secure. It also explains the setup of virtual machines used for visualising the use of the prototype. Finally the *Conclusion* summarise the findings of the thesis and the academic relevance of these findings.

Chapter 2

Motivation

The motivation section presents the setup of the current system, and why an extension is interesting for Trifork and from an academic standpoint.

2.1 Trifork's motivation

To reap the benefits of the growing IoT marked, Trifork has created a solution called *Secure Device Grid*[Tri]. The *Secure Device Grid* lets a customer communicate securely with the control unit of the customer's home automation system, with a smartphone. The customer is able to manage and operate the system, when not even at home, e.g. when at work or vacation.

This section will introduce the different IoT devices, consider them in a home automation system and clarify why an extension to the current system is needed.

2.1.1 Customer-Premises Equipment (CPE)

In the context of the *Secure Device Grid*, the control unit is called a *CPE*. An illustration of a *CPE* can be seen in figure 2.1. The *CPE* is an IoT device, which allows it to communicate using the LAN.

The *CPE* is created by a manufacturer. A manufacturer could be a company providing heating to customers' homes. This manufacturer is interested in selling *CPEs* to customers, which automate the heating process in the customers' homes. The *CPE* has a series of sensors and actuators connected wirelessly to it. The manufacturer has written the software facilitating the communication with the sensors and actuators. This means that the software Trifork is providing and thereby the software of this thesis, cannot communicate directly with sensors and actuators, but needs to access them through the manufacturer API. If needed for a home automation system, the manufacturer software will monitor the system. For an automated heating system, this would mean that the manufacturer software ensures that the correct temperature is maintained based on the desired- and current temperature.

Trifork's software, on the *CPE*, facilitates the secure communication with the smartphone. In the context of the automated heating system the communication would provide the manufacturer software with the desired temperature (operate), and provide the smartphone with the current temperature of the individual rooms (manage).

To recap, the manufacturer provides a home automation system and Trifork provides the setup allowing a smartphone to communicate with the home automation system.

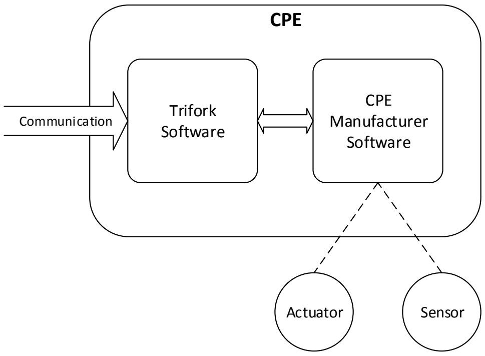


Figure 2.1: Detailed CPE view.

Throughout the thesis, we will use an example scenario, in which two different *CPEs* will be referenced; *CPE A* and *CPE B*. The two *CPEs* are examples, and help to provide a more realistic view of a *Secure Device Grid* system's capabilities.

CPE A can control the windows in a customer's home. The sensors of *CPE A* informs the manufacturer software when a window has been opened and when a window has been closed. The actuators of the *CPE A* facilitates the automatic opening- and closing of windows, by instruction from the manufacturer software.

CPE B is the *CPE* previously used as an example, the heat regulating *CPE*. The sensors of *CPE B* keeps the manufacturer software updated with the current temperature. The actuators control the radiators, thereby increasing or decreasing the temperature, by instruction from the manufacturer software.

2.1.2 Client

A *Client* corresponds to a customer's smartphone. For the rest of this thesis the smartphone will be referenced *Client*. It is more fitting, as the device communicating with the *CPEs* is not limited to a smartphone, but could be any device with the capability to access the internet. This software on the *Client* is provided entirely by Trifork.

2.1.3 Secure Device Grid server

The *Secure Device Grid Server* relays messages between *CPEs* and *Clients*, which are connected to different networks. This means that it's allows for devices that resides behind different firewalls to communicate securely.

2.1.4 Secure Device Grid setup

A *Secure Device Grid* setup with one *CPE*, like seen in figure 2.2, offers a wide range of automated processes for the customer.

Considering a setup with only *CPE B*. Customers would be able to set a different temperature for each room equipped with a sensor and an actuator. Room temperature could also be set to change during the day. To save money and not compromise the comfort, customers could make intervals where the radiators should be turned off. This could be during work hours or during the night where a lower temperature might be preferred. It could also be during a longer absence from home like vacation. The *Secure Device Grid* allows the customer, should they forget to turn off the radiators, to do it from anywhere in the world. The customer can also turn on the radiators, when on their way home from vacation, to ensure that their home is nicely temperate. This scenario, however, has no way to deal with the situation where the room becomes too hot.

Considering a setup with only *CPE A*. *CPE A* allows the customer to open and close windows from the *Client*. It makes it easier to open and close windows, that might be hard to reach for the customer, e.g. skylights. Ventilating is also done much easier. The customer could open all windows at the same time from the *Client* with a single touch, instead of manually opening each of them. The command could ensure that the windows automatically closes after five minutes to further help the customer.

A home containing both the home automation system of *CPE A* and the home automation system of *CPE B* would be able to carry out these automated processes and thereby automating a lot of manual work for the customer.

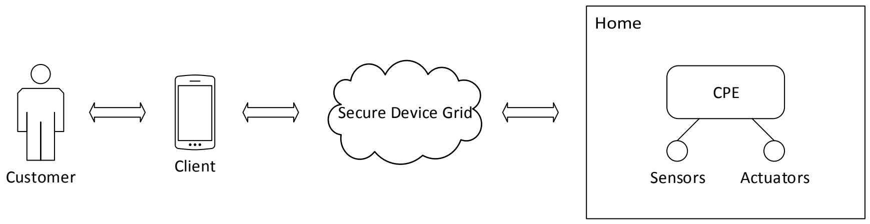


Figure 2.2: Secure Device Grid setup with 1 CPE.

Under the existing system, a home containing two *CPEs*, as shown in figure 2.3, could be thought of as two individual setups with one *CPE*. The *CPEs* have no knowledge of each other's existence.

The obvious question is: instead of having a *CPE A* and a *CPE B* why not just have one *CPE* controlling both windows and heating. The reason leads back to the manufacturers that Trifork work with. They aren't interested in sharing *CPEs* with one another, which results in homes with multiple *CPEs* unable to communicate. Having multiple *CPEs* also means that the customers can only automate processes within the individual home systems. This makes it impossible to have a sensor event of *CPE A* result in an actuator action in *CPE B*, and vice versa.

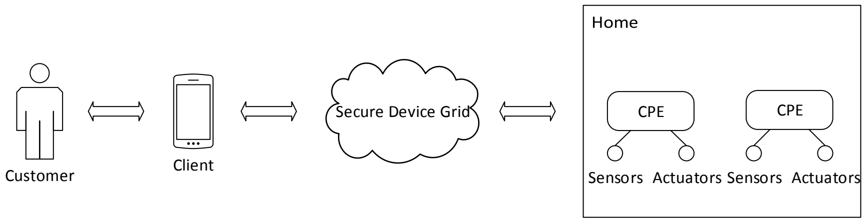


Figure 2.3: Secure Device Grid setup with 2 CPEs.

The current setup doesn't allow *CPEs* to communicate. The customer's *CPEs* are connected to the same LAN, making direct communication between them possible. If the *CPEs* are connected using the LAN, this would make it possible for sensor events of *CPE A* to result in actuator actions in *CPE B*, and vice versa.

The idea is to connect *CPE A* and *CPE B* with Trifork's software and thereby give the customer the illusion of an uniform home automation system, where a sensor event of *CPE A* results in a actuator action in *CPE B*. The customer would then be able to automate processes across *CPEs*. This approach wouldn't affect the companies creating the *CPEs*, as this could be implemented entirely by Trifork and distributed with Trifork's software.

An example of a new automated process could be turning off the radiator, when a window has been opened. The customer could then just open the window without worrying about turning off the heat. A similar automation could be made to ensure that the radiator is turned on again when the window has been closed.

This thesis is working with the problem of having a sensor event from one *CPE* result in an actuator action in another *CPE*. In addition to distributing the sensor event using LAN, the distribution must also be done securely.

2.2 Academic motivation

This section put Trifork's motivation into an academic perspective, to see how Trifork's motivation can translate into relevance for a thesis. Essentially Trifork needs a way to have two IoT devices communicate securely on a LAN.

How one *CPE* informs another *CPE* of a sensor event needs to be formalised. This is needed so we know which information is sent between *CPEs*, and how the internal software should process this information. It also helps to know which kind of information is exchanged from a security standpoint. It would be academically relevant to formalise the exchange of information from sensor to actuator as a policy-based system.

As for security between the *CPEs*, an investigation into what is required to consider an exchange of information secure, is needed. How the security requirements are met would also need an investigation. Here the state of the art authentication- and key establishment protocols, and the underlying cryptography they rely on are interesting. It would also be necessary to investigate Trifork's existing solution for secure communication in order to compare it to the alternative solution of this thesis.

2.3 Summary

The motivation of Trifork is to give a collection of different home automation systems, using *CPEs*, the illusion of being on home automation system. Allowing customers to create relevant scenarios across *CPE*, e.g. having the heat turned off, when a window is being opened. For this to happen, the *CPEs* will need a way to communicate and also to communicate securely.

The thesis therefor will need to investigate how to facilitate secure communication between *CPEs*, by looking into cryptography, protocol and how to translate events from one *CPE* to actions of another.

Chapter 3

Goal

The challenge with having actuators from one *CPE* act on sensor events from another *CPE*, is that no communication channel exists between the two.

The *Client* might be able to communicate with both *CPEs* securely through the *Secure Device Grid*, and thereby theoretically act as a relay server between the *CPEs*. This approach, however, is not possible, as the *Client*, being a smartphone, can't be expected to be available all the time. The approach isn't desirable anyways, as the *Client*'s battery life would decrease due to an increase of network traffic, and because latency between *CPEs* would be higher compared to using direct communication over the LAN.

Guidelines need to be created for how the communication from a sensor event to an actuator action is facilitated. This should be considered for home automation systems containing one or more *CPEs*. These guidelines would need to be addressed before considering, which additional means of secure communication is required, and before considering what is required for a communication to be considered secure.

3.1 Policy-based systems

A policy-based system consists of a set of policies determining how a system acts to certain events within that system. To enforce the policies of a policy-based system, it is done using the structure: *event-condition-action.*

When a policy-based system produces an *event*, this *event* triggers the *condition.* The *condition* then evaluates the *event* against the policies of the system, determining if the *event* should result in a certain *action* from the system. This way the policies of a policy-based system, determine how the system should act.

The policies, which the *condition* evaluates *events* against, are constructed using the form: if *event* then *action.* As there can exist many policies for a system, one *event* might result in multiple actions, and one action might be caused by multiple events.

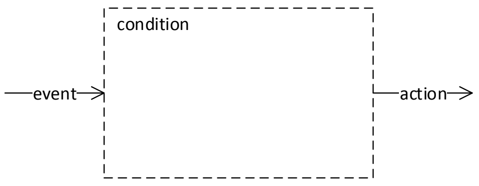


Figure 3.1: The *event-condition-action* structure.

The home automation system, in which Trifork are participating, follows the structure of a policy-based system. It is unclear if the manufacturer is aware of this. The structure, *event-condition-action*, enforcing the policies, are easily applied to a home automation systems using *CPEs.* In such systems the sensors would produce the events, the actuators would perform the actions and the *CPE* would be the condition, containing the logic enforcing the policies.

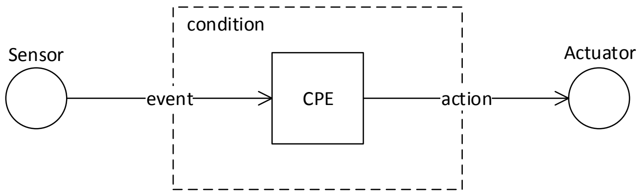


Figure 3.2: The *event-condition-action* structure using a *CPE.*

A policy-based system, using the *CPE B* heating automation system, could have multiple policies in place to ensure the desired temperature of the home. To ensure the desired temperature of the living room the following 2 policies are created. The policies are created on the assumption that *CPE B* receives events from a sensor corresponding of the current temperature of that specific room; and that the actions performed by the actuator increase or decrease the heat intensity of that specific room, where the heat intensity doesn't correspond to a specific temperature.

- **if** living room temperature exceeds 21 °C
  then decrease living room temperature
- **if** living room temperature is below 19 °C
  then increase living room temperature

The desired temperature for the living room, according to the policies, is 20 °C. The sensor, measuring the temperature of the living room, will periodically inform the *CPE* of the current temperature. If the temperature of the living room is too high, the *CPE* will inform the actuator to perform the action, *decrease living room temperature*; and if the temperature of the living room is to low, the *CPE* will inform the actuator to perform the action, *increase living room temperature.*

The policy-based system of the *CPE B* home automation system will ensure that the desired temperature of the living room is maintained. The system has a very low coupling, where sensors and actuators can easily be added to the system, and similar policies created to ensure the desired temperature of the rooms in question.

The challenge is now to apply the principle of a policy-based system to a home containing two *CPEs*, *CPE A* and *CPE B*. The goal is to have an event from one *CPE* result in an action from the other *CPE.* For now, the fact that this is not possible in the current system is ignored. A policy is created which depends on both home automation systems. The event is the living room window being opened, which is an event of the *CPE A*; and the action is turning off the heating of the living room, which is an action of the *CPE B.*

- **if** living room window is opened
  then turn off living room heating

Considering the *event-condition-action* structure of a policy-based system, the events are still provided to the condition from the sensor and the action is still performed by the actuator when the condition is fulfilled. Trifork cannot have one *CPE* communicate with sensors and actuators of another *CPE*, as this communication is facilitated by the manufacturer's software. This means that, instead of having one *CPE* enforcing policies, the structure now relies on two *CPEs* to enforce policies of the condition.

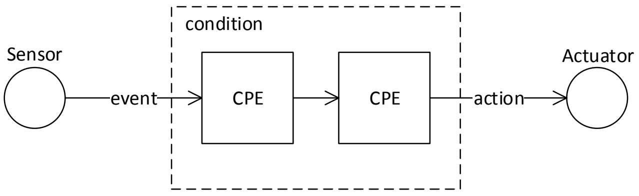


Figure 3.3: The *event-condition-action* structure using two *CPEs*.

This mean that the policy, *if living room windows is opened then turn off living room heating*, is applied over two *CPEs* instead of one. In practice this means that the original policy is split in two, and one part of the policy is provided to each of the *CPEs*. The event of the original policy is used as the event for the policy of *CPE A*, and the action of the original policy is used as the action for the policy of *CPE B*. This leaves *CPE A* without an action to perform and *CPE B* without an event to react on. These should facilitate the communication between the *CPEs*, where *CPE A* informs *CPE B* of an event happening. This means the action of *CPE A* corresponds to the event of *CPE B*.

Original policy:	**if** living room windows is opened
	then turn off living room heating
CPE A policy:	**if** living room windows is opened
	then inform *CPE B* of event
CPE B policy:	**if** informed of event by *CPE A*
	then turn off living room heating

Having two *CPEs* for a policy-based system introduced the need for having the *CPEs* communicate. A channel not previously considered by Trifork, and therefor not covered under the secure communication of the *Secure Device Grid*.

3.2 Policy-based security considerations

This section considers the interactions needed to enforce policies for a policy-based system containing two *CPEs*. The *CPEs* work together to give the customer the illusion, that the home automation systems of *CPE A* and *CPE B* are one home automation system, where policies can be created across *CPEs*. A challenge for the system is that the *CPEs* aren't aware of each other's existence, and that policies should be provided. The idea is to let the *Client* make the *CPEs* aware of each other's existence and to let the *Client* distribute the individual policies to the *CPEs* previously discussed. The *CPEs* both have a secure channel through the *Secure Device Grid* to the *Client*.

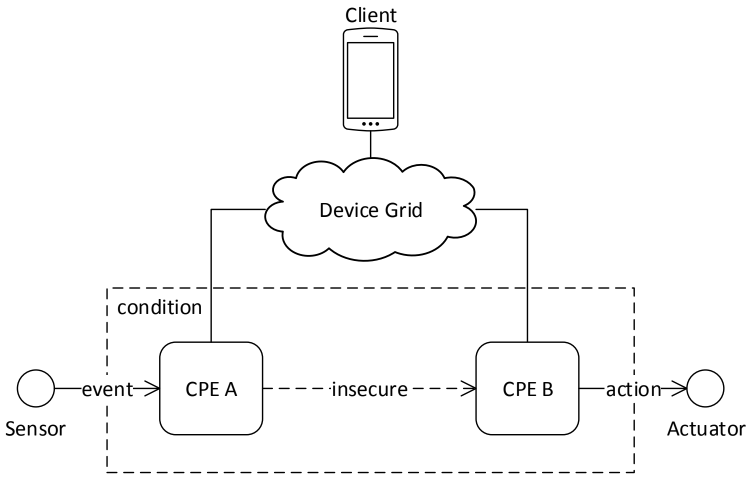


Figure 3.4: A home automation system containing 2 *CPEs*.

This means that in order to enforce policies, for a policy-based system containing two *CPEs*, the following interactions are needed: an interaction from the *Client* distributing the policies and making the *CPEs* aware of each other's existence; and an interaction between the *CPEs* following the structure *event-condition-action* for two *CPEs*. Below the interactions needed to distribute and enforce policies are described.

1. The *Client* provides *CPE A* and *CPE B* with individual policies
2. Hiatus until a sensor event, window opened, is sent to *CPE A*
3. If sensor event satisfies *CPE A*'s policy, it results in an action
4. *CPE B* is informed of sensor event. If sensor event satisfies *CPE A*'s policy, it results in the action, turn off heating
5. *Sequence starts over from 2.*

This is the overall sequence for the system. The *Client* distributes the policies and sends them to the *CPEs*. After distribution of the policies, the *Client* isn't part of the sequence anymore. Instead the *CPEs* enforce the policies independent of the *Client*.

When distributing the policies, the *Client* should be able to use the secure channel, already established through the *Secure Device Grid*, between the *Client* and the *CPEs*, which is why this protocol won't be considered.

The *CPE* have no way of communicating, it should therefor be considered how the *CPEs* should be able to communicate using the LAN. The *CPEs* have no way of communicating securely, which is why security needs to be considered for the protocol. Later it is also needed to consider, what the event, sent from *CPE A* to *CPE B*, should contain, in order for *CPE B* to understand the event. This is a design issue, and will therefor be discussed during design of the protocol.

3.3 Security goals for enforcing policies

This section discusses security properties and their relevance to enforcing policies. Well known properties will be evaluated. The CIA triad's properties; confidentiality, integrity and availability will be used. These are paired with the cryptographic goals from the *Handbook of Applied Cryptography*[Gre14]: confidentiality, data integrity, authentication and non-repudiation. Notice that some of the properties overlap. Confidentiality is mentioned for both, and data integrity and integrity are considered the same property.

3.3.1 Confidentiality

Confidentiality ensures that only authorised users gain read access to information. Considering the policies it means that no attacker should be able to read the policies neither at distribution or enforcement.

This is an important property of the system. A policy distributed to *CPEs* contain information about which properties needs to be fulfilled in order to trigger an action.; e.g. when a window is being opened. Information sent between *CPEs* could indicate that a windows is being opened. To secure and make the customer feel more protected policy distribution and enforcement need to ensure confidentiality.

3.3.2 Data integrity

Data integrity ensures protocol participants that messages has not been modified by an attacker. For a policy-based system it means that attackers aren't able to alter data in the exchanged messages for the distribution and enforcement of policies unnoticed.

Without data integrity the attacker might be able to change the policies when distributed, which could lead to a window being opened when it is favourable. If the data integrity is not provided when enforcing the policies. Here changing the data could result in a window not being closed when expected, or a windows being opened when not expected. Data integrity is needed to ensure a secure system.

3.3.3 Availability

Availability ensures that information concerned is accessible to the authorised user at all times. Considering the policies, it means that *Clients* and *CPEs* should be able to communicate at all time.

If an attacker floods the system with messages, executing a DoS attacker, it could result in policies not being distributed and enforced. If a *CPE*, experiencing an event, aren't able to inform the *CPEs* defined in the policy, this could lead to a windows not being closed. A *Client* unable to distribute a policy, definitely doesn't provide a good user experience, but it doesn't expose the system to any security threat, assuming that the existing policies are secure. The system is vulnerable to DoS attacks, regardless this property is disregarded, as protocols can't defend against DoS attacks. DoS attacks should instead be solved by Trifork's surrounding software.

3.3.4 Authentication

Authentication is a two-part property consisting of entity authentication and data origin authentication. Entity authentication is insurance that the two parties entering into a communication identify each other. Data origin authentication ensures data integrity and that data originates from the authenticated party.

This is an important property of the system. If authentication is not enforced, an attacker would be able to force actions on a *CPE* that should be done by trusted parties. In terms it could result in the attacker opening a window. An attacker could also force another policy at distribution. Authentication implies data integrity, which mean that arguments made for data integrity also apply for authentication. Authentication is an important property for the system.

3.3.5 Non-repudiation

Non-repudiation ensures that parties cannot deny ownership of information. Considering policies, it would mean that a *Client* wouldn't be able to deny distributing the policies, and that a *CPE* wouldn't be able to deny informing another *CPE* of an event happening. Information can still be trusted without non-repudiation.

This is not an important property of the system. There is no need for ownership between *Client* and *CPEs*. The same is true between *CPEs*. As long as information can be trusted, there is no need to apply non-repudiation. If the final solution provide non-repudiation it is because other arguments has provided a solution that implements non-repudiation.

3.4 Summary

The goal is to design a protocol to enforce the policies across *CPEs*. The protocol needs to enforce the security properties: confidentiality, data integrity and authentication, in order to protect customers from attackers. For now only the security properties are considered for secure communication. Security considerations for key management are included in section 5, where protocol requirements are defined.

Chapter 4

State of the art

This section looks at the cryptographic tools, that are relevant to creating a secure solution for Trifork's goal. This mean that the current system of Trifork is investigated along with the cryptographic techniques used by Trifork. Additionally other state of the art cryptographics are investigated to provide an alternative to the current solution of Trifork.

4.1 Trifork's existing system

This section gives an insight into the existing system and its solution for a secure system. It starts of by explaining the need of the Device Grid server and the solutions selected. After that security is considered, how to generate session keys and how to pair devices. The generation of session key are considered first as this is used for the Device pairing. Next the way of ensuring random for small devices is described. Finally the used crypto are presented.

In the motivation section, we saw how the *Client* (smartphone) and the *CPE* are suppose to communicate, in order for the *Client* to control the *CPE*. It is relatively easy for the devices to communicate, if they reside on the same LAN, but this not always the case. Trifork is working with a scenario where the *Client* and the *CPE* communicate using TCP from two different LANs, which means that they reside behind two different NATs. This means that the two devices have no way of contacting each other without the help of a mutually trusted third party.

4.1.1 Network address translation (NAT)

The introduction of NATs complicates communication between the two devices. Most NATs are outbound NATs[FSK05], which means that they allow connections to be established from within the LAN, but deny connections to be established from the internet. A device on a LAN doesn't initially have a public address where others might contact it. The NAT assigns the device a public port number, when it connects to a public ip address. This port number and the public ip address of the NAT corresponds to the public address of the device also called the *session endpoint*. The session endpoint is where, replies to the device for this session, can be send. When the NAT receives a reply on the session endpoint it translates the public port number to the local ip address and local port number of the device thereby routing the reply to the device (vice versa for a request).

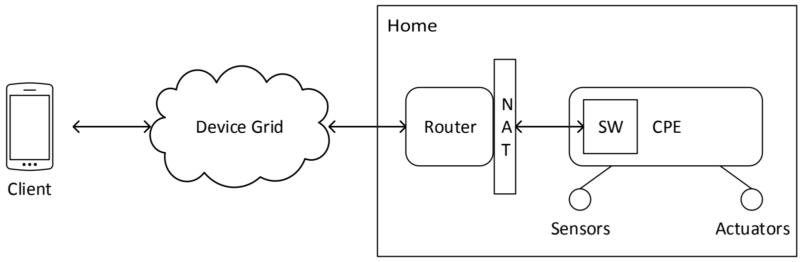


Figure 4.1: Device Grid Setup.

To solve this predicament, where both device resides behind a NAT, techniques called NAT traversal are used. There exists two popular NAT traversal approaches: Relaying and Hole punching.

4.1.1.1 Relaying

This approach is the one Trifork is using. Here a trusted server is uses to relay messages between the two devices. For Trifork this trusted server is the Secure Device Grid server. The server would keep an open connection to both devices, making relaying of messages conceptually easy. In reality only the *CPE* would have a keep an open connection with the server, and the *Client* would only have an open connection to the server when contacting the *CPE*. This is not a problem, as the initiative for communication lies entirely with the *Client*, and both devices would have an open connection with the server at this time.

The strength of relaying is that it always work, no matter what kind of NATs you are working with. The disadvantage is that efficiency depends on the server's processing power and the network bandwidth. The added link for the communication flow increases latency, and this is even if great processing power and great network bandwidth is available[FSK05].

4.1.1.2 Hole punching

For this approach the server doesn't relay the messages, but rather facilitates the connection between the devices. Using Trifork's setup as an example; the *Client* would inform the server that it wishes to communicate with the *CPE*. The server would then provide the *Client* with the *CPE*'s public endpoint for the *Client* to contact the *CPE* directly. At the same time the server sends a connection request to the *CPE* containing the *Client*'s public endpoint. The *Client* and *CPE* will now be able to communicate directly, without relying on the server to relay messages.

Hole punching allow for a much greater throughput. It has the downside of not working with all NATs. In fact only 64 % of NATs support TCP hole punching[FSK05]. Keeping in mind, that these numbers where published in 2005. An increase of support for hole punching has been reported, but regardless it still means that hole punching can't be relied on to always work.

4.1.1.3 NAT Conclusion

Trifork's reason for choosing the relaying approach rather than hole punching, derives from the need to serve all customers. It is more important to have a system that always works rather than having great throughput. Also the volume of data send between the *Client* and the *CPE* isn't that great and the transmission rate isn't that important. Hole punching is usually used in systems like VoIP, where great throughput outweighs the need for compatibility.

Relaying gives the *Client* and the *CPE* the means to communicate, but lacks the secure aspect, as all messages would be sent in plaintext. Therefor protocols ensuring a secure communication between *Clients* and *CPEs* are now considered.

4.1.2 CurveTun protocol

CurveTun is a Diffie-Hellman key exchange used for secure communication by Trifork. The protocol is used in three variation by Trifork, where the underlined part in message 3 changes based on these variations. First the overall flow of the protocol is described and after that the three variations will be discussed.

Message 1: Here A sends a random challenge to T, which is also A's public session key xG.

Message 2: T returns its own challenge, yG, to A, which is also T's public session key. The challenge is encrypted and ensured data integrity under the symmetric key derived from A's public session key and T's long-term private key. This means that the message can only be opened by one in possession of T's long-term public key and the private session key corresponding to xG. A trusts that the value yG comes from T, as T's private key has been used to encrypt the message.

Message 3: A sends the public session xG again, encrypted under the shared long-term symmetric key, which can be derived using A an T's long-term public- and private keys. This ensures T that the session key is shared with A. The long-term public key of A, pkA, and the *licens* value is encrypted under the session key. This mean that encrypted messages containing these two fixed values will differ for every protocol run. After message 3 A and T can engage in secure trusted communication using the session key, xyG.

$$\begin{aligned} &1.\ A \rightarrow T : xG \\ &2.\ T \rightarrow A : \{|yG|\}_{K_{txG}} \\ &3.\ A \rightarrow T : \{|\underline{pkA, licens}, \{|xG|\}_{K_{atG}}|\}_{K_{xyG}} \end{aligned} \tag{4.1}$$

As mentioned, how message 3 is perceived and what it includes depends on which of three variations are considered. The three variations are discussed below.

CPE → Server
This is the establishment of a secure connection between the *Server* and a *CPE*. The *CPE* has the role of A and the *Server* has the role of T. This variation includes both *pkA* and *licens*. The *CPE* has the long-term public key of the *Server*, but the *Server* doesn't have the long-term key of the *CPE*. This means that the *CPE* needs to provide it for the *Server*. The *Server* cannot trust the

long-term public key of A without being ensured that A can be trusted. This trust is provided by the *licens*, the *Server* has a long list of licenses it trusts, and if it trust the provided licens, it will trust the long-term public key. When the *Server* trusts the long-term public key, it can unpack xG to determine if it trusts the session key xyG.

Client → Server
This is the establishment of a secure connection between the *Server* and a *Client*. The *Client* has the role of A and the *Server* has the role of T. The *Client* connects to the *Server* when pairing with a *CPE*. This variation only includes the *Client*'s long-term public key pkA. The *Client* has the long-term public key of the *Server*, but the *Server* doesn't have the long-term key of the *Client*. The *Server* doesn't have anything to authenticate the *Client*with, neither a *licens* or a long-term public key. The protocol ensures that the *Server* and the *Client* share a session key. No trust is provided between the two, instead the responsibility to ensure trust is passed to the *CPE* (See the paring section 4.1.3).

Client → CPE
This is the establishment of a secure connection between a *Client*and a *CPE*. The *Client* has the role of A and the *CPE* has the role of T. This connection is not the pairing mentioned above, but is rather the connection between a *Client*and a *CPE* when they have been paired. This means that the two know each others' long-term public keys. This variation doesn't include any of the underlined part. In fact the trust in the session key is establish using the above mentioned step procedure.

4.1.2.1 Cryptographic functions of CurveTun

The CurveTun protocol relies on the Networking and Cryptography library (NaCl)[Ber], where it uses the function crypto_box for encrypting messages and the function crypto_box_open for decrypting messages.

crypto_box encrypts messages using the symmetric-key algorithm *Salsa*. In addition to the message the function also takes the following as input: a public key, a private key and a nonce; to create the ciphertext.

$$ciphertext = crypto\_box(message, nonce, public\_key, private\_key)$$

Symmetric encryption functions uses a shared secret to encrypt and decrypt messages. The crypto_box function takes a public key and a private key. Within the crypto_box function a shared secret is dereived from these keys using the

Curve25519 function, which is based on a elliptic curve Diffie-Hellman key exchange. In addition crypto_box offers data integrity by creating a MAC using Poly1305.

4.1.3 Pairing of Client and CPE

This protocol seeks to establish trust between a *Client*and a *CPE*; meaning exchanging their long-term public keys, and ensuring the *CPE* and *Client*that these public keys in fact can be trusted. The pairing is needed so that the *Client* might communicate with the *CPE* and thereby regulate e.g. the heat of the customers home.

Protocol 4.2 shows the interaction needed to ensure this trust. The protocol is complex, and a protocol description can be seen below. The protocol has three participants: A the *CPE*, S the *Server* and C the *Client*.

$$\begin{aligned}
&1.\ A \rightarrow S : start\ pairing \\
&2.\ S \rightarrow A : R_1, token \\
&3.\ A\ gen. : R_2 \\
&4.\ A\ shows : R_1, R_2 \\
&5.\ Enter\ C : R_1, R_2 \\
&6.\ C \rightarrow S : R_1 \\
&7.\ S \rightarrow C : token \\
&8.\ A \rightarrow S : token \\
&9.\ C \rightarrow S : token \\
&10.\ A \leftrightarrow C : connected \\
&11.\ A \leftrightarrow C : upgrade\ curvetun \\
&12.\ A \leftrightarrow C : PACE\ protocol : R_1, R_2
\end{aligned} \tag{4.2}$$

The protocol prerequisites a physical setup. The customer standing next to the *CPE*with the *Client*(smartphone) in hand. The customer is considered trusted, as the *CPE*is located within the home of the customer.

Step 1. The customer put the *CPE*in pairing mode. The *CPE* contacts the Server and sets up a secure channel, as described in section 4.1.2, and lets the *Server* know that it wishes to be paired.

Step 2. The *Server* returns a fixed length random string, R_1, and a connection token, *token*.

Step 3/4. The *CPE* then generates another fixed length string R_2, and shows R_1 concatenated with R_2 on its display.

Step 5. The customer enter the string into the *Client*. The idea here is that only the trusted customer would be able to enter R_1 concatenated with R_2 into the *Client* device.

Step 6. The *Client* contacts the *Server* and sets up a secure channel, as described in section 4.1.2, and sends R_1 to the *Server* indicating that the *Server* should provide the *Client* with the connection token corresponding to R_1. Referencing section 4.1.2 the *Server* doesn't trust the *Client*.

Step 7. The *Server* returns the connection token to the *Client*.

Step 8/9. The *CPE* and the *Client* informs the *Server* that they wish to be connected.

Step 10. The *Client* and the *CPE* are now connected through the *Server*. The *Server* provides the *Client* with the *CPE*'s public key and vice versa. The *Server* now relays the messages as previously discussed. This also means that the *Server* puts it on the *CPE* to determine for itself if it trusts the *Client*.

Step 11. The *Client* and the *CPE* exchange public session keys using Protocol 4.1. This means that they have a secure connection, but trust isn't provided yet.

Step 12. To establish trust, a protocol called the PACE protocol [BFK09] is used. It determines if the *Client* and *CPE* have the same value for R_1 concatenated with R_2. Remember that only the *Client*, owned by the customers initialising the pairing protocol, should be able to know this string and therefor be trusted. If the strings are identical the pairing is successful meaning that the *CPE* trusts the *Client* and vice versa. The two can now communicate securely using the CurveTun protocol from section 4.1.2.

Understanding the CurveTun- and the Paring protocol, it should be obvious why the *Server* doesn't need to trust the *Client* as long as the *CPE* does. It might not be obvious why the *Server* needs to trust the *CPE*, when the *Client* would trust the *CPE* regardless. The trust between *Server* and *CPE* is need for two main reasons. Only certified *CPEs* should be able to use the benefits of the Device Grid. If the server just accept any device with at public key, one could in theory connect its own device, taking advantage of the Device Grid infrastructure. The second reason is that Device Grid allows updates of the software in the *CPE* through the *Server*. Only a trusted *Server* should be able to push updates to the *CPE*.

4.1.4 Randomness considerations

There is a lot of *CPEs* connected to the Device Grid. These devices are in many ways identical, the same piece of hardware with the same piece of software. Only keys and licenses differ. This is necessary for the devices to not generate the same random value under the same circumstances, so that an attacker won't be able to deduce the random generation of one device and use it to attack another device.

To ensure a unique random among *CPEs*, a random feed of 32 bytes are written to the flash at production. When the *CPE* is running it keeps a random feed of 64 bytes in memory. The random feed in memory is constructed by running SHA-512 on the random feed in the flash, the uptime counter and realtime clock. These values are xored to obtain the 64 bytes random feed. This random value is constructed at boot and the formula can be seen below.

Construction of random 64 byte value in memory at boot
mem_rand = SHA-512(flash_rand) $\oplus$ SHA-512(uptime) $\oplus$ SHA-512(realtime)

To ensure that the random value in memory isn't guessed by an attacker, it needs to be updated. This is done by, every time the *CPE* receives trusted data from the *Server*, the decrypted value is run through SHA-512 and xored with the existing random value in memory. The decrypted value is used because it is confidential, which means, that should the value be known, security would already have been broken regardless. Formula for updating based on the received *Server* data.

Updating of random 64 byte value in memory
mem_rand = mem_rand $\oplus$ SHA-512(decrypted_server_data)

The random feed in flash is never updated. This means that after a power down, the booting sequence starts again with the same random feed. Flash memory has a finite number of writes, the random value in flash therefor isn't updated as this would wear on the flash and decrease the lifespan of the *CPE*.

4.2 Block Cipher

This section is based on chapter 7, Block Ciphers, from the book, *Handbook of Applied Cryptography*[MvOV01]. Block ciphers and block cipher modes are investigated, as they are widely used for symmetric-key encryption.

A block cipher is a deterministic function. It consists of two paired algorithms; one for encryption and one for decryption. Block ciphers provide confidentiality. The Advance Encryption Standard (AES) is a block cipher. Block ciphers are usually thought of as encryption- and decryption function in a symmetric key encryption scheme. Block ciphers are very generic, and can be used as a building block for pseudorandom number generators, stream ciphers, MACs and hash functions. This section approaches the block ciphers from the perspective of AES.

A block cipher encryption function, E, transforms n-bit plaintext blocks to n-bit ciphertext blocks and a block cipher decryption function, E^{-1}, transforms n-bit ciphertext blocks to n-bit plaintext blocks. This means that plaintext is input for the encryption function and that ciphertext is input for the decryption function. In addition, both functions takes a key K, which the texts are transformed according to.

$$\begin{aligned}\text{Encrypting: } & \textit{ciphertext} = E(\textit{plaintext}, K)\\ \text{Decrypting: } & \textit{plaintext} = E^{-1}(\textit{ciphertext}, K)\\ & \textit{plaintext} = E^{-1}(E(\textit{plaintext}, K), K)\end{aligned}$$

There exist a lot of block ciphers, such as AES. These block ciphers can be used in different modes, which have great impact on the efficiency and security. Depending on applications' restrictions, it might be necessary to favour efficiency, which degrades security; or security might be favoured hurting efficiency. Modes don't affect the block cipher, which is still a deterministic function. The mode determines how blocks are fed to the block cipher, which affects the resulting cipher- or plaintext.

The most popular block cipher modes are: Electronic Codebook, Cipher Block Chaining, and Counter.

4.2.1 Electronic Codebook (ECB)

ECB is a fairly simple block cipher mode. Figure 4.2 illustrates the encryption, E, and the decryption, E^{-1}, for ECB mode. Remember these corresponds to the AES encryption- and the decryption function.

For ECB mode, all n-bit plaintext blocks, $x_1, ..., x_j$, are being used individually along with the key K as input for the encryption function E. Plaintext blocks are being encrypted one by one, with the key K being the only value affecting the resulting ciphertext. This means that identical plaintext blocks will result in identical ciphertext blocks, under the condition that the same key is used.

Having a mode, where identical plaintext blocks encrypted result in identical ciphertext blocks, makes the ciphertext susceptible to data analysis. Attackers might be able to deduce the plaintext from the repeated ciphertext. Attackers might also be able to identify places in the ciphertext, where bits could be substituted to the attacker's benefit.

The ECB mode might be vulnerable to attacks, because it encrypts blocks individually only using an additional key as input, but it has positive impact on performance. When blocks are encrypted in ECB mode they don't rely on any previous computation, which means that encryption of blocks can be done in parallel. The same applies for decryptions.

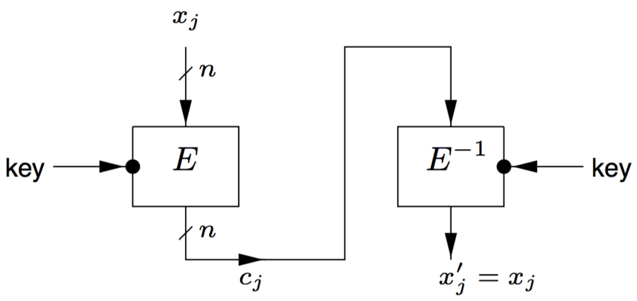


Figure 4.2: Electronic Codebook mode

The ECB mode is a very efficient block cipher mode, but has clear security weaknesses. The ECB mode should therefor only be used for application where the need for efficiency outweighs the need for security, and if other more secure modes do not meet the efficiency requirements.

4.2.2 Cipher Block Chaining (CBC)

CBC mode is a slight more complex block cipher mode than the ECB mode. Figure 4.3 illustrates the encryption, E, and the decryption, E^{-1}, for CBC mode. Compared to ECB mode it relies on additional data to create a strong encryption. The first plaintext block, x_0, is being xored with an initialisation vector, IV, which consist of random data; before being encrypted with key K. The additional plaintext blocks, $x_2, ..., x_j$, uses the preceding ciphertext block, c_{j-1}, to xored with before being encrypted with key K.

A different random initialisation vector should be used every time, as this will result in, identical plaintext blocks being encrypted to unique ciphertext blocks.

The use of the CBC mode makes the ciphertext resilient to data analysis, as attackers can't deduce anything from identical ciphertext blocks.

Using the CBC mode will impact efficiency negatively. When plaintext block are being encrypted, it relies on the encryption of the previous plaintext block. This means that CBC mode doesn't supported parallel encryption of plaintext block, but instead is forced to encrypt blocks sequentially.

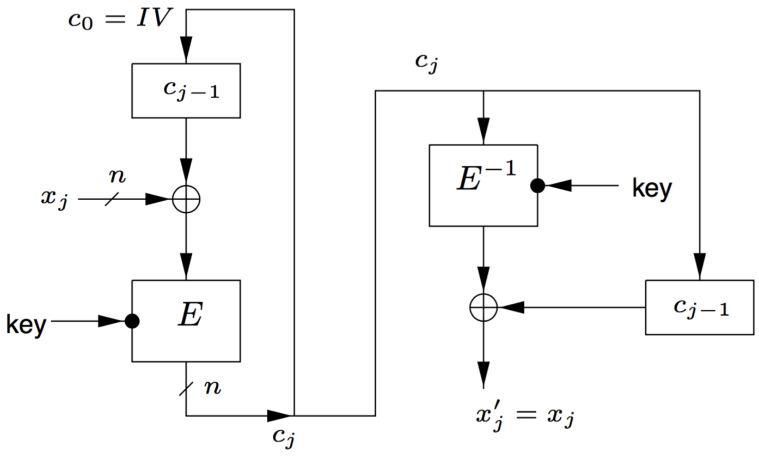


Figure 4.3: Cipher Block Chaining mode

The CBC mode is a very secure block cipher mode, but but isn't as efficient as ECB mode. The CBC mode is recommended mode for applications where strength of security is of high priority and where the efficiency speed up using ECB mode aren't required.

4.2.3 Counter (CTR)

The CTR mode is an efficient and secure block cipher mode. It can, as ECB mode, encrypt and decrypt text in parallel, and gives, as CBC mode, a new output ciphertext for the same plaintext, given the random provided input is different. The input to the encryption function, apart from the key, is a nonce and a counter corresponding to the block number being encrypted. The input doesn't rely on any previous computation, which is the reason it can be computed using the encryption function in parallel. The use of nonces makes sure that identical plaintext for different encryptions differs and addition of the counter makes sure that identical blocks of plaintext within the encryption differs.

CTR mode also differs from the two previous mode, by not encrypting the plaintext using the encryption function, but rather encrypting the plaintext by xoring it with the output of the encryption function. Decryption is done in a similar way. Here the ciphertext xored with the output of the encryption function results in the plaintext. Notice that that encryption function E^{-1} isn't used, but rather the E for both encryption and decryption.

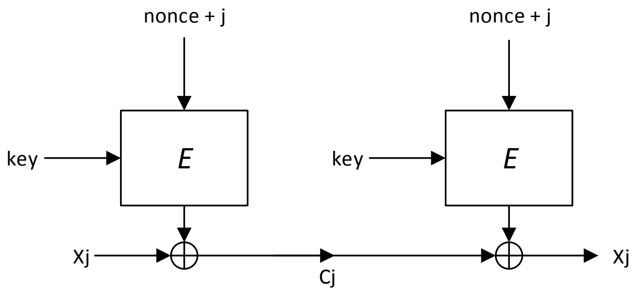


Figure 4.4: Counter mode

The CTR mode includes both the stregth of ECB mode and the stregth of the CBC, while not inheriting any of their weaknesses. This makes the CTR mode the superior cipher block mode, which is also the reason is it used along modern encryption functions like AES.

4.3 Advanced Encryption Standard (AES)

This section is based on the paper *AES Proposal: Rijndael* by Joan Daemen and Vincent Rijmen [DR98]. AES is the NIST standard for symmetric key encryption[Bar15]. It is therefor widely used in the industry, which is why it is considered for this system.

AES is a block cipher. Rijndael, which is the name from the paper, and AES is basically the same. The only the difference is that AES has some regulations, that implementations should follow. Rijndael has a variable length for the blocks and the key. For Rijndael the block and key length can be any multiple of 32-bit, starting at 128-bit and a ended at 256-bit. AES has a fixed block length of 128-bit, and three options for the key length: 128-bit, 192-bit and 256-bit.

AES is block cipher, which means that it can be used in ECB- and CBC mode. It also means that it has an encryption function and an decryption function. This section only focuses on how the encryption function work, as the decryption function is very similar. Remembering section 4.2, the decryption function is just the inverse of the encryption function.

4.3.1 AES encryption

The AES encryption function takes a plaintext block and a key as input, to produce a ciphertext block. The plaintext changes during the AES encryption algorithm, which is why it's called state instead. The state is visually represented as a two-dimensional array consisting of 16 bytes (Figure 4.5). According to *Rijndael*, the state always have 4 rows, which makes 32 bits pr. row. The number of columns depends on the block length. AES has a fixed block length of 128 bits, which makes: $128\text{-}bits/32\text{-}bits = 4\ columns$.

$$state = \begin{array}{|c|c|c|c|} \hline p_1 & p_5 & p_9 & p_{13} \\ \hline p_2 & p_6 & p_{10} & p_{14} \\ \hline p_3 & p_7 & p_{11} & p_{15} \\ \hline p_4 & p_8 & p_{12} & p_{16} \\ \hline \end{array}$$

Figure 4.5: Two-dimensional state array

The AES encryption is done in three overall steps: Initial Round, $N_r - 1$ Round and Final Round; which is visualised in the list below. The $N_r - 1$ Round is repeated $N_r - 1$ times, where N_r is the total number of rounds. The Final Round counts as the last round, which makes the number of rounds reach N_r.

The Initial Round isn't considered among the total number of rounds, but rather as a setup round before the actual rounds. Here the KeyExpansion function expands the key from the input, CipherKey, to $N_r + 1$ RoundKeys. This makes one RoundKey for each N_r and on for the InitialRound. A new RoundKey is given as input to AddRoundKey every time it is performed.

1. Initial Round(State,CipherKey)

   KeyExpansion(CipherKey,RoundKeys);
   AddRoundKey(State,RoundKey);]

2. $N_r - 1$ Rounds(State,RoundKey)

   SubBytes(State);

```
    ShiftRows(State);
    MixColumns(State);
    AddRoundKey(State,RoundKey);
```

3. Final Round(State,RoundKey)

```
    SubBytes(State);
    ShiftRows(State);
    AddRoundKey(State,RoundKey);
```

The Initial Round and the Final Round are each performed once. How many times the $N_r - 1$ Rounds are performed varies. According to *Rijndael*, the number of total rounds, N_r, depends on the key size and block size. AES has a fixed block size, which makes the number of rounds depend on the key size. Total number of rounds, N_r, based on key size can be seen in the list below.

- $128\text{-}bit = 10\ rounds$
- $192\text{-}bit = 12\ rounds$
- $256\text{-}bit = 14\ rounds$

The four round transformations: SubBytes, ShiftRows, MixColumns and AddRoundKey; will be described, to get an idea of, how the plaintext is encrypted.

4.3.2 SubBytes

In this transformation every $a_{i,j}$ is map to a $b_{i,j}$, e.g. $a_{0,1}$ is mapped to $b_{0,1}$. The transformation for the value $a_{i,j}$ to $b_{i,j}$ happens in the function S-box.

$$\begin{array}{|c|c|c|c|} \hline a_{0,0} & a_{0,1} & a_{0,2} & a_{0,3} \\ \hline a_{1,0} & a_{1,1} & a_{1,2} & a_{1,3} \\ \hline a_{2,0} & a_{2,1} & a_{2,2} & a_{2,3} \\ \hline a_{3,0} & a_{3,1} & a_{3,2} & a_{3,3} \\ \hline \end{array} \quad \xrightarrow{a_{i,j}} \text{S-box} \xrightarrow{b_{i,j}} \quad \begin{array}{|c|c|c|c|} \hline b_{0,0} & b_{0,1} & b_{0,2} & b_{0,3} \\ \hline b_{1,0} & b_{1,1} & b_{1,2} & b_{1,3} \\ \hline b_{2,0} & b_{2,1} & b_{2,2} & b_{2,3} \\ \hline b_{3,0} & b_{3,1} & b_{3,2} & b_{3,3} \\ \hline \end{array}$$

The S-box transformation is rather simple. It takes the 8 bit, $x_0, ..., x_7$, of a $a_{i,j}$ byte and inserts them in the function below.

$$\begin{bmatrix} y_0 \\ y_1 \\ y_2 \\ y_3 \\ y_4 \\ y_5 \\ y_6 \\ y_7 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 & 1 & 1 & 1 & 1 \\ 1 & 1 & 0 & 0 & 0 & 1 & 1 & 1 \\ 1 & 1 & 1 & 0 & 0 & 0 & 1 & 1 \\ 1 & 1 & 1 & 1 & 0 & 0 & 0 & 1 \\ 1 & 1 & 1 & 1 & 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 & 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 & 1 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 & 1 & 1 & 1 & 1 \end{bmatrix} \times \begin{bmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \\ x_6 \\ x_7 \end{bmatrix} \oplus \begin{bmatrix} 1 \\ 1 \\ 0 \\ 0 \\ 0 \\ 1 \\ 1 \\ 0 \end{bmatrix}$$

The $b_{i,j}$ byte's 8 bit, $y_0, ..., y_7$ are calculated by the equation below. The example shows y_0 being calculated, but a similar approach is used for $y_1, ..., y_7$.

$$y_0 = (1x_0 \oplus 0x_1 \oplus 0x_2 \oplus 0x_3 \oplus 1x_4 \oplus 1x_5 \oplus 1x_6 \oplus 1x_7) \oplus 1 \qquad (4.3)$$

This S-box transform is applied to all the bits in the a-matrix to get the resulting b-matrix.

4.3.3 ShiftRows

Here each each rows is shifted by an offset. The offset is based on the number of columns, N_B. For *AES* N_B is always 4, and the offset for each rows is therefor: $row_0 = 0$, $row_1 = 1$, $row_2 = 2$ and $row_3 = 3$.

$a_{0,0}$	$a_{0,1}$	$a_{0,2}$	$a_{0,3}$
$a_{1,0}$	$a_{1,1}$	$a_{1,2}$	$a_{1,3}$
$a_{2,0}$	$a_{2,1}$	$a_{2,2}$	$a_{2,3}$
$a_{3,0}$	$a_{3,1}$	$a_{3,2}$	$a_{3,3}$

$$\xrightarrow{ShiftRows}$$

$a_{0,0}$	$a_{0,1}$	$a_{0,2}$	$a_{0,3}$
$a_{1,1}$	$a_{1,2}$	$a_{1,3}$	$a_{1,0}$
$a_{2,2}$	$a_{2,3}$	$a_{2,0}$	$a_{2,1}$
$a_{3,3}$	$a_{3,0}$	$a_{3,1}$	$a_{3,2}$

4.3.4 MixColumns

Here each column of the state is transformed and mapped to the same column in the resulting state.

$a_{0,0}$	$a_{0,1}$	$a_{0,2}$	$a_{0,3}$
$a_{1,0}$	$a_{1,1}$	$a_{1,2}$	$a_{1,3}$
$a_{2,0}$	$a_{2,1}$	$a_{2,2}$	$a_{2,3}$
$a_{3,0}$	$a_{3,1}$	$a_{3,2}$	$a_{3,3}$

$$\xrightarrow{\substack{a_{0,j} \\ a_{1,j} \\ a_{2,j} \\ a_{3,j}}} \text{Mix} \xrightarrow{\substack{b_{0,j} \\ b_{1,j} \\ b_{2,j} \\ b_{3,j}}}$$

$b_{0,0}$	$b_{0,1}$	$b_{0,2}$	$b_{0,3}$
$b_{1,0}$	$b_{1,1}$	$b_{1,2}$	$b_{1,3}$
$b_{2,0}$	$b_{2,1}$	$b_{2,2}$	$b_{2,3}$
$b_{3,0}$	$b_{3,1}$	$b_{3,2}$	$b_{3,3}$

In the transformation the four input bytes $a_{0,j}, ..., a_{3,j}$ are inserted into the matrix multiplication below to get the resulting bytes $b_{0,j}, ..., b_{3,j}$.

$$\begin{bmatrix} b_0 \\ b_1 \\ b_2 \\ b_3 \end{bmatrix} = \begin{bmatrix} 02 & 03 & 01 & 01 \\ 01 & 02 & 03 & 01 \\ 01 & 01 & 02 & 03 \\ 03 & 01 & 01 & 02 \end{bmatrix} \times \begin{bmatrix} a_0 \\ a_1 \\ a_2 \\ a_3 \end{bmatrix}$$

4.3.5 AddRoundKey

In this transformation the state bytes are xored with the RoundKey bytes and mapped to the resulting bytes.

$$\begin{array}{|c|c|c|c|} \hline a_{0,0} & a_{0,1} & a_{0,2} & a_{0,3} \\ \hline a_{1,0} & a_{1,1} & a_{1,2} & a_{1,3} \\ \hline a_{2,0} & a_{2,1} & a_{2,2} & a_{2,3} \\ \hline a_{3,0} & a_{3,1} & a_{3,2} & a_{3,3} \\ \hline \end{array} \oplus \begin{array}{|c|c|c|c|} \hline k_{0,0} & k_{0,1} & k_{0,2} & k_{0,3} \\ \hline k_{1,0} & k_{1,1} & k_{1,2} & k_{1,3} \\ \hline k_{2,0} & k_{2,1} & k_{2,2} & k_{2,3} \\ \hline k_{3,0} & k_{3,1} & k_{3,2} & k_{3,3} \\ \hline \end{array} = \begin{array}{|c|c|c|c|} \hline b_{0,0} & b_{0,1} & b_{0,2} & b_{0,3} \\ \hline b_{1,0} & b_{1,1} & b_{1,2} & b_{1,3} \\ \hline b_{2,0} & b_{2,1} & b_{2,2} & b_{2,3} \\ \hline b_{3,0} & b_{3,1} & b_{3,2} & b_{3,3} \\ \hline \end{array}$$

$$a_{i,j} \oplus k_{i,j} = b_{i,j}$$

Every state byte, $a_{i,j}$, is simply xored with it corresponding key byte, $k_{i,j}$, to get the resulting state byte, $b_{i,j}$.

4.3.6 Summary

When all the rounds have been completed, one block of 128-bits has been encrypted. If used in counter mode the output from the encryption function would then needed to be xored with the plaintext block corresponding to the counter used for input.

4.4 Salsa20

This section is based on the paper, *The Salsa20 family of stream ciphers* by Daniel J. Bernstein [Ber08]. The Salsa20 stream cipher is relevant to investigate, as it is the cipher used for confidentiality in the *crypto_box*-function of the NaCl crypto library, used by Trifork. Salsa20 is considered to be faster than AES and to be just as secure, according to Bernstein. Salsa20 is not a NIST approved encryption standard.

A stream cipher is very similar to a block cipher. A block cipher used in counter mode could actually be considered a stream cipher. A cipher can be considered a stream cipher if a cipher outputs a cipher stream s which are then xored with plaintext p to get the ciphertext c. This is just as presented for counter mode for block ciphers. This means that block cipher such as AES could be used to generate the cipher stream, but other types of ciphers might also generate the cipher stream. Here the Salsa20 stream cipher generate the cipher stream s used for xoring, when encrypting and decrypting.

$$\begin{array}{rcl} p & = & 00111010 \\ s & = & 01010101 \\ \hline c & = & 01101111 \end{array} \qquad\qquad \begin{array}{rcl} c & = & 01101111 \\ s & = & 01010101 \\ \hline p & = & 00111010 \end{array}$$

Encryption Decryption

The Salsa20 encryption function uses three simple operations to generate the cipher stream.

1. 32-bit addition, producing the sum $a + b \bmod 2^{32}$ of two 32-bit words a, b;
2. 32-bit exclusive-or, producing the xor $a \oplus b$ of two 32-bit words a, b; and
3. constant-distance 32-bit rotation, producing the rotation $a <<< b$ of a 32-bit word a by b bits to the left, where b is constant.

The cipher stream is generated in 64 byte blocks corresponding to the plaintext blocks. Each cipher stream block is computed independently as it doesn't rely on computations from any previous blocks. To generate a stream block the following values are needed: a secret key, a nonce, a block number (counter) and 4 constants. Given that none of the input relies on any previous computations, the cipher stream blocks can be computed in parallel.

The input values: secret key, nonce, block number and 4 constants; are inserted into an initial block structure. The structure is depicted in figure 4.7. The block structure is a 64 byte block stream consisting of a 256-bit secret key, a 64-bit nonce, a 64-bit block-counter and four 32-bit constants. This block structure is now put through 20 rounds of the Salsa stream cipher. A round of the Salsa20 stream cipher consists of 5 steps using the three earlier mentioned computations. The 16 values in the block structure is called words.

$constant_1$	key_2	key_3	key_4
key_5	$constant_6$	$nonce_7$	$nonce_8$
$counter_9$	$counter_{10}$	$constant_{11}$	key_{12}
key_{13}	key_{14}	key_{15}	$constant_{16}$

Figure 4.7: A 16 word, 64-byte block structure

Step 1: All the words below a constant are altered. This makes key_5, $counter_{10}$, key_{15} & key_4. They are altered by using the 3 simple computations mentioned early this section. key_5 will be used as an example. To alter the value of key_5, the word above key_5 it is used, $constant_1$, and the word above $constant_1$ is used, key_{13}. For step 1, the constant 7 is used for left bit rotation.

$$key_5 = (key_5 \oplus (key_{13} + constant_1 \bmod 2^{32})) <<< 7$$

Step 2: All the words below-below a constant are altered. This makes $counter_9$, key_{14}, key_3 & $nonce_8$. They are altered by using the same 3 simple computations as in step 1. $counter_9$ will be used as example. To alter the value of $counter_9$, the word above-above $counter_9$ is used, $constant_1$, and the word above $counter_9$ is used, key_5. For step 2, the constant 9 is used for left bit rotation.

$$counter_9 = (counter_9 \oplus (key_5 + constant_{11} \bmod 2^{32})) <<< 9$$

Step 3: All words above a constant are altered. This makes key_{13}, key_2, $nonce_7$ & key_{12}. key_{13} will be used as an example. key_{13} uses the two words just above it for the computation: key_5 and $counter_9$. For step 3, the constant 13 is used for left bit rotation.

$$key_{13} = (key_{13} \oplus (key_5 + counter_9 \bmod 2^{32})) <<< 13$$

Step 4: All constants are altered. $constant_1$ will be used as an example. $constant_1$ uses the two words just above it for the computation: key_{13} and $counter_9$. For step 4, the constant 18 is used for left bit rotation.

$$constant_1 = constant_1 \oplus ((key_{13} + counter_9) \bmod 2^{32}) <<< 18$$

Step 5: The round ends by transposing the state array, to ready it for the next round. When the 20th round is done the last step is to add the resulting array to the initial array. This will result in the final array used for encrypting the plaintext or decrypting the ciphertext.

4.5 Message Authentication Code (MAC)

This section is based on Section 4.2.3, Message Authentication Codes (MACs), of the publication, *Recommendation for Key Management* by NIST [Bar15].

A MAC can be used to provide source and integrity authentication. The MAC is a checksum generated by a MAC-algorithm, which as input accepts at least: a message of a variable length and a secret key known only to the sender and receiver. The MAC can help to ensure that data hasn't changed. A change in a MAC could be the result of an error in the transmission of data or that an attacker has modified or forged the data.

Integrity for MACs are ensured by that the sender and the receiver share a secret key as MACs are both generated and verified using the same secret key. For this reason MACs are usually used to ensure integrity for symmetric encryption as sender and receiver already share a secret key. Non-repudiation is not possible, as both the sender and the receiver would be able to verify and generate the same MAC using their shared secret key.

4.6 Keyed-Hash Message Authentication Code (HMAC)

This section is based on the FIPS PUB 198-1, which is the NIST standard for HMAC[NIS08]. The $HMAC$ uses a hash function and a secret key to provide a MAC for messages thereby providing data integrity and data authentication to a symmetric key protocol.

The $HMAC$ function takes as input a secret key and a message of arbitrary length. Depending on the underlying hash function the algorithm will use a different block size B to compute the MAC. The block size B depends on which data block size the underlying hash algorithm operates upon.

The algorithm start by preprocessing the secret key K to K_0. How the key should by preprocessed is determined by the length of the key K. If the length of K equals B, K_0 is set to K. If the length of K is greater than B, K_0 is set to the hash of K appended with zeroes to obtain the length of B. If the length of K is less than B, K_0 is set to K appended zeroes to obtain the length of B.

$$\begin{aligned}&\text{if length of } K = B : K_0 = K\\&\text{if length of } K > B : K_0 = H(K) + 00...00\\&\text{if length of } K < B : K_0 = K + 00...00\end{aligned}$$

The next step is to xor K_0 with the value *ipad*, which produces the value: $K_0 \oplus ipad$. The *ipad* is an inner pad consisting of the byte $0x36$ repeated B times. The message is appended to this value and hashed.

$$H((K_0 \oplus ipad) + message)$$

The next step is to xor K_0 with the value *opad*, which produces the value: $K_0 \oplus opad$. The *opad* is an outer pad consisting of the byte $0x5c$ repeated B times.

The MAC is then computed by appending $H((K0 \oplus ipad) + message)$ to $K_0 \oplus opad$ and hashing it. This value can now be used to ensure data integrity and data authentication to a symmetric key protocol.

$$H((K_0 \oplus opad) + H((K_0 \oplus ipad) + message))$$

For the hash function H any hash function may be used, but the cryptographic strength of the $HMAC$ depends greatly upon the cryptographic strength of the underlying hash function. The NIST publication with recommendation on key management[Bar15], recommends a wide range of hash function for $HMAC$: SHA-1, SHA-224, SHA-512/224, SHA-256, SHA-512/256, SHA-384, SHA-512 and SHA3-512.

4.7 Poly1305-AES

This section is based on the paper, *The Poly1305-AES message-authentication code* by Daniel J. Bernstein [Ber05]. The Poly1305 is the algorithm used to create MACs and thereby provide data integrity to the NaCl crypto_box symmetric encryption used by Trifork.

Poly1305-AES is a secret-key message-authentication code (MAC). As suggested by the name, the Poly1305-AES uses the AES block-cipher algorithm to compute the MAC. Poly1305-AES computes a 16-byte MAC from the input: a variable length message m, a 16-byte nonce n, a 32-byte shared key. Below the function used for computing the MAC is presented.

$$(((c_1 r^q + c_2 r^q - 1 + ... + c_q r^1) \bmod 2^{130} - 5) + AES_k(n)) \bmod 2^{128}$$

The function requires some prepossessing on some of the input. The values $c_1, c_2, ..., c_q$ are computed from the message m of length l, where $q = l/16$ and $c_1, c_2, ..., c_q \in \{1, 2, 3, ..., 2^{129}\}$. The message m, $m[0], m[1], ..., m[l-1]$, is converted to the sequence of c integers by:

if $1 \leq i \leq [l/16]$ then:

$$c_i = m[16i-16] + 2^8 m[16i-15] + 2^{16} m[16i-14] + ... + 2^{120} m[16i-1] + 2^{128}$$

if l is not a multiple of 16 then:

$$c_q = m[16q-16] + 2^8 m[16q-15] + ... + 2^{8(l \bmod 16)-8} m[l-1] + 2^{8(l \bmod 16)}$$

The 32-byte secret key is a two part key: a 16-byte AES-key k and a 16-byte string r. The key r is used along the c integers, while the key k is used to encrypt the nonce n. Key r needs some bits cleared; $r[3]$, $r[7]$, $r[11]$, $r[15]$ are required to have their top four bits cleared and $r[4]$, $r[8]$, $r[12]$ are required to have their bottom two bits clear.

The Poly1305-AES MAC algorithm explicitly takes a 16-byte nonce as input. The nonce helps to ensure that the MAC is freshly generated to prevent replay attacks for protocols using the Poly1305-AES. Poly1305-AES provides the nonce n as input for AES and encrypts is with the secret key k to obtain a 16-byte string $AES_k(n)$.

The prime $2^{130} - 5$ is used in computing the MAC and is where the algorithm gets its name from. The algorithm is guaranteed secure if AES is secure. The use of AES algorithm to encrypt the nonce could easily be substituted by another algorithm. The NaCl crypto_box function uses the Salsa20 stream cipher instead of the AES block cipher.

4.8 Sub-groups

The sub-groups of $\mathbb{Z}_p^*$ and elliptic curves are widely used for asymmetric encryption schemes. The two will be investigated with more focus on the elliptic curves, as they present a more complex problem. Finally comparable strength for the sub-groups will be match with different encryption schemes.

4.8.1 Sub-group, $\mathbb{Z}_p^*$

The sub-group of $\mathbb{Z}_p^*$ is a fairly simple sub-group, which is based on the discrete logarithm problem. This problem relies on the hard problem of integer factorisation. In sense how to identify the integer x from a function: α^x, where the value of α is known. The base α is a primitive root modulo of p from $\mathbb{Z}_p^*$.

The value of α^x is usually used as public key for e.g. a Diffie-Hellman key exchange, where an attacker shouldn't be able to derive the private key x from the public key α^x. Increasing the prime defining prime-order sub-group $\mathbb{Z}_p^*$, would increase the key size an thereby increase the security of the private key x.

This concludes the sub-group of $\mathbb{Z}_p^*$, which turns out to be simple to understand, but a hard problem to break.

4.8.2 Sub-group, Elliptic curves

The sub-group of elliptic curves is a more complex sub-group than $\mathbb{Z}_p^*$. The elliptic curve discrete logarithm problem is significantly harder than the discrete logarithm problem of $\mathbb{Z}_p^*$. This results in a strength-per-key-bit is substantially greater in elliptic curve systems than in conventional discrete logarithm systems. This results in greater speed, smaller keys and smaller certificates. To understand how an elliptic curve crypto system work, one first need to know elliptic curves and its group operations.

Elliptic curves over the finite field $\mathbb{F}_p$. Let p be a prime. $\mathbb{F}_p$ is comprised of the set of integers $\{0, 1, 2, ..., p-1\}$. When working with elliptic curves over the finite field $\mathbb{F}_p$, there are two operations to be familiar with: addition and multiplication.

An elliptic curve E over $\mathbb{F}_p$ could have the form $y2 = x3+ax+b$, where $a, b \in \mathbb{F}_p$, and $4a^3 + 27b^2 \not\equiv 0(\text{mod} p)$. Further more the set $E(\mathbb{F}_p)$ consists of all points (x, y), $x \in \mathbb{F}_p$, $y \in \mathbb{F}_p$ that satisfy the equation E, together with the point at infinity.

P+Q=R

The addition for elliptic curves is when adding two point, $P + Q$ on the elliptic curve $E(\mathbb{F}_p)$ to get the point$R$. The addition is easily understood geometrically and is depicted in Figure 4.8. Here a line line is drawn through point P and Q, the third intersection on the elliptic curve is $-R$. To get R the point is mirrored in the x-axis.

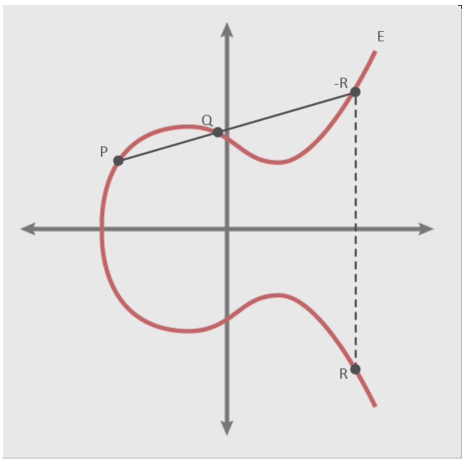


Figure 4.8: Addition of $P + Q = R$.

The formulas below is used for calculating point x and point y of R:

$$R_x = \frac{y_2 - y_1}{x_2 - x_1}^2 - x_1 - x_2$$

$$R_y = \frac{y_2 - y_1}{x_2 - x_1}(x_1 - x_3) - y_1$$

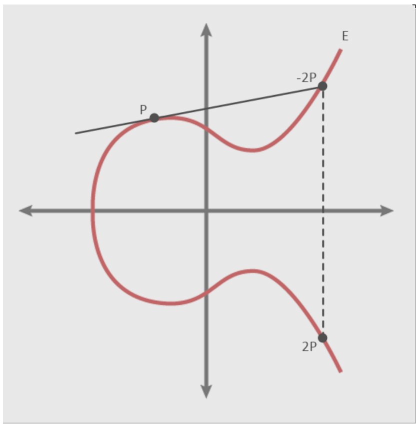
P+P=2P

$P + P$ is called point doubling. Geometrically it is very similar to $P + Q = R$. Instead of drawing a line through P and Q, the tangent of P is drawn (Figure 4.9a). The second intersection is then $-2P$, and mirroring it with the x-axis results in $2P$. It might be apparent that the $P + P$ operation actually is the multiplication operation $(2 \times P)$.

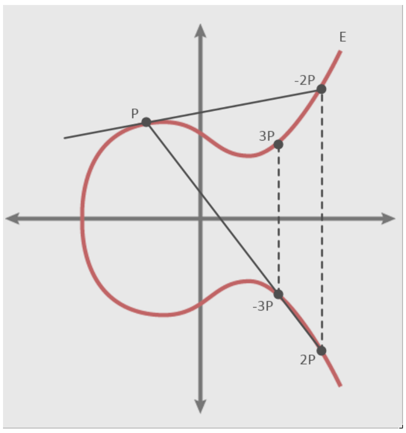
For point doubling the following formulas are used to calculate point x and point y of $2P$:

$$2P_x = \frac{3x_1^2 + a}{2y_1}^2 - 2x_1$$

$$2P_y = \frac{3x_1^2 + a}{2y_1}(x_1 - x_3) - y_1$$



(a) Point doubling $P + P = 2P$



(b) Scalar multiplication $3P$

Figure 4.9: Point doubling

Figure 4.9b also shows the multiplication $3P$. Here a line is drawn through the point P and $2P$ to get the point $-3P$, and again this point is mirrored in the x-axis to get the result $3P$. The formula of multiplication is $Q = kP$, where k is an integer $1 \leq k \leq n - 1$ (n is the order of the subgroup).

The computation of $Q = kP$ is called scalar multiplication, and is relevant for asymmetric encryption schemes. Here the value of Q would correspond to a public key and the value of k corresponds to a private key. P is the generator point of the elliptic curve and scalar multiplying it with the private key k results in the public keys Q. Computing the public key from the private key and generator is an inexpensive computation, whereas computing the private key from the public key and generator point is considered hard. This characteristic makes it a good sub-group for an asymmetric encryption scheme.

4.8.3 Comparable strengths of Asymmetric-key algorithms

This section compare the strength of the sub-groups usually used for asymmetric-key algorithms. Figure 4.1 shows how the key size in bits compare between the two sub-groups [Bar15]. A 2048 bit key of sub-group $\mathbb{Z}_p^*$ corresponds to a 224-255 bit key of the elliptic curve sub-group.

A Diffie-Hellman key exchange relies on the these two sub-group. The key size using elliptic curves is smaller, which turns out a valuable property to a Diffie-Hellman key exchange. The smaller key is significantly faster to generate

$\mathbb{Z}_p^*$	Elliptic curve
2048 bit	224-255 bit
3072 bit	256-383 bit

Table 4.1: Comparable key strength of sub-groups.

[SSG13], and a smaller key size is also significantly faster to transfer between protocol participants.

Signature generation algorithms like RSA and DSA uses the $\mathbb{Z}_p^*$ sub-group, whereas ECDSA uses the elliptic curve sub-group. Algorithms using the $\mathbb{Z}_p^*$ sub-group are faster at generating signatures and verifying signatures, but they again have the disadvantage of larger key. Apart from having to store these, you would also have to, if enforcing forward secrecy (more on this later), transfer these larger key for your Diffie-Hellman key exchange.

Using elliptic curves seems like a better choice than using $\mathbb{Z}_p^*$. ECDSA will therefor be investigated rather than RSA and DSA. ECDSA is DSA's counterpart, where DSA uses $\mathbb{Z}_p^*$ instead of elliptic curves.

4.9 Elliptic Curve Digital Signature Algorithm (ECDSA)

This section is based on the paper, *The Elliptic Curve Digital Signature Algorithm (ECDSA)* by Don Johnson, Alfred Menezes and Scott Vanstone [JMV01].

Domain paramers The parameters from this section which represent the most important domain parameters are:

1. The prime p that specifies the size of the finite field.
2. The coefficients a and b of the elliptic curve equation.
3. The generator point G (The initial point on the curve. In Figure 4.8 P is the generator point.
4. The order n of the subgroup.

The key pair is created by following these steps.

1. Select a random integer d in the interval $[1, n-1]$.
2. Compute $Q = dG$ (scalar multiplication of the generator point, G on the curve E.)
3. d is the private key; Q is the public key.

Signature generation The signature consists of the pair (r, s) generated from the following steps.

1. Select random integer k, $1 \leq k \leq n-1$.
2. Compute $kG = (x_1, y_1)$ and convert x_1 to an integer $\bar{x_1}$.
3. Compute hash(m) and convert to an integer e.
4. Compute $r = \bar{x_1} \bmod n$. If $r = 0$ select a new k.
5. Compute $s = k^{-1}(e + d \times r) \bmod n$. If $s = 0$ select a new k.

Signature verification For another party to verify the signature (r, s) on m, one also need the domain parameters (p, a, b, G, n) and public key Q.

1. Verify that r and s are integers in the interval $[1, n-1]$.
2. Compute hash(m) and convert to an integer e.
3. Compute $w = s^{-1} \bmod n$.
4. Compute $u_1 = ew \bmod n$.
5. Compute $u_2 = rw \bmod n$.
6. Compute $X = u_1 G + u_2 Q$.
7. If $X =$ point at infinity reject the signature.
   Otherwise, convert the x coordinate x_1 of X to an integer $\bar{x_1}$.
8. Compute $v = \bar{x_1} \bmod n$.
9. Accept the signature if and only if v = r.

Comparing the two procedures of DSA and ECDSA it is apparent that they are very similar only with the difference in subgroups.

4.10 Diffie-Hellman key exchange

This section is based on the paper, *New Directions in Cryptography* by Whitfield Diffie and Martin E. Hellman [DH76]. This is the original paper suggesting the Diffie-Hellman key exchange. The original sub-group of $\mathbb{Z}_p^*$ is therefor used. A Diffie-Hellman key exchange allow parties to share a key over an insecure channel. Previously keys would have to be pre-distributed using an already secure channel. This put an unnecessary delay on the actual communication. The Diffie-Hellman key exchange has been widely accepted in security field and is being used in countless applications.

When using the Diffie-Hellman key exchange the two parties *Alice* and *Bob* wishes to obtain a shared secret, S_{AB}, which may be used for future symmetric encryption and - decryption. *Alice* and *Bob* initially agrees on the two values α and p, which may be publicly known. The next step is to compute the public keys of *Alice* and *Bob*. A public key is computed by raising a private value to the power of α. In the protocol below *Alice*'s private value is x and *Bob*'s private value is y. When the public keys, α^x and α^y have been exchange *Alice* and *Bob* can now compute the secret key. *Alice* computes the secret key as: $S_{AB} = (\alpha^x)^y \bmod p$. *Bob* computes the secret key similary by using his own private value y and *Alice*'s public value.

$$\begin{aligned} &1.\ A \rightarrow B : \alpha^x \\ &2.\ B \rightarrow A : \alpha^y \end{aligned} \tag{4.4}$$

During the Diffie-Hellman key exchange only the public values α^x and α^y have been compromised to a possible eavesdropper. It is not possible to compute S_{AB} from α^x and α^y directly. An eavesdropper would have to extract x from α^x or y from α^y in order to achieve it. This is consider a computational hard problem, and is also known as the discrete logarithm problem.

This is the basic of a Diffie-Hellman key exchange. It doesn't encrypt or decrypt data, but rather help to obtain a shared secret between two parties, who do not share a secure channel.

4.10.1 Man-in-the-middle attack

The Diffie-Hellman key exchange protocol is secure when dealing with an eavesdropper, as S_{AB} can't be computed from α^x and α^y.

The protocol is vulnerable to a man-in-the-middle attack, where the attacker, *Mallory*, intercepts the messaging between *Alice* and *Bob* and alters it. This is due to the fact that the protocol doesn't include any form of authentication between *Alice* and *Bob*. This attack is depicted in Protocol 4.5.

It is a fairly simple attack where $Mallory$ intercepts the key exchange and negotiates and secret key with *Alice* and *Bob* respectively. *Alice* now thinks she shares a secret key with *Bob*, and vice versa, when they in fact are sharing one with $Mallory$.

$$\begin{aligned} &1.\ A \to I : g^x \\ &2.\ I \to B : g^i \\ &3.\ B \to I : g^y \\ &4.\ I \to A : g^i \end{aligned} \tag{4.5}$$

Alice would then encrypt her message send it to $Mallory$. $Mallory$ decrypts the data using the key she shares with *Alice*, encrypt it again using the secret key she shares with *Bob* and finally send it to *Bob*. *Bob* receives the message decrypts it using the secret key he shares with $Mallory$. *Alice* now believes she has sent a secret message to *Bob* and *Bob* believes he has received a secret message from *Alice* when in fact $Mallory$ knows the content of the message. The attack is very efficient and is something all public key exchanges are vulnerable to.

4.10.2 Elliptic curve Diffie-Hellman

As the name suggest the Elliptic curve Diffie-Hellman uses elliptic curves to obtain a shared secret. Knowing about the elliptic curves from the section about ECDSA, it is possible to redesign the protocol using elliptic curves. It is fairly simple. Alice and Bob must agree on the domain parameters. Remember the generator point G and the order n of the subgroup. The other parameters are also needed but only these two are needed to understand the procedure.

The sequence the depicted in protocol 4.6. Alice selects a random integer x as her private key, where $1 \leq a \leq n - 1$. Scalar multiplication is used to compute Alice's public key xG, where G is the generator point of the elliptic curve. Bob does the same to get his private- and public key. The public key are exchanged and a shared keys (point on the curve), K_{xy}, can be derived by scalar multiplication ones own private key with the other party's public key e.g. $y * xG$.

$$\begin{aligned} &1.\ A \to B : xG \\ &2.\ B \to A : yG \end{aligned} \tag{4.6}$$

The shared key is usually computed as the hashed value of the x-coordinate of the shared point S_{AB}.

4.11 Station-to-station (STS) protocol

This section is based on the paper, *Authentication and Authenticated Key Exchanges* by Whitfield Diffie, Paul C. Van Oorschot and Michael J. Wiener [DVOW92].

The paper offers a solution to the simple but very powerful man-in-the-middle attack on the Diffie-Hellman key exchange. It is resistant to an attacker that is able to see all messages, delete, alter, inject and redirect messages, initiate communications with another party and reuse messages from past communications. Its a two-party mutual authentication protocol, which provides the communication parties with assurance, that they know each other's identity. Further more it, as the Diffie-Hellman key exchange, allows the two parties to share a common key. This secret key may be used to provide data integrity. The protocol also offers perfect forward secrecy as the dynamically created secret key cannot be compromised should the long-term key be compromised.

The paper includes two examples of the STS protocol; a more basic one without the use of certificates and more complex one using certificates.

The basic STS protocol prerequisites that Alice and Bob know each other's public keys. The approach is very similar to the Diffie-Hellman key exchange and the security of the protocol also relies on the discrete logarithm problem of the exponential key. The sequence is depicted int Protocol 4.7.

$$\begin{aligned} &1.\ A \rightarrow B : \alpha^x \\ &2.\ B \rightarrow A : \alpha^y, \{sign_B([\alpha^x, \alpha^y])\}K_{xy} \\ &3.\ A \rightarrow B : \{sign_A([\alpha^x, \alpha^y])\}K_{xy} \end{aligned} \tag{4.7}$$

Alice creates a random number x and sends the value α^x to Bob. Bod creates his own random number y and creates the value α^y and the key K (α^{xy}). Bobs sends the exponential α^y and the signed hash value of α^x concatenated with α^y. Alice can now compute the key K and verify Bob's signature using Bob's public key. Finally Alice sends her signature of the concatenated α^x and α^y, so that Bob might verify Alice's signature using Alice's public key. The verification works because Alice and Bob themselves have influence over what should be signed. Alice trusts Bob because he encrypts his signature with K, which demonstrates that he in fact created y.

The protocol doesn't address how Alice and Bob end up with each other's public keys, which is a very essential part of the protocol. Without the pre-distributed public keys the protocol doesn't work. This is what the STS protocol with certificates addresses.

The STS protocol with certificates prerequisite that Alice and Bob have been provided with a certificate signed by a trusted party T. It is necessary that Alice and Bob both trust T and that they have T's public key. Alice and Bob do not know any of each other's keys and do not share any information. This version of the protocol doesn't rely on a network wide α and p. Alice and Bob's α and p are individual and included in the certificates, see Protocol 4.8. Alice's α and p are also used by Bob in order to obtain the shared secret.

$$\begin{aligned}&1.\ A \rightarrow B : \alpha, p, \alpha^x\\&2.\ B \rightarrow A : \alpha^y, Cert_B, \{sign_B([\alpha^x, \alpha^y])\}K_{xy}\\&3.\ A \rightarrow B : Cert_A, \{sign_A([\alpha^x, \alpha^y])\}K_{xy}\end{aligned} \tag{4.8}$$

The sequence of the STS protocol with certificates is very similar to the basic STS protocol. The difference is that Alice gets Bob public key from the certificate $Cert_B$. Alice can verify the public key of Bob as the certificate include T's signature hash over Bob's name and public key. When acquiring the public key, Alice can verify Bob's identity the same way as in the basic STS protocol. Bob gets Alice's public key in a similar fashion and verifies Alice's identity as in the basic STS protocol.

Once again this protocol doesn't explain how Alice and Bob acquires T's public key. This exchange is also vulnerable to the man-in-the-middle attack. The advantage is that this public key only needs to be distributed once, and that it can be used to verify numerous parties, knowing this public key. The public key of T is usually distributed at installation or by bootstrapping. The certificate can be requested from T in a secure manner when Alice knows T's public key.

4.12 Curve25519

This section is based on the paper, *Curve25519: new Diffie-Hellman speed records* by Daniel J. Bernstein [Ber06]. The Curve25519 is the Diffie-Hellman algorithm used to create a shared key from a public value and private value in the NaCl crypto_box symmetric encryption used by Trifork.

Curve25519 is a elliptic curve Diffie-Hellman function. It allows a sender and a receiver to obtain a shared secret without sharing any information considered secret. Curve25519 is considered very fast as is has obtain new speed records for high-security Diffie-Hellman computations. It is more than twice as fast as any other elliptic curve Diffie-Hellman functions, which has the same security level.

Curve25519 computations are limited to the group operation, scalar multiplication on $E(F_{p^2})$, where p is the prime number, $2^{255} - 19$, and E is the elliptic curve, $y^2 = x^3 + 486662x^2 + x$. The prime number p is also where Curve25519 gets its name.

Protocol 4.9 shows the basic Diffie-Hellman key exchange with respect to Curve25519. Alice and Bob's initial knowledge is their own private keys respectively: a and b. They both generate their public key, A and B, using the Curve25519 function with their private keys and the public string 9. Alice sends her public key to Bob and Bob sends his public key to Alice.

$$\begin{aligned} &1.\ A \rightarrow B : Curve25519(a, 9) \\ &2.\ B \rightarrow A : Curve25519(b, 9) \end{aligned} \tag{4.9}$$

Alice and Bob can now generate the shared secret Q by using their own private key and the other's public key as input for Curve25519. The only information an eavesdropper would be able to collect is the public keys. There is no way the eavesdropper would be able to compute the shared secret from the two public keys. A private key is needed to compute the shared secret, and the problem of distracting the private key from its public key is considered a very hard problem.

The protocol is vulnerable to the Diffie-Hellman protocol attack, so other means of security is needed to overcome this attack.

4.13 Summary

The State of the art chapter has investigated the existing system of Trifork, and cryptographic techniques relevant to secure communication. This includes symmetric and asymmetric encryption and Diffie-Hellman key exchange. Current state of the art cryptographic techniques has been investigated along the cryptographic techniques used by Trifork in the NaCl crypto library. The knowledge acquired can now be used to make responsible design choices.

Chapter 5

Protocol Requirements

This section refines the security goal from section 3.3 to protocol requirements, with regards to an attack model.

5.1 Protocol Goal

The overall goal of the protocols is to ensure the security properties from section 3.3, when enforcing policies. The security properties are confidentiality, data integrity and authentication. Additionally some refining goals are needed for the protocol.

Relay attack The protocol mustn't be vulnerable to relay attacks, meaning that authentication must ensure that no attacker is relaying messages to influence a *CPE*.
Replay attack The protocol mustn't be vulnerable to replay attacks, meaning that messages can't be reused to influence a *CPE*.
Crypto analysis The attacker mustn't be able to identify packages, meaning that the ciphertext must vary for each transfer.

5.2 Attack Model

In order to characterise a protocol secure, an attack model is needed. When an attack model is defined you know the kind of attacker you need to defend against. Protocols that complies with the attack model ensures the customer that the security properties are met according to the attack model.

The Dolev-Yao[DY83] attack model serves as a comprehensive attack model for distributing and enforcing policies. Here the attacker is only limited by the constraints of the used cryptographic methods.

1. An attacker controls the communications between all parties which means that the adversary can observe all messages sent, alter messages, insert new messages, delay messages or delete messages.
2. An attacker can obtain any session keys used in previous protocol runs.
3. An insider, who is part of the protocol, will not act malicious but according to the protocol.

What is important to notice is that, there exits the notion of a passive attacker and an active attacker. The passive attacker can observe all messages and thereby also store these for later use. The active attacker can in theory also do this, but we work with the distinction, where the active attacker can alter messages, insert new messages, delay messages and delete messages.

The active attacker is interested in impersonate one of the two communicating parties in order to exploit the influence of this party. It is therefor important for parties to authentication each other. The attacker might wish to change some values with in a message to influence some action. To prevent the action data integrity ensures that altered values are detected.

The passive attacker is interested in recording the messages and learning valuable information. For this reason confidentiality is needed. The passive attacker is therefor interested in breaking the encryption enforcing the confidentiality by either a flawed key exchange or leak of a secret key. Point 2 in the attack model might seem extreme, as attackers will be able to obtain any session keys from previous protocol runs. In term it means that should the long term shared key be used for such a protocol run it would be compromised, and all previous protocol runs would be exposed, breaking confidentiality. Therefor a new session key should be generated for each protocol run, ensuring confidentiality should the long term secret key be exposed. Generating a new session key for each protocol run is to implement *Forward Secrecy.*

Generating session keys and ensuring authentication is achieved through a series of messages, whereas data integrity in ensured message specific. Requirements for data integrity is therefor a more simple matter than requirements for authentication and establishing session key. For a symmetric encryption scheme MACs are used to ensure data integrity and an asymmetric scheme digital signatures are used.

5.3 Requirements for authentication and session key establishment

Requirements for generating session keys and ensuring authentication are more complex and exists in different degrees of security strength. Therefor a more thorough evaluation of requirements for generating session keys and ensuring authentication are needed. This section refines these requirements for authentication and session key establishment.

5.3.1 Authentication requirements

Authentication comes in two versions, as described in section3.3. One that deals with authentication between two parties, *Entity Authentication*, and one that incorporate entity authentication and data integrity, *Data Origin Authentication.* This section considers entity authentication and to which degree entity authentication should be realised. Data origin authentication aren't considered as, it is accomplished if entity authentication and data integrity are supported.

> *Entity authentication* is the process whereby one party is assured (through acquisition of corroborative evidence) of the identity of a second party involved in a protocol, and that the second has actually participated (i.e., is active at, or immediately prior to, the time the evidence is acquired).[MvOV01]

It is a requirement that the protocol enforcing policies, should support mutual authentication.

Mutual authentication occurs if both entities are authenticated to each other in the same protocol.[BM03]

Protocol 5.1 is an example of a protocol implementing mutual authentication, where B is authenticated to A in message 2, by returning A's random challenge, N_A, signed with B's private key. A is authenticated to B in a similar fashion in message 3. The protocol can by no means be considered secure for this system, but is a mere example of mutual authentication using signatures.

$$\begin{aligned} &1.\ A \rightarrow B : N_A \\ &2.\ B \rightarrow A : N_B, Sig_B(N_A) \\ &3.\ A \rightarrow B : Sig_A(N_B) \end{aligned} \tag{5.1}$$

Mutual authentication is important when enforcing policies. It ensures both entities that they in fact is communicating with a trusted *CPE*. Using unpredictable nonces together with mutual authentication will also make the protocol resilient against replay attacks.

It is a requirement that the protocol enforcing policies should support strong entity authentication.

Strong entity authentication of A to B is provided if B has a fresh assurance that A has knowledge of B as her peer entity.[BM03]

Protocol 5.2 provides strong authentication of B to A. The fresh assurance is provided by N_A signed by B and the inclusion of A's identity ensures that B is aware of A as the peer entity. If the identity wasn't included A could be sure that B was aware he was communicating with A. The inclusion of A's identity therefor provides a stronger sense of authentication. It also has the added benefit of protecting against reflection attack, should the protocol be vulnerable to it.

$$\begin{aligned} &1.\ A \rightarrow B : N_A \\ &2.\ B \rightarrow A : Sig_B(A, N_A) \end{aligned} \tag{5.2}$$

Protocol 5.3 illustrates how C could use a relay attack on a protocol not supporting strong entity authentication. Here C could make B trust C by using the authenticated messages from A. Identities are included as plaintext to help illustrate the weakening of authentication. By including the identity of A, C

won't be able to exploit this vulnerability.

$$\begin{aligned}1.\ & A \rightarrow C_B : A, B, N_A \\ 1'.\ & C \rightarrow B : C, B, N_A \\ 2.\ & B \rightarrow C : B, C, Sig_B(N_A) \\ 2'.\ & C_B \rightarrow A : B, A, Sig_B(N_A)\end{aligned} \tag{5.3}$$

5.3.2 Session key requirements

To get a quality session key, *Boyd*[BM03] lists two goals, that should be uphold. The session key should be good and key confirmation should be provided. It is therefor a requirement that the enforcement protocol should meet these goals. Should the policy distribution protocol require the need to provide new key material to the *CPEs* the same requirements apply to this key.

The following is the definition of the good key goal for shared session keys.

> The shared session key is a *good key* for A to use with B only if A has assurance that[BM03]:
>
> - the key is fresh
> - the key is known only to A and B and any mutually trusted parties

A similar one exists for public session keys.

> The public session key is a *good key* for A to use with B only if[BM03]:
>
> - the key is fresh
> - the corresponding private key is known only to B

To ensure that the key is fresh, nonces are used. In protocol 5.4, X_A and X_B are the values needed to create the session key. A knows that the values received from B, X_B, is fresh, because it resides along A's nonce N_A. The same is true for B in message 3. The session key feeds, X_A and X_B, are not confidential, so if the key should be know to only A, B and any mutually trusted parties, a Diffie-Hellman styled key exchange would be need here. Other protocols might not need Diffie-Hellman key exchange, if the session key feed could be kept confidential.

Protocol 5.4 is a key exchange protocol where both parties are providing key material. Freshness can also be accomplished for key transportation protocols. If X_B where confidential between A and B in protocol 5.4, X_B could be used as the session key, as A know it to be fresh, because the inclusion of N_A.

$$\begin{aligned} &1.\ A \rightarrow B : N_A \\ &2.\ B \rightarrow A : Sig_B(N_A, N_B, X_B) \\ &3.\ A \rightarrow B : Sig_A(N_B, X_A) \end{aligned} \tag{5.4}$$

Having a good session key will ensure that messages can't be identified by the ciphertext, as the same plaintext will results in different ciphertext, when using different session keys. Thereby the crypto analysis goal is also fulfilled.

The good key requirement concludes the requirements for authentication and session key establishment. The requirements that the policy enforcement protocol should support is:

- Mutual authentication
- Strong entity authentication
- Good key

Chapter 6

Protocol Design

This section investigates how to design the protocol used to enforce policies (Policy enforcement protocol). Before considering the protocol some general design decisions are to be made. A design of the policy setup is required: which information should the *CPEs* exchange to enforce the customer's policies. A freshness approach is also needed, where one of three should be chosen: counters, nonces or timestamps. When this is done different protocols will be considered for the Policy enforcement protocol.

6.1 Policy-based protocol

The design, of a policy-based protocol, is illustrated in figure 6.1. Here the event is triggered by the sensor of *CPE A* and the action is performed by the actuator of *CPE B*. Events and actions are specific to the sensors and actuators of a *CPE*. This mean that *CPE B* have no way of interpreting the event from *CPE A* directly. Because of this, informing *CPE B* of a sensor event by sending the event along wouldn't do much by itself. Some sort of mapping is needed between sensor events and actuator actions.

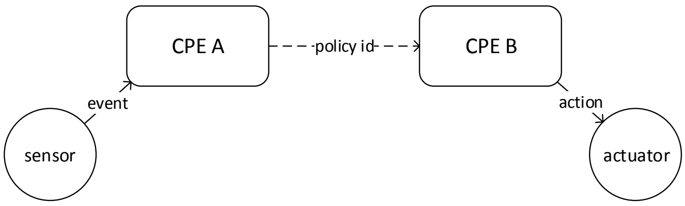


Figure 6.1: Policy-based sequence

Consider the customer's policy from section 3.1, which is seen below.

$$\begin{aligned} Policy &: \text{ IF window } a \text{ is opened THEN turn off radiator } b \\ Event &: \text{ window } a \text{ is opened} \\ Action &: \text{ turn off radiator } b \end{aligned}$$

This is an overall policy, and from this two sub-policies can be created. One for *CPE A* and one for *CPE B*. The idea is to have these sub-policies enforce the overall policy across *CPEs*. To supply a mapping between sensor event and actuator action, a policy id is introduced. The policy id is then send from the *CPE* experiencing a sensor event to the *CPE* performing the actuator action. The sub-policies of *CPE A* and *CPE B* can be seen below.

$$\begin{aligned} CPE\ A &: \text{ IF window } a \text{ is opened THEN send policy id } i \\ CPE\ B &: \text{ IF policy id } i \text{ received THEN turn off radiator } b \end{aligned}$$

When *CPE A* recognises the sensor event from the policy, it would then send the policy id to *CPE B*. *CPE B* would then match the policy id to an existing policy and perform the action associated with that policy.

Why not use the sensor event as mapping between *CPEs*, as seen below.

$$\begin{aligned} CPE\ A &: \text{ IF window } a \text{ is opened THEN send window } a \text{ is opened} \\ CPE\ B &: \text{ IF window } a \text{ is opened THEN turn off radiator } b \end{aligned}$$

There are two reasons for not doing this. The first is that by using a policy id, a fixed length random value can be used. The length of sensor events can't be counted on, to have a fixed length. Having a fixed length message means that the corresponding ciphertext will have a fixed length. This helps to conceal the intend of the message and thereby limits the attacker's opportunity to perform crypto analysis. The other reason isn't quite as scientific, but there is something reassuring by not explicitly sending, which sensor experiences an event and the content of that event. This is despite the information being encrypted.

6.2 Counters, Nonces or Timestamps

To comply with the requirements for authentication and session key establishment, assurance of freshness is needed. Freshness is usually achieved by introducing one of three approaches; counters, nonces or timestamps, into a protocol. The three approaches ensures that no number is used twice for a protocol. It is necessary to decide a freshness approach before deciding a protocol, as protocols vary based on the freshness approach.

Counters ensures that no number is used twice by having a synchronised counter between the communicating parties. For every new message the value is sent along and then incremented. It is necessary, to prevent attacks, that the state is synchronised at all time. Keeping the counter synchronised can be quite challenging. The state of counters must be stored for every communication partner. Should a counter be out of sync, it can be quite troublesome recovering, and synchronising the counter again. A user could synchronise the counter or another freshness approach could help recovering from a counter out of sync. Using another freshness approach sort of nullifies the reason for using counters in the first place.

Nonces rely on random challenges to authenticate and ensure fresh keys. A nonce N_A is generate and sent from A to B. B returns this nonce to A along with the message which has been encrypted. A can now confirm that the received nonce is in fact identical to N_A, which ensures A that the response from B is fresh. Using nonces can result in an increased number of interactions in a protocol, as both parties need to verify the random challenge sent. The minimum number of interactions is therefor three, where e.g. the counter- and timestamp approach can do with just two interactions.

Timestamps are used along with a synchronised time clock between the communicating parties to ensure freshness. A timestamp are sent along side the message which allows the receiver to validate the freshness according to the receivers own time clock. Using timestamps introduces some difficulties: it is necessary to agree on an interval where the sent message can be considered fresh, the time clock between the parties need to stay synchronised. A time clock out of sync, opens a protocol to exploits. Recovering from an unsynchronised time clock introduces the same inconvenience as a counter out of sync.

Nonces seem like the more attractive choice. There is no need to synchronise nonces, should they come out of sync, and implementation a protocol using random numbers rather than counters or timestamps seems likely to have better success of not introducing freshness errors. Based on these reasons protocols using nonces will be considered.

6.3 Policy enforcement protocol

The goal of this protocol is to enforce a policy created by a customer. More specifically transfer a policy id from one *CPE* to another, while still upholding the security goals and protocol requirement. The protocol needs to establish a session key with an authenticated party, and use this session key to encrypt the policy id before sending it to the intended *CPE*.

According to C. Boyd and A. Mathuria [BM03] there are three things to consider when designing a secure session key protocol: the number of users, the existing cryptographic keys, and the method of session key generation.

Number of Users
The protocol has two active users: *CPE A* and *CPE B*. The protocol can assume that both parties previously have had contact with a mutually trusted party.

Existing Cryptographic Keys
It is not possible to obtain a secure session key and authentication for a protocol, if there is no initial trust established. This mean that either the two communicating share a key, or they both trust a mutual third party, usually a server. The different ways of having initial trust are:

1. The *CPEs* already shared a secret key.
2. An online server is used. The *CPEs* share a key with a server. The server is contacted when a new session key is needed.
3. An offline server is used. This is essentially the usage of public key certificates, signed by a mutual trusted third party.

The protocol could support a shared secret key. This would mean that the *Client* would need to supply the two *CPEs* with a shared secret key.

The protocol couldn't support an online server, as the number of active users are 2, *CPE A* and *CPE B*. The only party both *CPEs* trust is the *Client*, which would need to act as the online server. The *Client* (smartphone) cannot be counted on to always be online, which is why this approach wouldn't work.

The protocol would be able to support an offline server. Here the *Client* could be used as the server. At distribution, the *Client* is already supplying the policies, which would make it possible to supply certificates along side them.

This means that option 1 and 3 is being consider as an approach for the Policy enforcement protocol.

Method of Session Key Generation
There exists three different ways of generating a session key: key transportation, key agreement and a hybrid of the two. The hybrid key generation is not considered as it is only viable for protocol with more that two active participants. Therefore only two are considered.

1. Key transportation, where one *CPE* generates the session key and thereafter transfers it to the other *CPE*.
2. Key agreement, where both *CPEs* provided input for the function generating the session key.

Both key transportation and key agreement can be used for the Policy enforcement protocol. Key agreement is usually preferred in protocols. Here input for the session key generation is provided by two different entities, which results in a more random secret key[BM03]. Using key agreement also allows one to use a Diffie-Hellman based key exchange, which ensures that an attacker wouldn't be able to derive the secret key from the exchanged information. Based on these reasons key agreement is selected, as the method for generating session keys.

The previous considerations leaves two protocols to consider, as the approach for the Policy enforcement protocol.

1. Shared secret with key agreement (section 6.3.1).
2. Offline server with key agreement (section 6.3.2).

Now two protocols will be suggested, one for each approach. After that the protocols will be evaluated, to see which one is the better choice, as the Policy enforcement protocol.

6.3.1 Policy enforcement protocol using shared key cryptography

Here we consider a protocol from the ISO standard, *ISO/IEC 11770-2*[ISO96], key establishment protocols using symmetric key encryption. Protocol 6.1 illustrates *Key Establishment Mechanism 6* (ISO-6) from this standard.

The underlined value is not part of the original protocol. It is the policy id encrypted with the session key. In order to save on the number of interactions, the policy id is sent as part of message 3. The reason for encrypting the policy id, with the session key, rather than only relying on the encryption of the long-term symmetric key, K_{AB}, is: to ensure that the same policy id encrypted will result in different ciphertext. The session key, $K_{R_AR_B}$, is derived from the two random generated key values R_A and R_B. The key values are kept confidential under the encryption of the shared secret key, K_{AB}, which makes it possible for A and B to use any function to generate the session key.

$$\begin{aligned}
&1.\ A \rightarrow B : N_A \\
&2.\ B \rightarrow A : \{|N_B, N_A, A, R_B|\}_{K_{AB}} \\
&3.\ A \rightarrow B : \{|N_A, N_B, R_A, \underline{\{policy_{id}\}_{K_{R_AR_B}}}|\}_{K_{AB}}
\end{aligned} \tag{6.1}$$

If the values for R_A and R_B and the function generating the session aren't selected with care, perfect forward secrecy might break, exposing the policy id. The values selected for the session key generation should therefor follow a Diffie-Hellman key exchange, making the session key secure. The Diffie-Hellman key exchange should use elliptic curves to help minimise the key size.

Protocol 6.2 has made the use of public session keys using elliptic curves explicit, and will be the protocol considered for the Policy enforcement protocol using shared key cryptography.

$$\begin{aligned}
&1.\ A \rightarrow B : N_A \\
&2.\ B \rightarrow A : \{|N_B, N_A, A, xG|\}_{K_{AB}} \\
&3.\ A \rightarrow B : \{|N_A, N_B, yG, \underline{\{policy_{id}\}_{K_{xyG}}}|\}_{K_{AB}}
\end{aligned} \tag{6.2}$$

The ISO-6 protocol using elliptic curve Diffie-Hellman satisfies all the requirements for the Protocol enforcement protocol. Mutual authentication is provided by challenge responses of the nonces N_A and N_B. Strong entity authentication is provide of B to A, by the inclusion of A's identity and A's nonce in message 2. The inclusion of A's identity and A's nonce, provides A with a fresh assurance that B has knowledge of A as his peer entity. The public session keys xG and yG can be considered good. A knows the public session key of B to be fresh by A's associated nonce N_A, and B knows the public session key of A to be fresh by B's associated nonce N_B

By fulfilling the requirements for authentication and session key establishment, the session key, xyG, can now be used to ensure confidentiality for the policy id. Data integrity can, for symmetric key protocols, be ensured by providing a MAC along the message, e.g. using *Poly1305* or a *HMAC* to compute such a MAC. The recipient will then be able to check if messages have been altered. The use of a fixed length policy id and a session key to encrypt it also ensures that a package can't be identified from the ciphertext alone, fulfilling the crypto analysis requirement.

Using protocol 6.2 the *Client* would need to supply the *CPEs* with a *sub-policy* and a *shared symmetric key*, paired with the *identity* of the *CPE* sharing the symmetric key.

6.3.2 Policy enforcement protocol using offline server cryptography

A protocol using an offline server is based upon certificates, where the protocol entities A and B, receive certificates. These certificates have been signed by a mutual trusted party, which ensures the validity of B's public key for A, and vice versa. The exchange of certificates is done for A and B to obtain each other's public keys. The certificate could also include domain parameters, but these are excluded in the following protocols to simplify expressions. Instead domain parameters are considered to be global.

The STS protocol rely on certificates to exchange public keys. Protocol 6.3 illustrates the STS protocol using certificates, and is similar to the STS protocol from the *State of the art* section. Protocol 6.3 uses the subgroup of elliptic curves.

$$\begin{aligned} &1.\ A \rightarrow B : xG \\ &2.\ B \rightarrow A : yG, Cert_B, \{|Sig_B([yG, xG])|\}_{K_{xyG}} \\ &3.\ A \rightarrow B : Cert_A, \{|Sig_A([xG, yG]), \underline{policy_{id}}|\}_{K_{xyG}} \end{aligned} \tag{6.3}$$

Looking at protocol 6.3, it becomes apparent, that strong entity authentication is not supported. B doesn't give A fresh assurance that B wishes to communicate with A; A's identity is not included in message 2. This means that an attacker could masquerade as B in regards to A, breaking strong entity authentication.

Protocol 6.4 includes the identity of A to ensure strong authentication. Including the identity of A, nullifies the need for the symmetric encryption over the signature in message 2 and -3[BM03], which is also expressed in protocol 6.4.

$$\begin{aligned} &1.\ A \rightarrow B : xG \\ &2.\ B \rightarrow A : yG, Cert_B, Sig_B([yG, xG, A]) \\ &3.\ A \rightarrow B : Cert_A, Sig_A([xG, yG, B]), \underline{\{|policy_{id}|\}_{K_{xyG}}} \end{aligned} \tag{6.4}$$

Protocol 6.4 now ensures strong entity authentication. Mutual authentication is ensured by using the public session keys xG and yG as challenge-responses. A knows is assured of B identity, because A trusts B's public keys from the certificate, and because only B would be able to supply this signature over A's public session key xG. Both strong entity authentication and mutual authentication are supported by the protocol satisfying the authentication requirements.

The session key requirements are also satisfied. A know the public session key of B to be fresh, as it is sent along A's public session key in message 2. The same freshness assurance is provide to B, where A's public session key is sent along B's public session key in message 3. The public keys do not need to be kept confidential. A and B only need to have assurance of data integrity of the keys. Data integrity is provided by signatures for the STS protocol.

In addition to satisfying all the protocol requirements, protocol 6.4 have some functional advantages over protocol 6.3. Test results are being referenced from appendix A.1. Comparing message 2 of the protocols; protocol 6.4 has a 16 byte smaller payload than protocol 6.3 and is 1,04168 times faster to compute. For message 3 the protocols have the identical payload. Message 3 isn't that different for the two protocols. The difference is that the signature isn't encrypted under the session key, K_{xyG}, for protocol 6.4. The difference doesn't affect the security of the protocols, but protocol 6.4's message 3 is computed 1,03160 times faster than message 3 of protocol 6.3. The difference in cryptographic computation time, probably won't affect the system, but why not use the faster of the two, when security isn't compromised.

There is no doubt that protocol 6.4 is the better protocol, compared to protocol 6.3. The question is if it is good enough. A and B are exchanging certificates, to obtain each other's public key. It might be better, to just have the *Client* distribute the public keys of the *CPEs*, rather than the certificate. Tests referenced is from appendix A.2. The test shows that storing four public key, from different *CPEs*, corresponds to storing one certificate. If also factoring in that verification isn't needed when receiving the certificate, having the public key stored rather than exchanging certificates seem more efficient. The system considered here, would rarely have more than four *CPEs*, which is why public key easily could be stored instead of using certificates. Protocol 6.5 is the version of the STS protocol without certificates.

$$\begin{aligned}
&1.\ A \rightarrow B : xG \\
&2.\ B \rightarrow A : yG, Sig_B([yG, xG, A]) \\
&3.\ A \rightarrow B : Sig_A([xG, yG, B]), \underline{\{|policy_{id}|\}_{K_{xyG}}}
\end{aligned} \tag{6.5}$$

Removing the certificates doesn't affect the assurance of the protocol requirements, as certificates are only relied upon to distribute public keys in a secure manner. Protocol 6.5 has been put through the model checker, *Open-Source Fixedpoint Model-Checker*, to identify possible attacks on the protocol. No attacks where found for 2 sessions, of the model checker (Appendix A.4).

Protocol 6.5 is chosen as the Policy enforcement protocol using an offline server, and the one to evaluate against the protocol, selected for the shared key cryptography. Using protocol 6.5 the *Client* would need to supply the *CPEs* with a *sub-policy* and a *public key*, paired with the *identity* of the public key owner.

6.3.3 ISO or STS for the Policy enforcement protocol

This section compares the two approaches, chosen as possible solutions for the Policy enforcement protocol: protocol 6.2 (ISO-6) and protocol 6.5 (STS). The section compares the efficiency of the two protocols and consider the storage needed for each approach.

1. N_A
2. $\{|N_B, N_A, A, xG|\}_{K_{AB}}$
3. $\{|N_A, N_B, yG, \underline{\{policy_{id}\}_{K_{xyG}}}|\}_{K_{AB}}$

(a) Protocol 6.2 (ISO-6).

1. xG
2. $yG, Sig_B([yG, xG, A])$
3. $Sig_A([xG, yG, B]), \underline{\{|policy_{id}|\}_{K_{xyG}}}$

(b) Protocol 6.5 (STS).

Figure 6.2: Protocols considered for the Policy enforcement protocol.

The efficiency of a protocol depends one three parameter, which is an expression of the total protocol run time. How many interactions, does the protocol require. How long does it take to ready messages, sent between *CPEs*. How long does it takes to transfer the messages.

The number of interactions are the same for both protocols (see figure 6.2). Therefor the number of interactions are disregarded.

Payload is a representation of the time it takes to transfer messages between *CPEs*, and is thereby relevant for efficiency. A smaller payload results in a faster transfer time. This is relevant for evaluating the protocols, as the payload for message 2 and -3 vary between the protocols.

Crypto time is a representation of the time it takes to ready a message, and is thereby relevant for efficiency. Message 2 and -3 are encrypted before sent and decrypted at reception. Therefore the time it takes to encrypt and decrypt will be measured for message 2 and -3.

Storage is the final thing considered. The two protocol approaches should require different amount of storage. It isn't relevant to the efficiency of the protocol, but it's relevant for smaller devices, like a *CPE*, as they might have limited storage space.

Evaluating the three parameters: payload, crypto time and key storage, for each protocol, should help identify the better approach for the Policy enforcement protocol.

6.3.3.1 Payload

Results for the payload can be found in appendix A.3. The reason for considering payloads, is that the size of the payload impact the transfer time of the message between the *CPEs*. The greater payload the longer transfer time. The more efficient protocol will therefor have the smaller payload.

Message 1: The ISO-6 protocol and the STS protocol has identical payloads.

Message 2: The ISO-6 protocol's payload is 80 B smaller than the payload of the STS protocol, which corresponds to a factor of 2.

Message 3: The ISO-6 protocol's payload is 56 B smaller than the payload of the STS protocol, which corresponds to a factor of 1.6.

The ISO protocol has, for message 2 and message 3, a smaller payload; payloads which is smaller by factors of 2 and 1.6. From a payload perspective the ISO-6 protocol is the better choice for the Policy enforcement protocol.

6.3.3.2 Crypto time

Results for the crypto time, for encrypting and decrypting message 2 and -3, can be found in appendix A.3. The reason, for considering crypto time, is that it together with the transfer time constitutes the total protocol run time. The better protocol, from a crypto time perspective, will therefor have the faster crypto time. Message 1 isn't considered, as no encryption or decryption is needed.

Message 2: The ISO-6 protocol's crypto time is 7,12 ms faster than the crypto time of the STS protocol, which corresponds to a factor of 3954.

Message 3: The ISO-6 protocol's crypto time is 7,11 ms faster than the crypto time of the STS protocol, which corresponds to a factor of 3476.

The ISO protocol has, for message 2 and message 3, a significantly faster crypto time; crypto times which is faster by factors of 3758 and 3128. From a crypto time perspective the ISO-6 protocol is the better choice for the Policy enforcement protocol.

6.3.3.3 Key storage

For key storage, Appendix A.2, is used, which contain key storage requirements tests for both protocols. The reason, for considering key storage, is that IoT devices have a limited amount of storage. The better protocol, from a key storage perspective, will therefor require the least amount of storage.

ISO protocol: The storage requirements for the ISO-6 protocol follow the function: $40B \times relations$

STS protocol: The storage requirements for the STS protocol follow the function: $40B \times relations + 96B$

Both protocols need to store a key and an identity for each relation they have. In addition, for the STS protocol, one needs to store one's own public- and private key, which constitutes the $96B$. This mean that the STS protocol would always require $96B$ more storage than the ISO protocol. For Trifork's setup, this is not true. In this setup, the *CPEs* would already store this private- and public key. For Trifork this means that neither protocol is better from a storage perspective. For others not having this setup, the ISO protocol is the better protocol from a storage perspective.

6.4 Conclusion on protocol design

The ISO-6 protocol turns out to be a better fit for the Policy enforcement protocol. It is better or equally as good as the STS protocol, when considering payload, crypto time and key storage. The ISO-6 protocol also meets all the protocol requirements from section 5, while being the more efficient protocol. The ISO protocol, protocol 6.2, will serve as the Policy enforcement protocol.

1. $A \rightarrow B : N_A$
2. $B \rightarrow A : \{|N_B, N_A, A, xG|\}_{K_{AB}}$
3. $A \rightarrow B : \{|N_A, N_B, yG, \underline{\{policy_{id}\}_{K_{xyG}}}|\}_{K_{AB}}$

Figure 6.3: Policy enforcement protocol (ISO-6).

Chapter 7

System design

The system design reflects how the Policy enforcing protocol might be implemented in a policy-based system. The design doesn't reflect the design of Trifork and the existing manufacturers' systems, but is the design of the prototype. It's designed to see how a sensor event from one *CPE* might lead to an actuator action for another *CPE*. The design is mainly centred around the software of the *CPE*, and how the protocol is incorporated in the system. The system design includes considerations for programs representing the *Client*, sensors and actuators. These are simple programs that, when implemented, helps to illustrate the flow of the policy-based system. The software, to the *CPE*, is provide by Trifork and a manufacturer. To make this clear, the design strives to separate the software into two parts, which work independently and only communicate using messages.

The system design includes a package diagram illustrating the layered architecture of the system. It also includes an overview of processes needed and their connectivity to illustrate the flow of the system. For the *CPE* software, there is provided an overview of running threads and how these communicate to enforce policies. Interaction diagrams are included for the protocol flow, to give a dynamic representation of the system. Finally a class diagram is include, it gives a static representation of the software designed for Trifork.

7.1 Package diagram

The package diagram shows the logical architecture of the *CPE* system, including Trifork's- and the Manufacturer's software. Notice in figure B.1, that the UI layer and the Data layer has been greyed out. Usually, when designing software systems, you rely on a layered architecture to separate your software. This allow engineers to substitute old layers for new, should the system, e.g. need a new user interface. Seen from this thesis' point of view, the *CPE* system doesn't rely on any user interface. The need for persistence of data isn't really important either. The data in need of persistence would be long-term keys paired with identities, and policies. Instead of using a data layer, keys are incorporated in the code, whereas policies will be distributed by the *Client*.

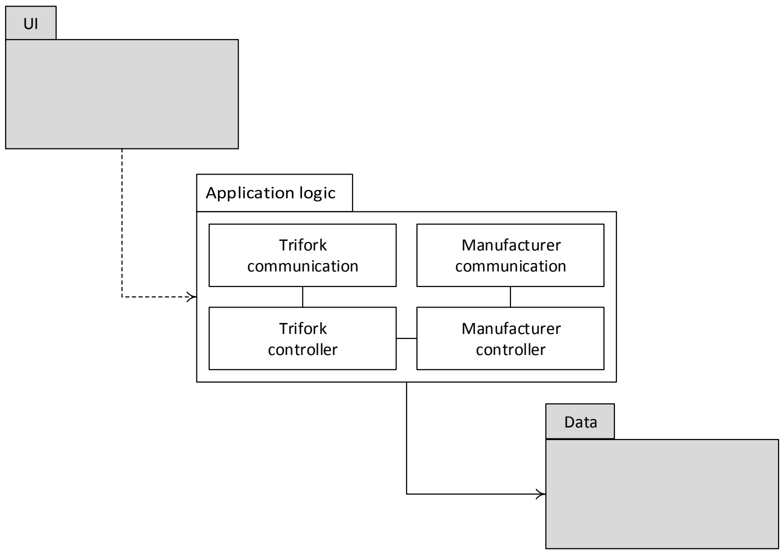


Figure 7.1: Package diagram for *CPE*.

The application logic layer embraces the scope for this thesis. Here the Policy enforcement protocol is implemented. It is only possible to make new additions to the software written by Trifork. New software additions can therefor only be made to the *Trifork communication-* and the *Trifork controller* package. I have also designed a simple *Manufacturer communication-* and *Manufacturer controller* package. This is done to simulate the entire flow of a sensor event from one *CPE* to the actuator action of another *CPE*.

The *Trifork communication* package facilitates all incoming- and outgoing communication with *CPEs* and *Clients*. It is here the Policy enforcement protocol will be implemented.

The *Trifork controller* package interprets the messages from *CPEs*, *Clients* and the *Manufacturer controller* package. If a message comes from a *CPE* the message is evaluated against the policies, and the *Manufacturer controller* packade is informed according to the policies, e.g. turn off heating. If the message is from a *Client* the message is translated to a policy, which later, is used for evaluation against messages from *CPEs*. If a message comes from the *Manufacturer controller* package, the message is evaluated against the policies, to see if any *CPE* should be informed of the event in the message.

The *Manufacturer communication* package facilitates all incoming- and outgoing communication with sensors and actuators.

The *Manufacturer controller* package interprets messages from the *Trifork controller* package and the *Manufacturer communication* package. If the message is from the *Trifork Controller* package, the *Manufacturer controller* package makes sure the correct actuator receives the message, by sending a message to the *Manufacturer communication* package. If the message is from the *Manufacturer communication* package, the message is from a sensor, and the *Trifork controller* package is informed of the sensor event.

7.2 Processes

This section gives insight into the processes needed to simulate a home automation system. Figure 7.2 illustrates these processes from a *CPE*'s perspective; the *CPE* with the bold border. The figure has five processes. These five processes run on five unique physical devices, which has wireless communication capabilities. The dashed lines illustrate the connectivity among the devices and the directional arrow identifies the initiator, e.g. the sensor process initiates communication with the *CPE*, and both *CPE* processes may initiate communication with the other *CPE* process.

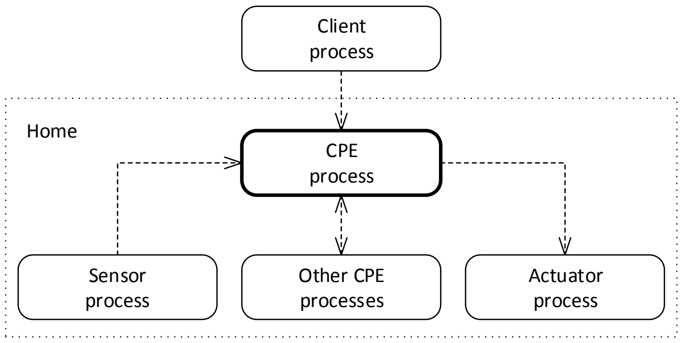


Figure 7.2: Processes in a home automation system and their connectivity.

The *Other CPE processes* box illustrates other *CPE* home automation systems within the customer's home. These processes are also connected to sensors, actuator and the *Client.* These processes and connections have been left out of figure 7.2 to provide a more simplistic illustration.

Having five processes on five different devices means that four different programs should be written. The sensor-, actuator- and *Client* program will be simple programs providing the necessary implementation to illustrate the flow through the system.

The *Client* program is connected to all the *CPEs* within a home, and should provide them with policies and key material

The sensor program needs to initiate contact with the *CPE* when an event happens and provide the *CPE* with the event.

The actuator program should accept connections from the *CPE* and perform the action provided by the *CPE.*

The communication between these three programs and the *CPE* does not provide any security, as it is just clear text sent between them to help reach the actual goal of the system. The goal of having the *CPEs* communicate securely using the Policy enforcement protocol, when policies are satisfied.

The *CPE* program implements the Policy enforcement protocol, that facilitates communication between *CPEs.* Therefore the *CPE* program is the more complex program, which is why the following design section will focus solely on the software design of the *CPE.* The *CPE* software is designed generically, which means that the same program can be used for multiple *CPEs*, only the provided policies will be different, thereby ensuring different behaviour for different *CPE*.

7.3 Threads

This section explains the inner workings of the *CPE* process. It looks at the threads running in the process, how the threads communicate and which messages are passed along between them. The *CPE* process is illustrated in figure 7.3. Messages within the process is passed along using message queues. Incoming- and outgoing messages of the process uses a TCP connection. The process is conceptually split in two by the dashed line. The right side is the manufacturer software and the left side is Trifork's software.

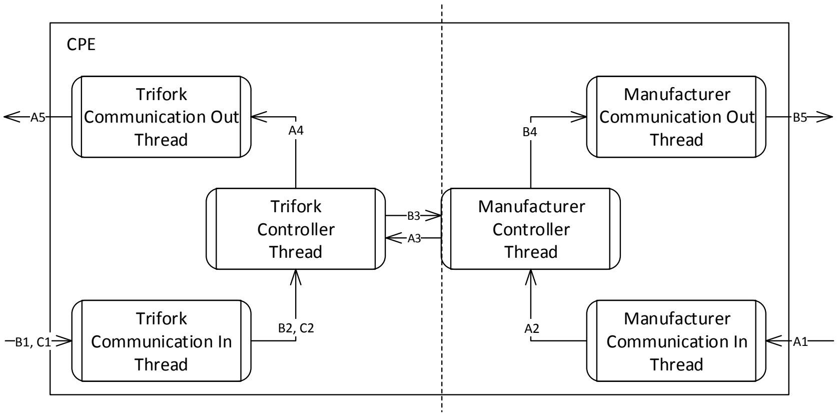

CPE
Trifork Communication Out Thread
A5
A4
Trifork Controller Thread
B3
A3
Trifork Communication In Thread
B1, C1
B2, C2
Manufacturer Communication Out Thread
B4
B5
Manufacturer Controller Thread
Manufacturer Communication In Thread
A2
A1

Figure 7.3: Threads in a *CPE* process and their connectivity.

Trifork's software consists of three threads: *Trifork Communication Out*, *Trifork Controller* and *Trifork Communication In*.

The *Trifork Communication Out* thread facilities the TCP communication, when the *CPE* itself initiates the contact to other *CPEs*. The thread therefore acts as the initiator *A*, for the Policy enforcement protocol, when connecting to other *CPEs*. The thread has a message queue and is only activated when messages is passed to the message queue by the *Trifork Controller*.

The *Trifork Communication In* thread facilities the TCP communication when the *CPE* is contacted by either the *Client* or other *CPEs*. The thread therefore acts as the responder B, for the Policy enforcement protocol, when contacted by other *CPEs*. The thread also receives policies and key material from the *Client*. The thread has a pointer to the *Trifork Controller*'s message queue, to which received messages are relayed. The thread is only active when a TCP connection is accepted.

The *Trifork Controller* evaluates the messages received from the manufacturer software and the *Trifork Communication In* thread against existing policies. If policies are satisfied either the manufacturer software is informed, using the message queue of the *Manufacturer Controller* thread; or other *CPEs* are informed, using the *Trifork Communication Out* thread as a relay. If a messages contain a new policy, it is instead added to the existing policies.

In theory only the messages and threads to the left of the dashed line, would be relevant to consider for the design. This is the software provided by Trifork, which is the only part we can add code to. The manufacturer software, which is to the right of the dashed line, is a very simple design. It reuses the classes designed for the Trifork software to save time on implementation.

The *Manufacturer Communication Out* thread sends a message to an actuator when the *Manufacturer Controller* thread instructs it to.

The *Manufacturer Communication In* thread sends a message to the *Manufacturer Controller* thread's message queue when receiving an event from a sensor.

The *Manufacturer Controller* thread relays messages between the *Trifork Controller* thread and the *Manufacturer Controller*'s communication threads.

7.3.1 Flows

Figure 7.3 also illustrates the three flows of the system: Flow A, -B and -C. It shows how different events trigger different flows. Flows are triggered by the initial messages: A1, B1 and C1. A message contains the type of entity the message is sent from, being it sensor, client or cpe. It also contains event information and may contain information about the action to perform. Messages contain information about: entity type, event and action.

Flow A: illustrates the flow through the *CPE*, from when the manufacturer software receives a sensor event, to the policy id corresponding to that event, has been sent to another *CPE* by Trifork's software.

Flow B: illustrates the flow through the *CPE*, from when Trifork's software receives a policy id from another *CPE*, to the corresponding actuator action has been sent to an actuator by the manufacturer software.

Flow C: illustrates the flow through the *CPE*, from when receiving policies and key material from a *Client*. Here the key material is stored by the *Trifork Communication In* thread. The policies are stored by the *Trifork Controller* thread to evaluate future policy ids and sensor events against.

A closing remark for the section. It has been emphasised in the design, that the manufacturer software and Trifork's software should be able to coexist in the same process while still be two individual systems, where they only communicate through an API styled interface using message queues.

7.4 Interaction diagrams

The following two pages contain interaction diagrams depicting the interactions of the Policy enforcement protocol between two *CPEs*. Figure 7.4 shows the interaction from the initiator's point of view, which corresponds to the actions taken by the *Trifork Communication Out* thread, when it receives message 4 of flow *A*. Figure 7.5 shows the interaction from the responder's point of view, which corresponds to the actions taken by the *Trifork Communication In* thread, when it receives message 1 of flow *B*.

7.4.1 Initiator sequence diagram

The sequence diagram of the initiator, in figure 7.4, is activated by the reception of a message on the class *ComOutTrifork*'s message queue. The key corresponding to *CPE*, which the message is intended to, is fetched from the *CpeRepo*. When the key is fetched, the *CPE* connects to the other *CPE* to inform it that a *CPE* wishes to communicate, before initiating the protocol. The *SocketTCP* class facilitates all the communication to and from the other *CPE*. The *ProtocolISO* class follows the steps of the protocol initiator *A* from the protocol design, when calling the function *Initiate*. The *NaclLibrary* class provides the *ProtocolISO* class with the necessary cryptographic functions. The protocol finishes with the last encrypted message being sent to the other *CPE*, and the *ComOutTrifork* class continuing to listen on its message queue.

7.4.2 Responder sequence diagram

The sequence diagram of the responder is very similar to the initiator sequence diagram. Here the *ComInTrifork* class continuously accepts connections, and is activated by the other *CPE* connecting to it. After the connected *CPE*'s id is received, the corresponding shared key is fetched. After that the *ProtocolISO* function, *Response*, is activated. It performs the necessary steps of the protocol responder *B* from the protocol design. The *NaclLibrary* is used for the necessary cryptographic functions. The sequence finishes by passing along the received message (policy id) to the *Trifork Controller* thread's message queue, and continues to accept incoming connections.

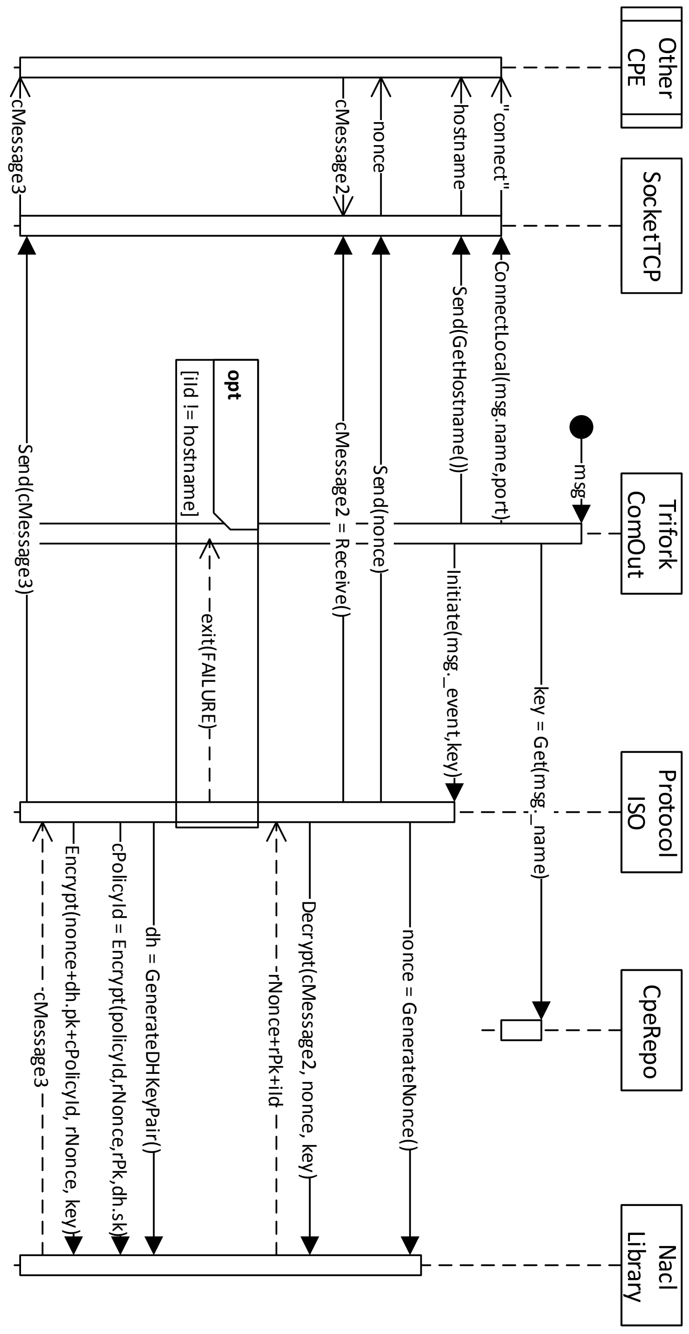


Figure 7.4: Sequence diagram, ISO protocol initiator.

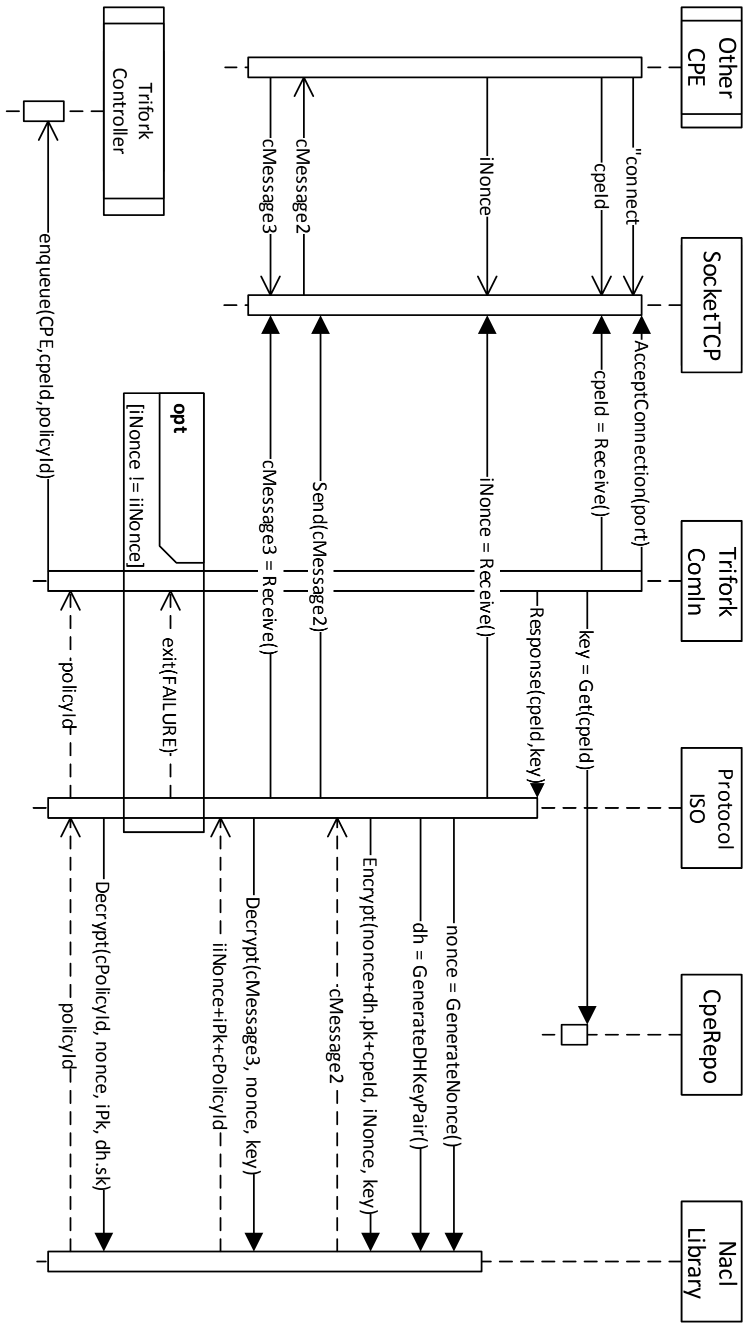


Figure 7.5: Sequence diagram, ISO protocol responder.

7.5 Class diagrams

The following class diagram, shown in figure 7.6, shows the static representation of the Trifork software. If interested the class diagram of the manufacturer software can be seen in appendix B.1.

The three classes representing the three threads of the Trifork software are: *TriforkController*, *TriforkComOut* and *TriforkComIn*. They all inherit from the *Thread* base class, which simplifies thread creation. The *Controller* class, which *TriforkController* inherits from is created to minimise redundancy between the *TriforkController* and the *ManufacturerController*, as they both have three references to the *MessageQueue* class. The *TriforkCommunication* class has been created to minimise redundancy between the *TriforkComOut* class and the *TriforkComIn* class. They both need an *ICommunication* class to accept connections and connect to other *CPEs* and they both need an *IProtocol* class to communicate securely with another *CPE*.

The classes of the *TriforkCommunication* class, *ICommunication* and *IProtocol*, are using the Strategy design pattern, which means that other implementations of the interfaces easily can be used instead. If another protocol, using symmetric encryption, is selected instead of the *ProtocolISO*, it just needs to implement the *IProtocol* using a *ICryptoLibrary* for it to work with the system. The same goes for the *NaclLibrary*, which uses Daniel Bernstein's crypto library. If another class is implemented, which relies on an AES implementation, it just need to implement the *ICryptoLibrary*. In fact the new implementation using AES could be used with the existing *ProtocolISO* class.

The design has been made generic where it makes sense, which makes it easy to change communication protocol, security protocol and crypto library. Other parts just use a simple class, that acts like a database, as the data layer is not provided. These are the repo classes, which contain identities, keys and policies for the *CPE*.

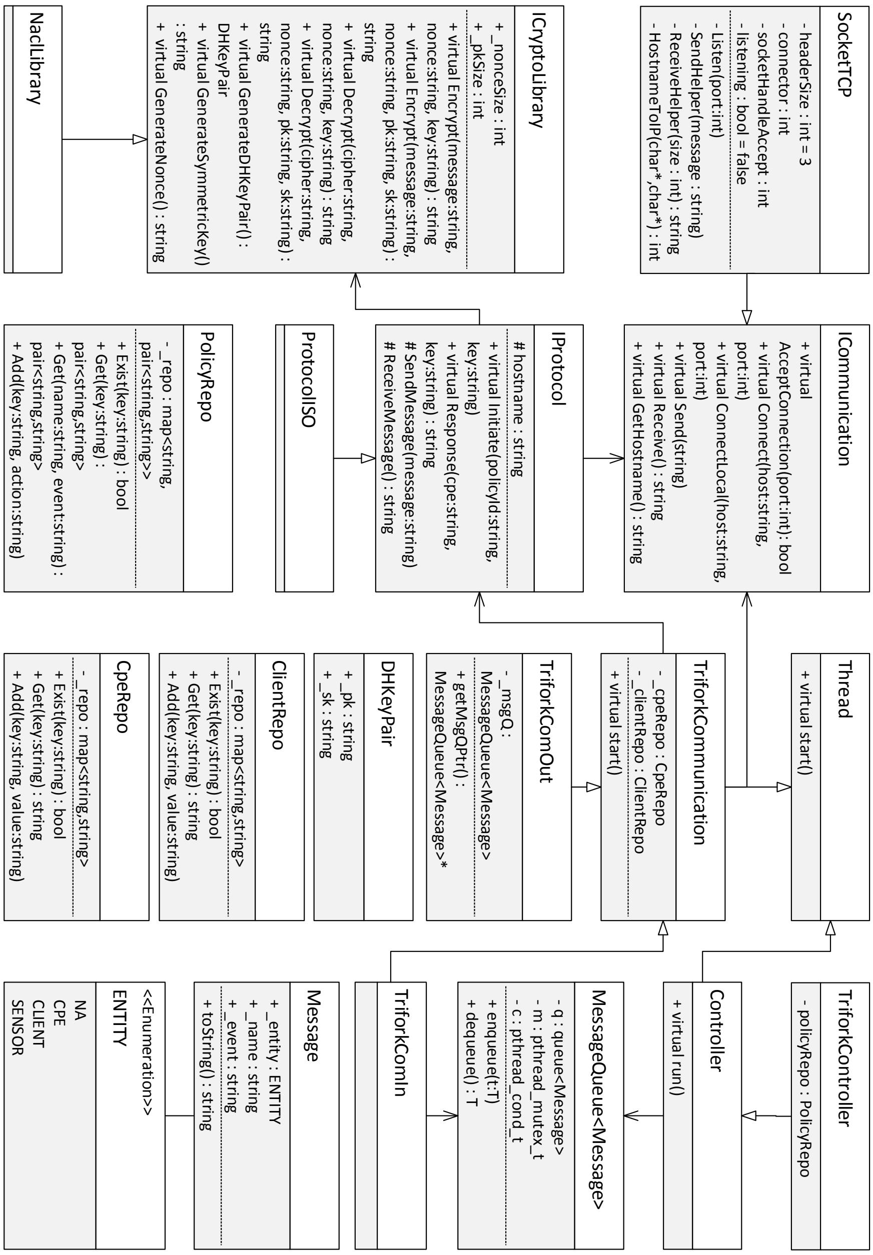

SocketTCP
- headerSize : int = 3
- connector : int
- socketHandleAccept : int
- listening : bool = false
- Listen(port:int)
- SendHelper(message : string)
- ReceiveHelper(size : int) : string
- HostnameToIP(char*,char*) : int
ICryptoLibrary
+ _nonceSize : int
+ _pkSize : int
+ virtual Encrypt(message:string, nonce:string, key:string) : string
+ virtual Encrypt(message:string, nonce:string, pk:string, sk:string) : string
+ virtual Decrypt(cipher:string, nonce:string, key:string) : string
+ virtual Decrypt(cipher:string, nonce:string, pk:string, sk:string) : string
+ virtual GenerateDHKeyPair() : DHKeyPair
+ virtual GenerateSymmetricKey() : string
+ virtual GenerateNonce() : string
NaclLibrary
ICommunication
+ virtual AcceptConnection(port:int): bool
+ virtual Connect(host:string, port:int)
+ virtual ConnectLocal(host:string, port:int)
+ virtual Send(string)
+ virtual Receive() : string
+ virtual GetHostname() : string
IProtocol
# hostname : string
+ virtual Initiate(policyId:string, key:string)
+ virtual Response(cpe:string, key:string) : string
# SendMessage(message:string)
# ReceiveMessage() : string
ProtocolISO
PolicyRepo
- _repo : map<string, pair<string,string>>
+ Exist(key:string) : bool
+ Get(key:string) : pair<string,string>
+ Get(name:string, event:string) : pair<string,string>
+ Add(key:string, action:string)
Thread
+ virtual start()
TriforkCommunication
- _cpeRepo : CpeRepo
- _clientRepo : ClientRepo
+ virtual start()
TriforkComOut
- _msgQ : MessageQueue<Message>
+ getMsgQPtr() : MessageQueue<Message>*
DHKeyPair
+ _pk : string
+ _sk : string
ClientRepo
- _repo : map<string,string>
+ Exist(key:string) : bool
+ Get(key:string) : string
+ Add(key:string, value:string)
CpeRepo
- _repo : map<string,string>
+ Exist(key:string) : bool
+ Get(key:string) : string
+ Add(key:string, value:string)
TriforkController
- policyRepo : PolicyRepo
Controller
+ virtual run()
MessageQueue<Message>
- q : queue<Message>
- m : pthread_mutex_t
- c : pthread_cond_t
+ enqueue(t:T)
+ dequeue() : T
TriforkComIn
Message
+ _entity : ENTITY
+ _name : string
+ _event : string
+ toString() : string
<<Enumeration>>
ENTITY
NA
CPE
CLIENT
SENSOR

Figure 7.6: Class diagram for Trifork's software.

Chapter 8

Implementation

The implementation chapter discusses the cryptographic techniques and functions used for the implementation of the Policy enforcement protocol. It concludes by describing the implementation of the protocol design.

8.1 Cryptographic techniques

This section explains the cryptographic techniques used for the Policy enforcement protocol. NaCl has been selected for cryptographic computations as Trifork is already using it. This means that the Salsa20 stream cipher will be used to ensure confidentiality, the Poly1305 MAC function will be used to provide data integrity and finally the Curve25519 elliptic-curve Diffie-Hellman function will be used for session key generation. Another approach for cryptographic techniques can be seen in table 8.1. Here AES could be substituted for Salsa20, an HMAC function could provide MACs instead of Poly1305 and a key generation based on a Diffie-Hellman key exchange using sub-group $\mathbb{Z}_p^*$. Even though the computations used for e.g. AES- and Salsa20 implementations are fairly simple, time and again, flawed implementations keep opening perfectly designed protocols to attacks such as side channel attacks[ZF05]. This is also a reason for selecting NaCl, as its considered secure against side channel attacks[Ber16].

	Cipher	**MAC**	**DHKE**
NaCl	Salsa20	Poly1305	Curve25519
Other	AES	HMAC	DH, $\mathbb{Z}_p^*$

Table 8.1: Cryptographic approaches.

The use of AES, HMAC and DH, $\mathbb{Z}_p^*$; could be relevant for other implementations where the customer might be more comfortable using NIST approved methods rather than NaCl. The *ICryptoLibrary* provides an interface, that the *NaClLibrary* class implements using the NaCl library. Another implementation of the *ICryptoLibrary* could be made using NIST approved techniques instead. This way Trifork's software could easily change between a NaCl library and a library using NIST approved techniques.

8.1.1 Linking NaCl with production code.

The NaCl crypto library can be downloaded from Daniel Bernstein's webpage[Ber]. It is build by running the file *do*. This will build the library for the environment. The library is linked to the production code using the paths below. Here the *nacl-20110221* folder is the NaCl library main folder.

Include path: ./nacl-20110221/build/ubuntu/include/amd64
Static library path: ./nacl-20110221/build/ubuntu/lib/amd64
Static library: nacl

The command for building the code for the *CPE* is shown below. Here the include path, the static library path and the static library are added to give the production code a reference to the NaCl library. If other object files are needed, they are added by project_ofiles. For this implementation the following object files are needed: *$(Static library path)/randombytes.o*, *SocketTCP.o* and *ProtocolISO.o*. Finally the *-lpthread* is added to allow the program to use threads.

```
g++ -I$(Include path) -L$(Static library path) $(project_ofiles)
      $(CPE).cpp -o $(CPE) -lnacl -lpthread
```

The *Makefile* for the entire program can be found on the USB in the folder *program*.

8.2 Policy enforcement protocol functions

This section describes the implementation of the two ProtocolISO functions: *Initiate* and *Response*; that are used for the Policy enforcement protocol. These are implemented based on the findings of the protocol design section. The *Initiate* function is used by protocol participant A and *Response* is used by protocol participant B. Before going into the protocol functions the functions used for encryption and decryption are explained.

8.2.1 Crypto helper functions

The protocol functions are using some helper functions, which handle encryption and decryption for the Policy enforcement protocol.

The first encryption function takes a message, a nonce and a symmetric key. The function uses the NaCl function *secret_box*, which is based on the Salsa20 stream cipher and the Poly1305 MAC. These both need a nonce and a key, to compute the ciphertext and MAC respectively. By using a nonce for ciphertext- and MAC generation, freshness can be assured indirectly for the encrypted message, rather than sending the nonce along in the message itself.

$$string\ Encrypt(string\ message, string\ nonce, stringkey)$$

The second encryption message is very similar to the first. The difference here is, that the message is encrypted using a private session key and a public session key, instead of a shared key. The function uses the NaCl function *crypto_box*. This function is also based on the Salsa20 stream cipher and the Poly1305 MAC. Additionally the Curve25519 elliptic-curve Diffie-Hellman function is used to generate the symmetric session key from the private session key and the public session key. This symmetric session key is then used to compute the ciphertext and MAC, similar to the encryption function above.

$$string\ Encrypt(string\ message, string\ nonce, string\ pk, string\ sk)$$

The decryption functions below are the opposites of the above encryption function. The functions are based on the NaCl decryption functions *secret_box_open* and *crypto_box_open*. These functions have the added functionality of checking data integrity, e.i. have the message been encrypted with the same key and nonce as used for decryption.

$$string\ Decrypt(string\ cipher, string\ nonce, string\ key)$$
$$string\ Decrypt(string\ cipher, string\ nonce, string\ pk, string\ sk)$$

When a nonce is used for MACs, it proves freshness of authentication for challenge-responses, without including the nonce in the message itself. This means that N_A can by excluded from message 2 and that N_B can be excluded from message 3, as they are already included as part of the MAC. The decryption function fails, if the correct nonce hasn't been used for the encryption.

8.2.2 The *Initiate* function

The *Initiate* function is supposed to send a policy id, *policyId*, using a symmetric key, *symKey*. The function starts by generating a nonce for challenge-response and a Diffie-Hellman key pair, containing a private- and public key. According to the protocol design the protocol starts by sending the nonce (*Message 1*).

Message 2 is received from the protocol responder. According to the design, this message is encrypted using the shared symmetric key between the two protocol participants. The message is also encrypted using the nonce. The nonce is part of the Poly1305 MAC, which means that the nonce isn't returned explicitly, but rather as part of the MAC. If decryption succeeds the protocol responder has used the correct nonce and symmetric key, which makes him trustworthy. The decrypted *Message 2* is then split into the three remaining values sent from the protocol responder: The protocol responder's nonce, the responder's public session key, and the identity of the *CPE*, the protocol responder believes to be communicating with. If this identity equals one's own identity, strong authentication is fulfilled, and the protocol can proceed.

Message 3 is then constructed by encrypting the policy id, with the session key, and encrypted again along with the the initiator's nonce and the initiator's public session key. The message is then finally encrypted using the provided nonce, $rNonce$, and the shared symmetric key, before sent to the protocol responder.

```
void ProtocolISO::Initiate(string policyId,string symKey)
{
  // Generates
  string nonce = _crypto->GenerateNonce();
  DHKeyPair dh = _crypto->GenerateDHKeyPair();

  // Message 1
  SendMessage(nonce);

  // Message 2
  string cM2 = ReceiveMessage();
  string m2 = _crypto->Decrypt(cM2,nonce,symKey);
  string rNonce = m2.substr(0,_crypto->_nonceSize);
  string rPk = m2.substr(_crypto->_nonceSize,_crypto->_pkSize);
  string iId = m2.substr(_crypto->_nonceSize+_crypto->_pkSize,
               m2.size()-_crypto->_nonceSize+_crypto->_pkSize);
  if(_hostname != iId) {exit(EXIT_FAILURE);}

  // Message 3
  string cPI = _crypto->Encrypt(policyId,rNonce,rPk,dh.sk);
  string m3 = _crypto->Encrypt(nonce+dh.pk+cPI,rNonce,symKey);
  SendMessage(m3);
}
```

8.2.3 The *Response* function

The *Response* function represents the other half of the Policy enforcement protocol. The function takes the *CPE* id of the *CPE* initiating the protocol and the symmetric key they share. The function starts by generating a nonce for challenge-response and a Diffie-Hellman key pair, containing a private- and public key. According to the protocol design the protocol starts by receiving the nonce, $iNonce$, of the protocol initiator (*Message 1*).

Message 2 is constructed by encrypting the *CPE* id, *cpeId*, of the protocol initiator, the responder's nonce, *nonce*, and the responder's public session key, *dh.pk*. The message is encrypted using the initiator's nonce, $iNonce$, and the shared symmetric key, $symKey$.

Message 3 is decrypted using the nonce, *nonce* and the symmetric key, $symKey$. Here the nonce is the one provided for message 2 to challenge the initiator. The nonce is used to verify the MAC along with the symmetric key. If the message is encrypted using the correct nonce and symmetric key, the message is decrypted. *Message 3* is then split into the message described in the protocol design: The nonce, $iNonce$; the public session key of the protocol initiator, and the encrypted policy id, *cPolicyId*. The $iiNonce$ is compared to the $iNonce$ value to ensure that the protocol participant is the same as the protocol initiator. Finally the policy id is decrypted using the session key: the public key of the protocol initiator and the protocol responder's private key.

```
string ProtocolISO::Response(string cpeId, string symKey)
{
  // Generates
  string nonce = _crypto->GenerateNonce();
  DHKeyPair dh = _crypto->GenerateDHKeyPair();

  // Message 1
  string iNonce = ReceiveMessage();

  // Message 2
  string m2 = nonce+dh.pk+cpeId;
  string cM2 = _crypto->Encrypt(m2, iNonce, symKey);
  SendMessage(cM2);

  // Message 3
  string cm3 = ReceiveMessage();
  string m3 = _crypto->Decrypt(cm3, nonce, symKey);
  string iiNonce = m3.substr(0, _crypto->_nonceSize);
  string iPk = m3.substr(_crypto->_nonceSize, _crypto->_pkSize);
  string cPI = m3.substr(_crypto->_pkSize+_crypto->_nonceSize,
              m3.size()-_crypto->_pkSize-_crypto->_nonceSize);
  if(iNonce != iiNonce) {exit(EXIT_FAILURE);}
  string policyId = _crypto->Decrypt(cPI, nonce, iPk, dh.sk);

  return policyId;
}
```

CHAPTER 9

Evaluation

The evaluation chapter presents the test ensuring, that the Policy enforcement protocol is secure, and the automated test ensuring that the negotiated output of the protocol is correct. Finally the prototype's test setup is explained.

9.1 Security evaluation of the Policy enforcement protocol

The OFMC is a model checker. It uses a Dolev-Yao intruder [BMV05], to analyse protocols for weaknesses. Weaknesses are determined based on the defined protocol goals. It is possible to set goals for authentication by using the keyword *authenticates* and to set goals for confidentiality by using the keyword *secret*.

Figure 9.1 shows the Policy enforcement protocol implemented in the OFMC syntax. Here the goals are set to: keep the policy id secret between the two protocol participants A and B, let A authenticate B by A's nonce challenge N_A, and to let B authenticate A by B's nonce challenge N_B.

The OFMC protocol implementation follows the implementation of the Policy enforcement protocol. A is the protocol initiator, and B is the protocol responder. This is just as represented in the protocol design.

Message 1: The protocol initiator, starts the protocol by sending a freshly generated nonce, N_A.

B now knows the nonce value N_A.

Message 2: The protocol responder sends back three values: B's nonce, N_B; B's public session key, $exp(g, Y)inv(mod(B))$; and the identity of A. A MAC is also provided to ensure data integrity of the message, notice here that A's nonce, N_A, is included as response to A's challenge. This is similar to what the $Poly1305$ MAC does for the implementation. Finally the message is encrypted using A and B's shared key, $\{message\}sym(A, B)$.

A now knows the three values: B's nonce, B's public session key and the identity of the protocol initiator. B has now been authenticated to A, and A can make sure of strong entity authentication, by comparing the protocol initiator's identity provided by B to A's own identity. Just as done for the implementation.

Message 3: The protocol initiator, sends three values to B: A's nonce, N_A; A's public session key, $exp(g, X)inv(mod(A))$; and the policy id encrypted by the session key, $\{|PolicyId|\}exp(exp(g, Y), X))$. Just as for the implementation. The nonce of B, N_B, is only included as part of the MAC, just as for the $Poly1305$ MAC.

B can now authenticate A with the MAC, which includes N_B, and ensure that A in fact initiated the protocol by comparing the two values of N_A received from the initiator. Just as done for the implementation. B now also knows the session key, and can therefore decrypt the policy id. All goals have been reached, and the protocol is done.

The policy id has been kept secret between A and B, A was authenticated to B and B was authenticated to A. This is at least true from what has been presented so far. The OFMC model checker does support the theory. It didn't find any attacks for the protocol implementation. The OFMC test can by found in the folder *test/ofmc* on the supplied USB, and can be confirmed using the command below:

$$./ofmc\text{-}mac\ Symmetric\_ISO6.AnB\ \text{–}numSess\ 2$$

```
Protocol: ISO6

Types:
   Agent      A, B;
   Number     NA, NB, X, Y, g, PolicyId;
   Function   mod, sym, mac;

Knowledge:
   A: A,B,sym(A,B),inv(sym(A,B)),mod,inv(mod(A)),g,mac;
   B: A,B,sym(A,B),inv(sym(A,B)),mod,inv(mod(B)),g,mac;

Actions:
   A -> B:  NA

   B -> A:  { NB, A, {exp(g,Y)}inv(mod(B)),
              mac(NA,NB,A,{exp(g,Y)}inv(mod(B))) }sym(A,B)


   A -> B:  { NA, {exp(g,X)}inv(mod(A)),
              {|PolicyId|}exp(exp(g,Y),X),
              mac(NB,NA,{exp(g,X)}inv(mod(A)),
                   {|PolicyId|}exp(exp(g,Y),X)) }sym(A,B)

Goals:
   A authenticates B on NA
   B authenticates A on NB
   PolicyId secret between A,B
```

Figure 9.1: OFMC test of the Policy enforcement protocol

9.2 Protocol integration test

The test in figure 9.2 was used during implementation of the ProtocolISO class, to test the communication between the two protocol participants. The two functions: *Initiator* and *Responder*; are thread functions, which represent the two protocol participants.

The *Responder* function simply accepts connections and when connected to, it starts responding to the Policy enforcement protocol. If the correct policy id is received, the test is successful.

The *Initiator* function connects to itself as the host, thereby connecting to the *Responder* function's open connection. The Policy enforcement protocol is then initiated using the function *Initiate* of the ProtocolISO class.

The test is a simple test for developing the Policy enforcement protocol functions of the ProtocolISO class: *Initiate* and *Response*. The protocol test can be found on the USB at *program/test_protocol*.

```
string cpeId = "CPE1";
string policyId = "12345678";
string secretKey = "10101010101010101010101010101010";

void *Initiator(void *ptr)
{
    SocketTCP tcp_initiator;
    tcp_initiator.ConnectLocal(cpeId, TRIFORK_PORT);
    ProtocolISO protocol(&tcp_initiator, new NaclLibrary());
    protocol.Initiate(policyId, secretKey);
}

void *Responder(void *ptr)
{
    SocketTCP tcp_responder;
    tcp_responder.AcceptConnection(TRIFORK_PORT);
    ProtocolISO protocol(&tcp_responder, new NaclLibrary());
    string receivedPolicyId = protocol.Response(cpeId, secretKey);

    if(receivedPolicyId == policyId) {
        cout << "TEST:SUCCESSFUL" << endl;
    }
    else {
        cout << "TEST:UNSUCCESSFUL" << endl;
    }
}
```

Figure 9.2: Thread functions used to integration test the Policy enforcement protocol

9.3 Prototype

I have successfully created four program, which correspond to the processes from the system design chapter. The four programs are: *mainClient*, *mainCPE*, *mainActuator* and *mainSensor*. They can be found in the folder *program* on the USB.

I have tested the programs using a virtual machine setup running the linux distribution ubuntu. Figure 9.3 illustrates the setup, using 3 virtual machines communicating using TCP. I have been successful in distributing policies from the *Client* to the *CPEs*. The policies are then being enforced by the *CPEs*. This has been successfully confirmed by initiating an event from the sensor. The *CPE*, connected to the sensor, contacts the other *CPE* and using the Policy enforcement protocol, they exchange the policy id corresponding the sensor event. The *CPE* receiving the event then successfully informs the actuator to perform an action.

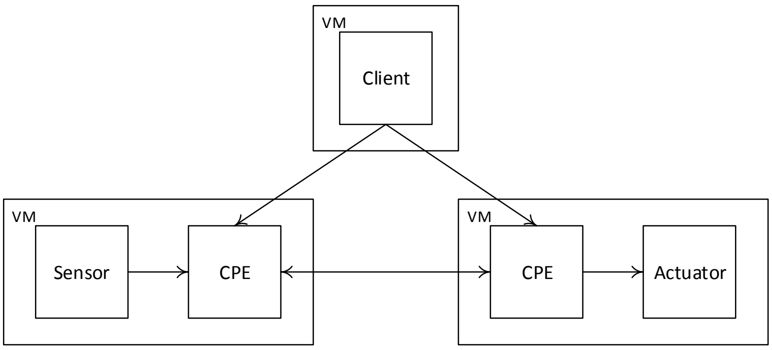


Figure 9.3: The test setup using three virtual machines (VM).

The prototype is successfully using the Policy enforcement protocol to securely transferring the policy id, when a sensor event fulfils a policy. Thereby giving the illusion of one home automation systems, when there is in fact is two different home automation systems using two different *CPEs*.

CHAPTER 10

Conclusion

Trifork would like to have *CPEs* communicate securely. This would allow customers to create automation process across multiple home automation systems. The proposed solution reshapes the existing system as a policy-based system, where sensor events from one *CPE* is paired with actuator actions by policy ids. This means, that only a policy id needs to be transferred between *CPEs*. The transfer of the policy id follows the security properties confidentiality, data integrity and authentication, to ensure security. Furthermore some refined requirements are created to address the Dolev-Yao attack model. These are mutual authentication, strong entity authentication and good key. The refined requirements help to prevent replay attacks, relay attacks and data analysis of package data. Two protocols, ISO-6 and STS, are then analysed for the protocol design. They both fulfil the protocol requirements, and are evaluated in order to select the better protocol, for transferring the policy id. The ISO-6 protocol is selected, as it is more efficient, and require less storage for key material. A system design and implementation is created for the prototype. It is created to imitate the flow from a sensor of one system to the action of an actuator of another system. The implementation uses the ISO-6 protocol for communication between *CPEs* and uses the NaCl crypto library for cryptographic computations. The security of the protocol implementation is evaluated using the OFMC model, which doesn't find any attacks on the protocol.

Working with Trifork's problem has given me a great insight into what is required to created a secure line of communication. You need to know, which type of attacker you are defending against and which security goals should be fulfilled. Without it, there is no way of evaluating, if the system is secure. The existing setup also have a great impact on how you design your system. Does protocol participants have a trusted third party available, or do you need to pre-distribute keys, in order to achieve a secure line communication. Symmetric keys encryption is way more efficient than asymmetric encryption, but asymmetric encryption has the benefit of scaling better than, as certificates can keep the need for key storage constant. Asymmetric encryption also provides non-repudiation.

I think the important thing to take away for the thesis is, that there is no one way of doing things, it all depends. It depends the existing setup, the needed for speed or the need for scalability, and are you defending against an active attacker or a passive attacker. All these external factors and requirements all help you to reach the best solution under the given circumstances.

Appendix A

Tests

A.1 Test of STS protocol payload and cryptographic computations

This test compares message 2 and 3 from the two version of the STS protocol; protocol 6.3 and protocol 6.4. The test files can be found on the USB in folder */tests/sts_test*.

Message 2 and message 3 differ some, and to see the affects of changing between the two, payload and computational time is considered. Only the part that differs in the two protocols are considered, which is why the public session keys and certificates are not included.

The results from the payload testing can be seen in figure A.2.

2. STS Protocol 6.3	$\{\vert Sig_B([yG, xG])\vert\}K_{xyG}$	144 bytes
2. STS Protocol 6.4	$Sig_B([yG, xG, A])$	128 bytes
3. STS Protocol 6.3	$\{\vert Sig_A([xG, yG]), \underline{policy_{id}}\vert\}K_{xyG}$	152 bytes
3. STS Protocol 6.4	$Sig_A([xG, yG]), \underline{\{\vert policy_{id}\vert\}K_{xyG}}$	152 bytes

Figure A.1: Payload for STS protocols.

The results from the computational testing can be seen in figure A.2. Results is an average of 1.000.000 computations.

2. STS Protocol 6.3	$\{\vert Sig_B([yG, xG])\vert\}K_{xyG}$	2.09207 ms
2. STS Protocol 6.4	$Sig_B([yG, xG, A])$	2.00835 ms
3. STS Protocol 6.3	$\{\vert Sig_A([xG, yG]), \underline{policy_{id}}\vert\}K_{xyG}$	2.01198 ms
3. STS Protocol 6.4	$Sig_A([xG, yG]), \underline{\{\vert policy_{id}\vert\}K_{xyG}}$	1.95034 ms

Figure A.2: Payload for STS protocols.

Protocol 6.4's message 2 is computed $2.09207/2.00835 = 1,04168$ times faster, that message 2 of protocol 6.3.

Protocol 6.4's message 3 is computed $2.01198/1.95034 = 1,03160$ times faster, that message 3 of protocol 6.3.

A.2 Storage requirements for different key approaches

Tabel A.1 shows the different storage need for keys using the ISO protocol and the STS protocol without certificates. For each relation ISO needs an id of 8 bytes and a key of 32 byte. STS need an id of 8 bytes and a public key of 32 bytes for every relation. Additionally STS also needs 64 bytes for ones own private key and 32 bytes for the corresponding public key.

Number of relations	ISO-6	STS no cert.
1	40	136
2	80	176
3	120	216
4	160	256
5	200	296

Table A.1: Comparison of storage between ISO and STS.

Tabel A.2 shows the storage requirements for the STS protocol with and without certificates. It only shows the difference is storage which is why ones own key pair isn't included. STS without certificates need an id of 8 bytes and a public key of 32 bytes for every relation. STS with certificates only need the certificate, with is why the size is constant.

Number of relations	STS without cert.	STS with cert.
1	40	160
2	80	160
3	120	160
4	160	160
5	200	160

Table A.2: Comparison of storage between STS with and without certificates.

A.3 Test of payload and efficiency

Table A.3 presents the payload for the different messages of protocol 6.2 (ISO) and protocol 6.5 (STS).

Message	ISO	STS	STS / ISO
1. $A \rightarrow B$	40 B	40 B	1
2. $B \rightarrow A$	80 B	160 B	2
3. $A \rightarrow B$	96 B	152 B	1.6

Table A.3: Comparison of payloads.

Tabel A.4 presents the computation time for encryption and decryption for the different messages of protocol 6.2 (ISO) and protocol 6.5 (STS). The encryption and decryption time are considered together to compensate for the difference in encrypting and decrypting signatures. This mean that it takes $7.12549ms$ to encrypt and decrypt the signature of message 2 for the STS protocol.

Message	ISO	STS	STS / ISO
2. $B \rightarrow A$	0.001802 ms	7.12549 ms	3954
3. $A \rightarrow B$	0.002046 ms	7.11178 ms	3476

Table A.4: Comparison of cryptographic computation time.

A.4 Test of STS protocol security with ofmc

```
Protocol: STS_Certificate

Types:
      Agent A,B,C;
      Number X,Y,g,PolicyId;
      Function pk,mod;

Knowledge:
      A: A,B,C,mod,inv(mod(A)),g,pk(A),inv(pk(A)),
         {A,pk(A)}inv(pk(C)),pk(C);
      B: A,B,C,mod,inv(mod(B)),g,pk(B),inv(pk(B)),
         {B,pk(B)}inv(pk(C)),pk(C);
      C: A,B,C,pk(C),inv(pk(C))
      where C != i

Actions:
      A -> B: A,{exp(g,X)}inv(mod(A))

      B -> A: {exp(g,Y)}inv(mod(B)),
              B,pk(B),{B,pk(B)}inv(pk(C)),
              {{exp(g,X)}inv(mod(A)),
              {exp(g,Y)}inv(mod(B))}inv(pk(B))

      A -> B: A,pk(A),{A,pk(A)}inv(pk(C)),
              {{exp(g,X)}inv(mod(A)),
              {exp(g,Y)}inv(mod(B))}inv(pk(A)),
              {|PolicyId|}exp(exp(g,Y),X)

Goals:
      A authenticates B on {exp(g,X)}inv(mod(A))
      B authenticates A on {exp(g,Y)}inv(mod(B))
      PolicyId secret between A,B
```

Appendix B

Diagrams

B.1 Manufacturer class diagram

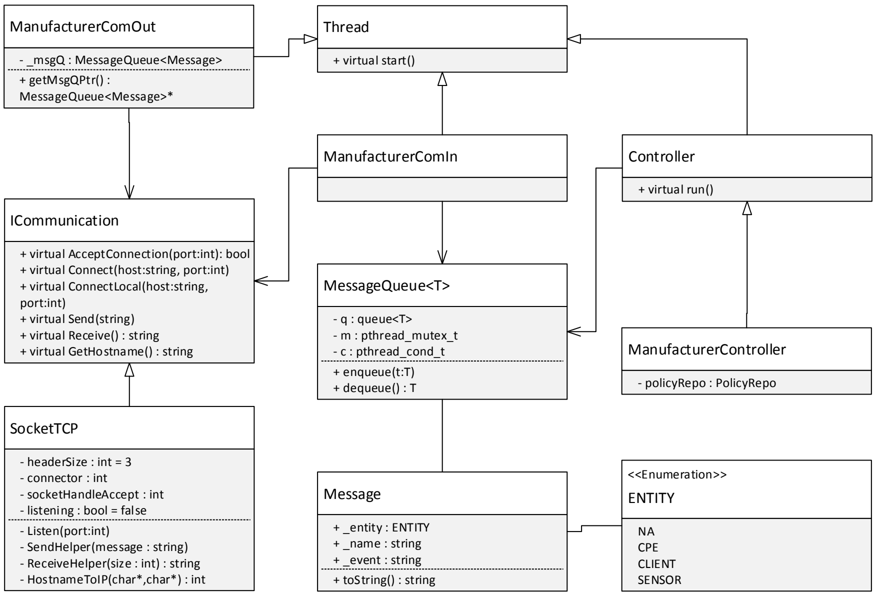

ManufacturerComOut
- _msgQ : MessageQueue<Message>
+ getMsgQPtr() : MessageQueue<Message>*
Thread
+ virtual start()
ManufacturerComIn
Controller
+ virtual run()
ICommunication
+ virtual AcceptConnection(port:int): bool
+ virtual Connect(host:string, port:int)
+ virtual ConnectLocal(host:string, port:int)
+ virtual Send(string)
+ virtual Receive() : string
+ virtual GetHostname() : string
MessageQueue<T>
- q : queue<T>
- m : pthread_mutex_t
- c : pthread_cond_t
+ enqueue(t:T)
+ dequeue() : T
ManufacturerController
- policyRepo : PolicyRepo
SocketTCP
- headerSize : int = 3
- connector : int
- socketHandleAccept : int
- listening : bool = false
- Listen(port:int)
- SendHelper(message : string)
- ReceiveHelper(size : int) : string
- HostnameToIP(char*,char*) : int
Message
+ _entity : ENTITY
+ _name : string
+ _event : string
+ toString() : string
<<Enumeration>>
ENTITY
NA
CPE
CLIENT
SENSOR

Figure B.1: Package diagram for *CPE*.

Bibliography

[Bar15] Elaine Barker. Recommendation for key management - part 1: General (revision 4). *DRAFT NIST Special Publication 800-57, Part 1, Rev. 4*, September 2015.

[Ber] Daniel J. Bernstein. https://nacl.cr.yp.to.

[Ber05] Daniel J Bernstein. The poly1305-aes message-authentication code. In *Fast Software Encryption*, pages 32–49. Springer, 2005.

[Ber06] Daniel J Bernstein. Curve25519: new diffie-hellman speed records. In *Public Key Cryptography-PKC 2006*, pages 207–228. Springer, 2006.

[Ber08] Daniel J Bernstein. The salsa20 family of stream ciphers. In *New stream cipher designs*, pages 84–97. Springer, 2008.

[Ber16] Daniel J. Bernstein, 2016.

[BFK09] Jens Bender, Marc Fischlin, and Dennis Kügler. Security analysis of the pace key-agreement protocol. In *Information Security*, pages 33–48. Springer, 2009.

[BM03] Colin Boyd and Anish Mathuria. *Protocols for Authentication and Key Establishment.* Springer Science & Business Media, 2003.

[BMV05] David Basin, Sebastian Mödersheim, and Luca Vigano. Ofmc: A symbolic model checker for security protocols. *International Journal of Information Security*, 4(3):181–208, 2005.

[DH76] Whitfield Diffie and Martin E Hellman. New directions in cryptography. *Information Theory, IEEE Transactions on*, 22(6):644–654, 1976.

[DR98] Joan Daemen and Vincent Rijmen. Aes proposal: Rijndael. 1998.

[DVOW92] Whitfield Diffie, Paul C Van Oorschot, and Michael J Wiener. Authentication and authenticated key exchanges. *Designs, Codes and cryptography*, 2(2):107–125, 1992.

[DY83] Danny Dolev and Andrew C Yao. On the security of public key protocols. *Information Theory, IEEE Transactions on*, 29(2):198–208, 1983.

[FSK05] Bryan Ford, Pyda Srisuresh, and Dan Kegel. Peer-to-peer communication across network address translators. In *USENIX Annual Technical Conference, General Track*, pages 179–192, 2005.

[Gara] Gartner. Gartner identifies the top 10 strategic technologies for 2012, http://www.gartner.com/newsroom/id/1826214.

[Garb] Gartner. Gartner identifies the top 10 strategic technologies for 2013, http://www.gartner.com/newsroom/id/2209615.

[Garc] Gartner. Gartner identifies the top 10 strategic technologies for 2014, http://www.gartner.com/newsroom/id/2603623.

[Gard] Gartner. Gartner identifies the top 10 strategic technologies for 2015, http://www.gartner.com/newsroom/id/2867917.

[Gre14] Sari Stern Greene. *Security Policies and Procedures: Principles and Practices*. Pearson Education, Inc., 2014.

[ISO96] ISO. Information technology - security techniques - key management - part 2: Mechanisms using symmetric techniques. *ISO/IEC 11770-2*, 1996.

[JMV01] Don Johnson, Alfred Menezes, and Scott Vanstone. The elliptic curve digital signature algorithm (ecdsa). *International Journal of Information Security*, 1(1):36–63, 2001.

[MvOV01] Alfred J. Menezes, Paul C. van Oorschot, and Scott A. Vanstone. *Handbook of Applied Cryptography*. ACM Digital Library, 2001.

[NIS08] NIST. The keyed-hash message authentication code (hmac). Technical report, National Institute of Standards and Technology, 2008.

[SSG13] Rounak Sinha, Hemant Kumar Srivastava, and Sumita Gupta. Performance based comparison study of rsa and elliptic curve cryptography. *International Journal of Scientific and Engineering Research*, 2013.

[Tri] Trifork. http://securedevicegrid.com/.

[ZF05] YongBin Zhou and DengGuo Feng. Side-channel attacks: Ten years after its publication and the impacts on cryptographic module security testing. *IACR Cryptology ePrint Archive*, 2005:388, 2005.